

An OCL-Based Formalization of the MOOSE Metric Suite

Aline Lúcia Baroni, Fernando Brito e Abreu

Universidade Nova de Lisboa - Departamento de Informática

{abaroni@leve.emn.fr, fa@di.fct.unl.pt}

Abstract - Design metrics are useful for several means, including the improvement of software quality, the identification of fault-prone classes, the prediction of maintenance efforts, the estimation of rework efforts, etc. However, many of the existing metrics suffer from an ill definition, which leads to different interpretations and to a subsequent lack of their use, due to their informal definitions. So, in spite of research studies, design metrics have not been widely utilized in the software industry. One of the major problems that limitates their use is the nonattendance of available tools to measure the metrics, which in turn can be a consequence of the metrics imprecise specification. This paper presents an approach used to formalize metric suites in a precise way, solving the ambiguity problems that can reduce their use. The MOOSE metrics – Metrics for Object-Oriented Software Engineering – serve to exemplify the simplicity and limitations of our approach.



1. INTRODUCTION

Most software developers face the risks of schedule slips and/or cost overruns. Effective resource allocation, reduction of design complexity, and adoption of effective software engineering techniques are some keys for resolving or reducing such risks. Design metrics, which are quantitative measures of the complexity of the software or design, have been suggested as useful means to assist in achieving these goals [Tang and Chen, 2002].

A number of design metrics have been studied and demonstrated as useful in several aspects, such as in understanding, assessing and evaluating the complexity of designs [Henry and Selig, 1990], estimating the complexity of software based on its design [Harrison, 1988], identifying fault-prone software units and plausible types of faults [Basili et al., 1996; Tang et al., 1999], and estimating required maintenance efforts [Li and Henry, 1993; Rombach, 1990; Abreu and Melo, 1996].

Nevertheless, despite research studies, design metrics have not been widely utilized, as expected, in the software industry. One of the major problems that have limited their use is the lack of available tools to measure the metrics, which in turn can be a consequence of their imprecise specification.

This paper acquaints with both the informal and formal definitions of one well-known set of metrics, named MOOSE – Metrics for Object-Oriented Software Engineering [Chidamber and Kemerer, 1993a; Chidamber and Kemerer, 1993b].

The metrics on this set were originally defined informally, using natural language, and the foremost contribution of this work remains on bringing up precision, through their formal definitions.

The definition of each metric is done with OCL [OMG], the library of functions FLAME [Baroni and Abreu, 2003] and the UML meta-model [OMG], at different levels of abstraction, including the meta-classes Package, Model Element, Generalizable Element, Classifier, Feature, Operation and Attribute, as shown in Figure 1. The FLAME (Formal Library for Aiding Metrics Extraction) library is itself formalized with OCL upon the UML meta-model.

Our approach is verified and validated for sake of correction and for guaranteeing the quality of the formalizations. The remaining part of this article is organized as follows. In section 2, we show some functions of FLAME, used to obtain the formal definition of the MOOSE metrics. In section 3, the formalization of MOOSE metrics is exemplified. Section 4 outlines our conclusions and further work.

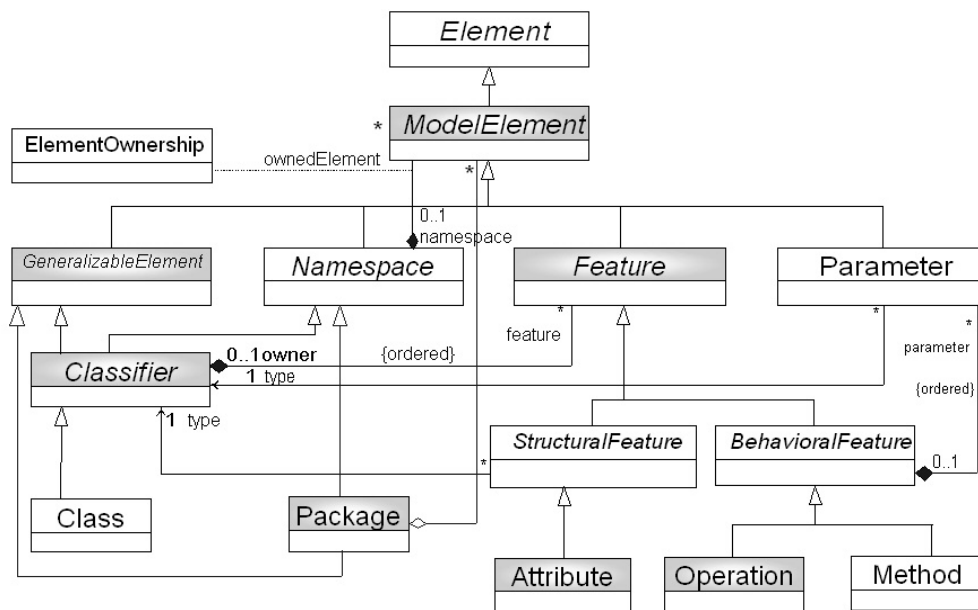


Figure 1 - Different Levels of Abstraction of the Formalized Metrics

2. FLAME FUNCTIONS FOR MOOSE FORMALIZATION

This section presents the formal definition of some functions that compose FLAME, using OCL and part of the UML meta-model as background.

The functions are classified as general, set, percentage or counting functions. General functions are those that return booleans. Set functions return set of elements, which can have the type of any meta-class in the UML meta-model. Percentage functions return a value representing a percentage and, finally, counting functions return integers.

GeneralizableElement General Functions

| Name | isRoot |
|---------------------|--|
| Informal Definition | Indicates whether the GeneralizableElement has ascendants or not. A true value indicates it has no ascendants and a false value indicates it has ascendants. |
| Formal Definition | <u>GeneralizableElement:: isRoot(): Boolean</u> = self.isRoot |
| Comments | As an alternative, the result could be: = parents() -> isEmpty() or = PARN() = 0 |

GeneralizableElement Set Functions

| Name | Children |
|---------------------|--|
| Informal Definition | Set of directly derived Classes of the current GeneralizableElement. |
| Formal Definition | <u>GeneralizableElement:: children(): Set(GeneralizableElement)</u> = self.generalization -> collect(g g.parent) -> excluding(self) -> asSet |

| | |
|---------------------|--|
| Name | parents |
| Informal Definition | Set of Classes from which the current GeneralizableElement derives directly. |
| Formal Definition | <u>GeneralizableElement:: parents(): Set(GeneralizableElement)</u> = self.specialization -> collect(c c.child) -> asSet() -> excluding(self) |

GeneralizableElement Counting Functions

| | |
|---------------------|--|
| Name | CHIN – Children Number |
| Informal Definition | Number of directly derived Classes. |
| Formal Definition | <u>GeneralizableElement:: CHIN(): Integer</u> = children() -> size() |

| | |
|---------------------|---|
| Name | PARN – Parents Number |
| Informal Definition | Number of Classes from which the current GeneralizableElement derives directly. |
| Formal Definition | <u>GeneralizableElement:: PARN(): Integer</u> = parents() -> size() |

Classifier Set Functions

| | |
|---------------------|---|
| Name | feature2AttributeSet |
| Informal Definition | Subset of Attributes (from one set of Features) belonging to the current Classifier. |
| Formal Definition | <u>Classifier:: feature2AttributeSet(s: Set(Feature)): Set(Attribute)</u> = s -> select(f: Feature f.ocIsKindOf(Attribute)) -> collect(f f.ocAsType(Attribute)) -> asSet |

| | |
|---------------------|--|
| Name | Feature2OperationSet |
| Informal Definition | Subset of Operations (from one set of Features) belonging to the current Classifier. |
| Formal Definition | <u>Classifier:: feature2OperationSet(): Set(Operation)</u> = s -> select(f: Feature f.ocIsKindOf(Operation)) -> collect(f f.ocAsType(Operation)) -> asSet |

| | |
|---------------------|---|
| Name | allFeatures |
| Informal Definition | Set containing all Features of the Classifier itself and all its inherited Features. |
| Formal Definition | <u>Classifier:: allFeatures(): Set(Feature)</u> = self.feature -> union(self.parents() -> collect(g g.oclAsType(Classifier).allFeatures()) -> flatten -> asSet) |
| Comments | It can be alternatively defined as: = newFeatures() -> union(allInheritedFeatures()) |

| | |
|---------------------|--|
| Name | allAttributes |
| Informal Definition | Set containing all Attributes of the Classifier itself and all its inherited Attributes (both directly and indirectly). |
| Formal Definition | <u>Classifier:: allAttributes(): Set(Attribute)</u> = feature2AttributeSet(self.allFeatures()) |
| Comments | Previously defined in the UML meta -model, but redefined here. It can be alternatively defined as: = newAttributes() -> union(allInheritedAttributes()) or = self.allFeatures() -> select(f f.oclIsKindOf(Attribute)) -> collect(f f.oclAsType(Attribute)) -> asSet |

| | |
|---------------------|--|
| Name | allOperations |
| Informal Definition | Set containing all Operations of the Classifier itself and all its inherited Operations. |
| Formal Definition | <u>Classifier:: allOperations(): Set(Operation)</u> = feature2OperationSet(self.allFeatures()) |
| Comments | Previously defined in the UML meta -model, but redefined here. It can be alternatively defined as: = newOperations() -> union(allInheritedOperations()) or = self.allFeatures() -> select(f f.oclIsKindOf(Operation)) -> collect(f f.oclAsType(Operation)) -> asSet |

| | |
|---------------------|--|
| Name | allOppositeAssociationEnds |
| Informal Definition | Set of all AssociationEnds, including the inherited ones, that are opposite to the Classifier. |
| Formal Definition | <u>Classifier:: allOppositeAssociationEnds(): Set(AssociationEnd)</u> = self.oppositeAssociationEnds() ¹ -> union(self.parents()) -> collect(g g.oclAsType(Classifier).allOppositeAssociationEnds()) -> flatten -> asSet() |

| | |
|---------------------|---|
| Name | coupledClasses² |
| Informal Definition | Set of Classifiers to which the current Classifier is coupled (excluding inheritance). |
| Formal Definition | <u>Classifier:: coupledClasses(): Set(Classifier)</u> = self.allOppositeAssociationEnds().type -> union(self.allAttributes().type -> union(self.allOperations().parameter.type -> union(self.allOperations().type))) -> reject(c: Classifier c.oclIsKindOf(DataTypee)) -> asSet() |
| Comments | This function includes the coupled classes corresponding to: <ul style="list-style-type: none"> - Classes directly associated with the current one; - Class Attributes; - Class Operations Parameters type; - Class Operations return type. The function does not include: <ul style="list-style-type: none"> - Method local Attributes; - Attributes from other Classes used by the Classifier's Methods; - Receivers from messages sent by the Classifier's Methods and - The Data Types. |

ModelElement Set Functions

| | |
|---------------------|---|
| Name | client |
| Informal Definition | Set containing all direct clients of the ModelElement. |
| Formal Definition | <u>ModelElement:: client(): Set(ModelElement)</u> = self.supplierDependency -> collect(d : Dependency d.client) -> flatten -> asSet |

¹ The function *oppositeAssociationEnds()* is defined in the UML meta-model.

² A discussion about the *coupledClasses()* function takes place on [Baroni, 2002]. This reference also contains a complete reference of the FLAME functions.

| | |
|---------------------|---|
| Name | allClients |
| Informal Definition | Set containing all the ModelElements that are clients of this ModelElement, including the clients of these ModelElements. This is the transitive closure. |
| Formal Definition | ModelElement:: allClients(): Set(ModelElement) = self.client() -> union(self.client() -> collect(m : ModelElementImpl m.allClients()) -> flatten) -> asSet |

3. MOOSE FORMALIZATION WITH FLAME

Several studies have been conducted to validate the MOOSE metrics and have shown that they are useful quality indicators for predicting fault-prone classes [Basili et al., 1996; Tang et al., 1999] and maintenance effort [Li and Henry, 1993], as well as being significant economic variable indicators [Chidamber et al., 1998].

This section introduces the formalization of the MOOSE metrics, done with the FLAME functions presented on the previous section, OCL and the UML meta-model. All our research was conducted with the version 1.3 of the meta-model. Although the MOOSE set is widespread, not all the metrics are design ones, and some of them cannot be formalized upon the UML core meta-model.

For the complete description of the metrics, their corresponding usefulness and evaluation, refer to [Chidamber and Kemerer, 1993a; Chidamber and Kemerer, 1993b].

| | |
|---------------------|--|
| Name | WMC - Weighted Methods per Class |
| Informal Definition | The sum of complexities of the Methods in the current Class. If all method complexities are considered to be unique, WMC is equal to the number of Methods. |
| Formal Definition | Classifier:: WMC(): Integer = self.allOperations() -> size() |
| Comments | The authors do not propose any algorithm for calculating the complexities of methods. As such, in the formalization above, the complexities were considered unitary. |

| | |
|---------------------|--|
| Name | DIT - Depth of Inheritance Tree |
| Informal Definition | The length of the longest path of inheritance from the current Class to the root of the tree. |
| Formal Definition | Classifier:: DIT(): Integer = if self.isRoot() then 0 else if PARN() = 1 then 1 + self.parents() -> iterate(elem: GeneralizableElement; acc: Integer = 0 acc + elem.oclAsType(Class).DIT()) else self.parents() -> iterate(elem: GeneralizableElement; acc: Integer = 0 acc + elem.oclAsType(Class).DIT()) endif endif |

| | |
|---------------------|---|
| Name | NOC – Number of Children |
| Informal Definition | The number of classes that inherit directly from the current Class. |
| Formal Definition | Classifier:: NOC(): Integer = self.CHIN() |

| | |
|---------------------|---|
| Name | CBO – Coupling Between Objects |
| Informal Definition | The number of other Classes that are coupled to the current one. |
| Formal Definition | Classifier:: CBO(): Integer = self.coupledClasses() -> size() |

| | |
|---------------------|--|
| Name | RFC – Response for a Class |
| Informal Definition | The number of Methods in the current Class that might respond to a message received by its object, including Methods both inside and outside of this Class. |
| Formal Definition | Classifier:: RFC(): Integer = (self.allOperations() -> union(self.allOperations().method.allClients().oclAsType(Operation))) -> asSet() -> size()) |
| Comments | $RFC = \{M\} \dot{\bigcup}_{all\ i} \{R_i\}$ where $\{R_i\}$ = set of Methods called by Method i and $\{M\}$ = set of all Methods in a Class. R_i is dependent on the implementation of the Method i . |

| | |
|---------------------|---|
| Name | LCOM – Lack of Cohesion in Methods |
| Informal Definition | The degree of similarity of Methods in the current Class (by counting instance variables sets used by all possible Method pairs). |
| Formal Definition | Classifier:: LCOM(): Integer = |
| Comments | The set of instance variables used by one Method is known only after completing the implementation of the Method. So this metric is code dependent, and can not be extracted in the design phase. |

The MOOSE set of metrics is not completely related with design. This restricts the formalization. For instance, the metric LCOM depends on the source code. Moreover, it is necessary to note another limitation considering this set. The metric WMC has, in this document, provides only the simplest implementation regarding complexities (it considers all the methods' complexities as unitary). However, different implementations could be offered. We considered the simplest case.

4 CONCLUSIONS AND FURTHER WORK

Software measurement contributes to software quality from various aspects, such as understandability, complexity, reliability, testability and maintainability, as well as performance and productivity of software projects [Tang et al., 2002]. With the pervasive popularity and adaptation of object-oriented programming languages and methodologies in software development, software metrics tailored to object-

oriented characteristics are essential to improve the object-oriented process and products.

Based on this, we built the library FLAME, and with it we formalized different metric suites, as the MOOD and MOOD2 – Metrics for Object-Oriented Design [Abreu, 1995; Abreu, 1998], MOOSE – Metrics for Object-Oriented Software Engineering [Chidamber and Kemerer, 1993a; Chidamber and Kemerer, 1993b], EMOOSE – Extended MOOSE [Li et al., 1995] and QMOOD – Quality Model for Object-Oriented Design [Bansiya and Davis, 1997].

In this work we were mainly concerned about design metrics extracted from UML class diagrams. As future work, we plan to investigate which are relevant metrics for dealing not only with the static structure of models, but also with its behavior. For such, we can investigate the possibilities of metrics formalization (and even creation) based on other UML diagrams. Additionally, in this work, the version 1.3 of the UML meta-model was used. We plan to adapt the contribution presented here to the newest version of this meta-model.

5. REFERENCES

- Abreu, F. B. [1995]. *Design Metrics for Object-Oriented Software Systems*. workshop on Quantitative Methods for Object-Oriented Systems Development (ECOOP'95), Aarhus, Denmark, August 7th - 11th.
- Abreu, F. B.; Melo, W. L. [1996]. *Evaluating the Impact of Object-Oriented Design on Software Quality*. 3rd International Software Metrics Symposium (Metrics'96), Berlin, Germany, March.
- Abreu, F. B.; Cuche, J. S. [1998]. *Collecting and Analyzing the MOOD2 Metrics*. Workshop on Object-Oriented Product Metrics for Software Quality Assessment (ECOOP'98), Brussels, Belgium, pages 258-260, July 21st.
- Baroni, A. L. [2002]. *Formal Definition of Object-Oriented Design Metrics*. Master Thesis. Vrije Universiteit Brussel, Belgium.
- Baroni, A. L.; Abreu F. B. [2003]. *A Formal Library for Aiding Metrics Extraction*. Workshop on Object-Oriented Reengineering (ECOOP'03), Darmstadt, Germany.
- Basili, V.; Briand, L.; Melo, W. L. [1996]. *A Validation of Object-Oriented Design Metrics as Quality Indicators*. IEEE Transactions on Software Engineering, 22(10), pages 751-760.
- Bansiya, J.; Davis, C. [1997]. *An Object-Oriented Design Quality Assessment Model*. University of Alabama, EUA.
- Chidamber, S. R.; Kemerer, C. F. [1993a]. *A Metrics Suite for Object Oriented Design*. WP No.249, MIT Sloan School of Management, Cambridge, MA, EUA.
- Chidamber, S. R.; Kemerer, C. F. [1993b]. *MOOSE: Metrics for Object Oriented Software Engineering*. Workshop on Processes and Metrics for Object-Oriented Software Development (OOPSLA'93), Washington DC, EUA, September.
- Harrison, W. [1988]. *Using Software Metrics to Allocate Testing Resources*. Journal of Management Information Systems.
- Henry, S.; Selig, C. [1990]. *Predicting Source-Code Complexity at the Design Stage*. IEEE Software.
- Li, W.; Henry, S. [1993]. *Object-Oriented Metrics that Predict Maintainability*. Journal of Systems and Software, 23(2), pages 111-122.
- Li, W.; Henry, S.; Kafura, D.; Schulman, R. [1995]. *Measuring Object-Oriented Design*. JOOP (July / August), pages 48-55.
- OMG; Rational Software Corporation [1999]. *Unified Modeling Language Specification (version 1.3)*. Object Management Group.
- Rombach, D. [1990]. *Design Measurement: Some Lessons Learned*. IEEE Software.
- Tang, M.H.; Chen, M.H.; Kao, M. [1999]. *Investigating Test Effectiveness on Object-Oriented Software - A Case Study*. 12th Annual International Software Quality Week.
- Tang, M. H.; Chen, M. H. [2002]. *Measuring OO Design Metrics From UML*. UML2002, Dresden, Germany, October.