Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1
© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A FORMAL DEFINITION OF METRICS FOR OBJECT ORIENTED DESIGN: MOOD METRICS

MERYEM LAMRANI, YOUNES EL AMRANI, AZIZ ETTOUHAMI
Conception and System Laboratory, University Mohammed V Agdal,
Computer Science Departement PB 1014 Rabat, Morocco
E-mail: {lamrani, elamrani, touhami}@fsr.ac.ma

ABSTRACT

Software design metrics, since their apparition, suffer from a lack of formalism in their definition opening room to ambiguities and thus to misleading results. Although, several studies attempted to bring rigor to most well-known suite of metrics, the degree of formalism used to define them, constitutes a significant obstacle towards the built of solid tools support, considered as the key point to an easy integration of measurement in the industry. This paper is a logical continuation of a previous published work where a Z-based formalization of the CK metrics is presented, offering an innovative and easy to follow methodology which successfully manages to provide a solid definition of metrics that deals with complexity, coupling and cohesion. While this work brings formalism at the classifier level, we proceed, in the present, to propose formalism for an overall quality measurement of the object-oriented systems, introducing the invisibility concept formalization and extending the quality indicator properties to encapsulation and polymorphism.

Keywords: *Formal Methods, Z Language, UML Metamodel, MOOD Metrics*

1. INTRODUCTION

“It was a great step in science when men became convinced that, in order to understand the nature of things, they must begin by asking, not whether a thing is good or bad, noxious or beneficial, but of what kind it is? And how much is there of it? Quality and Quantity were then first recognized as the primary features to be observed in scientific inquiry” [21]. This quote of the Scottish physicist and mathematician James Clerk Maxwell (1831 - 1879) highlights the importance of identifying the nature of the entity to take into consideration when it comes to quality and also the major role of measurement in any scientific field. As software engineering differentiates itself from other hard sciences such as mathematics and physics, especially for its subjectivity aspects, several studies and experiments have shown that software metrics, when applied earlier in the software life cycle (i.e. design phase), can help considerably the improvement and control of software quality over specific software properties such as efficiency, complexity, understandability and reuse [8]. Many software quality indicators have been identified and successfully verified in helping reduce risks, detect faultiness and thus managing both time and cost estimation control [20]. Some of the most relevant ones are encapsulation, inheritance and polymorphism.

Encapsulation means hiding the internal specification of an object that do not contribute to its essential characteristics and showing only the external interface; typically, the structure of an object is hidden, as well as the implementation of its methods [19]. Inheritance is a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes [19]. Finally, polymorphism is a concept in type theory wherein a name (such as a variable declaration) may denote instances of many different classes as long as they are related by some common superclass [19].

These concepts, according to theoretical and experimental results, have a strong capability to build a flexible system.

Among existing suite of metrics defined that emphasize the above properties, we especially consider in this contribution, the well-known Metrics for Object Oriented Design suite [6], also called the MOOD metrics, for their commonly recognized ability to provide useful results and information about the whole object-oriented system quality.

© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

This paper joins the multiple efforts done to facilitate the introduction of software design metrics into the industry field. Based on a recently published formalization methodology [4], it aims to provide precise and complete formalized definition of the proposed software design metrics. The choice of the Z language [1-2] is justified by its structure: the grouping concept introduced by schema, its maturity and the possibility of checking consistency using proof theorems [18].
The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents a brief overview of the Z-based formalization approach. Section 4 illustrates the formal definitions of the MOOD metrics and finally, Section 5 draws conclusion and future work.

2. RELATED WORK

In this section we review existing sets of metrics that measure a system quality and existing formalization approaches up-to-date.

Described below, the most relevant metrics suite concerned with the software design phase and which continues to attract interest nowadays due to their significant results considered as quality indicators.

- ***Metrics for Object-Oriented Software Engineering (MOOSE):*** a metric suite for Object-oriented design defined by Chidamber and Kemerer [7], also known as the CK metrics. Even if, their definition contains some ambiguities due to the degree of formalism used to express them, they are still frequently cited in several contributions and improved by other authors for being useful as quality indicators over many characteristics related to complexity, inheritance, coupling, cohesion and messaging.
- ***Metrics for Object-Oriented Design (MOOD):*** introduced by Fernando Brito e Abreu [6] and evaluated by many other authors, they have shown their capabilities to measure efficiently a whole quality system according to the measured properties such as encapsulation, inheritance and polymorphism. However, some weaknesses have also been found resulting in the extension to MOOD2 metrics that filled the lack of measures of reuse and external coupling of the initial MOOD suite.
- ***Quality Model for Object-Oriented Design (QMOOD):*** proposed by Jagdish Bansiya and Carl G Davis [9] and presented as a hierarchical model. It aims to inform about six quality factors that are functionality, effectiveness, understandability, reusability, flexibility and extendibility following the set of ISO 9126 as an initial set of quality attributes.

Even though, we choose the MOOD metrics [6] for their system-wide measurements, this contribution, also, aims to demonstrate the ability of the proposed formalism model [4] to adapt and expand to any set of object-oriented metrics which are likely to be defined over the UML metamodel [3].
Among design metrics formalization expressed on top of metamodels, Monperrus et al. [10] define a model-driven measurement approach also called MDM approach where metrics are implemented as an instance of a metric specification metamodel. El-Wakil et al. [11] use XQuery language [12] to build metrics expression for UML class diagrams. Baroni et al. [13] propose a Formal Library for Aiding Metrics Extraction (FLAME) [14] where OCL [15] is used to express metrics definition. McQuillan et al. [16] based their work on Baroni's approach and extended the UML metamodel 2.0 to offer a framework for metric definitions. Likewise, Reissing [17] extends the UML metamodel on which metrics definitions are based and then use this model to express known metrics suites with set theory and first order logic. Similarly, Lamrani et al. [4] presents an approach to formalize object-oriented design metrics using Z language but instead of extending the UML metamodel, it uses the original OMG standard [3] specifications and then proceeds to its formalization as a basis for the formal metrics expressions.
All significant efforts in this area involve the use of metamodels with non-formal (XQuery, SQL...) or semi-formal languages, especially OCL [15] which remains a language that was initially designed to express constraints on UML class diagrams. Although, OCL benefits from concise and friendly syntax, it suffers from the absence of a metamodeling approach since it uses an EBNF grammar instead, preventing a complete integration with the rest of UML. Besides, its semantics lack formal definitions leading to misunderstanding and unclear issues as explained in Baar [23] practice report. It enumerates number of OCL weaknesses such as restrictions on nested sets and the undeterministic iterate-construct. As a consequence, OCL limitations constitute significant obstacles towards the rigorous expression of software metrics definitions.
The formal approach [4] proposed in this contribution benefits from a multi level formalization which consists in expressing formally

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

the UML metamodel and then the metrics definitions using the Z language, known for its maturity and ability to apply proof theorems.

3. Z-BASED FORMALIZATION

This section describes the formal approach adopted to express the metrics definitions rigorously. This approach [4] is based on a leveled formalization. At first, it gives a formal specification of the UML metamodel [3] part on which the second level, consisting on metrics formalization, is defined.

Units

This methodology is an adaptation of the Laurent Henocque contribution [5] about the formal specification of object-oriented constraint programs where we can find the following basics:

- **ObjectReference:** a set of object references as an uninterpreted data type.
  [ObjectReference]
- **ReferenceSet:** a finite set of object references used to model object types.
  ReferenceSet == Φ *ObjectReference*
- **CLASSNAME:** class names defined using free type syntax of Z.
  CLASSNAME ::= ClassElement | ClassNamedElement | ...
- **ObjectDef:** a predefined super class for all future classes.

∪_*ObjectDef*_________________
→*ref: ObjectReference*
→*class:CLASSNAME*
∠________________________

- **Instances:** a function mapping class names to the set of instances of that class

→***instances***: *CLASSNAME* ϕ *ReferenceSet*

- **NIL:** Undefined Object

→***NIL***: *ObjectDef*

- **Class:** implemented via two constructs:

1. **A class definition:** a schema in which we find, in its invariant part, both the class attributes and the inheritance relationships and in its predicate part, specification of class invariants.

∪_***ClassDefElement***______________
→*name:* seq *CHAR*
∠________________________

2. **A class specification:** a combination of a class definition extended with the ObjectDef and class references.

ClassSpecElement | *ClassDefElement f* [*ObjectDef* | *class* = *ClassElement*]

In the current contribution, this approach [4] is used to formalize UML class structures consisting in inheritance, relationships and aggregation. It is also extended with the notion of visibility for the sake of metrics formalization presented later on Section 5.

UML Metamodel Formalization

The following schema is extracted and combined according to the UML metamodel specifications [3]. It consists on the core package.

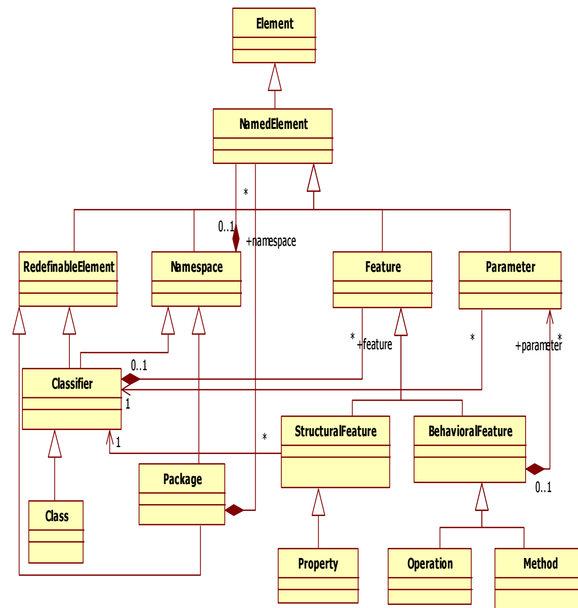

Fig 1. A Partial Representation Of The UML Metamodel Core Package

Since the formalization of the MOOD metrics introduces the notion of visibility which is not included in the described approach [4], we propose

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

in this paper, an extension that will incorporate this notion as part of the Z-formalized approach:

The visibilities package is a subpackage of the Abstractions package that provide the basic constructs from which visibility semantics will be constructed [3]. Represented in the UML metamodel standard by the following (figure 1):

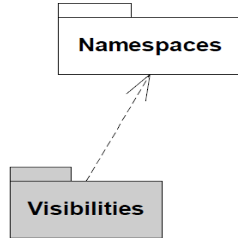

Fig 2. The Visibilities Package

Where the elements defined in this package are:

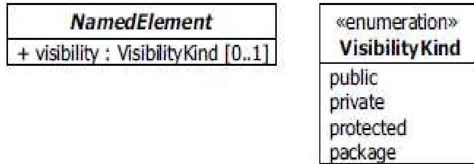

Fig 3. The Elements In The Visibilities Package

In our formal approach, we represent the enumeration by its equivalent in Z language:

VisibilityKind ::= *public* | *private* | *protected* | *package*

The visibilityKind is an enumeration type that contains literals to determine the visibility of elements in a model [3]

We, then, proceed to the redefinition of the NamedElement Class already existing in the previous approach. We, especially, add the visibility attribute that will constrain the usage of a namedElment either in namespaces or in access to the element [3]. This attribute is defined as a power set of the enumeration visibilityKind with a predicate indicating that a namedElement will have at most one kind of visibility.

∪_*ClassDefNamedElement*___________
→*ClassDefElement*
→*visibility:* Π *VisibilityKind*
∩_______________
→*# visibility*) 1
∠____________________

4. FORMAL DEFINITION OF MOOD METRICS

The MOOD metrics suite [6] was proposed by F. B e Abreu team. It aims to respond to a number of criteria listed below:

1. Metrics determination should be formally defined.
2. Non-size metrics should be system size independent.
3. Metrics should be dimensionless or expressed in some consistent unit system.
4. Metrics should be obtainable early in the life-cycle.
5. Metrics should be down-scalable.
6. Metrics should be easily computable.
7. Metrics should be language independent.

MOOD metrics are all system-wide measurements; they are indicators for the following properties:

1. Information hiding Factor: the measurement of hiding factor of encapsulation at attribute level and method level. Two metrics of the MOOD set are concerned: AHF for attribute and MHF for method.

- ***AHF : Attribute Hidden Factor:***

The ratio of hidden attributes (private and protected) to total attributes defined.

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_v(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Where:

Ad: defined attributes;
Av: visible attributes;
Ah: hidden attributes
And: $Ad(Ci) = Av(Ci) + Ah(Ci)$

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The Z specification of AHF:

→*AHF: ObjectDef* ξ *Package* ϕ P
∩______________
→A *o: ObjectDef; P: Package; p:* Z*; A:* Π *Property; C:* Π *Class*
→ | *CN* (*o*, *P*) > 1
→ *f PDAN* (*o*, *P*) > 0
→ *f C* = *allClasses* (*o*, *P*)
→ *f* (A *c: C* ∞ *A* = *allAttributes* (*o*, *c*))
→ *f* (A *a: A* ∞ *p* = *p* + 1 - *APV* (*a*, *P*))
∞ *OHF* (*o*, *P*) = *p* div *PDAN* (*o*, *P*)

Where 1 – APV(a,P) indicates the percentage of the classes in the package from which attributes is not visible.

- ***MHF: Method Hiding Factor :***

The ratio of hidden methods (private and protected) to total methods defined

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_v(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Where:

Md: defined methods*;*
Mv: visible methods*;*
Mh: hidden methods

And : *Md* (*Ci*) = *Mv* (*Ci*) + *Mh* (*Ci*)

The Z specification of MHF:

→*MHF: ObjectDef* ξ *Package* ϕ P
∩______________
→A *o: ObjectDef; P: Package; p:* Z*; M:* Π *Operation; C:* Π *Class*
→ | *CN* (*o*, *P*) > 1
→ *f PDON* (*o*, *P*) > 0
→ *f C* = *allClasses* (*o*, *P*)
→ *f* (A *c: C* ∞ *M* = *allOperations* (*o*, *c*))
→ *f* (A *m: M* ∞ *p* = *p* + 1 - *OPV* (*m*, *P*)) ∞ *MHF* (*o*, *P*) = *p* div *PDON* (*o*, *P*)

Where 1 – OPV(m,P) indicates the percentage of the classes in the package from which operations is not visible.

2. Inheritance Factor: The degree to which the class architecture of an Object-0riented system makes use of inheritance for both methods and attributes.

1. *AIF: Attribute Inheritance Factor:*

The ratio of inherited attributes to total attributes defined.

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} = 1 - \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Where:

Aa: Attributes availables;
Ad: Attributes defined;
Ai: Attributes inherited (not ovverriden)

And: *Aa*(*Ci*) = *Ad* (*Ci*)+*Ai*(*Ci*) / *Ad*(*Ci*) = *An* (*Ci*)+*Ao*(*Ci*)

The Z specification of AIF:

→*AIF: ObjectDef* ξ *Package* ϕ P
∩______________
→A *o: ObjectDef; P: Package* | *PAAN* (*o*, *P*) > 0
→ ∞ *AIF* (*o*, *P*) = *PIAN* (*o*, *P*) div *PAAN* (*o*, *P*)

2. *MIF: Method Inheritance Factor:*

The ratio of inherited methods to total methods defined.

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} = 1 - \frac{\sum_{i=1}^{TC} M_d(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Where:

Ma: Methods availables;
Md: Methods defined;
Mi: Methods inherited (not overriden)

And: *Ma*(*Ci*) = *Md* (*Ci*)+*Mi*(*Ci*) / *Md*(*Ci*) = *Mn* (*Ci*)+*Mo*(*Ci*)

© 2005 - 2013 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The Z specification of MIF:

→*MIF: ObjectDef* ξ *Package* ϕ P
∩_______________
→A *o: ObjectDef; P: Package* | *PAON* (*o*, *P*) > 0
→ ∞ *MIF* (*o*, *P*) = *PION* (*o*, *P*) div *PAON* (*o*, *P*)

3. Polymorphism Factor: Measurement of the degree of overriding in the class inheritance tree. It represents the actual number of possible different distinct polymorphic situation for a class Ci.

.

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}\left[M_n(C_i)\times DC(C_i)\right]}$$

Where:

Mo: Methods overridden;
Mn: New methods;
DC: Descendants count

The Z specification of PF:

→*PF: ObjectDef* ξ *Package* ϕ P
∩_______________
→A *o: ObjectDef; P: Package* | *PAON* (*o*, *P*) > 0
→ ∞ *PF* (*o*, *P*) = *POON* (*o*, *P*) div *PAON* (*o*, *P*)

4. Coupling Factor: informs about the relationship between modules. It represents the ratio of the maximum possible number of couplings in the system to the actual number of coupling. As a reminder, a class is coupled to another class if it calls methods of another class.

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is_client(C_i,C_j)\right]}{TC^2 - TC - 2\times\sum_{i=1}^{TC} DC(C_i)}$$

Where:

is_client(C_c,C_s)*: the client - supplier relationship;*

TC: Total number of classes.

The Z specification of COF:

→*COF: ObjectDef* ξ *Package* ϕ P
∩_______________
→A*o:ObjectDef; P:Package |CN(o,P)>1*
∞ *COF(o,P) = sqrt(ICLN(o,P) div* →
((CN(o,P) CN(o,P))-CN(o,P)))*

ICLN stands for Internal Coupling Links Number and it represents the Number of distinct coupling relations where both the client and the supplier Classes belong to the current Package (excludes inheritance) [14].

→*ICLN: ObjectDef* ξ *Package* ϕ N
∩_______________
→A*o:ObjectDef; P:Package; C:* Π *Classifier |*
→*C=internalSupplierClasses(o,P)* ∞ *ICLN(o,P) = #C*

5. CONCLUSION AND FUTURE WORK

Up to now, we presented a new approach to formally express software design metrics based on a formal definition of the UML metamodel. We then successfully applied this approach to the well-know suite of metrics: the CK metrics and currently to the MOOD metrics, showing the ability of this model to be expanded, in one hand, to include new notions such as invisibility and to adapt, in other hand, to other suite of metrics based in their definitions on the UML metamodel.

We plan in the future to pursue our ongoing work to propose a relevant tool support that will automate the calculation of design metrics over proposed models. This tool will help to inform about the quality of a design model and is intended to facilitate the integration of measurements in the industry.

REFRENCES:

[1] M. Spivey: The Z Notation. Prentice-Hall, 1992.
[2] J. Woodcock and J. Davies, “Using Z: Specification, Proof and Refinement,” Prentice

© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Hall International Series in Computer Science, 1996.

[3] The Object Management Group, UML 2.3 superstructure specification, 2010. http://www.omg.org/spec/uml/2.3/

[4] M. Lamrani, Y. El Amrani and A. Ettouhami, “Formal Specification of Software Design Metrics,” in The Sixth International Conference on Software Engineering Advances (ICSEA) Barcelona, October 2011.

[5] L. Henocque, “Z specification of Object Oriented Constraint Programs” RACSAM, 2004.

[6] F. B Abreu, “The MOOD Metrics Set In Proc.’” ECOOP'95 Workshop on Metrics, (1995).

[7] S. R. Chidamber and C. F. Kemerer, “A metric suite for Object Oriented Design,” Journal IEEE Transactions on Software Engineering vol. 2, pp. 476 – 493, 1994.

[8] K. Mazhar, R. A. Khan and M. H. Khan, “Significance of Design Properties in Object Oriented Software Product Quality Assessment,” In TECHNIA – International Journal of Computing Science and Communication Technologies, vol. 3, January, 2011. (ISSN 0974-3375)

[9] B. Jagdish, “A Hierarchical Model for object-oriented Design Quality Assessment,” In IEEE Transaction on software engineering, vol. 28, January, 2002.

[10] M. Monperrus, J. M. Jézéquel, J.Champeau and B. Hoeltzener, “A Model-driven Measurement Approach,” In Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'2008), 2008.

[11] M. M. El-Wakil, A. El-Bastawisi, M. B. Riad and A. Fahmy, “A novel approach to formalize Object-Oriented Design,” In 9th International Conference on Empirical Assessment in Software Engineering (EASE 2005), April, 2005.

[12] XQuery 1.0 Standard by W3C XML Query Working Group. http://www.w3.org/TR/2010/REC-xquery-20101214/

[13] A. L. Baroni and F.B. Abreu, “An OCL-Based Formalization of the MOOSE Metric Suite,” In Proceedings of the 7th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QUAOOSE'2003), Darmstadt, 2003.

[14] A. L. Baroni and F. B. Abreu, “A Formal Library for Aiding Metrics Extraction,” In International Workshop on Object-Oriented Re-Engineering at ECOOP, 2003.

[15] The Object Management Group, Object Constraint Language 2.2, 2010 http://www.omg.org/spec/OCL/2.2/

[16] J.A. McQuillan and J. F. Power, “Towards re-usable metric definitions at the meta-level,” In PhD Workshop of the 20th European Conference on Object-Oriented Programming (ECOOP 2006) Nantes, July 2006.

[17] R. Reissing, “Towards a model for object-oriented design measurement,” In ECOOP'01 Workshop QAOOSE, 2001.

[18] R. Barden, S. Stepney and D. Cooper, “The use of Z. In J. E. Nicholls, editor,” In Proceedings of the 6th Z User Meeting, York, UK, 1991, Workshops in Computing, pp. 99–124. Springer, 1992.

[19] G. Booch: Object-Oriented Analysis and Design with Applications (2nd ed.), Benjamin Cummings, 1994.

[20] Y. Jiang, B. Cukic, T. Menzies and N. Bartlow, “Comparing Design and Code Metrics for Software quality Prediction,” In PROMISE, 2008.

[21] L. C. W. Garnett, “Valentine from A Telegraph Clerk ♂ to a Telegraph Clerk ♀,” The Life of James Clerk Maxwell, 631, 1882.

[22] I. Meisels and M. Saaltink, “The Z/EVES 2.0 Reference Manual,” Technical Report TR-99-5493-03e, ORA Canada, October, 1999.

[23] T. Baar, “Experiences with the UML/OCL-Approach to Precise Software Modeling: A Report from Practice, ” In Proc. Net.ObjectDays, Erfurt, 2000.

Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1
© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: **1992-8645** www.jatit.org E-ISSN: **1817-3195**

APPENDIX:

The following is a formal description of the previously used functions throughout the metrics formalization expressions. It was entirely written and verified with Z/EVES tool [22]. Complementary functions are defined in the Annexe Section of the previous contribution [4].

% Subset of Attributes (from one set of Features) belonging to the current Classifier.

→*feature2AttributeSet: ObjectDef* ξ Π *Feature* ϕ Π *Property*
∩_______________
→A *o: ObjectDef; S:* Π *Feature*
→ | *instances o.class = Feature*
→ *f S =* { *f: Feature* | *oclIsKindOf* (*o, Property*) = *TRUE* }
→ ∞ *feature2AttributeSet* (*o, S*) = { *f: S* | *oclAsType* (*o, Property*) = *o* }

% Subset of Operations (from one set of Features) belonging to the current Classifier.

→*feature2OperationSet: ObjectDef* ξ Π *Feature* ϕ Π *Operation*
∩_______________
→A *o: ObjectDef; S:* Π *Feature*
→ | *instances o.class = Feature*
→ *f S =* { *f: Feature* | *oclIsKindOf* (*o, Operation*) = *TRUE* }
→ ∞ *feature2OperationSet* (*o, S*) = { *f: S* | *oclAsType* (*o, Operation*) = *o* }

% Set of Features declared in the Classifier, including overridden Operations.

→*definedFeatures: ObjectDef* ξ *ReferenceSet* ϕ Π *Feature*
∩_______________
→A *o: ObjectDef; C: ReferenceSet; p:* Π *Feature*
→ | *instances o.class = Feature*
→ *f C = Classifier*
→ *f p =* { *f: Feature* | *f* ε *C* } ∞ *definedFeatures* (*o, C*) = *p*

% Set of Classes from which the current GeneralizableElement derives directly.
→*parents: ObjectDef* ξ *RedefinableElement* ϕ Π *RedefinableElement*
∩_______________
→A *o: ObjectDef; r: instances ClassRedefinableElement*
→ | *instances o.class = RedefinableElement*
→ ∞ *parents* (*o, r*)
→ = { *r': RedefinableElement*
→ | *instances ClassRedefinableElement* χ *instances o.class* }

% Set of directly derived Classes of the current GeneralizableElement.

→*children: ObjectDef* ξ *RedefinableElement* ϕ Π *RedefinableElement*
∩_______________
→A *o: ObjectDef; r: RedefinableElement* | *instances o.class = RedefinableElement*
→ ∞ *children* (*o, r*)
→ = { *r': RedefinableElement*
→ | *instances o.class* χ *instances ClassRedefinableElement* }

% Set containing all Features of the Classifier itself and all its inherited Features.

→*allFeatures: ObjectDef* ξ *Classifier* ϕ Π *Feature*
∩_______________
→A *o: ObjectDef; c: Classifier; r: RedefinableElement*
→ ∞ *allFeatures* (*o, c*) = Y {(*allFeatures* ((*oclAsType* (*o, Classifier*)), *c*))}

% Set containing all Attributes of the Classifier itself and all its inherited Attributes (both directly and indirectly).

→*allAttributes: ObjectDef* ξ *Classifier* ϕ Π *Property*
∩_______________
→A *o: ObjectDef; c: Classifier; S:* Π *Property*
→ | *S = feature2AttributeSet* (*o*, (*allFeatures* (*o, c*)))
→ ∞ *allAttributes* (*o, c*) = *S*

% Set containing all Operations of the Classifier itself and all its inherited Operations (both directly and indirectly).

→*allOperations: ObjectDef* ξ *Classifier* ϕ Π *Operation*
∩_______________
→A *o: ObjectDef; c: Classifier; S:* Π *Operation*
→ | *S = feature2OperationSet* (*o*, (*allFeatures* (*o, c*)))
→ ∞ *allOperations* (*o, c*) = *S*

% Set of Classes to which the current Class is coupled (excluding inheritance).

→*coupledClasses: Classifier* ϕ Π *Classifier*
∩_______________
→A *c: Classifier; S:* Π *Classifier*
→ | *S =* { *c': Classifier* | *hasAttribute* (*c, c'*) = *TRUE* }
→ ∞ *coupledClasses c = S*

→*allClientOperations: ObjectDef* ξ *Classifier* ϕ Π *Operation*
∩_______________
→A *o: ObjectDef; c: Classifier; C:* Π *Classifier; M:* Π *Operation*
→ | *coupledClasses c = C f M =* Y { *c': C* ∞ (*allOperations* (*o, c'*)) }
→ ∞ *allClientOperations* (*o, c*) = *M*

% Number of Classes in the Package.

→*CN: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N | *C = allClasses* (*o, P*) *f* →*n* = # *C* ∞ *CN* (*o, P*) = *n*

% Number of Classes in the considered Package where the Attribute can be accessed.

Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1
© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

→AVN: Property ξ Package ϕ N
∩_______________
→A a: Property; P: Package; n: N | n = FVN (a, P) ∞ AVN (a, P) = n

% Number of Classes in the considered Package where the Operation can be accessed.

→OVN: Operation ξ Package ϕ N
∩_______________
→A m: Operation; P: Package; n: N | n = FVN (m, P) ∞ OVN (m, P) = n

% Percentage of Classes in the considered Package where the Attribute can be accessed (excludes the Classifier where the Attribute is declared).

→APV: Property ξ Package ϕ Z
∩_______________
→A o: ObjectDef; a: Property; P: Package; q: Z
→ | CN (o, P) > 1 ƒ q = (AVN (a, P) - 1) div (CN (o, P) - 1)
→ ∞ APV (a, P) = q

% Percentage of Classes in the considered Package where the Operation can be accessed (excludes the Classifier where the Operation is declared).

→OPV: Operation ξ Package ϕ Z
∩_______________
→A o: ObjectDef; m: Operation; P: Package; q: Z
→ | CN (o, P) > 1 ƒ q = (OVN (m, P) - 1) div (CN (o, P) - 1)
→ ∞ APV (m, P) = q

% Number of Attributes defined in the Class

→definedAttributes: ObjectDef ξ Classifier ϕ Π Property
∩_______________
→A o: ObjectDef; C: Classifier; A: Π Property
→ | A = feature2AttributeSet (o, (definedFeatures (o, Classifier)))
→ ∞ definedAttributes (o, C) = A

% Number of Operations defined in the Class
→definedOperations: ObjectDef ξ Classifier ϕ Π Operation
∩_______________
→A o: ObjectDef; C: Classifier; O: Π Operation
→ | O = feature2OperationSet (o, (definedFeatures (o, Classifier)))
→ ∞ definedOperations (o, C) = O

% Number of Defined Attributes in the Classifier

→DAN: ObjectDef ξ Classifier ϕ N
∩_______________
→A o: ObjectDef; C: Classifier; A: Π Property | A = definedAttributes (o, →C) ∞ DAN (o, C) = # A

% Number of Defined Operations in the Classifier

→DON: ObjectDef ξ Classifier ϕ N
∩_______________
→A o: ObjectDef; C: Classifier; O: Π Operation | O = definedOperations →(o, C) ∞ DON (o, C) = # O

% Number of Available Attributes in the Classifier.

→AON: ObjectDef ξ Classifier ϕ N
∩_______________
→A o: ObjectDef; C: Classifier; O: Π Operation | O = allOperations (o, →C) ∞ AON (o, C) = # O

% Number of Available Operations in the Classifier.

→AAN: ObjectDef ξ Classifier ϕ N
∩_______________
→A o: ObjectDef; C: Classifier; A: Π Property | A = allAttributes (o, C)
→ ∞ AAN (o, C) = # A

% Set of directly inherited Features.

→directlyInheritedFeatures: ObjectDef ξ Classifier ϕ Π Feature
∩_______________
→A o: ObjectDef; C: Classifier; F: Π Feature; R: Π RedefinableElement
→ | R = parents (o, C)
→ ƒ F
→ = Y { r: R
→ ∞ (definedFeatures ((oclAsType (o, Classifier)),
→ RedefinableElement)) }
→ ∞ directlyInheritedFeatures (o, C) = F

% Set of all inherited Attributes (both directly and indirectly).

→allInheritedAttributes: ObjectDef ξ Classifier ϕ Π Property
∩_______________
→A o: ObjectDef; C: Classifier; A: Π Property
→ | A = feature2AttributeSet (o, (allInheritedFeatures (o, C)))
→ ∞ allInheritedAttributes (o, C) = A

% Set containing all Operations of the Classifier itself and all its inherited Operations (both directly and indirectly).

→allInheritedOperations: ObjectDef ξ Classifier ϕ Π Operation
∩_______________
→A o: ObjectDef; C: Classifier; O: Π Operation
→ | O = feature2OperationSet (o, (allInheritedFeatures (o, C)))
→ ∞ allInheritedOperations (o, C) = O

% Number of inherited Attributes in the Classifier.

→IAN: ObjectDef ξ Classifier ϕ N
∩_______________
→A o: ObjectDef; C: Classifier; A: Π Property
→ | A = allInheritedAttributes (o, C) ∞ IAN (o, C) = # A

% Number of inherited Operations in the Classifier.

→ION: ObjectDef ξ Classifier ϕ N
∩_______________
→A o: ObjectDef; C: Classifier; O: Π Operation
→ | O = allInheritedOperations (o, C) ∞ ION (o, C) = # O

% Number of Operations defined in the Class that override inherited ones.
→overriddenOperations: ObjectDef ξ Classifier ϕ Π Operation
∩_______________
→A o: ObjectDef; C: Classifier; IO, DO: Π Property
→ | IO = allInheritedOperations (o, C) ƒ DO = definedOperations (o, C)
→ ∞ overriddenOperations (o, C) = IO I DO

Journal of Theoretical and Applied Information Technology
10th March 2013. Vol. 49 No.1
© 2005 - 2013 JATIT & LLS. All rights reserved·

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

% Number of overridden Operations in the Classifier

→*OON: ObjectDef* ξ *Classifier* ϕ N
∩_______________
→A *o: ObjectDef; C: Classifier; O:* Π *Operation*
→ | *O = overriddenOperations* (*o*, *C*) ∞ *OON* (*o*, *C*) = # *O*

% Number of defined Attributes in the Package
→*PDAN: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + DAN* (*o*, *c*))
∞ *PDAN* (*o*, *P*) = *n*

% Number of defined Operations in the Package

→*PDON: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + DON* (*o*, *c*))
∞ *PDON* (*o*, *P*) = *n*

% Number of inherited Attributes in the Package.

→*PIAN: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + IAN* (*o*, *c*))
∞ *PIAN* (*o*, *P*) = *n*

% Number of available Attributes in the Package.
→*PAAN: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + AAN* (*o*, *c*))
∞ *PAAN* (*o*, *P*) = *n*
% Number of available Operations in the Package.
→*PAON: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + AON* (*o*, *c*))
∞ *PAON* (*o*, *P*) = *n*

% Number of inherited Operations in the Package.

→*PION: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + ION* (*o*, *c*))
∞ *PION* (*o*, *P*) = *n*

% Number of overridden Operations in the Package

→*POON: ObjectDef* ξ *Package* ϕ N
∩_______________
→A *o: ObjectDef; P: Package; C:* Π *Class; n:* N
→ | *C = allClasses* (*o*, *P*) *f* (A *c: C* ∞ *n = n + OON* (*o*, *c*))
∞ *POON* (*o*, *P*) = *n*

% Set of supplier Classes in the current Package.

→*internalSupplierClasses: ObjectDef* ξ *Package* ϕ Π*Classifier*
∩_______________
→A*o: ObjectDef; P: Package; C:* Π*Classifier |*
C=supplierClasses(o,P) ∞ *internalSupplierClasses(o,P) = C*