Model Metrication in MOVA: A Metamodel-Based Approach using OCL

Manuel Clavel, Marina Egea, and Viviane Torres da Silva

Universidad Complutense de Madrid, Spain
clavel@sip.ucm.es
marina_egea@fdi.ucm.es
viviane@fdi.ucm.es

Abstract. We present the metrication facility available in the MOVA tool and discuss its metamodel-based design: metrics in MOVA are OCL queries over the instances (automatically generated by the tool) of the MOVA metamodel corresponding to the user-depicted models. Thanks to this metamodel-based design, although the MOVA tool provides a collection of predefined metrics, the users can also write and execute their own metrics using the OCL editor included in the MOVA tool.

1 Introduction

Software metrics have been widely used to improve productivity and quality during the software development life-cycle. Metrics have been applied to the software design, to the software implementation, and also to the software development process itself. The ability to measure the software provides a quantitative basis for its development and validation [23]. In particular, the early use of metrics in the software life-cycle can provide quantitative indicators and predictors of structural problems [3].

Different approaches have been proposed for defining metrics which vary from the use of natural language to the use of mathematical formalisms. Both extremes present difficulties: the former may cause ambiguous definitions and the latter may require a formal background to understand the definitions themselves [1]. The Object Constraint Language(OCL) [17] provides an interesting balance. Although a formal language, OCL has been developed as a business modeling language to be easy to read and write for those without a strong mathematical background. In addition, OCL is part of the UML [18] standard and it can be used to precise and query UML models: as a constraint language, OCL helps to precise the models; as a query language, it helps to analyze them. The use of OCL for defining metrics was first proposed by [11]: in this approach, metrics are defined as OCL queries which are evaluated over the instances of the metamodel that correspond to the models to be measured.

MOVA [24] is a modeling and validation tool: it allows the users to draw UML class and object diagrams; write and check OCL constraints; write and evaluate OCL queries; define OCL operations to be used in constraints and queries; and write and evaluate OCL metrics. In this paper we present the metrication facilities provided by MOVA, and discuss its metamodel-based design, which follows the ideas proposed in [11]. To the best of our knowledge MOVA is the only available modeling tool that allows its users to write and execute their own metrics directly over the user models. In fact, to help the users in writing and executing their own metrics, MOVA provides an OCL editor and an internal translator from MOVA user models to their corresponding MOVA metamodel instances.

The MOVA tool is still an experimental tool: it is being developed at the Universidad Complutense de Madrid by the MOVA group as part of a broader effort for integrating rigorous modeling and validation into the industrial software engineering process. The latest version of the tool, along with its user manual and a collection of examples, is available at `http://maude.sip.ucm.es/mova`.

Organization The paper is organized as follows. In Section 2 we recall the MOF architecture. Then, in Section 3, we discuss the uses of OCL and explain, in particular, how it can be used to define metrics as query operation over the metamodel; here we also discuss the features that are expected in a tool that supports this metamodel-based approach for defining metrics. Next, in Section 4, we introduce our MOVA tool and describe how its supports OCL metrics through its OCL editor and its metamodel instance generator. Finally, in Sections 5 and 6 we discuss related work, and draw our conclusions and future work.

2 The MOF Architecture

The Meta-Object Facility (MOF) [16] defines an abstract language for specifying metamodels. A metamodel is in effect an abstract language for some kind of metadata. Examples include the metamodel for UML and the MOF itself. UML and MOF are normally viewed in the context of the classical four-layered metadata architecture:

– The meta-metamodel layer (M3) contains meta-metamodel elements with which to define metamodels. MOF is an example of a meta-metamodel.
– The metamodel layer (M2) contains metamodel elements with which to define models. A metamodel is an instance of a meta-metamodel, meaning that each element of the metamodel is an instance of an element in the meta-metamodel. The UML metamodel, which is an instance of MOF, is an example of a metamodel.
– The model layer (M1) contains model elements with which to model different problems domains. A user model is an instance of the UML metamodel, and contains both model elements and snapshots of instances of these model elements.
– The bottom layer (M0) contains the run-time instances of the model elements defined in a model.

This structure can be applied recursively many times: what is a metamodel in one case can be a model in another case, and this is what happens with UML and MOF. UML is a metamodel from which users can define their own models. Similarly, MOF is a metamodel from which users can define their own models. However, from the perspective of MOF, UML is viewed as a user model that is based on MOF as its metamodel. Figure 1 illustrates the four-layer metamodel hierarchy and how these meta-layers relate to each other. As noted above, every model is an instance of the metamodel on which it is based: that is, there is a snapshot of the metamodel associated with each model, as it is indicated by the mappedTo arrow in Figure 1.

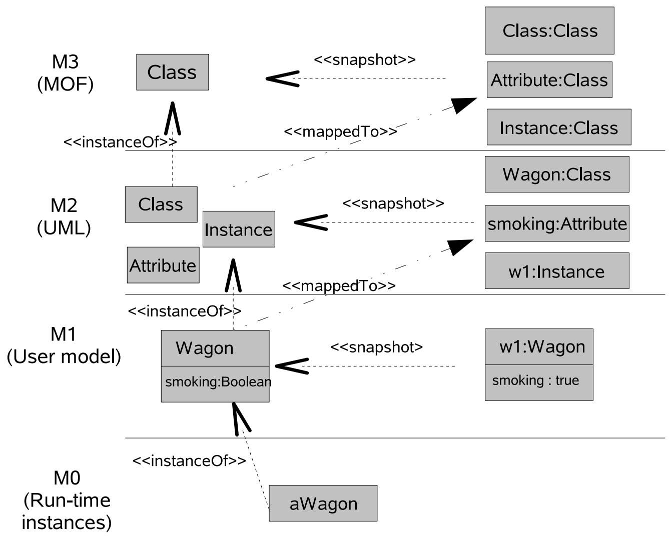


Fig. 1. The **MOF** architecture

3 The OCL Language and its Use in Metrics Definition

Within UML, the Object Constraint Language (OCL) [17] is the standard for specifying constraints and defining queries over user models. Every expressions written in OCL relies on the types that are defined in the (meta) models. As a constraint language, OCL helps to precise the user models. Consider, for example, the user model shown in Figure 2. It describes a simple railway system: a train may own wagons; wagons can be either smoking or non-smoking; there are first-class wagons; and wagons may be connected to other wagons (their predecessor and successor wagons).
To require that each train should contain at least one smoking wagon, we can add to our model the following OCL invariant:

context Train **inv:** AtLeastOneSmokingWagon
self.wagon→exists(w|w.smoking)

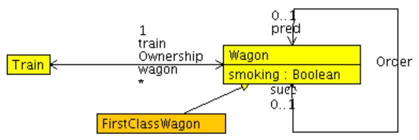


Fig. 2. An example of user model: a railway system.

As a query language, OCL helps to analyze the snapshots of the user models: for example, to obtain the number of smoking wagons linked to a train, we can use the following OCL operation:

context Train::numSmokingWagons(): Integer
body: self.wagon→select(w|w.smoking)→size()

From the perspective of MOF, the UML metamodel is a user model, which can be precised and analyzed using OCL. In fact, the UML metamodel specification [18] defines the well-formedness rules of each metaclass as a (possibly empty) set of OCL invariants for the metaclass, which must be satisfied by all instances of that metaclass for the model to be meaningful. In many cases, additional operations on the metaclasses are needed for the OCL expressions, which are also defined in [18] using OCL. For example, the UML metamodel specifies with the following OCL invariant that generalization hierarchies must be acyclical:

context Classifier
inv: not self.allParents()→includes(self).

This invariant uses the query allParents() that gives all direct and indirect ancestors of a generalized classifier:

context Classifier::allParents(): Set(Classifier)
body: self.parents()→union(self.parents()→collect(p| p.allParents())).

The definition of allParents() uses also the query parents() that gives all immediate ancestors of a generalized classifier:

context Classifier::parents(): Set(Classifier)
body: self.generalization→collect(g|g.general).

The use of OCL to define design metrics was first proposed in [11]. In this approach, metrics are OCL query operations defined in the UML metamodel which are applied to the snapshots of this metamodel corresponding to the user models under consideration. For example, the number of ancestors of a classifier can be obtained using the following query operation:

context Classifier::numAllParents(): Integer
body: self.allParents()→size()

In principle, any tool implementing an OCL evaluator could be used to apply metrics to UML models using this approach. However, in practice, the following features are also desirable:

Modeling GUI The tool should have a graphical interface for building user models or/and a facility for importing models built using other modeling tools, since defining models using a textual description is an error-prone and tedious task for the users.

Instance Generator The tool should automatically built the snapshots of the UML metamodel which correspond to the models built by the user with the graphical interface. Mapping models to their associated snapshots of the UML metamodel is also an error-prone and tedious task for the users. Additionally, the formal definition of this mapping should be publically available, since the users must understand it in order to define their own metrics and to evaluate the results of their applications to models.

OCL Editor The tool should provide an editor of OCL expressions with type-checking and model-based syntax-guiding facilities, since writing metrics, i.e. OCL expressions over the UML metamodel, can be again an error-prone and tedious task for the users.

Metrics Library The tool should provide a collection of frequently-used query operations (including metrics) over the UML metamodel, which the users can directly use to measure their models and/or to write their own metrics.

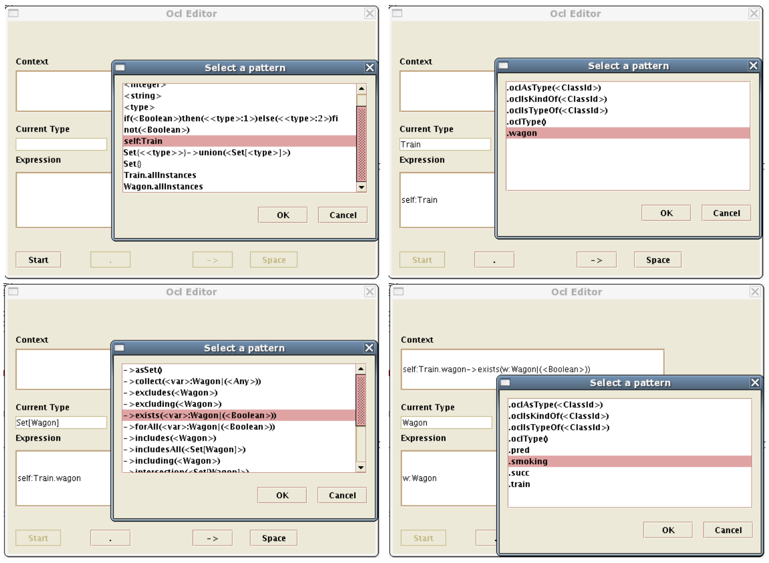
4 The MOVA Tool

The MOVA tool is a Java IDE for the ITP/OCL tool [9], a text-input, experimental modeling and validation tool for UML+OCL user models. The ITP/OCL tool relies on the equational specification of UML+OCL models summarized in [8]. The ITP/OCL is written entirely in Maude [7], a rewriting-based programming language. The MOVA tool is freely available at `http://maude.sip.ucm.es/mova`, along with its user manual and a collection of examples. Currently, the MOVA tool allows the user to draw UML class and object diagrams, write and check OCL invariants, write and evaluate OCL queries, define OCL operations to be used in invariants and queries, and write and evaluate OCL metrics. The MOVA tool is still an experimental tool. However, it features two applications which, to the best of our knowledge, are unique among the available modeling and validation tools: an OCL editor, with type-checking and model-based syntax-guiding, and a metamodel-based metrics application, with automatic generation of metamodel snapshots from user models.

4.1 The OCL editor

It assists the user in writing OCL expressions in three different contexts: adding an invariant to a class diagram; writing a query about an object diagram; and defining the body of an operation.

To write an expression the user selects *patterns* from lists that are built at “run-time” when the buttons Start, Dot, Arrow, or Space are pressed. For example, Figure 3 shows the sequence of patterns that are selected in order to add the invariant AtLeastOneSmokingWagon (defined in Section 3) to the railway model shown in Figure 2. The actual patterns shown in the lists depend on the model under consideration, the current type of the expression, and the button that has been pressed: the Start button is used to start writing an expression (top-left screenshot in Figure 3); the Dot button is used to access a property of a class (top-right and bottom-right screenshots in Figure 3); the Arrow button is used to access a property of a collection (bottom-left screenshot in Figure 3); and the Space button is used to introduce a logical or an arithmetic operator. The current type of an expression is shown in the Current Type subwindow.



Train
self:Train

Set[Wagon]
self:Train.wagon

self:Train.wagon->exists(w:Wagon|(<Boolean>))
Wagon
w:Wagon

Fig. 3. A sequence of pattern selections.

Patterns can contain *holes*, which are *types* enclosed in angle brackets possibly followed by an index. For example, the exists-pattern contains the hole ⟨Boolean⟩. When a pattern with holes is selected the user has to fill its holes with expressions of the appropriate type. Each of these expressions is written in a new editor window, that is a *child* of the window in which the pattern has been selected. The type of the hole to be filled in a child window is shown in its Target Type subwindow, and the context in which this hole appears is shown in its Context subwindow. For example, to write the invariant AtLeastOneSmokingWagon, the hole ⟨Boolean⟩ in the exists-pattern is filled with the expression w:Wagon.smoking of type Boolean, which is written in a child window with Boolean as its Target Type and self:Train.wagon→exists(w:Wagon| ⟨Boolean⟩) as its context.

4.2 The metrics application

It allows the users to apply their own metrics to their user models. Metrics are written using the OCL editor as query operations over the snapshots of the MOVA metamodel corresponding to the user models. Here we introduce each of the key components of this application.

The MOVA Metamodel It is a subset of the UML metamodel: basically, it only specifies the metamodel elements which are required to define both user models and snapshots of these models. The metamodel elements are named following the UML metamodel: an exception to this rule is the Class element which, for technical reasons, has been renamed to MovaClass. The MOVA metamodel is described in [10].

The MOVA instance generator The tool internally generates the snapshots of the metamodel corresponding to the user models, using the mapping defined in Appendix A which basically follows the one described in [18]. To illustrate this mapping, we show in Figure 4 the snapshot of the metamodel corresponding to the railway model shown in Figure 2.

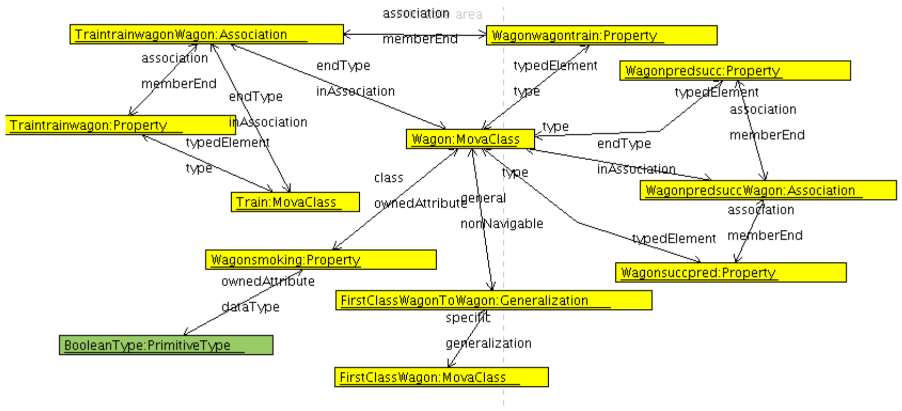
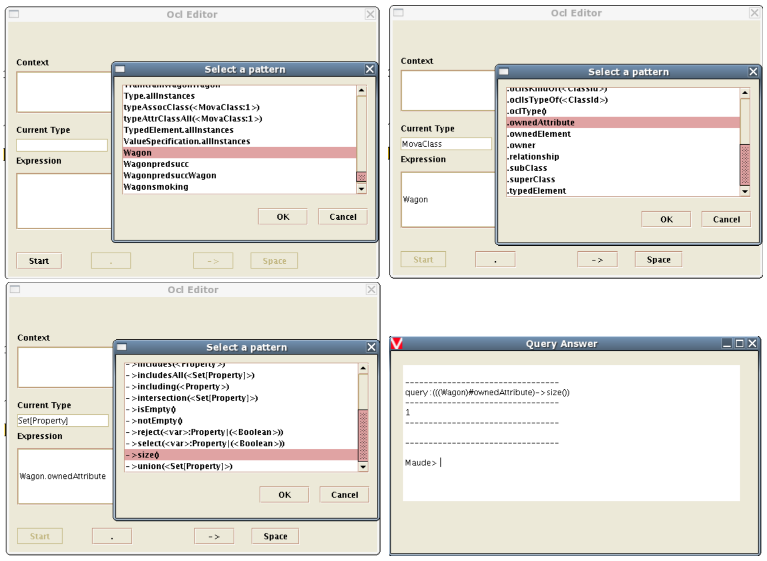


Fig. 4. The snapshot of the metamodel for the railway model.

Notice that, in this snapshot, for example,

– The class Wagon is mapped to the object Wagon of the class MovaClass.
– The class FirstClassWagon is mapped to the object FirstClassWagon of the class MovaClass.
– The generalization between Wagon and FirstClassWagon is mapped to the Generalization-object FirstClassWagonToWagon, which is linked to the object FirstClassWagon with the association generalization↔specific, and to the object Wagon with the association nonNavigable↔general.

The MOVA editor for metrics The OCL editor assists the users in writing their metrics as query operations. The patterns, of course, will correspond to the classes, attributes, roles, generalizations, and associations included in the metamodel, and to the objects and links included in the snapshots of the metamodel corresponding to the user models under consideration. For example, Figure 5 shows the sequence of patterns that can be selected in order to query about the number of attributes owned by the class Wagon in the railway model shown in Figure 2, along with the result of this query.



```
--------------------------------
query : (((Wagon)#ownedAttribute)->size())
--------------------------------
1
--------------------------------

--------------------------------

Maude> |
```

Fig. 5. A sequence of (meta) pattern selections and the result of the query.

The MOVA metrics library To ease the task of writing queries, the editor for metrics offers additional patterns that correspond to query operations (including basic metrics) which are frequently-used in defining complex metrics. For example, the query operations allParents() [18] is included in the MOVA metrics library, although renamed to parentClassAll. Using this operation is easy to write, for example, the metric inherAttrClassAll(c: MovaClass) that returns the total number of inherited attributes of a class c, namely,
parentClassAll(c)→collect(c1|c1.ownedAttribute)→size.
The complete list of these operations, with their OCL definitions, is shown in Appendix B. They are inspired by the metrics defined in [3, 21, 26]: the differences are mostly due to the fact that they are defined over different metamodels.

5 Related Work

There are many issues, both theoretical and pragmatical, about software and design metrics open to discussion: how to translate (if possible) software metrics from/to models [4, 3, 2]; how they should be correlated [20]; how to use them as indicators of the quality of systems [22]; how good is to measure design models and the metrics to use [3, 14, 21, 15]; how to avoid ambiguity in metrics definition [6, 3]; how to empirically validate metrics for design models [19, 13, 5]; and how to build/use tools for applying metrics to design models [27, 12, 22]. Although centered around different concerns, the discussions frequently run intermingled, which, in some cases, handicap their progress to find proper solutions.

The use of OCL for defining design metrics was first proposed in [11], and further explored in [3, 14, 22, 21, 20]: in this approach, metrics are defined as OCL queries which are evaluated over the instances of the metamodel that correspond to the models to be measured.

The library FLAME [3] consists of a collection of metrics defined in OCL over the UML 1.3 metamodel. The library has not yet been ported to the UML 2.0 metamodel and, at the moment, it is only used through the USE tool [25].[1] In our view, this situation is not ideal. First, the USE tool does not provide a graphical interface for defining user models; instead they must be introduced using a textual description. This is an issue when the models to be measured are based on a metamodel different from UML metamodel 1.3 (whose textual description is included as an example in the USE release). Second, the USE tool does not provide editing facilities for introducing OCL expressions: they must be manually typed within the textual description of their contextual classes. Finally, the USE tool does not generate, from the textual descriptions of the user models, the corresponding instances of the UML 1.3 metamodel: they must be manually drawn using the USE graphical interface for building snapshots of user models (in this case, the UML 1.3 metamodel).

The dMML (Defining Metrics at the Meta Level) tool [22] extends the UML metamodel with a separate package containing a single abstract class Metrics, so that metrics are defined as (possibly parameterized) query operations of concrete classes of this abstract class. Although a prototype of dMML was described in [22], at the moment of this writing, we have not been able to access the tool. Fortunately, a new release of the tool is expected in the near future, which is announced to work with Octopus 2.2.0 and Eclipse 3.1.[2]

6 Conclusions and Future Work

In this paper we have presented the metrics application available in the MOVA tool. MOVA [24] is a UML modeling and validation tool. It provides a modeling GUI that helps the users to draw both models and snapshots of these models. It

[1] Personal communication with F. Brito e Abreu, March 2007.
[2] Personal communication with Jacqueline McQuillan, March 2007.

also provides a rewriting-based OCL evaluator that helps the users to validate their models. Finally, it provides an editor that helps the users in writing OCL expressions in three different contexts: adding an invariant to a user model; formulating a query about a snapshot of a user model; and defining the body of query operations. This editor comes with type-checking and model-based syntax-guiding facilities.

Following the ideas proposed in [11], metrics in MOVA are OCL queries over snapshots of the MOVA metamodel. Inside the metrics application, MOVA supports this metamodel-based approach in two ways: first, by internally generating the snapshots of the metamodel corresponding to the models which are built using the modeling GUI; second, by automatically selecting these snapshots as the context of the expressions which are written using the OCL editor. To the best of our knowledge, MOVA is the only available modeling tool that provides this kind of support for writing and executing user-defined OCL metrics. In addition, MOVA provides a collection of frequently-used query operations (including metrics) over the metamodel, which are available to the users in the OCL editor inside the metrics application. The MOVA metrics library is inspired by the metrics defined in [3, 21, 26].

Currently, the MOVA metrics application suffers of two main shortcomings. Firstly, the MOVA metamodel is only a subset of the UML metamodel: in particular, packages and non-query operations are not specified and, consequently, metrics associated with these elements can not yet be defined. Secondly, the MOVA editor does not support the full OCL syntax; in particular, let expressions and Sequence types are not yet supported. In addition, the metrics application interface does not provide yet saving and loading facilities for user-defined metrics. To overcome these shortcomings, we plan to extend in the near future the MOVA metamodel, the MOVA OCL editor, and the MOVA metrics application interface.

References

1. A. Baroni, S. Braz, and F. Abreu. Using OCL to formalize object-oriented design metrics. In *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002), June 11th, 2002*, Lecture Notes in Computer Science. Springer-Verlag, 2002.
2. A. Baroni and F. Brito e Abreu. Measuring OO design metrics from UML. In *International Conference on the Unified Modeling Language*, Dresden, Germany, October 2002.
3. A. L. Baroni and F. Brito e Abreu. A Formal Library for Aiding Metrics Extraction. In *International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*, Darmstadt, Germany, July 2003.
4. V. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions*, 22:751–761, October 1996.
5. B. Bernardez, M. Genero, A. Duran, and M. Toro. A controlled experiment for evaluating a metric-based reading technique for requirements inspection. *Metrics*, 00:257–268, 2004.

6. L.C. Briand, J.W. Daly, and J.K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, January/February 1999.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
8. M. Clavel and M. Egea. Equational specification of UML+OCL static class diagrams, 2006. `http://maude.sip.ucm.es/~clavel/pubs/`.
9. M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In *11th International Conference on Algebraic Methodology and Software Technology AMAST'06, Kuressaare, Estonia, July 5–8, 2006*, volume 4019 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
10. M. Clavel, M. Egea, and V. Torres da Silva. MOVA user manual (version 0.3.1). Facultad de Informática, Universidad Complutense de Madrid, `http://maude.sip.ucm.es/mova`.
11. F. Brito e Abreu. Using OCL to formalize object oriented metrics definitions. Technical Report ES007/2001, FCT/UNL and INESC, Portugal, June 2001. `http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/others/MOOD_OCL.pdf`.
12. M. El-Wakil, A. El-Bastawisi, M. Riad, and A. Fahmy. A novel approach for formalize object oriented design metics. In *EASE'05: Proceedings of the Evaluation and Assessment in Software Engineering*, Keele, UK, April 2005.
13. M. Genero, M. Piattini, and C. Calero. Early measures for UML class diagrams. *L'OBJET*, 6(4), 2000.
14. M. Goulao and F. Brito e Abreu. Formalizing metrics for COTS. In *ICSE Workshop on Models and Processes for the Evaluation of COTS Components*, Edimburgh, Scottland, May 2004.
15. H. Kim and C. Boldyreff. Developing software metrics applicable to UML models. In *6th ECOOP Workshop on Quantitative approaches in Object-oriented engineering*, Malaga, Spain, June 2002.
16. Object Management Group. MetaObject Facility specification v.1.4. Technical report, Object Management Group, Frammingam, Mass, April 2002. `http://www.omg.org/docs/formal/02-04-03.pdf`.
17. Object Management Group. Object Constraint Language specification. Technical report, Object Management Group, Frammingam, Mass, 2004. `http://www.omg.org`.
18. Object Management Group. Unified Modeling Language specification, v.2.1.1. Technical report, Object Management Group, Frammingam, Mass, 2007. `http://www.uml.org`.
19. M. Marchesi. OOA metrics for the Unified Modeling Language. In *Second Euromicro Conference on Software Maintenance and Reengineering*, Florence, Italy, March 1998.
20. J.A. McQuillan and J. F. Power. On the application of software metrics to UML models. In *Models in Software Engineering - Workshops and Symposia at MoDELS 2006*, volume 4364 of *Lecture Notes in Computer Science*. Springer, 2007.
21. J.A. McQuillan and J.F. Power. A definition of the Chidamber and Kemerer metrics suite for the Unified Modeling Language. Technical Report NUIM-CS-TR-2006-03, Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland, October 2006.
22. J.A. McQuillan and J.F. Power. Towards the re-usability of software metric definitions at the meta level. In Dave Thomas, editor, *Proceedings of the ECOOP'06*

Doctoral Symposium and PhD Students Workshop, volume 4067 of *Lecture Notes in Computer Science*. Springer, 2006.

23. E.E. Mills. Software metrics. Technical Report SEI-CM-12-1.1, Seattle University, 1988.
24. The Modeling and Validation (MOVA) Group. The MOVA tool, 2006. `http://maude.sip.ucm.es/mova/`.
25. Database Systems Group. The USE tool, 2006. `http://www.db.informatik.uni-bremen.de/projects/USE/`.
26. `http://www.sdmetrics.com/`. SDMetrics: The software design metrics tool for the UML.
27. F. Wilkie. Tool support for measuring complexity in heterogeneous object-oriented software. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 152, Washington, DC, USA, 2002. IEEE Computer Society.

A The MOVA Instance Generator

We introduce in this appendix the mapping that defines the correspondence between MOVA user models and instances of the MOVA metamodel. In what follows, $(X + Y)$ denotes the identifier that results from concatenating the identifiers X and Y.

Given a class diagram D, we create a MOVAMetamodel-object diagram ($\mathsf{Meta} + D$), in which we insert:

- a PrimitiveType-object $\underline{\mathsf{IntegerType}}$,
- a PrimitiveType-object $\underline{\mathsf{StringType}}$, and
- a PrimitiveType-object $\underline{\mathsf{BooleanType}}$.

Then,

- For each class c in D, we insert in ($\mathsf{Meta} + D$):
  - a MovaClass-object $\underline{c}$; and
  - for each of its attributes at of type t, we insert
    - a Property-object $(\underline{c{+}at})$,
    - an ownedAttribute↔class-link between $(\underline{c{+}at})$ and $\underline{c}$,
    - if t is a primitive type, an ownedAttribute↔dataType-link between $(\underline{c{+}at})$ and $(\underline{t + \mathsf{Type}})$, and
    - if t is an enumeration type, an ownedAttribute↔dataType-link between $(\underline{c{+}at})$ and $\underline{t}$.
- For each enumeration class e in D, we insert in ($\mathsf{Meta} + D$):
  - an Enumeration-object $\underline{e}$; and
  - for each of its literals el,
    - an EnumerationLiteral-object $(\underline{e{+}el})$, and
    - an enumeration↔enumerationLiteral-link between $\underline{e}$ and $(\underline{e{+}el})$.
- For each generalization between a subclass c_1 and a superclass c_2 in D, we insert in ($\mathsf{Meta} + D$):
  - a Generalization-object $(\underline{c_1 + \mathsf{To} + c_2})$,

  - a generalization↔specific-link between $(\underline{c_1 + \mathsf{To} + c_2})$ and $(\underline{c_1})$, and
  - a nonNavigable↔general-link between $(\underline{c_1 + \mathsf{To} + c_2})$ and $(\underline{c_2})$.
- For each association between a class c_1 (with role r_1) and a class c_2 (with role r_2) in D, we insert in (Meta + D):
  - an Association-object $(\underline{c_1 + r_1 + r_2 + c_2})$,
  - a Property-object $(\underline{c_1 + r_1 + r_2})$,
  - a Property-object $(\underline{c_2 + r_2 + r_1})$,
  - an inAssociation↔endType-link between $(\underline{c_1 + r_1 + r_2 + c_2})$ and $(\underline{c_1})$,
  - an inAssociation↔endType-link between $(\underline{c_1 + r_1 + r_2 + c_2})$ and $(\underline{c_2})$,
  - an association↔memberEnd-link between $(\underline{c_1 + r_1 + r_2 + c_2})$ and $(\underline{c_1 + r_1 + r_2})$,
  - an association↔memberEnd-link between $(\underline{c_1 + r_1 + r_2 + c_2})$ and $(\underline{c_2 + r_2 + r_1})$,
  - a typedElement↔type-link between $(\underline{c_1 + r_1 + r_2})$ and $\underline{c_1}$, and
  - a typedElement↔type-link between $(\underline{c_2 + r_2 + r_1})$ and $\underline{c_2}$
- For each association class c_3 between a class c_1 (with role r_1) and a class c_2 (with role r_2) in D, we insert in (Meta + D):
  - an AssociationClass-object $\underline{c_3}$,
  - a Property-object $(\underline{c_1 + r_1 + r_2})$,
  - a Property-object $(\underline{c_2 + r_2 + r_1})$,
  - an inAssociation↔endType-link between $\underline{c_3}$ and $\underline{c_1}$,
  - an inAssociation↔endType-link between $\underline{c_3}$ and $\underline{c_2}$,
  - an association↔memberEnd-link between $\underline{c_3}$ and $(\underline{c_1 + r_1 + r_2})$,
  - an association↔memberEnd-link between $\underline{c_3}$ and $(\underline{c_2 + r_2 + r_1})$,
  - a typedElement↔type-link between $(\underline{c_1 + r_1 + r_2})$ and $\underline{c_1}$, and
  - a typedElement↔type-link between $(\underline{c_2 + r_2 + r_1})$ and $\underline{c_2}$

B The MOVA Metrics Library

We introduce in this appendix the metrics/queries included in the MOVA metrics library. Each metric/query is first informally explained and then formally defined/implemented as an OCL query over the MOVA metamodel. When a metric/query is inspired by a metric defined in another library, we indicate so.

Class Metrics/Meta Queries

- *The number of proper attributes of the class c.* [3, NumAttr]
  context numAttrClass(c: MovaClass): Integer
  body: c.ownedAttribute→size()
- *The set of the immediate successors of the class c* [3, children].
  context childClass(c: MovaClass): Set(Classifier)
  body: c.nonNavigable→collect(g:Generalization|g.specific)
- *The number of the immediate successors of the class c* [3, NOC].
  context numChildClass(c: MovaClass): Integer
  body: childClass(c)→size()
- *The number of proper connectors of the class c* [3, NConnectors].
  context numConnClass(c: MovaClass): Integer
  body: c.inAssociation→size() + c.generalization→size() + c.nonNavigable→size()

– *The set of the immediate ancestors of the class c* [3, parents].
  context parentClass(c: MovaClass): Set(MovaClass)
  body: c.generalization→collect(g|g.general.oclAsType(MovaClass))
– *The set of all the ancestors of the class c* [3, allParents].
  context parentClassAll(c: MovaClass): Set(MovaClass)
  body: parentClass(c)→union(parentClass(c)→collect(p|parentClassAll(p)))
– *The set of the immediate inherited attributes of the class c.*
  context inherAttrClass(c: MovaClass): Set(Property)
  parentClass(c)→collect(c1|c1.ownedAttribute)
– *The set of all inherited attributes of the class c* [3, allInheritedAttributes].
  context inherAttrClassAll(c: MovaClass): Set(Property)
  body: parentClassAll(c)→collect(c1|c1.ownedAttribute)
– *The number of all inherited attributes of the class c* [3, IAN].
  context numInherAttrClassAll(c: MovaClass): Integer
  body: inherAttrClassAll(c)→size
– *The set of all (proper and inherited) attributes of the class c* [21, allAttributes,allAccessibleAttributes].
  context attrClassAll(c: MovaClass): Set(Property)
  body: inherAttrClassAll(c)→union(c.ownedAttribute)
– *The set of data types of all the (proper and inherited) attributes of the class c* [21, typesOfAllAccessibleAttributes].
  context typeAttrClassAll(c: MovaClass): Set(Type)
  body: attrClassAll(c)→collect(p|p.dataType)
– *The value of checking whether the class c has an attribute of end type t* [21, hasAttribute].
  context hasAttrClass(c: MovaClass, t: Type): Boolean
  body: typeAttrClassAll(c)→includes(t)
– *The set of the immediate inherited associations of the class c.*
  context inherAssocClass(c: MovaClass): Set(Association)
  body: parentClass(c)→collect(c1|c1.inAssociation)
– *The set of all inherited associations of the class c.*
  context inherAssocClassAll(c: MovaClass): Set(Association)
  body: parentClassAll(c)→collect(c1|c1.inAssociation)
– *The number of all inherited associations of the class c.*
  context numInherAssocClassAll(c: MovaClass): Integer
  body: inherAssocClassAll(c)→size
– *The set of all (proper and inherited) associations of the class c* [21, associations].
  context assocClassAll(c: MovaClass): Set(Association)
  body: inherAssocClassAll(c)→union(c.inAssociation)
– *The set of data types of all the (proper and inherited) attributes of the class c* [21, typeOfAssociations].
  context typeAssocClassAll(c: MovaClass): Set(Type)
  body: assocClassAll(c)→collect(p|p.endType)
– *The value of checking whether the class c has an association with end type t* [21, hasAssociation].

context hasAssocClass(c: MovaClass, t: Type): Boolean
body: typeAssocClassAll(c)→includes(t)

– *The value of checking whether the class* c1 *is coupled with the class* c2 [21, avgCoupledTo].
context avgCoupledTo(c1: MovaClass, c2: MovaClass): Boolean
body: hasAttrClass(c1, c2) or hasAssocClass(c1, c2)

– *The set of classes that are coupled with the class* c [21, avgCouplings].
context avgCouplings(c: MovaClass): Set(MovaClass)
body: MovaClass.allInstances→excluding(c)
→select(c1|avgCoupledTo(c1,c) or avgCoupledTo(c,c1))

– *The number of classes that are coupled with the class c.*
context numAvgCouplings(c: MovaClass): Integer
body: avgCouplings(c)→size()

Diagram Metrics

– *The number of classes in the diagram* [26, NumCls].
context numClass(): Integer
body: MovaClass.allInstances→size()

– *The number of associations in the diagram* [26, Assoc].
context numAssoc(): Integer
body: Association.allInstances→size()

– *The number of generalizations in the diagram* [26, Genrs].
context numGenr(): Integer
body: Generalization.allInstances→size()

– *The number of classifiers in the diagram* [3, Genrs].
context numClasf(): Integer
body: Classifier.allInstances→size()

– *The number of connectors in the diagram* [26, Connectors].
context numConn(): Integer
body: Relationship.allInstances→size()

– *The number of attributes in the diagram* [3, PAAN].
context numAttr(): Integer
body: (MovaClass.allInstances→collect(c|attrClassAll(c)))→asSet()→size()

– *The number of all inherited attributes in the diagram* [3, PIAN].
context numInherAttr(): Integer
body: (MovaClass.allInstances→collect(c|inherAttrClassAll(c)))→asSet()→size()