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 on 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 discarded 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 discarded 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:
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 = \emptyset 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.

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.

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

ClassSpecElement \equiv ClassDefElement \land \{ ObjectDef | class = ClassElement \}

The \equiv symbol offers a different way to define a schema and the logical operator \land allows the extension.

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

ClassDefNamedElement \rightarrow ClassDefClassifier

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.

\forall c: Classifier \cdot \#(pc \{ c \}) < 1

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

\[
\text{hasNamedElement}: \text{Namespace} \leftrightarrow \text{NamedElement}
\]

\[
\text{hasNamedElement} \mapsto \text{NamedElement} \rightarrow \text{Namespace}
\]

The \( \rightarrow \) symbol represents the partial function and the \( \mapsto \) 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:

\[
\begin{align*}
\text{Element, NamedElement, Namespace, } \ldots & : \text{ReferenceSet} \\
\text{Element} & = \text{instances ClassElement} \cup \text{NamedElement} \\
\text{NamedElement} & = \text{instances ClassNamedElement} \cup \text{Namespace} \cup \text{RedefinableElement} \cup \text{Feature} \\
\text{...} & \\
\text{instances ClassElement} & = \{ \ o: \text{ClassSpecElement} \mid o.\text{class} = \text{ClassElement} \cdot o.\text{ref} \} \\
\text{instances ClassNamedElement} & = \{ \ o: \text{ClassSpecNamedElement} \mid o.\text{class} = \text{ClassNamedElement} \cdot o.\text{ref} \} \\
\text{...} & \\
\forall i : \text{instances ClassElement} & \exists x : \text{ClassSpecElement} \cdot x.\text{ref} = i \\
\forall i : \text{instances ClassNamedElement} & \exists x : \text{ClassSpecNamedElement} \\
\text{x.}\text{ref} & = i \\
\end{align*}
\]

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].

\[
\text{Classifier:: DIT( ) : Integer} \\
= \text{if self.isRoot( ) then 0 } \\
\text{else if PARN( ) = 1 then } \\
1 + \text{self.parents( ) -> iterate( elem: GeneralizableElement; acc: Integer = 0 } \\
\text{ | acc + elem.oclAsType( Class ).DIT( ) } \\
\text{ else } \\
\text{self.parents( ) -> iterate( elem: GeneralizableElement; acc: Integer = 0 } \\
\text{ | acc + elem.oclAsType( Class ).DIT( ) } \\
\text{ endif}
\]

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].
oth types
d to be unique, WMC is
\textit{oclIsTypeOf}: ObjectDef \times \text{ReferenceSet} \rightarrow \text{Boolean}
\forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} = t \cdot \text{oclIsTypeOf} \\
(o, t) = \text{TRUE} \\
\forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} \neq t \cdot \text{oclIsTypeOf} \\
(o, t) = \text{FALSE}

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

\textit{oclIsKindOf}: ObjectDef \times \text{Classifier} \rightarrow \text{Boolean}
\forall o: \text{ObjectDef}; c: \text{Classifier} \mid \text{oclIsKindOf} \\
(o, c) = \text{FALSE}

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

\textit{oclAsType}: ObjectDef \times \text{ReferenceSet} \rightarrow \text{ObjectDef}
\forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} = t \cdot \text{oclAsType} \\
(o, t) = o \\
\forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \neg \text{instances } o.\text{class} = t \cdot \text{oclAsType} \\
(o, t) = \text{NIL} \\
\forall o: \text{ObjectDef}; t: \text{ReferenceSet} \mid \text{instances } o.\text{class} \subseteq t \\
\cdot \exists r: \text{ObjectDef} \mid r.\text{ref} = o.\text{ref} \land \text{instances } r.\text{class} = t \cdot \text{oclAsType} \\
(o, t) = r

With \text{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 \text{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 \text{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 \text{ObjectDef}.

Finally, the third one is when the object’s type is part of the \text{ReferenceSet} given in parameter. In that case, the expression \text{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 \text{ReferenceSet} in parameter.

\textit{B. Formalizing the CK metrics}

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

- \textbf{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.

\textit{WMC}: ObjectDef \times \text{Classifier} \rightarrow \mathbb{N}
\forall o: \text{ObjectDef}; c: \text{Classifier} \mid \text{allOperations } (o, c)
\cdot \ WMC (o, c) = \# S

- \textbf{Number of Children}: counts the number of children classes that inherit directly from the current class.

\textit{NOC}: ObjectDef \times \text{Classifier} \rightarrow \mathbb{N}
\forall o: \text{ObjectDef}; c: \text{Classifier} \mid n = \text{CHIN} (o, c) \cdot \text{NOC} (o, c) = n

- \textbf{Depth of Inheritance Tree}: measures the length of the inheritance chain from the current class to the root.

\textit{DIT}: ObjectDef \times \text{RedefinableElement} \rightarrow \mathbb{N}
\forall o: \text{ObjectDef}; r: \text{RedefinableElement} \mid \text{isRoot} (o, r) = \text{TRUE} \cdot \text{DIT} (o, r) = 0
\forall o: \text{ObjectDef}; r: \text{RedefinableElement}; R: \text{RedefinableElement} \mid R = \text{parents} (o, r)
\cdot \exists S: \text{DIT} (o, r) = n
\cdot \text{S}: \text{P N}
\cdot \text{PARN} (o, r) > 1
\cdot \text{R} = \text{parents} (o, r)
\cdot \text{S} = \{ \text{depth} \mid \forall r': \text{R} \cdot \text{depth} = \text{DIT} (o, r') \}
\cdot \text{n} = \text{max} S \cdot \text{DIT} (o, r) = n

- \textbf{Coupling Between Classes}: the number of coupling with other classes.

\textit{CBO}: ObjectDef \times \text{Classifier} \rightarrow \mathbb{N}
\forall o: \text{ObjectDef}; c: \text{Classifier} \mid \text{coupledClasses} (o, c)
\cdot \text{CBO} (o, c) = \# C

- \textbf{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 \mid \text{RS} \mid where \text{RS} is the response set for the class expressed as:

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

with:

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

\[
\text{LCOM3} = \frac{(m - \text{sum}(mA)/a)/(m-1)}
\]

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

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], EMOOOSE- 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

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.

%Number of directly derived Classes.

CHIN: ObjectDef × RedefinableElement → N

∀ o: ObjectDef; r: RedefinableElement; S: P RedefinableElement
| instances o.class = RedefinableElement
• children (o, r) = { r: RedefinableElement
| instances o.class ⊂ instances ClassRedefinableElement }
%Number of Classes from which the current RedefinableElement
derives directly.

\[ \forall \, o: \text{ObjectDef}; \ r: \text{RedefinableElement}; \ S: \text{P RedefinableElement} \]
\[ S = \text{parents} \,(o, \ r) \cdot \text{PARN} \,(o, \ r) = \# \, S \]

%Indicates whether the RedefinableElement has ascendants or not.

\[ \text{isRoot: ObjectDef} \times \text{RedefinableElement} \rightarrow \text{Boolean} \]
\[ \forall \, o: \text{ObjectDef}; \ r: \text{RedefinableElement} \]
\[ \text{PARN} \,(o, \ r) = 0 \cdot \text{isRoot} \,(o, \ r) = \text{TRUE} \]
\[ \forall \, o: \text{ObjectDef}; \ r: \text{RedefinableElement} \quad \text{PARN} \,(o, \ r) \neq 0 \]
\[ \cdot \, \text{isRoot} \,(o, \ r) = \text{FALSE} \]

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

\[ \text{allFeatures: ObjectDef} \times \text{Classifier} \rightarrow \text{P Feature} \]
\[ \forall \, o: \text{ObjectDef}; \ c: \text{Classifier}; \ r: \text{RedefinableElement} \]
\[ \cdot \, \text{allFeatures} \,(o, \ c) = \cup \{ \text{allFeatures} \,(\text{oclAsType} \,(o, \ c)) \}
\]

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

\[ \text{allAttributes: ObjectDef} \times \text{Classifier} \rightarrow \text{P Property} \]
\[ \forall \, o: \text{ObjectDef}; \ c: \text{Classifier}; \ S: \text{P Property} \]
\[ \cdot \, \text{allAttributes} \,(o, \ c) = \text{S} \]

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

\[ \text{allOperations: ObjectDef} \times \text{Classifier} \rightarrow \text{P Operation} \]
\[ \forall \, o: \text{ObjectDef}; \ c: \text{Classifier}; \ S: \text{P Operation} \]
\[ \cdot \, \text{allOperations} \,(o, \ c) = S \]

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

\[ \text{typesOfAllAccessibleAttributes: Classifier} \rightarrow \text{P Classifier} \]
\[ \forall \, o: \text{ObjectDef}; \ c: \text{Classifier}; \ S: \text{P Property}; \ F: \text{P Feature} \]
\[ \cdot \, \text{typesOfAllAccessibleAttributes} \,(o, \ c) = T \]

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

\[ \text{hasAttribute: Classifier} \times \text{Classifier} \rightarrow \text{Boolean} \]
\[ \forall \, c, \ c': \text{Classifier} \]
\[ \cdot \, \text{hasAttribute} \,(c, \ c') = \text{TRUE} \]
\[ \forall \, c, \ c': \text{Classifier} \]
\[ \cdot \, \text{hasAttribute} \,(c, \ c') = \text{FALSE} \]

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

\[ \text{coupledClasses: Classifier} \rightarrow \text{P Classifier} \]
\[ \forall \, c: \text{Classifier} \]
\[ \cdot \, \text{coupledClasses} \,(c) = S \]

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

\[ \text{allClientOperations: ObjectDef} \times \text{Classifier} \rightarrow \text{P Operation} \]
\[ \forall \, o: \text{ObjectDef}; \ c: \text{Classifier}; \ S: \text{P Operation} \]
\[ \cdot \, \text{allClientOperations} \,(o, \ c) = M \]