Specifying Class Hierarchies and MOOSE Metrics in Z

Younès El Amrani
LCS laboratory, faculty of Sciences
University Mohamed V-Agdal
Rabat, Morocco
e-mail: elamrani@fsr.ac.ma

***Abstract*—Metrics put into numbers the quality of software's design and contribute to reinforce an organization's software development competitive advantage. Ultimately, an organization would gain impressive benefits in terms of quality, costs, cycle time and productivity in using metrics to quantify software artifacts. Metrics should be formally defined to ensure every stakeholder understands what is measurable in design, and what is actually measured. The formal specification should be easily formulated. A short and concise formal model is introduced in this article and is used to specify the MOOSE metrics suite. The formal specification proposed provides, for the first time, an unambiguous specification of the LCOM metric.**

Keywords-Metric; Design; Quality; Measure; MOOSE; LCOM; Z; Formal Specification.

I. INTRODUCTION

Even though the main concern of this article is to formally specify the MOOSE metrics suite [1], one should put one's mind to understand object-oriented terminology at first place. This article reviews the basic concepts in object-oriented terminology in the formal specification language Z [2]. Z provides not only a means for formulating concise specifications, but also an integrated framework for conducting proofs. One of the advantages of using pure Z is that one is unencumbered by many of the complications evolved in syntax-extensions introduced to reflect object-oriented concepts.

Section 2 is devoted to related works. Section 3 settles notation for expressing object-oriented concepts and reviews those features of object-orientation that will emerge again in Section 4 later on. Many aspects of object-orientation, of which there is abundance, are not covered. Only those that are needed to specify the MOOSE metrics suite [1] are covered. For fuller coverage, the reader is referred to the standard reviews published on the subject, some of which are mentioned in the references, such as [3]. Our main purpose is to set landmarks that will help readers to navigate through the concepts behind all object-oriented design metrics. Section 4 is intended to specify formally the MOOSE metrics suite [1] using Z. The remaining Section 5 is used to conclude and to explore future works that extend this research.

II. RELATED WORKS

It is vitally important to precisely specify the metrics used in software engineering to gain confidence in obtained measurements. Such precision is particularly important since the object-oriented paradigm abounds in terms and concepts. Introducing formalisms into the paradigm is important to the establishment of a sound theoretical foundation for the measurements in software engineering. The reader is referred to existing surveys, such as [3], in the combination of the object-oriented paradigm and formal specification. Three combinations are possible [3]: the first incarnates in a full transformation into an object-oriented language, the second proposes extension to the syntax of the formal language to cope with object-oriented concepts and necessitates to set up a transformational semantic, and finally, the third proposes to specify the system in an object-oriented fashion to keep available the proof system. The model proposed in the Section 3 belongs to the third approach.

Most of the third approach's specifications fall into two basic styles, depending on whether the properties are modeled as functions from identities to property values or modeled by a value in the object state. Hall's style [4-5] falls in the latter approach, whereas France's style [6-7] falls in the former approach. Both styles specify functional properties, called methods or function-members in object-oriented jargon, using schema operation. This common feature in both styles has tremendous consequences on the expressiveness of the specifications. The first limitation is that "there is no way of stating that a subclass must have all the operations of its superclass" [4] and a second limitation, in both styles, is that: "if the methods of different subclasses are in fact different in any way at all, it is not possible to give them the same name in Z" [4]. In the next section, the model presented circumvents these two limitations and empowers the object-oriented paradigm with an expressive formal model. The MOOSE metrics suite [1] is formally specified to illustrate the usefulness of the model. The MOOSE metrics suite [1] is also formally specified using the formal language Z in [8]; the main difference between our specification and [8] is that they used a formal specification of the UML [9] metamodel to express the same set of metrics, whereas in this article a smaller and more concise model is achieving more by adding several object-oriented consistency rules in the inheritance tree specification. However, object-oriented

 ISBN: 978-1-61208-230-1

consistency rules are not the main target of this article; the reader is referred to [10] for a devoted article to UML [9] consistency rules using Z and Hall's style. The scientific contribution of this work is to formally define the object-oriented concepts that could not be defined in other models [14], notably the concept of virtual methods. The formal definition of the MOOSE metrics is provided to illustrate how the problem of a formal definition of some metrics, like Lack of Cohesion Metric (LCOM), encountered when using other models [14] is circumvented then overcome in the model presented in this article.

III. FORMAL SPECIFICATION OF CLASS HIERARCHIES IN Z

At the heart of formal specification in Z is the ability to introduce new datatypes and to define functions and operations that manipulate their values. Datatypes can be introduced as given sets. The model proposed, uses five given sets: ID is the set of all identifiers, SIGNATURE is used for method's signature. NAME is the set of all names, including methods' names and variables' names. EXPRESSION defines the set of all expressions found in methods' bodies. Finally, the given set TYPE is the set of all variable types in the specification.

[ID, TYPE, SIGNATURE, EXPRESSION, NAME]

Sets can also be defined using Z enumerated sets. Only one enumerated set is used in this model. It is used to specify the concept of visibility for properties (methods and attributes)

Visibility ::= public | private | protected | package

Identifiers' sets for the main set are specified as subsets of ID: the set of all identifiers introduces earlier.

ClassID, ObjectID, AttributeID, MethodID, PropertyID: ℙ ID

⟨AttributeID, MethodID⟩ partition PropertyID

The method and attributes are modeled as Cartesian products. This specification allows different methods to share the same name (this is commonly called operator's overloading)

Method == MethodID × Visibility × NAME × SIGNATURE
Attribute == AttributeID × Visibility × NAME × TYPE

Variable and attribute are the same in the model: they define two syntaxic equivalences.

Variable == Attribute
VariableID == AttributeID

The method's body is itself modeled, the state variables is the set of all the variables used in the method's body. Whereas methods is the set of all the methods called in a given implementation. A set of expressions is defined. To specify a sequential execution, the power set can be replaced by seq EXPRESSION. Finally, the complexity of the method is provided as an instance variable.

MethodBody
variables: ℙ AttributeID
calls: ℙ MethodID
expressions: ℙ EXPRESSION
complexity: ℕ

A small set is defined as a return value for get functions. It can easily be replaced by a Boolean set.

YesNo ::= Yes | No

We use a forward declaration for a method that checks whether a method is abstract in a class ancestry. The method's complete definition will be defined later on.

isMethodAbstractInParentClass: MethodID × ClassID ⇸ YesNo

The method getMethodID is used to obtain the method's ID.

getMethodID: Method ⟶ MethodID

∀ method: Method; methodid: MethodID; visibility: Visibility; name: NAME;
signature: SIGNATURE
• method = (methodid, visibility, name, signature)
∧ getMethodID method = methodid

Now we can define a class. The defined attributes are separated from the inherited attributes (iattributes) as well as the defined methods are separated from the inherited methods (imethods).

Class
self: ClassID
parents: ℙ *ClassID*
children: ℙ *ClassID*
attributes: ℙ *Attribute*
methods: ℙ *Method*
iattributes: ℙ *Attribute*
imethods: ℙ *Method*
isAbstract: YesNo
implementation: ℙ (*MethodID* × *MethodBody*)

∃ *m: methods* | *getMethodID m* ∉ dom *implementation* • *isAbstract = Yes*
∃ *m: imethods*
| *isMethodAbstractInParentClass* ((*getMethodID m*), *self*) = *Yes*
∧ *getMethodID m* ∉ dom *implementation* • *isAbstract = Yes*

 ISBN: 978-1-61208-230-1

The first predicate states that if a defined method (not inherited) has no implementation then the class is abstract: the condition is sufficient. The second predicate states that if a method is abstract in class' ancestry and has not been attributed an implementation, then the class is abstract.

Inheritance is specified with a relation named *inheritFrom*. When a class C_1 inherits from a class C_2, then the pair (C_1,C_2) belongs to *inheritsFrom*. This relation is sematically equivalent to the relation *subSuper* used in Hall's style.

$$inheritsFrom: Class \leftrightarrow Class$$

$$inheritsFrom = \{ C_1: Class; C_2: Class \mid C_1 \in getClassFromID (C_2.parents) \bullet (C_1, C_2) \}$$

Now, the inheritance tree can be formally specified.

InheritanceTree

$$children: Class \rightarrow \mathbb{P}\ Class$$
$$parents: Class \rightarrow \mathbb{P}\ Class$$
$$offspring: Class \rightarrow \mathbb{P}\ Class$$
$$ancestry: Class \rightarrow \mathbb{P}\ Class$$

$$\forall C: Class \bullet children\ C = inheritsFrom (\{C\}) \wedge C \notin children\ C \wedge children\ C = getClassFromID (C.children)$$
$$\forall C: Class \bullet parents\ C = inheritsFrom^{\sim} (\{C\}) \wedge C \notin parents\ C \wedge parents\ C = getClassFromID (C.parents)$$
$$\forall C: Class \bullet offspring\ C = inheritsFrom^{+} (\{C\}) \wedge C \notin offspring\ C$$
$$\forall C: Class \bullet ancestry\ C = (inheritsFrom^{\sim})^{+} (\{C\}) \wedge C \notin ancestry\ C$$

The first and the second predicate use the *inheritsFrom* relation to specify the children and the parents of a class. The third and the fourth predicate define respectively the offspring as the transitive closure of the relation *inheritsFrom* whereas ancestry is the transitive closure of the inverse relation.

Two utility functions named *getAncestryOf* and *getOffspringOf* are formally introduced now. Both functions use relation *inheritsFrom*. This two functions are introduced now because both are used in the next section introducing the formal definition of the MOOSE metrics suite [1].

The first utility function *getAncestryOf* returns the ancestry of the class provided as input.

The method *isMethodAbstractInParentClass*, previously declared, can now be defined (Z does not allow using a function before declaring it) The definition is provided in the second predicate following the first predicate which defines the function *getAncestryOf*.

$$getAncestryOf: Class \rightarrow \mathbb{P}\ Class$$

$$\forall C: Class \bullet getAncestryOf\ C = inheritsFrom^{+} (\{C\})$$

$$isMethodAbstractInParentClass = \{ mid: MethodID; Cid: ClassID; C: Class; ancestry: \mathbb{P}\ Class \bullet \textbf{if } C = getClassFromID\ Cid \wedge ancestry = getAncestryOf\ C \wedge mid \notin \cup \{ C_1: ancestry \bullet (dom\ C_1.implementation) \} \textbf{ then } (mid, Cid) \mapsto Yes \textbf{ else } (mid, Cid) \mapsto No \}$$

The second utility function *getOffspringOf* returns the offspring of the class provided as input.

$$getOffspringOf: Class \rightarrow \mathbb{P}\ Class$$

$$\forall C: Class \bullet getOffspringOf\ C = (inheritsFrom^{\sim})^{+} (\{C\})$$

The formal specification of the MOOSE metrics suite [1] is now illustrated in Section 4.

IV. FORMAL SPECIFICATION OF THE MOOSE METRICS SUITE

The MOOSE Metrics Suite defines a set of six metrics: NOC is the total number of children in a class, DIT measures the depth of the inheritance tree, LCOM measures the lack of cohesion in the set of methods in a class, RFC measures the response for a class, WMC is the weighted methods per class it measures the complexity of the set of methods of the class, finally CBO measures the coupling between object. In the subsequent subsection, we define formally and precisely this set of metrics.

A. The NOC metric

The NOC metric is defined informally as the number of children of a given class. Its formal definition is straightforward in the model introduced in section 3.

$$NOC: Class \rightarrow \mathbb{N}$$

$$\forall C: Class \bullet NOC\ C = \#\ C.children$$

B. The DIT metric

The DIT metric is defined informally as the longest path from the input class to the inheritance tree root. Firstly the formal specification of the set of all paths leading to the root is provided, secondly the maximum length is specified and that is DIT.

A function *isRoot* is used to check if a class is a root. A class is a root when is has no parent.

$$isRoot: Class \rightarrow YesNo$$

$$\forall C: Class \bullet \textbf{if } C.parents = \varnothing \textbf{ then } isRoot\ C = Yes \textbf{ else } isRoot\ C = No$$

 ISBN: 978-1-61208-230-1

The function *getClassFromID* returns the class associated to the input *ClassID*

$$getClassFromID: ClassID \rightarrowtail\!\!\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\!\!\rightarrow Class$$

The function *allPathLengthToRoot* returns the set of all paths to root.

$$allPathLengthToRoot: Class \rightarrow \mathbb{P}\,\mathbb{N}$$

$$\forall\, C: Class \bullet allPathLengthToRoot\ C = \{\ path: \text{seq}\ Class;\ i: \mathbb{N} \mid i \in 1\ ..\ \#\,path \wedge path\ 1 = C \wedge path\ i \in getClassFromID\ (\ (path\ (i - 1)).parents\ ) \wedge isRoot\ (last\ path) = Yes \bullet \#\,path\ \}$$

The function *maxi* returns the maximum of a set of Integers.

$$maxi: \mathbb{P}\,\mathbb{N} \rightarrow \mathbb{N}$$

$$\forall\, I: \mathbb{P}\,\mathbb{N} \bullet \forall\, n: I \bullet \exists_1\, m: I \mid m > n \bullet maxi\ I = m$$

The DIT metric is now straightforward to define formally: it is the longest path to root.

$$DIT: Class \rightarrow \mathbb{N}$$

$$\forall\, C: Class \bullet DIT\ C = maxi\ (allPathLengthToRoot\ C)$$

C. The LCOM metric

The LCOM metric comes also easily, the two sets P and Q defined in [1] are formally specified as follow:

$$P: Class \rightarrow\!\!\!\!\!\!\!\!\rightarrow\, \mathbb{P}\ (Method \times Method)$$

$$\forall\, C: Class \bullet P\ C = \{\ m_1: C.methods;\ m_2: C.methods \mid (C.implementation\ (getMethodID\ m_1)).variables \cap (C.implementation\ (getMethodID\ m_2)).variables = \varnothing \bullet (m_1, m_2)\ \}$$

P is the set of all methods couples that do not use any variable (attribute) in common.

$$Q: Class \rightarrow\!\!\!\!\!\!\!\!\rightarrow\, \mathbb{P}\ (Method \times Method)$$

$$\forall\, C: Class \bullet Q\ C = \{\ m_1: C.methods;\ m_2: C.methods \mid (C.implementation\ (getMethodID\ m_1)).variables \cap (C.implementation\ (getMethodID\ m_2)).variables \neq \varnothing \bullet (m_1, m_2)\ \}$$

Q is the set of methods couples which implementations have some attributes in common.

LCOM is equal to zero if there a more couples in Q than in C, otherwise it is equal to the difference between the two.

$$LCOM: Class \rightarrow\!\!\!\!\!\!\!\!\rightarrow\, \mathbb{N}$$

$$\forall\, C: Class \bullet \textbf{if}\ \#\ (Q\ C) > \#\ (P\ C)\ \textbf{then}\ LCOM\ C = 0\ \textbf{else}\ LCOM\ C = \#\ (P\ C) - \#\ (Q\ C)$$

D. The RFC metric

The RFC metric computes how many different calls can occur as a response to a message received by a class. Of course defined methods and inherited methods are counted and added to the number of different calls that occur in implementations.

$$RFC: Class \rightarrow \mathbb{N}$$

$$\forall\, C: Class \bullet RFC\ C = \#\ C.methods + \#\ C.imethods + \#\ (\cup\ \{\ mob: MethodBody \mid mob \in C.implementation\ (\ getMethodID\ (\ C.methods\ )\ ) \bullet mob.methods\ \}) + \#\ (\cup\ \{\ imob: MethodBody \mid imob \in C.implementation\ (\ getMethodID\ (\ C.imethods\ )\ ) \bullet imob.methods\ \})$$

E. The WMC metric

The WMC metric computes the sum of methods complexities. The method complexity is a state variable of the method's body. A function that sums all the complexities of a set of method's bodies is defined and is used to compute the complexity of a class.

$$sumComplexity: \mathbb{P}\ MethodBody \rightarrow \mathbb{R}$$

$$\forall\, B: \mathbb{P}\ MethodBody \bullet \textbf{if}\ B = \varnothing\ \textbf{then}\ sumComplexity\ B = 0\ \textbf{else}\ \exists\, mb: B \bullet sumComplexity\ B = mb.complexity + sumComplexity\ (B \setminus \{mb\})$$

$$WMC: Class \rightarrow \mathbb{R}$$

$$\forall\, C: Class \bullet \exists\, B: \mathbb{P}\ MethodBody \mid B = C.implementation\ (\ getMethodID\ (\ C.methods\ )\ ) \bullet \textbf{if}\ B = \varnothing\ \textbf{then}\ WMC\ C = 0\ \textbf{else}\ \exists\, mob: B \bullet WMC\ C = mob.complexity + sumComplexity\ (B \setminus \{mob\})$$

 ISBN: 978-1-61208-230-1

F. The CBO metric

The remaining metric from the MOOSE metrics suite [1] is the CBO metric. This metric computes the coupling of a class with all the other classes of a provided design.

First, the function useMethods is defined. It has the value Yes if at least a method of one class uses one methods of the other class:

$$useMethods: Class \times Class \rightarrow\!\!\!\!\!\!\bullet\ YesNo$$

$$\forall C_1: Class; C_2: Class \bullet \textbf{if } \mathrm{dom}\ C_1.implementation \cap (getMethodID\ (C_2.methods)) \neq \emptyset \textbf{ then } useMethods\ (C_1, C_2) = Yes \textbf{ else } useMethods\ (C_1, C_2) = No$$

Second, the function useVariables is defined. It has the value Yes if at least a method of one class uses some attributes of the other class:

$$getAttributeID: Attribute \rightarrow AttributeID$$

$$\forall attribute: Attribute; attributeid: AttributeID; visibility: Visibility; name: NAME; type: TYPE \bullet attribute = (attributeid, visibility, name, type) \land getAttributeID\ attribute = attributeid$$

$$useVariables: Class \times Class \rightarrow\!\!\!\!\!\!\bullet\ YesNo$$

$$\forall C_1: Class; C_2: Class \bullet \textbf{if } \cup \{\ mob: MethodBody \mid mob \in C_1.implementation\ (getMethodID\ (C_1.methods)) \bullet mob.variables\ \} \cap (getAttributeID\ (C_2.attributes)) \neq \emptyset \textbf{ then } useVariables\ (C_1, C_2) = Yes \textbf{ else } useVariables\ (C_1, C_2) = No$$

Then, the coupling between two classes is defined:

$$CBO_1: Class \times Class \rightarrow \{0, 1\}$$

$$\forall C_1: Class; C_2: Class \bullet \textbf{if } useVariables\ (C_1, C_2) = Yes \lor useVariables\ (C_2, C_1) = Yes \lor useMethods\ (C_1, C_2) = Yes \lor useMethods\ (C_2, C_1) = Yes \textbf{ then } CBO_1\ (C_1, C_2) = 1 \textbf{ else } CBO_1\ (C_1, C_2) = 0$$

An object oriented design is formally specified as a set of classes.

$$Design == \mathbb{P}\ Class$$

And the coupling metric for a class is the sum of all coupling with other classes except the class itself.

$$CBO: Class \times Design \rightarrow\!\!\!\!\!\!\bullet\ \mathbb{N}$$

$$\forall C: Class; design: Design \bullet \textbf{if } design \setminus \{C\} = \emptyset \lor C \notin design \textbf{ then } CBO\ (C, design) = 0 \textbf{ else } \exists C_1: design \setminus \{C\} \bullet CBO\ (C, design) = CBO_1\ (C, C_1) + CBO\ (C, (design \setminus \{C_1\}))$$

All presented specifications have been thoroughly checked using the Z/EVES [12] system.

V. CONCLUSION AND FUTURE WORK

This article provided a formal specification for object-oriented concepts and illustrated the power of the proposed specification by providing a complete and formal definition of the MOOSE metrics suite [7]. A formal definition of the MOOD metrics suite [11] and others metrics can be specified with the model presented in Section 3. Additional object-oriented consistency rules can be specified by adding predicates in the inheritance tree. Concepts like the overriding in object-oriented paradigm can easily be specified with this framework. There are many object-oriented concepts that could be clarified and put in a clear mathematical predicate along the road. All the specifications presented in this article have been thoroughly tested using the Z/EVES [12] system. Because of its importance to the subsequent development of software engineering, the proposed formal specification of MOOSE metrics should be extended, in future work, to the set of metrics reviewed in [13].

[1] Chidamber S.R. and Kemerer, C.F.: A metric suite for Object Oriented Design. J. Trans. on Soft. Eng. vol. 20. IEEE Press, New York (1994)

[2] Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall International, Oxford (1998)

[3] Ruiz-Delgado, A., Pitt, D., Smythe, C.: A Review of Object-oriented Approaches in Formal Methods. J. Comp. vol. 38, pp. 777-784 (1995)

[4] Hall, J.A.: Specifying and Interpreting Class Hierarchies in Z. In: Bowen J.P., Hall J.A. (eds.) Cambridge 1994. Z User Workshop, pp. 120-138. Springer, New York (1994)

[5] Hall, J.A.: Using Z as a Specification Calculus for Object-Oriented Systems. In: Bjorner, D., Hoare, C.A.R., Langmaack, H. (eds.) VDM and Z, Third International Symposium on VDM Europe Kiel, 1990. LNCS, vol. 428, pp. 290-318. Springer, Heidelberg (1990)

[6] France, R.B., Bruel, J.M., Larrondo-Petrie, M.M., Shroff, M.: Exploring the Semantics of UML Type Structures with Z. In: Proceedings of the Formal Methods for Open Object-based Distributed Systems. FMOODS, pp. 247-257. Springer, New York (1997)

[7] Shroff, M., France, R.B.: Towards a Formalization of UML Class Structures in Z. In: 21th Computer Software and Application. COMPSAC, pp. 646-651. IEEE Press, New York (1997)

[8] Lamrani, M., El Amrani, Y., Ettouhami, A.: Formal Specification of Software Design Metrics. In: Sixth International Conference on Software Engineering Advances. Barcelona (2011)

[9] The Object Management Group: UML 2.3 superstructure specification. http://www.omg.org/spec/uml/ (09/11/2012)

[10] El Miloudi, K., El Amrani, Y., Ettouhami, A.: An Automated Translation of UML Class Diagrams into a Formal Specification to

 ISBN: 978-1-61208-230-1

Detect UML Inconsistencies. In: Sixth International Conference on Software Engineering Advances. Barcelona (2011)

[11] Abreu, F.B.: The MOOD Metrics Set. In: Workshop on Metrics, ECOOP. Aarhus (1995)

[12] 12. Saaltink, M.: The Z/EVES System. In: Bowen, J.P., Hinchey, M.G., Hill, D. (eds.) Ten International Conference of Z Users Reading 1997. LNCS, vol. 1212, pp. 72-85. Springer, Heidelberg (1990)

[13] Xenos, M., Stavrinoudis, D., Zikouli, K., Christodoulakis, D.: Object Oriented Metrics: A Survey. In: Proceedings of the Federation of European Software Measurement Association. FESMA 2000. Madrid (2000)

[14] Wieringa, R.: A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. In: ACM Computing Surveys. Vol. 30, No. 4, pp. 459-527, New-York (1998) doi=10.1145/299917.299919

 ISBN: 978-1-61208-230-1