Formal Specification of Software Design Metrics

Meryem Lamrani
Laboratoire Conception et Systèmes
University Mohammed V Agdal
Department of Computer Science
BP 1014 RP Rabat, Morocco
lamrani@fsr.ac.ma

Younès El Amrani
Laboratoire Conception et Systèmes
University Mohammed V Agdal
Department of Computer Science
BP 1014 RP Rabat, Morocco
elamrani@fsr.ac.ma

Aziz Ettouhami
Laboratoire Conception et Systèmes
University Mohammed V Agdal
Department of Computer Science
BP 1014 RP Rabat, Morocco
touhami@fsr.ac.ma

Abstract—**Given the significant interest in applying formal methods to object oriented paradigms, this paper presents a formal approach to define software design quality metrics upon a formal specification of the UML metamodel using the Z language. This multi-level formalization benefits greatly to design metrics as it allows a non ambiguous interpretation and a more rigorous definition, which, in turn, can assist the implementation of tools to measure the software design quality for industrial application. Our achievement gives precise meaning to software design metrics definitions in order to facilitate verification and validation. We, especially, applied our approach to one of the most well known set of metrics: the CK metrics.**

Keywords-formalization; UML metamodel; Z; CK metrics;

I. Introduction

"*Door meten tot weten*" [24] is a famous saying of the Dutch physicist and Nobel laureate Kamerlingh Onnes (1853 - 1926) literally translated as "Through measurement to knowledge". It attests that the quantifying process leads to a better insight and understanding over the measured element. The software engineering area is no exception. It has been widely recognized that the use of software metrics, for being considered as quality indicators, can accurately help improve the final results and keep time and cost estimation under control while assuring quality according to the desired properties.

At first, code metrics such as cyclomatic complexity measure or lines of code measure were defined and applied to track faultiness during software development but have soon shown a weak side for being measured till the implementation phase, which is already a very late phase considering the whole software life cycle. Since then, many software metrics concerned with the design phase were defined and commonly known as design metrics. A combination of both code and design metrics has also been explored with positive results [25].

Several authors have proposed various design metrics such as the MOOD and MOOD2 (Metrics for Object-Oriented Design) [28], MOOSE (Metrics for Object-Oriented Software Engineering) also known as the CK metrics [5], EMOOSE (Extended MOOSE) [29] and QMOOD (Quality Model for Object-Oriented Design) [30]. Most of them are lacking rigor and formalism in their definition.

This paper addresses the problematic lying in software measurement area due to the lack of formalization. Therefore, we present an approach to define formally software design metrics using the Z language [1, 2] over our proposed formal specification of the UML metamodel [3] based on the Laurent Henocque [4] transformation of UML class structures concept. This approach is intended to provide precise and complete formalized definition of software design metrics.

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 presents a brief overview of the Z language. Section 4 illustrates the Z formalization of the UML metamodel. Section 5 introduces an approach to formalize software design metrics definition and finally, conclusions are drawn in Section 6.

II. Related work

Measurement has always been a fundamental step to understandability and control. When it comes to quality, measurement is obviously more difficult to obtain due to its subjectivity, however, some of its aspects can be measured and verified and thus be considered as objective. Software engineering, for being a very recent field and especially a more human-intensive discipline [26], suffers from a lack of measurement which, undeniably, leads to an out of control in delivery and cost estimation of the software production.

With a massive research concerns, measurement has reached an early stage of the software life cycle. Therefore, the software design metrics were defined according to the commonly approved properties considered as quality indicators.

Many software metrics exist nowadays [5-7] however their practical use remains unpopular in the software industry mostly because of their ambiguity and non reliability [8]. Knowing that measurements have to be standard to mean the same thing to everyone, metrics should enforce their definitions using formal methods to become more useful, convenient and trust worthy.

Among authors who attempt to give a formal definition of software metrics, Baroni et al. [10], which proposed a Formal Library for Aiding Metrics Extraction (FLAME) [9] that uses OCL [11] as a metric definition language. El-Wakil et al. [12] built metric definitions using XQuery [13] language. McQuillan et al. [14] based their work on Baroni's approach and extended the UML metamodel 2.0 to offer a framework for metric definitions. Harmer and Wilkie [15] expressed metric definitions as SQL queries over a relational schema. Goulao et al. [16] also used the Baroni's approach for defining component based metrics and used the UML 2.0 metamodel as a basis for their definitions. In all related approaches, the UML metamodel is described in a subset of UML itself, supplemented by a set of well-formedness rules provided in OCL and natural language (English). Unfortunately, these approaches neither offer the possibility to check certain system properties nor they exclude the ambiguous use of UML itself to express the UML metamodel. Whereas in this article, there are two main contributions: the first contribution is to express UML metamodel in a formal language without any reflexive reference to UML, it results in more clarity. The second contribution is to express the CK metrics in a rigorous definition that enables to check certain system properties involving metrics. This could not be achieved with previous definitions using OCL.

In this paper, a Z formal model of UML metamodel is described. The model is enough general to express any set of metrics defined upon the UML metamodel 2.3. Then the authors provide a formal definition of the CK metrics. Expressing, for the first time, the CK metrics in a state-based formal method.

III. Z OVERVIEW

Z [1, 2] is a formal specification language originally created by J.-R. Abrial and then developed by the Programming Research Group at Oxford. Its notation is based upon set theory and mathematical logic, which consists in a first-order predicate calculus.

One aspect of the Z notation is the schemas. The notion of schema in Z is closely related to a class structure in Object-oriented concept. It combines two parts: a declaration part and a predicate part. Another particularity of Z is the use of types. Types in Z can be either basic or composite.

We used Z notation to build our formalization because of its maturity and the ability to check consistency of the design using proof theorems unlike the Object-Z [17] language, which was specifically developed to gain facilities with object oriented specification aspects to the detriment of formalization advantages mentioned earlier for Z language.

Some authors proposed a formalization of UML class constructs using PVS specification language (PVS-SL) [31], a language based on higher-order logic, where relationships and other constituents of UML diagrams are represented as PVS theories. Other approaches suggested the use of Description Logics (DLs) [32-33] where Object-oriented concepts are modeled in means of *concepts* (unary relations) and *relations* (n-ary relations). However, most attempts were done using Z. Among them, there are Hall [18-19] and Hammond [20], which, in their approaches, supported class, association and inheritance. Malcolm Shroff and Robert B. France [21-22] based their approach on the Hall and Hammond's Z formalization approach of the class structures with the particularity of introducing inheritance relationship as an attribute in the inheriting class. We disgarded Hall's original approach because it predates UML definition and it does not consider aggregation which is used in the core backbone of the UML metamodel. We also disgarded France's modeling because it uses a global system approach, he models properties of objects as functions from identities to property values. This approach is less appealing than the intuitive encapsulation of each object's state which is more natural to object-oriented thinking.

After investigating these different methods, we choose the Laurent Henocque approach [4], which was elaborated to give a formal specification to Object Oriented Constraint Programs. This choice is mostly justified by the approach to represent inheritance and aggregation relationships and also its responds to our need for a formalization of the object system as part of the specification.

Since the objective of this paper is to present a formalization of design metrics, we settled for providing a description of the Henocque approach [4], gradually through our formalization of the UML metamodel.

IV. Z FORMALIZATION OF THE UML METAMODEL

The UML metamodel is the result of many years of effort to standardize software engineering practices. Itself defined in UML, it is considered as the standard model to represents object models using UML. The following transformation concerns the core backbone of the UML metamodel, captured and reconstituted from the UML metamodel 2.3.

A. Different Level of Abstractions of the Metrics

Definition of each metric considered in the formalization is done upon the UML metamodel at different levels of abstraction:

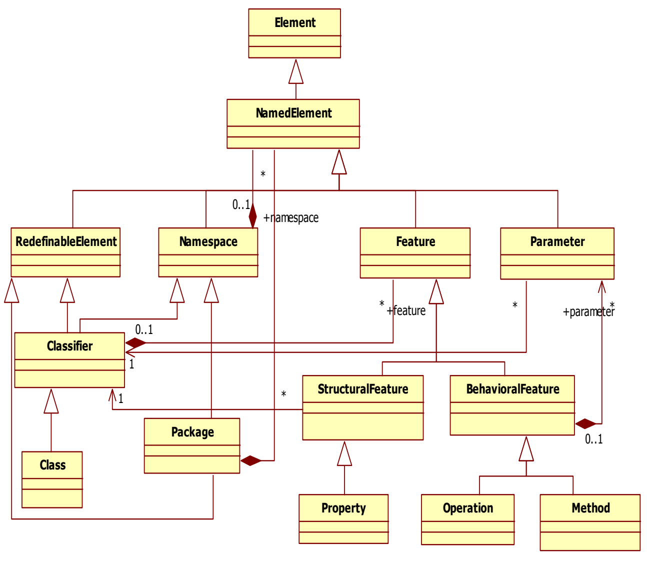

Figure 1. A fragment of the core backbone of the UML metamodel

B. *Z Transformation*

The following formalization is analyzed and validated using Z/EVES tool [23].

At the beginning, Laurent Henocque [4] defines an uninterpreted dataType [***ObjectReference***] considered as a set of object references and [***ReferenceSet***] as a finite set of object references later used to model object types.

ReferenceSet == $\mathbb{F}$ *ObjectReference*

For practical reasons, a global class names is defined using free type declaration syntax:
CLASSNAME ::= ClassElement | ClassNamedElement | ...

A function instances describes the mapping between class names and the set of instances of that class

| ***instances***: *CLASSNAME* $\rightarrow$ *ReferenceSet*

And then, he defines ***ObjectDef*** as a predefined super class for all future classes. This class will be used to bijectively map each object to a unique individual from the set ObjectReference.

An instance of each class presented is identified by its respective object identifier ***ident*** which is of type declared as a basic type.

ObjectDef
ref: ObjectReference
class: CLASSNAME

For our metrics transformation, we extend the ObjectDef with a NIL object to represent a undefined object.

| ***NIL***: *ObjectDef*

According to Henocque [4], each class is implemented via two constructs:

- **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*

with [CHAR] being a given set containing all characters. The attribute name was introduced in this transformation because the Z/EVES tool [23] does not allow the construction of an empty class. In the following, even though the UML metamodel class constructs contains attributes and predicates, we will only focus on the relationship between classes in order to simplify readability of our metrics transformation.

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

ClassSpecElement $\widehat{=}$ *ClassDefElement* $\wedge$ [*ObjectDef* | *class* = *ClassElement*]

The $\widehat{=}$ symbol offers a different way to define a schema and the logical operator $\wedge$ allows the extension.

As stated in the first part of the class constructs, inheritance relationship is defined in the class definition:

ClassDefNamedElement
ClassDefElement

ClassDefClassifier
ClassDefNamespace
ClassDefRedefinableElement

In both cases, simple inheritance or multiple inheritance, the inheritance relationship is built simply by importing the schema definition of inherited superclasses into the class that inherit from them.

Beside the inheritance relationship, we are also concerned with the aggregation and relations with multiplicities. General relations are free of constraints, which mean that every tuple can be accepted. The multiplicity is naturally stated in the predicate part as the cardinal of related target objects for each source object.

| *pc: Parameter* $\leftrightarrow$ *Classifier*
| $\forall$ *c: Classifier* $\bullet$ # (*pc* ({*c*})) $\leqslant$ 1

The aggregate relation is more constrained than a general one, thus we have to change the type of relation to make a distinction between both. In the different aggregate relations given in our UML metamodel fragment, the multiplicity is of 0..1 which means that each component occurs in at most one composite. Consequently, its relational inverse is an injective partial function.

$hasNamedElement: Namespace \leftrightarrow NamedElement$

$hasNamedElement^{\sim} \in NamedElement \rightarrowtail\!\!\!\!\!\!\!\!\!\;+ \;\; Namespace$

The $\rightarrowtail\!\!\!\!\!\!\!\!\!\;+ \;\;$ symbol represents the partial function and the $\sim$ stands for the relational inverse.

And finally, we define class types for a better understanding of what the types really represent. They are defined using an axiomatic definition:

Element, NamedElement, Namespace, … : ReferenceSet

$Element = instances\ ClassElement \cup NamedElement$
$NamedElement$
$\quad = instances\ ClassNamedElement \cup Namespace \cup RedefinableElement \cup Feature$
. . .
$instances\ ClassElement$
$\quad = \{\ o: ClassSpecElement \mid o.class = ClassElement \bullet o.ref\ \}$
$instances\ ClassNamedElement$
$\quad = \{\ o: ClassSpecNamedElement \mid o.class = ClassNamedElement \bullet o.ref\ \}$
. . .
$\forall\ i: instances\ ClassElement \bullet \exists\ x: ClassSpecElement \bullet x.ref = i$
$\forall\ i: instances\ ClassNamedElement \bullet \exists\ x: ClassSpecNamedElement \bullet x.ref = i$
.. .. ..

The type sets defined in the declaration part correspond to the existing classes of our given model. Each type is defined as a finite set of object references. The predicate part describes the properties of these sets. First, we have a type equal to the union of the corresponding class instances and the type of all its subclasses. And then, that each object reference is used at most once for an object which means that no two distinct object bindings share the same object reference.

V. An Approach to Formalize Design Quality Metrics Definitions

Among existing metrics, we will discuss the CK Metrics [5] proposed by Chidamber and Kemerer, one of the most well known suites of Object-oriented metrics. These metrics help measuring different aspects of an Object-Oriented design including complexity, coupling and cohesion. Several studies [26-27] have confirmed their usefulness as quality indicators.

An OCL formalization of the CK metrics was proposed by the authors Baroni et al. [10], defined using functions formalized in FLAME [9]. Although, OCL is based on mathematical logic, it still does not provide a formally defined semantics, furthermore, its syntax is given by a grammar description and no metamodel is available unlike the metamodel of UML which means that it suffers from an absence of well-formedness rules.

Considering that most metrics formalization efforts are made in OCL but yet still unpopular in the software industry, we argue that a more rigorous method of formalization should be explored in order to overcome OCL limitations.

As a simple example, the expression iterate, used in the OCL formalization of the DIT metrics, is known to be potentially non-deterministic since there is no precision on order evaluation leading to different possible results[34].

```
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
```

Also, in each metrics defined with OCL, we could find many OCL keywords (self, asSet…) and predefined functions (OclAsType, OclIsKindOf…) that are not precise enough semantically. Therefore, we propose a formal definition for those frequently used predefined functions in order to obtain a complete and precise definition of the CK metrics.

A. Formalizing OCL Predefined Functions

OclIsTypeOf and OclIsKindOf have the same signature. They are both applied to an object, take a type as parameter and return a Boolean as a result. The only difference is that the first one deals with the direct type of the object when the second one determines whether the type given in parameter is either the direct type or one of the supertypes of the object.

When it is certain that the actual type of the object is the subtype, the object can be re-typed using the OclAsType operation. Otherwise, the expression is undefined.

We propose a Z-formalization of these predefined operations using the Henocque approach [4].

$oclIsTypeOf: ObjectDef \times ReferenceSet \rightarrow Boolean$

$\forall\, o: ObjectDef;\ t: ReferenceSet \mid instances\ o.class = t \bullet oclIsTypeOf\ (o, t) = TRUE$

$\forall\, o: ObjectDef;\ t: ReferenceSet \mid instances\ o.class \neq t \bullet oclIsTypeOf\ (o, t) = FALSE$

The formalization is given as an axiomatic function. It takes the ObjectDef and a ReferenceSet as parameter and it returns a Boolean. When instances of o.class referering to the object's type is equal to the type given in parameter the expression of OclIsTypeOf is true. When both types are not the same, the operation return false.

$oclIsKindOf: ObjectDef \times ReferenceSet \rightarrow Boolean$

$\forall\, o: ObjectDef;\ t: ReferenceSet \mid instances\ o.class \subseteq t \bullet oclIsKindOf\ (o, t) = TRUE$

$\forall\, o: ObjectDef; t: ReferenceSet \mid \neg instances\ o.class \subseteq t \bullet oclIsKindOf\ (o, t) = FALSE$

When the type of the object given in parameter (expressed as instances o.class) is part of the ReferenceSet given in parameter, the expression oclIsKindOf returns true. Otherwise, it returns false.

$oclAsType: ObjectDef \times ReferenceSet \rightarrow ObjectDef$

$\forall\, o: ObjectDef;\ t: ReferenceSet \mid instances\ o.class = t \bullet oclAsType\ (o, t) = o$

$\forall\, o: ObjectDef;\ t: ReferenceSet \mid \neg\ instances\ o.class \subseteq t \bullet oclAsType\ (o, t) = NIL$

$\forall\, o: ObjectDef;\ t: ReferenceSet \mid instances\ o.class \subset t$
$\bullet \exists\, r: ObjectDef \mid r.ref = o.ref \wedge instances\ r.class = t \bullet oclAsType\ (o, t) = r$

With oclAsType operation we distinguish between three cases:

The first one is when the type given in parameter corresponds to the object's type, which means the result of applying oclAsType is the object itself.

The second one is when the object's type is not the same nor is it a part of the ReferenceSet given in parameter, which means that the expression is undefined and in that case we return the NIL value defined earlier as an extension to ObjectDef.

Finally, the third one is when the object's type is part of the ReferenceSet given in parameter. In that case, the expression OclAsType returns an object which has the same reference as the object in entry (that means it is the same object) but having as type the ReferenceSet in parameter.

B. *Formalizing the CK metrics*

Each of the above metrics refers to an individual class and not to the whole system.

- ***Weighted Methods Complexity:*** the sum of the complexity of all methods for a class. If all method complexities are considered to be unique, WMC is equal to the number of methods.

$WMC: ObjectDef \times Classifier \rightarrow \mathbb{N}$

$\forall\, o: ObjectDef;\ c: Classifier;\ S: \mathbb{P}\ Operation \mid S = allOperations\ (o, c)$
$\bullet\ WMC\ (o, c) = \#\, S$

- ***Number of Children:*** counts the number of children classes that inherit directly from the current class.

$NOC: ObjectDef \times Classifier \rightarrow \mathbb{N}$

$\forall\, o: ObjectDef;\ c: Classifier;\ n: \mathbb{N} \mid n = CHIN\ (o, c) \bullet NOC\ (o, c) = n$

- ***Depth of Inheritance Tree:*** measures the length of the inheritance chain from the current class to the root.

$DIT: ObjectDef \times RedefinableElement \rightarrow \mathbb{N}$

$\forall\, o: ObjectDef;\ r: RedefinableElement \mid isRoot\ (o, r) = TRUE \bullet DIT\ (o, r) = 0$

$\forall\, o: ObjectDef;\ r: RedefinableElement;\ R: \mathbb{P}\ RedefinableElement;\ n: \mathbb{N};\ S: \mathbb{P}\ \mathbb{N}$
$\mid PARN\ (o, r) \geqslant 1$
$\wedge R = parents\ (o, r)$
$\wedge S = \{\ depth: \mathbb{N} \mid \forall\, r': R \bullet depth = DIT\ (o, r')\ \}$
$\wedge n = max\ S \bullet DIT\ (o, r) = n$

- ***Coupling Between Classes:*** the number of coupling with other classes.

$CBO: ObjectDef \times Classifier \rightarrow \mathbb{N}$

$\forall\, o: ObjectDef;\ c: Classifier;\ C: \mathbb{P}\ Classifier \mid C = coupledClasses\ (o, c)$
$\bullet\ CBO\ (o, c) = \#\, C$

- ***Response for Class:*** 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. It can be defined as | RS | where RS is the response set for the class expressed as:

$$RS = \{ M \} \cup \text{all } i\ \{ R\ i \}$$

with:
- **{ Ri}** = set of methods called by method *i*
- **{ M }** = set of all methods in the class.

$$\begin{array}{|l} \mathit{RFC: ObjectDef} \times \mathit{Classifier} \rightarrow \mathbb{N} \\ \hline \forall\, o: \mathit{ObjectDef};\ c: \mathit{Classifier};\ m, mc: \mathbb{P}\ \mathit{Operation} \\ \quad \mid\ m = \mathit{allOperations}\ (o, c) \wedge mc = \mathit{allClientOperations}\ (o, c) \\ \quad \bullet\ \mathit{RFC}\ (o, c) = \#\, m + \#\, mc \end{array}$$

- ***Lack of Cohesion of Methods:*** The degree of similarity of methods in the current class. This metric was first improved by Chidamber and Kemerer themselves, calling it LCOM2, then by Henderson-Sellers by proposing the following expression:

$$\text{LCOM3} = (\text{m-sum(mA)/a})/(\text{m-1})$$

with:
- **m:** number of methods in a class.
- **a:** number of attributes in a class.
- **mA:** number of methods that access the attribute a.
- **sum(mA):** sum of all mA over all the attributes in the class.

$$\begin{array}{|l} \mathit{LCOM3: ObjectDef} \times \mathit{Classifier} \rightarrow \mathbb{N} \\ \hline \forall\, o: \mathit{ObjectDef};\ c: \mathit{Classifier};\ m: \mathbb{P}\ \mathit{Operation};\ a: \mathbb{P}\ \mathit{Property};\ n: \mathbb{N} \\ \quad \mid\ m = \mathit{allOperations}\ (o, c) \\ \quad\ \ \wedge\ a = \mathit{allAttributes}\ (o, c) \\ \quad\ \ \wedge\ (\forall\, A: a \bullet\ n = n + \mathit{sum}\ \langle\!\langle \mathit{mA}\ (A, m)\rangle\!\rangle) \\ \quad \bullet\ \mathit{LCOM}\ (o, c) = (\#\, m - n \operatorname{div} \#\, a) \operatorname{div} \#\, a - 1 \end{array}$$

VI. Conclusion and Future Work

In this work, we were mainly concerned about the formal definition of the CK metrics as a restricted application of our formalization approach, which consists on expressing formally the UML metamodel and then giving a formal definition of software design quality metrics for the sake of validation and verification.

As future work, we plan to extend our contribution to MOOD and MOOD2 - Metrics for Object-Oriented Design [28], EMOOSE- Extended MOOSE [29] and QMOOD Quality Model for Object-Oriented Design [30]. We, also, plan to build a support tool that will, first, automate the formal Z representation of design models according to our UML metamodel formalization and then, implement already formalized metrics expressions to automate their calculation and compare results.

References

[1] M. Spivey, "The Z Notation," Prentice-Hall, 1992.

[2] J. Woodcock and J. Davies, "Using Z: Specification, Proof and Refinement," Prentice 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] Laurent Henocque, "Z specification of Object Oriented Constraint Programs," RACSAM , 2004.

[5] Shyam R. Chidamber and Chris F. Kemerer, "A metric suite for Object Oriented Design," Journal IEEE Transactions on Software Engineering Volume 20 Issue 6, 1994, pp. 476 – 493.

[6] Lorenz M. and Kidd J., "Object-Oriented Software Metrics," Prentice Hall Object-Oriented Series, 1994.

[7] Norman E. Fenton and Lawrence Peeger S., "Software Metrics: A Rigorous and Practical Approach," International Thompson Computer Press, 1996.

[8] L. Briand, J. Daly and J.Wüst, "A unified framework for coupling measurement in object-oriented systems," IEEE Transactions on Software Engineering 25, 1999.

[9] Aline L. Baroni and F. Brito e 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, Germany, 2003.

[10] Aline L. Baroni and F. Brito e Abreu, "A Formal Library for Aiding Metrics Extraction," International Workshop on Object-Oriented Re-Engineering at ECOOP, 2003.

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

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

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

[14] Jacqueline A. McQuillan and James 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, France, 3-7, July 2006.

[15] F. Wilkie And T. Harmer, "Tool support for measuring complexity in heterogeneous object-oriented software," In Proceedings of IEEE International Conference on Software Maintenance, Montreal, Canada, 2002.

[16] M. Goulao and F. Brito e Abreu, "Formalizing metrics for COTS," In Proceddings of the ICSE Workshop on Models and Processes for the Evaluation of COTS Components, Edinburgh, Scotland, 2004.

[17] D. Duke, P. King, G.A. Rose, and G. Smith, 1991. The Object-Z Specification Language, version 1, Technical Report 91-1, Department of Computing Science, University of Queensland, Australia.

[18] J. A. Hall, "Specifying and Interpreting Class Hierarchies in Z," In Bowen and Hall, pp. 120-138.

[19] J.P. Bowen and J.A. Hall, editors, Z User Workshop, Cambridge 1994, Workshops in Computing. Springer-Verlag, New York , 1994.

[20] J. A. R. Hammond, "Producing Z specifications from Object-Oriented Analysis," In Bowen and Hall, pp. 316-336.

[21] Robert B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and M. Shroff, "Exploring the Semantics of UML Type Structures with Z", In: Proceedings of the Formal Methods for Open Object-based Distributed Systems (FMOODS'97), Springer, pp. 247-257.

[22] M. Shroff and Robert B. France, "Towards a Formalization of UML Class Structures in Z", In Proceedings of the 21st Computer Software and Application Conference (COMP-SAC'97), IEEE Press, 646-651.

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

[24] 'The Significance of Quantitative Research in Physics', Inaugural Address at the University of Leiden (1882). In Hendrik Casimir, Haphazard Reality: Half a Century of Science (1983), 160-1.

[25] Y. Jiang, B. Cukic, T. Menzies, and N. Bartlow: "Comparing Design and Code Metrics for Software quality Prediction": PROMISE (2008).

[26] Sandro Morasca: Software Measurement (2007).

[27] Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand, and Walcelio L. Melo: "A Validation of Object-Oriented Design Metrics as Quality Indicators": In IEEE Transactions on Software Engineering, vol**.** 22, NO. 10, October 1996, pp. 751 – 761,

[28] F. Brito e Abreu and R. Carapuça, "Object-Oriented Software Engineering: Measuring and Controlling the Development Process," 4th Int. Conf. on Software Quality, McLean, VA, USA, 3-5 October 1994.

[29] W. Li, S. Henry, D. Kafura and R. Schulman, "Measuring object-oriented design," *Journal of Object-Oriented programming*, vol. 8, NO. 4, pp. 48-55**.** July/August 1995.

[30] J. Bansiya and C. Davids: "Automated metrics and object-oriented development," *Dr. Dobbs Journal*, pp. 42–48, December 1997.

[31] Demissie B. Aredo, I. Traore, and K. Stølen: *"Towards a formalization of UML Class Structure in PVS,"* Research Report no. 272, Department of Informatics, University of Oslo, August 1999.

[32] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini: "Reasoning on UML Class Diagrams in Description Logics," In Proceedings of IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD 2001). 2001.

[33] L. Efrizoni, W.M.N Wan-Kadir and, R. Mohamad: "Formalization of UML Class using Description Logics," In the International Symposium in Information Technology (ITSim), 2010.

[34] M. Richters and M. Gogolla: "On Formalizing the UML Object Constraint Language," In Proceedings of the 17th International Conference on Conceptual Modeling. Springer-Verlag, London, UK, 1998.

Appendix:

The whole specification, of 426 lines in Latex, was entirely written and verified using the Z/EVES tool but for space reason, this appendix does not contain all of it.

The following is a description of the previously declared functions in the metrics formalization chapter.

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

$$\begin{array}{l}
\mathit{feature2AttributeSet: ObjectDef} \times \mathbb{P}\ \mathit{Feature} \rightarrow \mathbb{P}\ \mathit{Property} \\
\hline
\forall\ \mathit{o: ObjectDef;\ S:}\ \mathbb{P}\ \mathit{Feature} \\
\quad \mid\ \mathit{instances\ o.class = Feature} \\
\qquad \land\ \mathit{S} = \{\ \mathit{f: Feature} \mid \mathit{oclIsKindOf\ (o,\ Property) = TRUE}\ \} \\
\quad \bullet\ \mathit{feature2AttributeSet\ (o,\ S)} = \{\ \mathit{f: S} \mid \mathit{oclAsType\ (o,\ Property) =} \\
\mathit{o}\ \}
\end{array}$$

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

$$\begin{array}{l}
\mathit{feature2OperationSet: ObjectDef} \times \mathbb{P}\ \mathit{Feature} \rightarrow \mathbb{P}\ \mathit{Operation} \\
\hline
\forall\ \mathit{o: ObjectDef;\ S:}\ \mathbb{P}\ \mathit{Feature} \\
\quad \mid\ \mathit{instances\ o.class = Feature} \\
\qquad \land\ \mathit{S} = \{\ \mathit{f: Feature} \mid \mathit{oclIsKindOf\ (o,\ Operation) = TRUE}\ \} \\
\quad \bullet\ \mathit{feature2OperationSet\ (o,\ S)} = \{\ \mathit{f: S} \mid \mathit{oclAsType\ (o,\ Operation) =} \\
\mathit{o}\ \}
\end{array}$$

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

$$\begin{array}{l}
\mathit{definedFeatures: ObjectDef} \times \mathit{Classifier} \rightarrow \mathbb{P}\ \mathit{Feature} \\
\hline
\forall\ \mathit{o: ObjectDef;\ c: Classifier;\ p:}\ \mathbb{P}\ \mathit{Feature} \\
\quad \mid\ \mathit{instances\ o.class = Feature} \land \mathit{p} = \{\ \mathit{f: Feature} \mid \mathit{f} \in \mathit{Classifier}\ \} \\
\quad \bullet\ \mathit{definedFeatures\ (o,\ c) = p}
\end{array}$$

%*Set of Classes from which the current GeneralizableElement derives directly.*

$$\begin{array}{l}
\mathit{parents: ObjectDef} \times \mathit{RedefinableElement} \rightarrow \mathbb{P}\ \mathit{RedefinableElement} \\
\hline
\forall\ \mathit{o: ObjectDef;\ r: instances\ ClassRedefinableElement} \\
\quad \mid\ \mathit{instances\ o.class = RedefinableElement} \\
\quad \bullet\ \mathit{parents\ (o,\ r)} \\
\qquad = \{\ \mathit{r': RedefinableElement} \\
\qquad\qquad \mid\ \mathit{instances\ ClassRedefinableElement} \subset \mathit{instances\ o.class}\ \}
\end{array}$$

%*Set of directly derived Classes of the current GeneralizableElement.*

$$\begin{array}{l}
\mathit{children: ObjectDef} \times \mathit{RedefinableElement} \rightarrow \mathbb{P}\ \mathit{RedefinableElement} \\
\hline
\forall\ \mathit{o: ObjectDef;\ r: RedefinableElement} \mid \mathit{instances\ o.class =} \\
\mathit{RedefinableElement} \\
\quad \bullet\ \mathit{children\ (o,\ r)} \\
\qquad = \{\ \mathit{r': RedefinableElement} \\
\qquad\qquad \mid\ \mathit{instances\ o.class} \subset \mathit{instances\ ClassRedefinableElement}\ \}
\end{array}$$

%*Number of directly derived Classes.*

$$\begin{array}{l}
\mathit{CHIN: ObjectDef} \times \mathit{RedefinableElement} \rightarrow \mathbb{N} \\
\hline
\forall\ \mathit{o: ObjectDef;\ r: RedefinableElement;\ S:}\ \mathbb{P}\ \mathit{RedefinableElement} \\
\quad \mid\ \mathit{S = children\ (o,\ r)} \bullet \mathit{CHIN\ (o,\ r)} = \#\ \mathit{S}
\end{array}$$

%Number of Classes from which the current RedefinableElement derives directly.

PARN: ObjectDef $\times$ *RedefinableElement* $\rightarrow \mathbb{N}$

$\forall$ *o: ObjectDef; r: RedefinableElement; S:* $\mathbb{P}$ *RedefinableElement*
  $\mid$ *S = parents* (*o*, *r*) $\bullet$ *PARN* (*o*, *r*) = # *S*

%Indicates whether the RedefinableElement has ascendants or not.

isRoot: ObjectDef $\times$ *RedefinableElement* $\rightarrow$ *Boolean*

$\forall$ *o: ObjectDef; r: RedefinableElement* $\mid$ *PARN* (*o*, *r*) = 0 $\bullet$ *isRoot* (*o*, *r*) = *TRUE*
$\forall$ *o: ObjectDef; r: RedefinableElement* $\mid$ *PARN* (*o*, *r*) $\neq$ 0
  $\bullet$ *isRoot* (*o*, *r*) = *FALSE*

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

allFeatures: ObjectDef $\times$ *Classifier* $\rightarrow \mathbb{P}$ *Feature*

$\forall$ *o: ObjectDef; c: Classifier; r: RedefinableElement*
  $\bullet$ *allFeatures* (*o*, *c*) = $\cup$ {(*allFeatures* ((*oclAsType* (*o*, *Classifier*)), *c*))}

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

allAttributes: ObjectDef $\times$ *Classifier* $\rightarrow \mathbb{P}$ *Property*

$\forall$ *o: ObjectDef; c: Classifier; S:* $\mathbb{P}$ *Property*
  $\mid$ *S = feature2AttributeSet* (*o*, (*allFeatures* (*o*, *c*)))
  $\bullet$ *allAttributes* (*o*, *c*) = *S*

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

allOperations: ObjectDef $\times$ *Classifier* $\rightarrow \mathbb{P}$ *Operation*

$\forall$ *o: ObjectDef; c: Classifier; S:* $\mathbb{P}$ *Operation*
  $\mid$ *S = feature2OperationSet* (*o*, (*allFeatures* (*o*, *c*)))
  $\bullet$ *allOperations* (*o*, *c*) = *S*

% *Types (Classifiers) of all attributes that are accessible within the current Classifier.*

typesOfAllAccessibleAttributes: Classifier $\rightarrow \mathbb{P}$ *Classifier*

$\forall$ *o: ObjectDef; c: Classifier; S:* $\mathbb{P}$ *Property; F:* $\mathbb{P}$ *Feature; T:* $\mathbb{P}$ *Classifier*
  $\mid$ *S = allAttributes* (*o*, *c*) $\wedge$ *feature2AttributeSet* (*o*, *F*) = *S* $\wedge$ *F* $\subseteq$ *T*
  $\bullet$ *typesOfAllAccessibleAttributes c = T*

% *True if the first Classifier has an accessible attribute of type given as second Classifier.*

hasAttribute: Classifier $\times$ *Classifier* $\rightarrow$ *Boolean*

$\forall$ *c, c': Classifier* $\mid$ *c'* $\in$ *typesOfAllAccessibleAttributes c*
  $\bullet$ *hasAttribute* (*c*, *c'*) = *TRUE*
$\forall$ *c, c': Classifier* $\mid$ *c'* $\notin$ *typesOfAllAccessibleAttributes c*
  $\bullet$ *hasAttribute* (*c*, *c'*) = *FALSE*

% *Set of Classifiers to which the current Classifier is coupled (excluding inheritance).*

coupledClasses: Classifier $\rightarrow \mathbb{P}$ *Classifier*

$\forall$ *c: Classifier; S:* $\mathbb{P}$ *Classifier*
  $\mid$ *S* = { *c': Classifier* $\mid$ *hasAttribute* (*c*, *c'*) = *TRUE* }
  $\bullet$ *coupledClasses c = S*

% *Set of Operations that might respond to a message received by its object.*

allClientOperations: ObjectDef $\times$ *Classifier* $\rightarrow \mathbb{P}$ *Operation*

$\forall$ *o: ObjectDef; c: Classifier; C:* $\mathbb{P}$ *Classifier; M:* $\mathbb{P}$ *Operation*
  $\mid$ *coupledClasses c = C* $\wedge$ *M* = $\cup$ { *c': C* $\bullet$ (*allOperations* (*o*, *c'*)) }
  $\bullet$ *allClientOperations* (*o*, *c*) = *M*