Journal of Theoretical and Applied Information Technology
10th September 2013. Vol. 55 No.1

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

SPECIFYING CLASS HIERARCHIES IN Z

YOUNES EL AMRANI
CCM Team, LRI laboratory, Faculty of Sciences, Mohammed V Agdal University, Rabat, MOROCCO
E-mail: elamrani@fsr.ac.ma

ABSTRACT

The main target of this research is to provide a formal meta model for object-oriented systems. It provides a formal definition of the object-oriented concepts along with major consistency rules for object-oriented systems. This research is a contribution to the formalization of object-oriented systems. Other existing models fail to define the notion of virtual function and virtual class. In this article both concepts are specified in the proposed model and used to clarify the OO related concepts. To illustrate the expressiveness of the model a formal specification of the MOOD metric suite is provided using the model. The formal definition of the POF metric is successfully defined, providing one of the first Z formal specification for the POF metric thereof.

Keywords: *Formal Language, Z, Metrics, MOOD, Formal Model, Software Engineering, Measurement*

1. INTRODUCTION

Formal methods for software development are becoming necessary as software became an unavoidable part of everyday life. To handle the large scale software systems complexity, these formal methods should be combined with object orientation that provides a sound development methodology which supports modularity, re-usability, encapsulation and polymorphism for collections of interacting objects whose behaviors are specified by classes. The object-oriented paradigm is itself a source of potential confusion: the rich terminology makes it difficult to formalize the profusion of terms and concepts. This paper presents a formal specification of class hierarchies in Z. The provided framework is used to formally specify the MOOD metrics suite. The rest of this paper is organized as follows: Section 2 discusses related works. Section 3 presents a formal specification of class hierarchies in Z. Section 4 presents the extensional view of the model. In section 5, the inheritance tree is provided along with UML consistency rules formally specified in Z. The section 6 illustrates the formal specification of the MOOD metrics suite using the proposed model. Finally, conclusion and perspectives are drawn in section 7.

2. RELATED WORK

There is a great amount of research on the combination of formal methods and object-orientation [1]. Three main approaches were identified: the first approach is to specify the system in an object-oriented fashion that keeps the proof systems and tool support available. The second approach extends the syntax, using transformational semantics: new constructs should be mapped to the non-object-oriented version. The third approach incarnates in the definition of a new formal language, not necessarily compatible with the original one, syntax and semantics are generally redefined. The third approach requires the fundamental notions of classes, inheritance, polymorphism and encapsulation to be formally defined, and integrated within the semantic model of the formal language. The second approach modifies the syntax and requires a new set of tools to enable the proof system to mathematically manipulate the new features. Finally, the first approach is the best trade-off: it keeps the proof system available and discards away from the formal language any hindering terminology. The adopted approach in this paper emanates from the first approach. The first approach to the object-orientation using Z [2] could be partitioned into two dominant 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 [3-4] belongs to the latter approach, and France's style [5-6] belongs to the former approach. However, in both styles, the operations are specified using schema-operations. The name of the operation is the name given to the schema-operation: in other words the name of the operation is not separated from the specification of the implementation. In both styles it makes impossible to override an operation in Z [2]

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

because a schema-operation cannot be renamed; whereas in this paper, the name of an operation is separated from its implementation, which makes the specification powerful enough to define the overriding object-oriented concept, and allows, for the first time, to specify formally the POF from the MOOD metrics suite [7] thereof. This paper provides also a specification of methods' implementations that enables the formal specification of the CBO metric from the MOOD metrics suite [7] in a concise and complete way. Such a specification could not be achieved before for a simple reason: there is no way to specify, in a given specification, if a schema-operation uses state-variables from another schema, in order to calculate coupling.

This article provides, on one hand, a formalization of object-oriented consistency rules, found in the UML standard [9]. Several rules are provided in the predicates of the formal specification of the inheritance tree. On a second hand, the formal specification of the MOOD metrics [7] is provided to illustrate the expressiveness of the presented model.

3. THE INTENTIONAL VIEW

Z [1] is a formal specification language created by J.-R. Abrial and developed further by the Programming Research Group at Oxford. It is based upon set theory and first order logic. One essential construct of the Z notation is the schema. The notion of schema in Z relates to the concept of a class in object-oriented. A class can have attributes and methods, both share common features. The intentional view of a class is the formal definition of a class and its properties.

We start by specifying class methods and attributes.

3.1 Basic Sets Specification

To set the ground to specify the central concept of a class, this model starts by the specification of the following five given sets.

$$[ID, TYPE, SIGNATURE, EXPRESSION, NAME]$$

These five sets are used to specify the type, the name and identifiers for classes, methods and attributes. The given sets are the most basic construct in Z. It is used when no further specification is needed to specify a concept.

Actually the given set ID is used to define the set of all unique identifiers throughout the specification. ID is used to define objects, methods and attributes identifiers.

$$ClassID, ObjectID, AttributeID, MethodID, PropertyID: \mathbb{P}\, ID$$
$$\langle AttributeID, MethodID \rangle \text{ partition } PropertyID$$

3.2 The Visibility Specification

The notion of visibility is crucial in object-oriented methodology; it is used to determine how attributes and methods are inherited by subclasses. The formal definition of visibility is represented by a Z enumerated set:

$$Visibility ::= public \mid private \mid protected \mid package$$

3.3 Attributes And Methods

Attributes and methods define what is called properties in the object-oriented terminology. Their identifiers define a partition of all properties identifiers. The double equality introduces a syntactic equivalence that alleviates the formalism. Two syntactic equivalences are introduced: Method and Attribute. The use of the Cartesian product or the syntactic equivalent name is strictly equivalent.

$$Method == MethodID \times Visibility \times NAME \times SIGNATURE$$
$$Attribute == AttributeID \times Visibility \times NAME \times TYPE$$

Attributes are called variables and vice-versa. The sets Attribute and Variable are formally syntactically equivalent.

$$Variable == Attribute$$
$$VariableID == AttributeID$$

Now, one can define access functions to access members of Cartesian products as follows:

$$getMethodID: Method \rightarrow MethodID$$
$$\forall\, method: Method; methodid: MethodID;$$
$$visibility: Visibility; name: NAME;$$
$$signature: SIGNATURE$$
$$\bullet\ method = (methodid, visibility, name, signature)$$
$$\wedge\ getMethodID\ method = methodid$$

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

3.4 Method Body Specification

A genuine separation between method's implementation and the other characteristics of a method (visibility, signature, etc) is guaranteed by providing a separate specification of the method's body.

MethodBody

variables: ℙ VariableID
calls: ℙ MethodID
expressions: ℙ EXPRESSION
complexity: ℕ

3.5 Abstract Method Specification

A function implementation in the class will associate a method to its body. This separation allows postponing the definition of a method implementation, providing room for the concept of abstract methods. This separation allows also changing the method body of any method by modifying the mapping of the function implementation. This provides the ground for the OO concept of method overloading.

Complexity is defined here as an integer. An axiomatic declaration of the function called isMethodAbstractInParentClass is provided to define the concept of abstraction for class and methods. Since Z does not support forward declarations, the full specification of these axiomatic definitions comes after the Class specification. However, a preliminary declaration is provided then the full definition follows in an axiomatic definition.

YesNo ::= Yes | No

isMethodAbstractInParentClass: MethodID × ClassID → YesNo

3.6 Class Specification

Now, the concept of a class can be formally specified.

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

The formal specification of the class separates inherited properties, namely inherited methods and inherited attributes, from properties defined in the class itself. The former are named imethods and iattributes, the latter are named methods and attributes.

3.7 Abstract Class Specification

A state variable named isAbstract is used to specify if the class is abstract or not. The type YesNo can be easily replaced by a Boolean type.

The first predicate states that if a defined method has no implementation defined, it implies that the class is abstract.

The second predicate states that if an inherited method has not been associated to an implementation in the class's ancestry, and has no implementation associated in the current class, it implies that the current class is abstract.

We draw the attention of the reader that the concept of abstraction has never been defined in previous definitions of object-oriented constructs in Z, B or AMS.

3.8 Inheritance Specification

A formal definition of the inheritance relationship is formally provided by the relation inheritsFrom, it can be compared the relation subSuper defined in [3] and used to define

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

inheritance.

$$\begin{array}{|l} \text{getIDFromClass: Class} \rightarrowtail \text{ClassID} \\ \hline \forall\ C: \text{Class} \bullet \text{getIDFromClass}\ C = C.\text{self} \\ \text{getClassFromID: ClassID} \rightarrowtail \text{Class} \end{array}$$

The relation inheritsFrom formally specifies the inheritance relationship between two classes.

$$\begin{array}{|l} \text{inheritsFrom: Class} \leftrightarrow \text{Class} \\ \hline \text{inheritsFrom} \\ \quad = \{\ C_1\text{: Class; } C_2\text{: Class} \mid C_1 \in \\ \quad \text{getClassFromID}\ (C_2.\text{parents}) \bullet (C_1, C_2)\ \} \end{array}$$

The function getAncestry is obtained with the transitive closure of inheritsFrom, it is used, in our model, to formally specify the function that checks whether a method is abstract or not in a class.

$$\begin{array}{|l} \text{getAncestryOf: Class} \rightarrow \mathbb{P}\ \text{Class} \\ \hline \forall\ C: \text{Class} \bullet \text{getAncestryOf}\ C = \\ \quad \text{inheritsFrom}^+\ (\{C\}) \\ \text{isMethodAbstractInParentClass} \\ \quad = \{\ \text{mid: MethodID; Cid: ClassID;} \\ \quad C\text{: Class; ancestry: } \mathbb{P}\ \text{Class} \\ \qquad \bullet\ \textbf{if}\ C = \text{getClassFromID Cid} \\ \qquad\quad \wedge\ \text{ancestry} = \text{getAncestryOf}\ C \\ \qquad\quad \wedge\ \text{mid} \notin \cup \{\ C_1\text{: ancestry} \bullet \\ (\text{dom}\ C_1.\text{implementation})\ \} \\ \qquad\quad \textbf{then}\ (\text{mid, Cid}) \mapsto \text{Yes} \\ \qquad\quad \textbf{else}\ (\text{mid, Cid}) \mapsto \text{No}\ \} \end{array}$$

The getOffspringOf function is obtained by using the transitive closure of the relation inverse inheritsFrom.

$$\begin{array}{|l} \text{getOffspringOf: Class} \rightarrow \mathbb{P}\ \text{Class} \\ \hline \forall\ C: \text{Class} \bullet \text{getOffspringOf}\ C = \\ \quad (\text{inheritsFrom}^{\sim})^+\ (\{C\}) \end{array}$$

4. THE EXTENSIONAL VIEW

The extensional view of a class relates a class identifier to the set of all existing objects that are instances of that class. Each object is in turn characterized by an identifier. The function instances is performing this association: mapping a class identifier to the set of its instances.

The given set OBJECT is the set of all objects in the system.

$$[OBJECT]$$

The relation instances, relates a Class to the set of its instances.

$$\begin{array}{|l} \text{instances: ClassID} \leftrightarrow \text{OBJECT} \end{array}$$

Now, we can define the concept of disjoint classes. The domain of instances contains all the identifiers of classes. The objects of the system define the range of the relation instances.

The concept of disjoint classes comes naturally as a relation that states the following: two classes are disjoint if the sets of their instances are disjoint.

$$\begin{array}{|l} \text{disjointClass: Class} \leftrightarrow \text{Class} \\ \hline \text{disjointClass} \\ \quad = \{\ C_1\text{: Class; } C_2\text{: Class} \\ \qquad \mid (\text{instances}\ (\{C_1.\text{self}\})) \cap \\ \qquad\qquad (\text{instances}\ (\{C_2.\text{self}\})) = \varnothing \\ \qquad \bullet\ (C_1, C_2)\ \} \end{array}$$

The getAttributeVisibility method returns the visibility of the attribute in parameter.

$$\begin{array}{|l} \text{getAttributeVisibility: Attribute} \rightarrow \text{Visibility} \\ \hline \forall\ \text{attribute: Attribute; attributeid: AttributeID;} \\ \text{visibility: Visibility;} \\ \quad \text{name: NAME; type: TYPE} \\ \quad \bullet\ \text{attribute} = (\text{attributeid, visibility, name, type}) \\ \qquad \wedge\ \text{getAttributeVisibility attribute} = \text{visibility} \end{array}$$

The getMethodVisibility function returns the visibility of the method provided as parameter to the function.

$$\begin{array}{|l} \text{getMethodVisibility: Method} \rightarrow \text{Visibility} \\ \hline \forall\ \text{method: Method; methodid: MethodID;} \\ \text{visibility: Visibility; name: NAME;} \\ \quad \text{signature: SIGNATURE} \\ \quad \bullet\ \text{method} = (\text{methodid, visibility, name, signature}) \\ \qquad \wedge\ \text{getMethodVisibility method} = \text{visibility} \end{array}$$

Journal of Theoretical and Applied Information Technology
10th September 2013. Vol. 55 No.1

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

5. THE INHERITANCE TREE

We define now the inheritance tree. This is the most central definition that provides most the properties of the OO properties in the schema predicate. Properties required in the inheritance relationship appear as predicates in the second part of the schema.

InheritanceTree

children: Class → P Class
parents: Class → P Class
offspring: Class → P Class
ancestry: Class → P Class

∀ C: Class • C ∉ children C

∀ C: Class • children C = inheritsFrom ({C})

∀ C: Class •
children C = getClassFromID (C.children)

∀ C: Class • C.children =
getIDFromClass (inheritsFrom ({C}))

∀ C: Class • C ∉ parents C

∀ C: Class • parents C = inheritsFrom ⁻ ({C})

∀ C: Class • parents C =
getClassFromID (C.parents)

∀ C: Class • C.parents =
getIDFromClass (inheritsFrom ⁻ ({C}))

∀ C: Class • offspring C = inheritsFrom ⁺ ({C})

∀ C: Class • C ∉ offspring C

∀ C: Class • ancestry C =
(inheritsFrom ⁻) ⁺ ({C})

∀ C: Class • C ∉ ancestry C

∀ C: Class • C.imethods
= ∪ { C1: (inheritsFrom ⁻) ⁺ ({C});
m: Method | m ∈ C1.methods ∧
getMethodVisibility m ≠ private • {m} }

∀ C: Class • C.iattributes
= ∪ { C1: (inheritsFrom ⁻) ⁺ ({C});
a: Attribute | a ∈ C1.attributes ∧
getAttributeVisibility a ≠ private • {a} }

∀ C: Class
• if { mid: MethodID | mid
∉ ∪ { C1: ((inheritsFrom ⁻) ⁺ ({C})) ∪ {C}
• (dom C1.implementation) } } ≠ ∅
then C.isAbstract = Yes
else C.isAbstract = No

∀ C1: Class; C2: Class
• (C1, C2) ∈ disjointClass
⇒ ((inheritsFrom ⁻) ⁺ ({C1})) ∩
((inheritsFrom ⁻) ⁺ ({C2})) = ∅

The predicates thirteen and fourteen, state that private properties of a given class are not inherited by its offspring:

∀ C: Class • C.imethods
= ∪ { C1: (inheritsFrom ⁻) ⁺ ({C});
m: Method | m ∈ C1.methods ∧
getMethodVisibility m ≠ private • {m} }

∀ C: Class • C.iattributes
= ∪ { C1: (inheritsFrom ⁻) ⁺ ({C});
a: Attribute | a ∈ C1.attributes ∧
getAttributeVisibility a ≠ private • {a} }

The predicate fifteen states the abstraction concept:

∀ C: Class
• if { mid: MethodID | mid
∉ ∪ { C1: ((inheritsFrom ⁻) ⁺ ({C})) ∪ {C}
• (dom C1.implementation) } } ≠ ∅
then C.isAbstract = Yes
else C.isAbstract = No

The sixteen predicate establishes a constraint, easily violated in object-oriented designs, stating that two disjoint classes must not have any offspring in common:

∀ C1: Class; C2: Class
• (C1, C2) ∈ disjointClass
⇒ ((inheritsFrom ⁻) ⁺ ({C1})) ∩
((inheritsFrom ⁻) ⁺ ({C2})) = ∅

The first predicate states that a class cannot be among its own children. The fifth predicate states that a class cannot be among its own parents. The tenth predicate states that a class cannot belong to

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

the set of its offspring. The twelfth predicate states that a class cannot be among its own ancestry.

The model is used in the next section to formalize the MOOD metrics suite [7].

6. THE MOOD METRIC SUITE

The MOOD metrics suite consists of a set of six metrics, all of which are formally defined. An additional metric from the MOOSE metric suite named CBO [8] is formally defined and then used to define the COF metric.

The MOOD metric suite is defined on a set of classes. A formal specification of a set of classes is required. A set of classes is called a package in the OO terminology. If we add all relationships between classes and constraints on classes in the package, we obtain the Design. We need to formally specify a package as a set of classes.

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

6.1 The Coupling Factor

The COF metric [7] is the sum of all class's coupling for each class in a package, divided by all the possible couplings. If a package contains N classes, the maximum of class couplings would be N*(N-1) in other words, each class is coupled to all the other classes in the package.

A class is said to be coupled to another class if it uses one of its methods or if it uses, at least one, of its variables.

The function useMethods indicates if the coupling occurs due to a method use.

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

$$\forall C_1: Class; C_2: Class \bullet \textbf{if}$$
$$\cup \{\ mob: ran\ C_1.implementation \bullet mob.calls\ \}$$
$$\cap (getMethodID$$
$$(C_2.methods \cup C_2.imethods)) \neq \varnothing$$
$$\textbf{then}\ useMethods\ (C_1, C_2) = Yes$$
$$\textbf{else}\ useMethods\ (C_1, C_2) = No$$

Two classes are also coupled if one uses the variables of the other, the formal function useVariables checks whether a class uses the variables of a given class in its own methods' implementation. The function useVariables indicates if the coupling occurs due to a variable usage.

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

$$\forall C_1: Class; C_2: Class \bullet \textbf{if} \cup$$
$$\{\ mob: ran\ C_1.implementation \bullet mob.variables\}$$
$$\cap (getAttributeID$$
$$(C_2.attributes \cup C_2.iattributes)) \neq \varnothing$$
$$\textbf{then}\ useVariables\ (C_1, C_2) = Yes$$
$$\textbf{else}\ useVariables\ (C_1, C_2) = No$$

It is now possible to state formally when two classes are coupled. The following function returns takes two classes as parameters and returns 1 if the two classes are coupled, otherwise it returns 0. It is used to formally define the CBO metric.

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

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

The CBO metric from [8] counts classes to which a class is coupled in a given package:

$$CBO: Class \times Package \rightarrow \mathbb{N}$$

$$\forall C: Class; package: Package$$
$$\bullet\ \textbf{if}\ package \setminus \{C\} = \varnothing \vee C \notin package$$
$$\textbf{then}\ CBO\ (C, package) = 0$$
$$\textbf{else}\ \exists C_1: package \setminus \{C\}$$
$$\bullet\ CBO\ (C, package) = CBO_1\ (C, C_1) +$$
$$CBO\ (C, (package \setminus \{C_1\}))$$

Now, the COF metric [7] can be formally defined, it is the sum of all coupling count in a given package, divided by all the possible couplings.

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

```
COF: Package → ℝ
∀ package: Package
 • if package = ∅
   then COF package = 0
   else ∃ C: package • COF package
  = (CBO (C, (package \ {C})) + COF (package \ {C}))
              / (# package * # package - # package)
```

6.2 The Attribute Hidden Factor

The AHF metric [7] computes the percentage of attributes that are hidden in the package. A function that returns the sum of hidden and visible attributes is first defined, and then used to specify formally the AHF metric.

```
hiddenAndVisibleAttributeCount: Package → ℕ
∀ package: Package
 • if package = ∅
then hiddenAndVisibleAttributeCount package = 0
  else ∃ C: package
   • hiddenAndVisibleAttributeCount package
   = # {  a: C.attributes
      | getAttributeVisibility a = private }
        + # {  a: C.attributes
             | getAttributeVisibility a
               ∈ {public, package, protected} }
        + # C.iattributes
        + hiddenAndVisibleAttributeCount
                              (package \ {C})
```

Now we can formally define the AHF metric.

```
AHF: Package → ℕ
∀ package: Package
 • if package = ∅
  then AHF package = 0
  else ∃ C: package
   • AHF package
    = (# {  a: C.attributes
       | getAttributeVisibility a = private }
      + AHF (package \ {C}))
      div hiddenAndVisibleAttributeCount package
```

6.3 The Method Hidden Factor

The MHF metric [7] counts the percentage of methods that are private in the package. A function that counts the total number of hidden and visible methods is first defined then used in the formal specification of MHF. It is worth noting here, that a visible method, defined in root, would appear in all the properties imethod of the root's offspring, therefore it will be counted several times. The definition could be modified to count it only once, depending on the software engineer's interpretation of the MHF metric.

```
hiddenAndVisibleMethodCount: Package → ℕ
∀ package: Package
 • if package = ∅
then hiddenAndVisibleMethodCount package = 0
  else ∃ C: package
   • hiddenAndVisibleMethodCount package
   = # {  m: C.methods |
       getMethodVisibility m = private }
     + # {  m: C.methods
          | getMethodVisibility m
            ∈ {public, protected, package} }
     + # C.imethods
     + hiddenAndVisibleMethodCount (
                              package \ {C})
```

The MHF metric is now formally defined.

```
MHF: Package → ℕ
∀ package: Package
 • if package = ∅
      then MHF package = 0
  else ∃ C: package
 • MHF package
 = (# {  m: C.methods |
         getMethodVisibility m = private }
   + MHF (package \ {C}))
   div hiddenAndVisibleMethodCount package
```

6.4 The Attribute Inherited Factor

The AIF metric [7] counts the percentage of attributes that are inherited in the package. A function that counts the total number of defined and inherited attributes is first defined then used in the formal specification of AIF.

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

definedAndInheritedAttributeCount: Package → ℕ

∀ package: Package
• if package = ∅
  then
   definedAndInheritedAttributeCount package = 0
  else ∃ C: package
   • definedAndInheritedAttributeCount package
    = # C.attributes + # C.iattributes
     + definedAndInheritedAttributeCount
                    (package \ {C})

Now, we can formally define the AIF metric.

AIF: Package → ℕ

∀ package: Package
 • if package = ∅
  then AIF package = 0
  else ∀ C: package
   • AIF package
    = (# { C_1: getAncestryOf C; a: Attribute
      | a ∈ C_1.attributes
      ∧ getAttributeVisibility a ≠ private • a }
      + AIF (package \ {C}))
   div definedAndInheritedAttributeCount package

6.5 The Method Inherited Factor

The MIF [7] metric counts the percentage of methods that are inherited in the package. A function that counts the total number of defined and inherited methods is first defined then used in the formal specification of MIF.

definedAndInheritedMethodCount: Package → ℕ

∀ package: Package
 • if package = ∅
    then definedAndInheritedMethodCount
                    package = 0
  else ∃ C: package
   • definedAndInheritedMethodCount package
    = # C.methods + # C.imethods
     + definedAndInheritedMethodCount
                    (package \ {C})

Now we can formally define the MIF metric,

MIF: Package → ℕ

∀ package: Package
 • if package = ∅
  then MIF package = 0
  else ∀ C: package
   • MIF package
    = (# { C_1: getAncestryOf C; m: Method
      | m ∈ C_1.methods
      ∧ getMethodVisibility m ≠ private • m }
     + MIF (package \ {C}))
    div definedAndInheritedMethodCount
                    package

6.6 The Polymorphism Factor

The POF [7] metric provides a percentage of the methods that are overridden in the package. A function that returns a method's name is defined then used in the POF formal specification.

getMethodName: Method → NAME

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

Now, the POF metric can be formally defined.

POF: Package → ℝ

∀ package: Package
• if package = ∅
  then POF package = 0
 else ∃ C: package
  • ∀ C_1: getAncestryOf C
  • POF package
  = (# { m: C_1.methods
    | getMethodName m
     ∈ getMethodName (C.methods)
     ∧ getMethodVisibility m ≠ private • m }
      + POF (package \ {C}))
     / (# (getOffspringOf C) * # C.methods)

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

The MOOD metric suite has been defined in order to illustrate the clarity and expressiveness of the model provided in the sections 3 and 4. The proposed model provides a solid ground to specify any metric family with the same expressiveness.

7. CONCLUSION

The main target of this research is to provide a formal meta model for object-oriented systems by formally defining the main object-oriented concepts. The MOOD [7] metrics set is formally specified in section 6 to illustrate the expressiveness of the model. Any other metric suite could have been used instead. The section 6 is not the primary goal of this research, it was provided only to exemplify the expressiveness of the proposed model.

This paper provides also several consistency rules for object-oriented systems. These consistency rules can easily be augmented by providing additional predicates in the inheritance tree provided in section 5. This research is a contribution to the formalization of object-oriented systems. Other models belonging to the first approach discussed in section 2 fail to define the notion of virtual function and virtual class; these models fall short to cover all the OO concepts. This is why we hope this article will help at clarifying the concepts used in the dominant OO methodology of this last decade. All the presented specifications were thoroughly tested using the Z/EVES [10] system.

REFRENCES:

[1] A. Ruiz-Delgado, D. Pitt and C. Smythe, "A Review of Object-oriented Approaches in Formal Methods", *J. Comp*. Vol. 38, 1995, pp. 777-784.

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

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

[4] J.A. Hall, "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*, Springer, (Heidelberg). LNCS, vol. 428, 1990, pp. 290-318.

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

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

[7] F.B. Abreu, "The MOOD Metrics Set. In: Workshop on Metrics", *ECOOP*, Aarhus, 1995.

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

[9] The Object Management Group: UML 2.3 superstructure specification. http://www.uml.org/ (last access, March 01, 2013)

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