In: Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'2008)

Model-driven Engineering Metrics for Real Time Systems

M. Monperrus[1,2], J-M. Jézéquel[2], J. Champeau[1], B. Hoeltzener[1]

1: ENSIETA - 2, rue F. Verny - F29206 Brest Cedex
2: INRIA/IRISA - Campus de Beaulieu - F-35042 Rennes Cedex

Abstract

As with other software development processes, model-driven engineering of real time software systems include quality assurance and measurement. Model-driven engineering (MDE) supports the development of real-time software systems by means of a set of languages, processes, methods and tools. To measure the models, a dedicated measurement software has to be developed, which is costly. In this paper, we propose a framework to concisely define and automatically implement an open-ended family of metrics for real time software systems. The overall contribution of this approach is to give an instant, reliable and low cost implementation of model metrics seamlessly integrated into modeling tools.

1 Introduction

As with other software development processes, model-driven engineering of real time software systems include quality assurance [1]. Assessing the quality of a product built from a set of models, involves the measurement of the resulting target. However, to be able to drive the development process all along the life cycle, it is needed to measure the models themselves as well.

Model-driven engineering (MDE) supports the development of real-time software systems by means of a set of languages, processes, methods and tools [2]. Domain specific modeling languages (DSML) are specified and built so as to raise the level of abstraction and increase the productivity.

To measure the models, a dedicated measurement software has to be developed, which is costly. Time-to-market and cost constraints do not allow to build an ad hoc metric software. Indeed, the NASA recommends that *the cost of measurement should not add more than 2 percent to the software development or maintenance effort* [3].

In this paper, we propose a framework to concisely define and automatically implement an open-ended family of metrics for real time software systems. This framework generates the whole implementation and environment, both integrated into a modeling tool. This framework enables to concentrate only on the conceptual level of the development of metrics. The overall contribution of this approach is to give an instant, reliable and low cost implementation of metrics seamlessly integrated into modeling tools.

The remainder of this paper is organized as follows. In section 2, we explore the research question. We then present our proposal (section 3), applied to the measurement of a large scale AADL model (section 4). We finish with a related work (section 5) and conclude.

2 Research Question

In table 1, we list the high level requirements of a real time model measurement software. Despite the small number of requirements, each of them hides an intrinsic complexity. For instance, requirement #2 implies to have an open modeling environment, to know its internals and to master the programming languages used to implement it.

Indeed, a paper relating industrial experiences of software measurement points out five lessons [4]. The second lesson states that *software measurement should focus on a particular area to limit the cost of measurement [...]*. Measurement is part of software economics.

Model driven engineering involves models as inputs, and other models or source code as outputs. This generative approach seeks to cut off the development costs. An analysis of the requirements presented in table 1 concludes that they are all addressable by code generation. This is emphasized in the last column of the table. To sum up, our research question is how to reduce the cost of development of real time model measurement software. Our intuition is to leverage model-driven engineering to generate the measurement software from an high level description of what to measure.

#	Description	Generatable
1	The model measurement tool is accessible via a command-line interface.	Yes
2	The model measurement tool integrates into a modeling environment.	Yes
3	The code includes an API to implement estimation models on top of metrics.	Yes
4	The model measurement tool is extensible and gives a way to add new model metrics.	Yes.

Table 1: Main requirements for a measurement tool.

3 Solution

3.1 Architecture

In this section, we present the architecture of model driven measurement software (first introduced in [5]).

The design of a DSML environment starts with the metamodeling activity. DSMLs are specified by the domain expert with a metamodel which identifies the concepts of the domain and their relationships. The domain expert also knows what needs to be measured. Hence, we propose to give him a way of defining the domain metrics. This is achieved by a metric metamodel that identifies the good concepts for the metric specification. Compared to an ad hoc measurement software development, the domain expert defines the metrics at a higher level of abstraction than code. These two functions are depicted in figure 1.

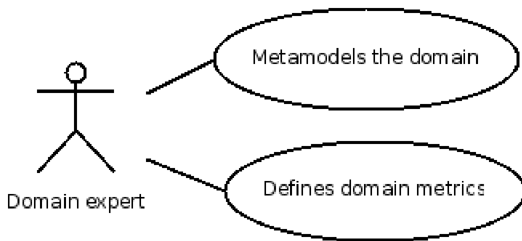

Figure 1: The domain expert function

Tools have to be implemented to create, edit and browse models as well as to check the conformity to the metamodel. However, there exists toolchains that generate this modeling environment conform to a metamodel (e.g. EMF[1] / Topcased[2]). We propose to add a module to such a toolchain that compiles the declarative description of metrics directly in the DSML modeling environment to augment it with metrics capabilities. This is depicted in figure 2.

[1]Eclipse Modeling Framework, see www.eclipse.org/emf

[2]Topcased is an open source CASE environment, see www.topcased.org

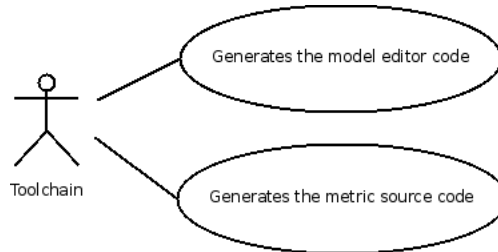

Figure 2: The toolchain function

Finally, the DSML engineer is able to check its model against the metric values, as shown in figure 3. Changes are instantly reflected in the metric module of the modeling environment. This approach has three major advantages:

- metrics are defined at a higher level of abstraction without caring about how this can be implemented in a DSML modeling environment;
- the implementation of the enhanced editor is generated;
- DSML engineers can edit and measure models in the same environment.

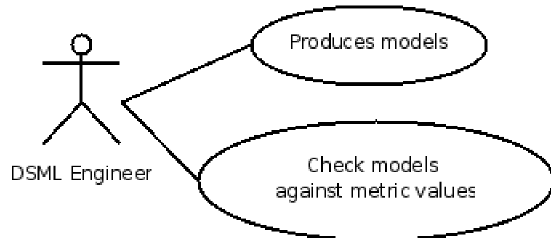

Figure 3: The engineer function

To sum up, the whole process is sketched on figure 4. This architecture generates the whole measurement software integrated into a modeling tool. This eventually gives an enhanced modeling environment with metric capabilities.

3.2 Metric Metamodel

In this section, we present the metric metamodel. This metamodel comes from the fact that a lot of metrics in the literature are a kind of count metrics, i.e. the number of a given kind of elements. The difficulty is to precisely define the elements that needs

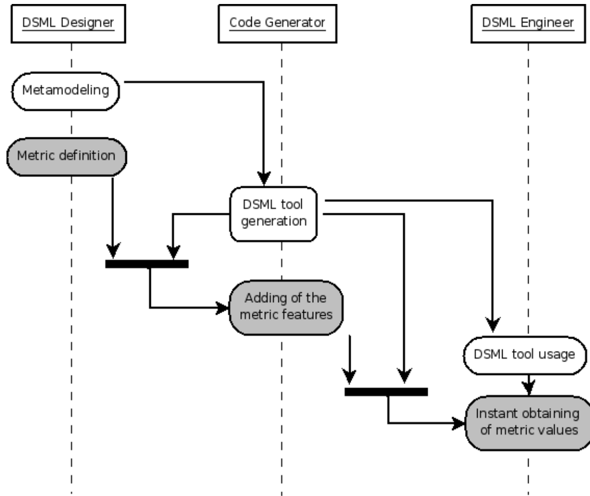

Figure 4: Model-driven engineering of metrics

to be counted. The definition of what to count is called a predicate. Ratio, sums, weighted sums can then be computed on top of these count metrics. All these abstractions have been gathered in a metamodel presented in the next paragraphs. We first present the general mechanisms to specify metrics, then the specification of predicates [6] in two distinct paragraphs.

In figure 5 is shown the backbone of the metric metamodel. The domain expert defines a *MetricSpecificationSet*, composed of *MetricSpecification*. Since the term "metric" is polymorphic, we use the term metric specification to be opposed to metric values. A *MetricSpecification* is either a *PredicateMetricSpecification*, a *ConstantMetricSpecification*, that simply handles constant values in metric expressions, and a *DerivedSpecification* that handle arithmetic operators and more complex ones, e.g. the exponential function (hidden in the figure).

A *PredicateMetricSpecification* contains a *Predicate*. A predicate is a function that takes an object as input and makes tests on it to determine if it has to be counted. Predicates refer to the concepts expressed in the metamodel. The tests are made on types, attribute values and/or referenced objects. A predicate can be as long and complex as needed. The metric value of a *PredicateMetricSpecification* is the number of elements of a given model that satisfy the predicate.

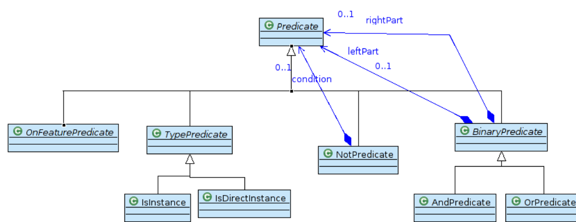

Figure 6: Main part of the predicate metamodel

As shown in figure 6, 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 handle tests on the type of the model element *IsInstance* and *IsDirectInstance*: *IsInstance* tests the metaclass the model element; *IsDirectInstance* tests the metaclass or one of its superclasses. The remaining class is central and handles tests on features of a given model element. In this context, a feature means a slot (e.g. a string "foo"), or a reference to another model element. This *OnFeaturePredicateClass* is further presented in the next paragraph.

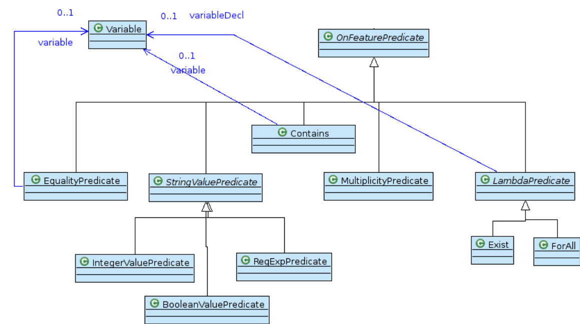

Figure 7: Features management in the metric metamodel

The figure 7 shows what are the possible tests on a given feature, which are:

- *EqualityPredicate* enables to test whether a reference points to a model element referred by a variable;
- *StringValuePredicate* enables to test the value of a slot (e.g. a boolean equals to "false" or an integer equals to "13");
- *ContainsPredicate* enables to test if a collection contains a model element referred by a variable;
- *MultiplicityPredicate* enables to test the actual size of a collection (not the multiplicity of the reference in the metamodel);
- *LambdaPredicate* enables to apply a predicate to all elements of a collection. It is subclassed in *Exists* and *ForAll* to express first-order logic quantifiers.

Given that a predicate can be as long and complex as needed and that predicate metrics can be composed together, this metamodel let the domain expert define an open-ended family of metrics. We show in the next sections our implementation choices and apply this framework to an AADL model.

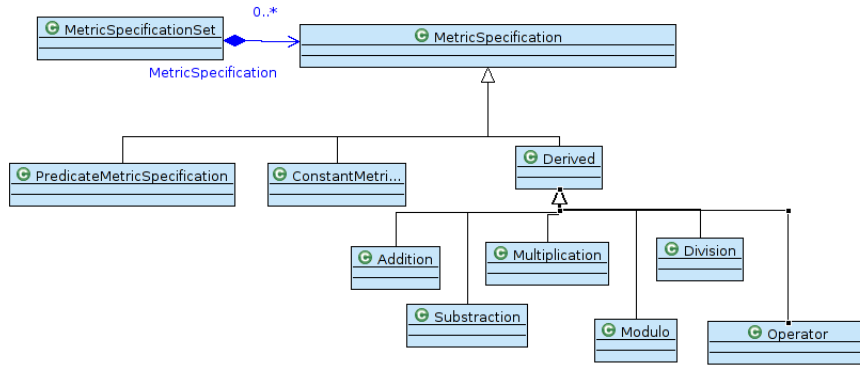

Figure 5: General mechanisms of the metric metamodel

3.3 Implementation

Our prototype relies on the following technologies: the metric metamodel is expressed in EMF/Ecore; the metric specification can be expressed as an EMF model, or using Kermeta [3]. For each metric specification, a Kermeta program is generated. This Kermeta program is built on top of a metric library, implemented as an aspect on the metric metamodel.

Since, EMF and Kermeta work inside Eclipse, we use the Eclipse API to listen to changes on models files in order to automate the computation of metric values. This prototype is a proof that our approach solves the requirements of table 1 by means of model-driven engineering.

4 Case study

Our case study consists of the generation of a measurement software for AADL models. We then compute and present the metric values for an open source industrial AADL model. The Architecture Analysis & Design Language (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 [8]. An excerpt of the AADL metamodel is given figure 8.

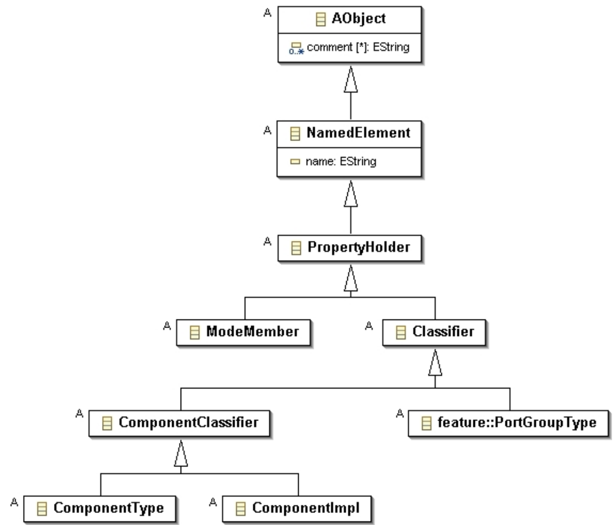

Figure 8: Excerpt of the AADL metamodel

The AADL model used is an open source model of a flight display system. It was created by Rockwell Collins and is part of the available resources on the AADL website. It consists of 30400 model elements. An excerpt of this model is given figure 9 with the standard textual syntax.

```
bus Node_Switch
  properties
    Network_Speed => 120 Kbps;
    Max_Transmission_Rate => 160 Kbps;
    Max_Packet_Size => 1 KB;
end Node_Switch;

thread Periodic_1_Hz
  properties
    Dispatch_Protocol => Periodic;
    Period => 1000 Ms;
end Periodic_1_Hz;
```

Figure 9: Excerpt of the AADL model used

We chose 9 metrics used by the real time system

[3] a model-driven language and workbench, see `www.kermeta.org` and [7]

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 2: The generated measurement software output

architects of an automotive company. Our contribution coupled to AADL give them the way to automate and obtain reliable values compared to their actual Word/Excel practices. The metric definitions are presented in the list below. D* stands for derived, P* stands for predicate based metric, as in the metamodel.

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., $D4 \rightarrow 1$, it is extremely hard to predict the load. In this case, real time systems architects augment their security margins.

For sake of brevity we present only two formalized metric specifications below (in this case study the predicates are written in Kermeta). Note that, the complexity of the predicate depends on the metric and the metamodel architecture.

```
// expression of the predicate of N1
result:= component::DeviceImpl.isInstance(this)

// expression of the predicate of N4
result:= this.asType(component::ThreadImpl)
  compType.properties.propertyAssociation.exists{ x |
    x.propertyValue.exists{z |
      (z.asType(~property::EnumValue)
        .enumLiteral.name == "Aperiodic"
      or z.asType(~property::EnumValue)
        .enumLiteral.name == "Sporadic")}}
```

The framework compiles the specification of metrics into a Kermeta program built on top of a Kermeta library. With the proposed framework the declarative specification of metrics is about 40 lines of code, mainly to expressed the predicates in Kermeta. The whole implementation of the generated measurement software is about 1000 lines. Given that Kermeta is more concise than Java, and that the library is the result of our measurement know-how, the added value of our generative approach to measurement software is significant. The workload of the metric definition by the domain expert is reduced to its formalization in textual or graphical syntax of the metric metamodel. The implementation is entirely automated.

This generated measurement software takes the large scale AADL model mentionned above and produce the needed metric values, as shown in table 2. According to architects, the most important metrics are P5, P3, P4. They all indicate that the model considered is manageable in regard to their know-how. For instance, we note that the schedulability difficulty indicator D3 is less than the empirical upper bound (30). According to project managers, P1, P2 and P3 are important. They need to be saved in a database, in order to ground future development process analysis, cost and risk estimation.

This case study illustrated the applicability of our model-driven approach to the measurement of real-time models. The framework is intended to obtain low cost measurement software, not only for AADL, but also for any real time systems metamodel.

5 Related Works

Metamodels and metrics Misic et al. [9] express a generic object-oriented metamodel using Z. With this metamodel, they express a function point metric using the number of instances of a given type. They also express a previously introduced metric called the *system meter*. Along the same line, [10] extends the UML metamodel to provide a basis for metrics expression and then use this model to express known metrics suite with set theory and first order logic. Our approach also uses object models, set theory and logic. However, our approach is fully executable.

In [11], the authors address 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. We also believe that metrics are needed in early life cycle and the genericity of the predicate based metrics makes it usable for design metrics. But it is also usable at any moment of a product life-cycle, since a step of a design process can be considered as a domain per se.

On the implementation issue, [12] 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. We also believe that it is needed to access the semantic elements of source code to express metrics. But the implementation presented in [12] is complex and tightly coupled with the source code and the metamodel unlike the predicate based metric and our associated prototype which are metamodel-independent. In [13], the authors use XQuery to express metrics. Metrics are then computed on XMI files. Our implementation relies also on XML. But thanks to the Eclipse Modeling Framework, it has a direct and complete access to the metamodel whereas when considering syntactic XML and XQuery, one has no direct access to the metamodel.

Baroni et al. [14] propose to use OCL to express metrics. They use their own metamodel exposed in a previous paper. Likewise, in [15], the authors use Java bytecode to instantiate the Dagstuhl metamodel and express known cohesion metrics in OCL. Our implementation relies on Kermeta which permits OCL-like queries.

Note that the scope of all these works are object oriented metrics. In this paper, the considered metrics are real time model metrics.

Abstraction level for expressing metrics On the genericity of metrics, Alikacem et al. [16] emphasize on the need not to hard-code metrics and to express metrics in a generic manner and refer to the proposition made in [17]. In this paper, Mens et al. define a generic object-oriented metamodel and generic metrics. They then show that known software metrics are an application of these generic metrics. The predicate-based metric is close to the *NodeCount* of [17]. Since *NodeCount* aims primarily at computing classical OO metrics such as the number of methods per class, *NodeCount* has a local view and considers a root node. On the contrary the predicate-based metric is at the model level and considers the model as a whole.

Marinescu et al. [18] propose a simplified implementation of object-oriented design metrics with a metric language called SAIL. Logically, some of the key mechanisms of SAIL, namely navigation, selection, set arithmetic are present in the predicates presented in this paper. Compared to them, we leverage generative techniques to integrate the domain-specific metrics into a DSML environment.

6 Conclusion

The current state of the art of real time systems development is moving toward more high level descriptions of systems. The Architecture Analysis & Design Language (AADL) is one of them. However for AADL, as for any metamodel of real time systems, the developement cost of the modeling environment, as well as its measurement features, has to be controlled.

To address this issue, we presented a way to generate a measurement software for measuring real time models. From a set of metric specifications made by the domain expert, the whole tool is generated and is able to compute metric values right out of the box. To address industrial scale and quality, the proof-of-concept research prototype has to be redeveloped in an future project or partnership in order to generate a whole Java/EMF Eclipse plugin for the model measurement. Moreover, further research is currently made to enrich the metric metamodel and thus widen the space of possible metric definitions.

References

[1] S. H. Kan, *Metrics and Models in Software Quality Engineering.* Reading, MA: Addison Wesley, 1995.

[2] D. C. Schmidt, “Model-driven engineering,” *IEEE Computer*, vol. 39, pp. 25–31, February 2006.

[3] N. S. Program, "Software measurement guidebook," tech. rep., National Aeronautics and Space Administration, 1995.

[4] E. van Solingen, R.; Berghout, "Integrating goal-oriented measurement in industrial software engineering: industrial experiences with and additions to the goal/question/metric method (gqm)," pp. 246–258, 2001.

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

[6] M. Monperrus, J.-M. Jézéquel, J. Champeau, and B. Hoeltzener, "Measuring models," in *Model-Driven Software Development: Integrating Quality Assurance* (J. Rech and C. Bunse, eds.), IDEA Group, to appear.

[7] P. A. Muller, F. Fleurey, and J. M. Jézéquel, "Weaving executability into object-oriented meta-languages," in *Proceedings of MODELS/UML 2005*, 2005.

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

[9] V. B. Misic and S. Moser, "From formal metamodels to metrics: An object-oriented approach," in *TOOLS '97: Proceedings of the Technology of Object-Oriented Languages and Systems-Tools - 24*, (Washington, DC, USA), p. 330, IEEE Computer Society, 1997.

[10] R. Reissing, "Towards a model for object-oriented design measurement," in *ECOOP'01 Workshop QAOOSE*, 2001.

[11] M.-H. Tang and M.-H. Chen, "Measuring OO design metrics from UML," in *Proceedings of MODELS/UML'2002*, UML 2002, 2002.

[12] T. J. Harmer and F. G. Wilkie, "An extensible metrics extraction environment for object-oriented programming languages," in *Proceedings of the International Conference on Software Maintenance*, 2002.

[13] M. El Wakil, A. El Bastawissi, M. Boshra, and A. Fahmy, "A novel approach to formalize and collect object-oriented design-metrics," in *Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering*, 2005.

[14] A. Baroni, S. Braz, and F. Abreu, "Using OCL to formalize object-oriented design metrics definitions," in *ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering*, 2002.

[15] J. A. McQuillan and J. F. Power, "Experiences of using the dagstuhl middle metamodel for defining software metrics," in *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java*, 2006.

[16] E. Alikacem and H. Sahraoui, "Generic metric extraction framework," in *Proceedings of IWSM/MetriKon 2006*, 2006.

[17] T. Mens and M. Lanza, "A graph-based metamodel for object-oriented software metrics," *Electronic Notes in Theoretical Computer Science*, vol. 72, pp. 57–68, 2002.

[18] C. Marinescu, R. Marinescu, and T. Gîrba, "Towards a simplified implementation of object-oriented design metrics.," in *IEEE METRICS*, p. 11, 2005.