

Technical Report ES007/2001 (June 2001)

Using OCL to formalize object oriented metrics definitions

Fernando Brito e Abreu
FCT/UNL and INESC
Portugal
fba@inesc.pt

Abstract

We propose to standardize object-oriented metrics definitions using the Object Constraint Language (OCL), a part of the Unified Modeling Language (UML) standard, and a meta-model of the modeling formalism. OCL allows specifying invariants, preconditions, postconditions and other types of constraints.

To illustrate this approach, we describe the MOOD2 metrics in OCL, based upon the meta-model of our object design modeling formalism – the GOODLY language. The outcome is, we believe, an elegant, precise and straightforward way to define metrics that may help to overcome several current problems. Besides, it is a natural approach since we are using object technology to define metrics on object technology itself.

1. Introduction

1.0 Formalization in object oriented specifications

The lack of formalization has been long felt in the object-oriented software modeling area [Meyer1985, Wand1989]. Although the basic elements of the diagrammatic specifications used in this area are very powerful and obvious, and it is easy to understand how they fit together, several modeling details, such as uniqueness and referential restraints, limitations and other constraints are expressed ambiguously, or even cannot be conveyed at all by those graphical notations. As pointed out in [Cook1994], there is a difference between precision and detail. In object-oriented modeling we often leave out the details but, at the same time, we need the precision. Since accuracy and unambiguity in specification have been, for many years, the aims of the branch of computer science known as “formal methods”, attempts have been made to combine them with object-oriented modeling. These attempts have followed four different roads.

One road was that of extending and adapting an existing formal language with object-oriented constructs like in Object-Z [Duke1991] and VDM++, an extension of VDM [Jones1990]. This approach is not in line with industrial practice trends to use the simple, but powerful, graphical notations in object-oriented analysis and design. In fact, most practitioners are not at ease in using traditional formal specification languages, since they usually require a strong mathematical background.

A second road was that of complementing diagrammatic notations with some existing formal language constructs, like for instance in Syntropy, where a subset of Z [Spivey1992] was combined with OMT [Rumbaugh1991]. Also in this road are the ROOA [Moreira1996] and Metamorphosis [Araújo1998] approaches. This is a compromise solution, joining the benefits of graphical modeling with those of a formal language. However, two drawbacks can be identified here. First, the conceptual gap between the two formalisms. Second, the already mentioned difficulty of using a “traditional” formal language does not fade away. Consequently, modeling practitioners practice continued to be, during the 90’s, a combination of graphical modeling with natural language descriptions to fill-in-the-blanks.

A third road was that proposed in the BON (Business Object Notation) object-oriented method [Waldén1995]. There, a constraint language is used to express design by contract modeling issues, as advocated by Bertrand Meyer [Meyer1995]. At the time of its publication BON was, among the popular analysis and design methods,

perhaps the only one to use a full-fledged assertion mechanism, allowing analysts to specify both the structure of a system and its semantics (constraints, invariants, properties of the expected results) [Meyer1997]. Besides graphical and tabular notations, BON uses a textual one to express assertions. This notation includes some constructs as “**delta a**” to specify that a feature can change an attribute “a”, “**forall**” and “**exists**” to express logic formulae of first-order predicate calculus, and set operators such as “**member_of**”. This notation bridges somehow the semantic gap problem previously mentioned, but still has a drawback – no widespread acceptance. Perhaps that was due to the fact that BON is somehow tied to the Eiffel language world. Besides, that acceptance often comes from standardization and shortly after BON was proposed, the joint initiative that would give birth to UML was already full speed ahead.

The last and more promising road to solve the problem in hand is called OCL - Object Constraint Language [Warmer1999], a part of the UML standard, version 1.1, published by the OMG (Object Management Group) [OMG1997]. This standard includes several types of object-oriented analysis and design model diagrams, with their semantic clarified by the use of OCL. The latter is a language to specify invariants, preconditions, postconditions, guard conditions, and other types of constraints (restrictions) on parts of object-oriented models and, therefore, supports design by contract. OCL is a formal, yet simple notation, to be used jointly with UML diagrams and whose syntax has some similarities to those of object-oriented languages such as Smalltalk, C++ or Eiffel. It is underpinned by mathematical set theory and logic, like in formal languages, but was designed for usability and is easily grasped by anybody familiar with object-oriented modeling concepts in general, and UML notation in particular. This last road brings the best of the previous ones: simplicity and powerfulness of the graphical notations, with preciseness and unambiguity granted by formality, in a very usable and conceptually integrated fashion. Moreover, since it is a part of UML, it has become a *de jure* standard.

2.0 Formalization in object-oriented metrics

Theoretical validation of software metrics definitions has received considerable attention. The first well-known approach to do this validation was an axiomatic one, where nine properties were proposed [Weyuker1988]. Although this approach was criticized by several authors [Cherniavsky1991, Fenton1991, Zuse1993], Henderson-Sellers argues that they do provide an initial framework that can be adapted and refined for object-orientation [Henderson-Sellers1996a, pp.75]. Another approach to metrics definition validation is based on measurement theory and is mainly concerned with the measurement scales and corresponding metrics interpretation and usefulness [Zuse1989, Fenton1991, Zuse1991]. According to it, the scale type of a metric (e.g. ordinal, interval, ratio or absolute) determines the types of statistical tests we can use to analyze the measurement values we get. For instance, parametric tests require interval or ratio scale measures.

Along with the validity problem is the one of the metrics definition formalization itself. This has always been an Achilles’ heel in the metrics area. Without clear and precise definitions it is impossible to build adequate metrics extraction tools, experiments replication is hampered, and results interpretation will be flawed. In the first known book on the subject of object-oriented metrication, for instance, all metrics are defined in natural language [Lorenz1994]. Nevertheless, most authors have used a combination of set theory and simple algebra to express their metrics [Chidamber1994, Abreu1995, Abreu1996a, Henderson-Sellers1996a]. Even when expressed mathematically, the metrics may be ill defined, like for instance some of the Chidamber set [Churcher1995].

This ill definition problem is due, in our opinion, to two reasons:

- i) metrics definitions are usually presented without the corresponding context, that is, without expressing which is the corresponding meta-model where the entities of interest and their interrelationships are expressed;
- ii) metrics definition is done without an underlying formal specification approach that uses the former meta-model as contextual input; this formal specification should specify, among other things, under which conditions the metrics are applicable.

In this technical report we present an approach to de define adequately context and metrics that solves this problem. We use UML and OCL to build that meta-model and then we express the metrics as meta-model operations. The metrics applicability limitations are defined with OCL pre-conditions. The metrics result itself is formally defined with OCL post-conditions.

This report is organized as follows: in section 2 we briefly introduce the OCL types and language syntax; our object design meta-model is then presented using UML and made more precise with OCL invariants. The

MOOD2 metrics set is used as an example, in section 3, to show the feasibility and meaningfulness of the proposed approach. Some related work is mentioned and compared to our approach in section 4. The last section presents some conclusions and identifies further work.

2. Using the Object Constraint Language

1.0 Introduction

OCL is the result of an effort to combine formalism soundness with usability and has its roots in the Syntropy method [Cook1994]. Their creators¹ wanted to produce a «precise, unambiguous language that can be easily read and written by all practitioners of object technology and their customers. This means that the language must be understood by people who are not mathematicians or computer scientists.» [Warmer1999, p.8].

As previously mentioned, OCL allows to express three kinds of constraints: invariants, pre-conditions and post-conditions which are predefined in UML as standard stereotypes: «invariant», «precondition» and «postcondition». Constraints convey a number of benefits, namely improved precision and better design documentation, resulting in better (unambiguous) communication among the parties involved, such as designers, users, programmers, testers or managers.

Invariants are constraints that represent conditions that must be met by all instances of the class, at all times. Their context is, therefore, a class, hereafter represented in the first line, underlined, as in:

```
Sequence  
self.oclIsKindOf( Collection )  
-- sequence inherits from Collection and therefore its instances can be used where a collection is  
allowed (this is a comment)
```

The dot notation is used for attribute access. In the above example the *oclIsKindOf* boolean operation is applied to the attribute *self*. This is a special implicit attribute that allows to reference the context object (the class instance).

The dot notation is also used to navigate in the class diagram through associations. If the role name of an association is identified in the UML model, then it is used in the navigation. Otherwise, the name of the target class is used, in lowercase letters. We will see many examples of these navigations in the following section.

Now let us focus on the assertions whose scope is an operation.

Pre-conditions are constraints that must be true for an operation to be executed. In the design by contract paradigm, they traduce the rights of the object that offers the service or, if you want, the client responsibilities.

Post-conditions are constraints that must be true when the operation ends its execution. They traduce the obligations to be fulfilled by the object that offers the service.

The context of both pre and post-conditions is, therefore, an operation, as in the following extract from the *Sequence* type definition:

```
Sequence::prepend(object: T): Sequence(T)  
post: result->size() = self@pre->size() + 1  
post: result->at(1) = object
```

Operations, like *prepend* above, can have input parameters and must have a return type. The “::” sign is a scope indicator. The “->” sign is used for applying an operation to a collection. In the previous case, the *prepend* operation is defined in the scope of the *Sequence* class. The *result* keyword represents the object returned by the operation, whose type is identified in the operation signature (a generic type *T*, in this case). The *@pre* suffix allows to use the value of the characteristic to which it is applied at the moment where the operation is called, that is, its original value before the operation is applied. We can have several pre and post conditions defined within the same operation.

¹ - From within the IBM Insurance division.

All objects in OCL have a type, derived from *OclAny*, which determines the applicable operations. There is a set of predefined types including basic ones (*Boolean*, *Integer*, *Real*, *Enumeration* and *String*) and collection types (*Collection*, *Set*, *Bag* and *Sequence*). Figure 1 summarizes the OCL types hierarchy.

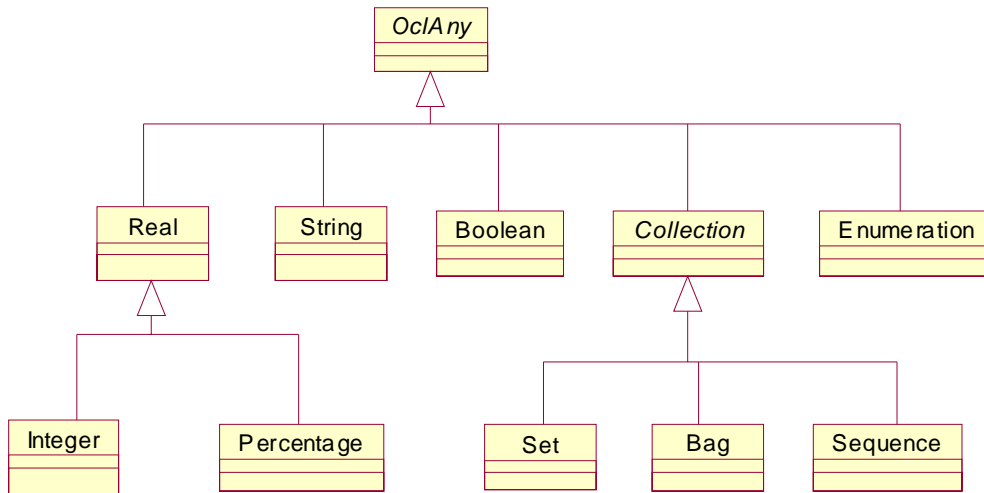


Figure 1 – OCL types

The basic types have a number of operations defined on them, as represented in Table 1.

Type	Operations
Boolean	=, not, and, or, xor, implies, if-then-else
Real	=, +, -, *, /, abs, floor, max, min, <, >, <=, >=
Integer	=, +, -, *, /, abs, div, mod, max, min
String	=, size, toLower, toUpper, concat, substring

Table 1 – Operations defined in basic OCL types

Sets do not allow duplicates and their elements are not ordered, *Bags* are similar but allow duplicates and *Sequences* have an order imposed on their elements and allow duplicates. By definition, the result of navigating through just one association is a *Set*², and through more than one association with multiplicity *many* is a *Bag*. The *Collection* class is an abstract class from which the previous three are derived. This can be expressed in OCL in the following manner:

Collection

Collection.allInstances->select(oclType = Collection)->isEmpty()

-- the *allInstances* operation returns the set of all objects of the named class and of all its subclasses;

-- this operation is defined in the *OclType* meta-class.

OCL types are open to specialization. In the Catalysis approach [D'Souza1998], for instance, the *Set* and *Sequence* type operations were extended. Here, we derived a new type from the OCL *Real* type. The ***Percentage*** type is a constrained *Real* because its instances can only have values in the interval [0, 1]. Since *Percentage* is a value type [Warmer1999, p.22], its instances are values. Therefore, we can write the following class invariant:

Percentage

(self >= 0) and (self <= 1)

-- 0 is 0% and 1 is 100%

² - Unless the association is adorned with the {ordered} tag, in which case we get a *Sequence*.

OCL is a declarative typed language. Expressions in OCL are free of side effects, which means that the state of the objects does not change by the application of an OCL expression. These expressions can range from simple comparisons (e.g. an attribute having an upper limit) to complex navigations in a class diagram through their associations. Since it is a typed language, it is possible to check expressions for validity during modeling³. OCL does not specify what happens when a constraint is broken. This problem is deferred to the implementation since the constraint and exception handling mechanisms are supported differently by available programming languages.

In expressions we can use operations defined in OCL types as well as those belonging to the UML model classes upon which we are writing constraints. However, since OCL is side-effect free, only selectors⁴ are allowed. The most frequently used operations when navigating on the class diagrams are those that manipulate collections. Figure 2 details the ones in OCL types *Collection* and *Set* that we will use often.

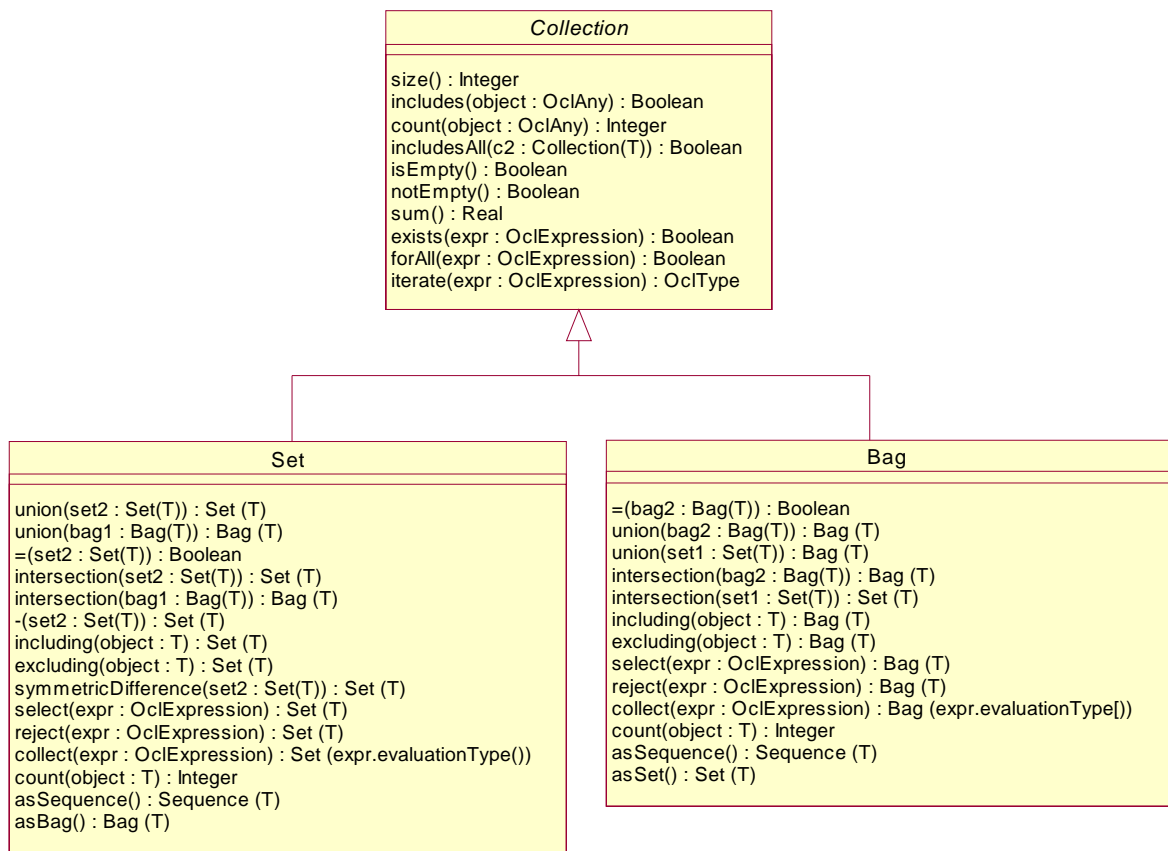


Figure 2 - OCL Collection, Set and Bag types

We will introduce more details of the OCL syntax in the next section where our object-oriented meta-model will be presented.

³ - For this purpose a free OCL parser can be found in <http://www.software.ibm.com/ad/ocl>

⁴ - query operations which return a value but do not change the object state; in UML their *isQuery* boolean label is true.

2.0 GOODLY meta-model

The GOODLY⁵ language allows the textual representation of object-oriented design information such as modules, classes and its inheritance hierarchies and parameterization, attributes, operations and their parameters or message exchanges [Abreu1999]. It also allows the expression of visibility rules as well as the inclusion of trace information of external design parts such as classes and modules, an important feature for large systems. It addresses reusability at the design level at two levels of abstraction: specification and module. There are no pre-defined types in GOODLY. All types from specifications converted into GOODLY are considered, namely the ones defined for the original formalism. As in pure object-oriented languages, such as Smalltalk or Eiffel, the only typing mechanism is the class. All imported types are considered to be classes in GOODLY even if, in hybrid languages, its internal memory (state) and allowed operations on its instances are not formally encapsulated (e.g. atomic types, such as *double*, in C++).

Since it is a design language, GOODLY is not computationally complete: neither algorithmic capabilities, nor control flow structures, are present. We have used this language mainly as a common intermediate formalism allowing the extraction of quantitative data (structural software metrics) [Abreu1998b] and also to obtain class coupling information that we have been using for object-oriented modularity assessment and reengineering [Abreu2000, Abreu2001a]. We generate GOODLY from the underlying design of systems originally described using other formalisms, either upstream (analysis and design models supported in CASE tools), or downstream (object-oriented program source code). A code extract in this language is included in appendix B.

In this report we will use the GOODLY meta-model to put the metrics definitions in context. We will now introduce this meta-model incrementally along with a series of identified OCL constraints.

The structural unit at the highest abstraction level is the **specification** (Figure 3). A specification is an identified package formed by a set of interrelated design parts. A specification is produced by a named person, team or company, and is made available as a whole and not only partially. A specification may “use” other specifications. By this, we mean that in order to provide the services for which they were conceived, the components in a specification (the “using” one) may depend on the collaboration of components in others (the “used” ones). For instance, an executable system (e.g. an application) is usually built upon the parts made available by several specifications.

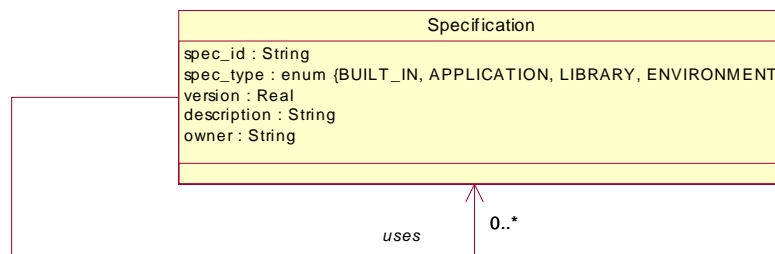


Figure 3 –The Specification type in the GOODLY meta-model

Specification identifiers are strings and should always be unique. This type check and uniqueness constraint can be expressed as follows:

Specification

spec_id.oclIsKindOf(String)

Specification.allInstances->forAll(s1, s2: Specification | s1 <> s2 implies s1.spec_id <> s2.spec_id)

Notice that the “<>” operator is used above in two distinct situations. In the first (*s1<>s2*) we are comparing objects and thus this operator is the one defined in the *OclAny* type, the supertype of all OCL types. In the second (*s1.spec_id <> s2.spec_id*) we are comparing strings. The semantic of string comparison is defined in the OCL *String* type.

⁵ - a Generic Object Oriented Design Language? Yes!

Each specification mentions which others it must use directly, so that the origin of all used symbols is known. By other words, if the specification A uses symbols of specifications B1 and B2, and B1 uses symbols defined in specification C11 and C12, then this “indirect” use in A of symbols defined in C11 and C12 is not enlisted in A.

It does not make sense to explicit that a specification uses itself, since that is implicit. This invariant can be expressed in OCL like this:

Specification
not (uses->includes(self))

There are several specifications types:

- **BUILT_IN** - set of standard types (classes) embedded in a given formalism (language); their interface is supposed to be independent across several platforms (environments) that support the same formalism;
- **ENVIRONMENT** - set of types (classes) that are included in the development environment being used that generally extends the built-in types in a proprietary unconstrained way; an environment specification always uses a given built-in specification;
- **LIBRARY** - set of types (classes) that are packaged together to facilitate a given set of semantically cohesive building blocks (reusable components), not available in the adopted environment; they are often produced by a third party; a library specification always uses a given built-in specification and may optionally be tailored to be used in one or more environments, if it relies on the services they provide;
- **APPLICATION** - set of collaborating classes that perform a specified set of functionalities; an application specification always uses a given built-in specification, is usually tailored to be used in one or more environments, and may also rely upon the services of one or more libraries.

To model this, the *spec_type* attribute is of the enumeration type. Possible values of such attributes are represented in OCL expressions with the # prefix. For instance, the next invariant states that an APPLICATION type specification must have a non-empty **main** specification:

Specification
spec_type = #APPLICATION implies (main_spec->notEmpty())

To say that a library specification always uses a given built-in specification we write:

Specification
spec_type = #LIBRARY implies (uses->exists(spec_type = #BUILT_IN))

A specification is organized as a set of modules (Figure 4). A **module** is a set of classes (types) grouped by a given aggregation criterion. The specification and module abstraction levels correspond, in the UML meta-model, to two nested packaging levels. We have not yet come across systems where these two levels were not enough for system decomposition.

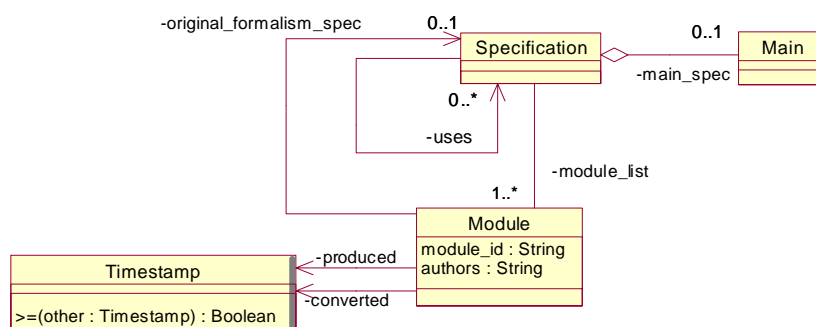


Figure 4 – Modules in the GOODLY meta-model

Within the same specification, module identifiers should always be unique. We can express this in OCL as:

Specification

module_list->forall(m1, m2: Module | m1 <> m2 implies m1.module_id <> m2.module_id)

Either the module belongs to a specification of BUILT_IN type, or it must refer which was the original formalism used in its production. This can be expressed in OCL in the following fashion:

Module

specification.spec_type = #BUILT_IN xor original_formalism_spec->notEmpty()

When referred, the original formalism specification is always of #BUILT_IN type. The same in OCL is:

Module

original_formalism_spec->notEmpty() implies original_formalism_spec.spec_type = #BUILT_IN

A module cannot be converted into GOODLY before being produced. In most situations the original formalism is other than GOODLY itself. Otherwise, the conversion date will be the same as the production one. This can be easily expressed by

Module

*if original_formalism_spec.spec_id = ("GOODLY") then
 converted = (produced)
else
 converted > (produced)
endif*

The utility *Timestamp* class⁶ has, among others, the relational operators defined in it, with the following interface:

Timestamp::=(other: Timestamp): Boolean
Timestamp::<>(other: Timestamp): Boolean
Timestamp::>(other: Timestamp): Boolean
Timestamp::>=(other: Timestamp): Boolean
Timestamp::<(other: Timestamp): Boolean
Timestamp::<=(other: Timestamp): Boolean

The basic component of a module is the **class**. Each class must have a unique identifier within each module:

Module

class_list->forall(c1, c2: Class | c1 <> c2 implies c1.class_id <> c2.class_id)

The GOODLY language supports the specification of both single and multiple inheritance. However, inheritance cannot be circular:

Class

not (self.Ascendants()->includes(self))

Where *Ascendants()* is a function that returns the set of classes from which the current class derives directly or indirectly.

Each **class** has both a set of **attributes** (comprising both instance variables and class variables), that characterize the object or class state⁷ and a set of **operations**⁸ that characterize the object behavior.

⁶ - Utility classes are drawn with a shadow.

⁷ - In GOODLY we do not distinguish (for now) between instance variables and class variables (static attributes).

⁸ - In other formalisms these are often designated by methods or function members.

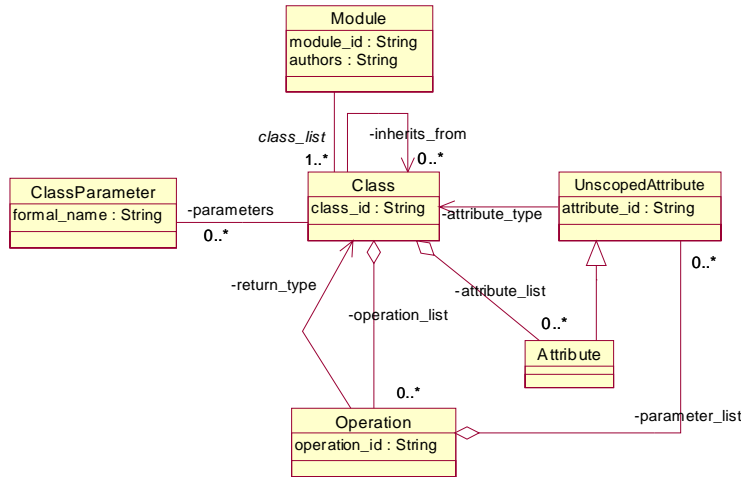


Figure 5 – Classes and its features in the GOODLY meta-model

Each attribute⁹ has an identifier, a type (class) and a scope. Within the same class, attribute identifiers should always be unique:

Class

attribute_list->forall(*a1*, *a2*: Attribute /
a1 <> *a2* implies (*a1*.attribute_id <> *a2*.attribute_id))

Each operation has an interface and a body or implementation. The interface includes its identifier, the formal parameter list and corresponding type(s), the returning type, its scope and traceability information. Within the same class, operations' signature (identifier plus the parameter list) should always be unique:

Class

operation_list->forall(*o1*, *o2*: Operation /
o1 <> *o2* implies (*o1*.operation_id <> *o2*.operation_id or (*o1*.parameter_list <> *o2*.parameter_list)))

The **scope** or range of an attribute or operation can be defined in a very detailed fashion in GOODLY (Figure 6). The scope is characterized by the visibility that components (classes) have on the attribute or operation. Invisibility implies inability to use. The following scope options can be used:

- **PRIVATE** scope – only the class where the attribute or operation is defined can “see” it; this scope is not cumulative with any other;
- **PROTECTED** scope – the class where the attribute or operation is defined, and its descendants, can “see” it;
- **DISCRIMINATED** class scope – the identified class can “see” the attribute or operation; that class must be defined in the same specification, although it can be declared in another module;
- Discriminated class hierarchy scope (**CLASS_TREE**) – the identified class and its descendants can “see” the attribute or operation; that class is supposed to be defined in the same specification, although it can be declared in another module.
- **MODULE** scope – all classes in the module that contains the class where the attribute or operation is defined can “see” it;
- Specification (**SPEC**) scope – all classes in the specification that contains the class where the attribute or operation is defined can “see” it;
- **PUBLIC** scope – all classes can “see” the attribute or operation, regardless of being defined in the same or other specification.

Only the DISCRIMINATED and CLASS_TREE scope types lead to the indication of scope classes. In all other the classes within range are implicit. The explicitation requirement can be stated as an OCL invariant:

⁹ - in other formalisms these are designated by instance variables or data members;

Scope

$(scope_type = \#DISCRIMINATED \text{ or } scope_type = \#CLASS_TREE) \text{ xor } (scoped_class \rightarrow isEmpty())$

The scope of an attribute or operation always includes the own class where it is defined. Therefore it is useless to include it explicitly in the scope clause. Formally we have the following invariant:

Operation

$not(scope_list.scoped_class \rightarrow includes(class))$

Attribute

$not(scope_list.scoped_class \rightarrow includes(class))$

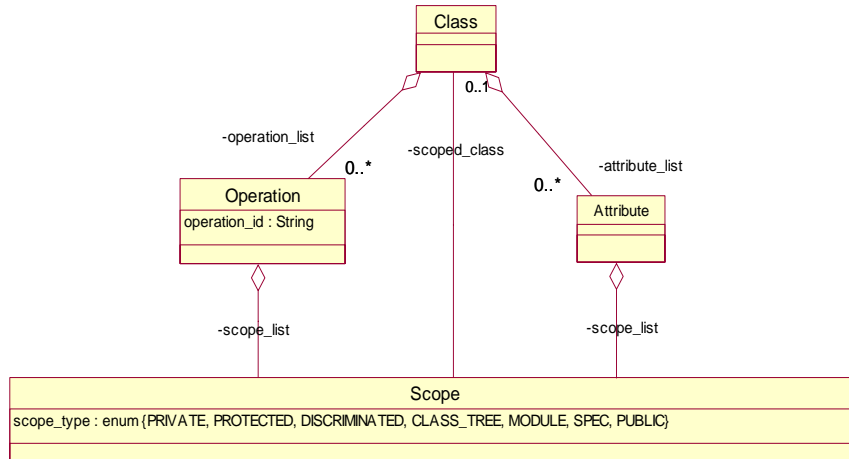


Figure 6 – Scope of attributes and operations in the GOODLY meta-model

Both the main section of a specification and each and every operation have an implementation body. The latter may have local attributes defined on it, may employ attributes from named classes and can issue requests (send messages) to instances of the same or of other classes (Figure 7).

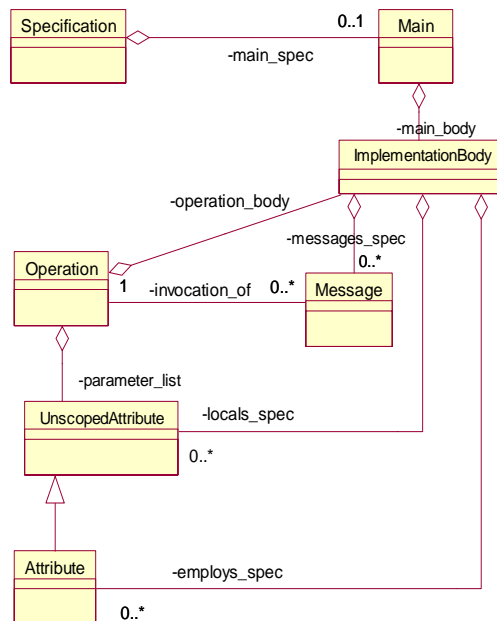


Figure 7 – The operations implementation body in the GOODLY meta-model

During the **trace** process (static binding) that occurs during linking, the classes used are marked if their declaration was, or not, found. The identified ones are those whose specification and module we know:

TracedClass

```

if self.identified then
  the_class.module.specification = origin_spec and (the_class.module = origin_module)
else
  origin_spec->isEmpty() and origin_module->isEmpty() and the_class->isEmpty()
endif

```

The meta-model extract corresponding to the trace information is represented in Figure 8.

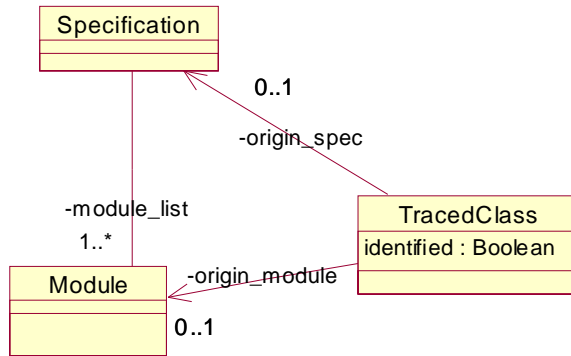


Figure 8 - Trace information in the GOODLY meta-model

Now it is time to wrap-up. The whole GOODLY meta-model is represented in Figure 9.

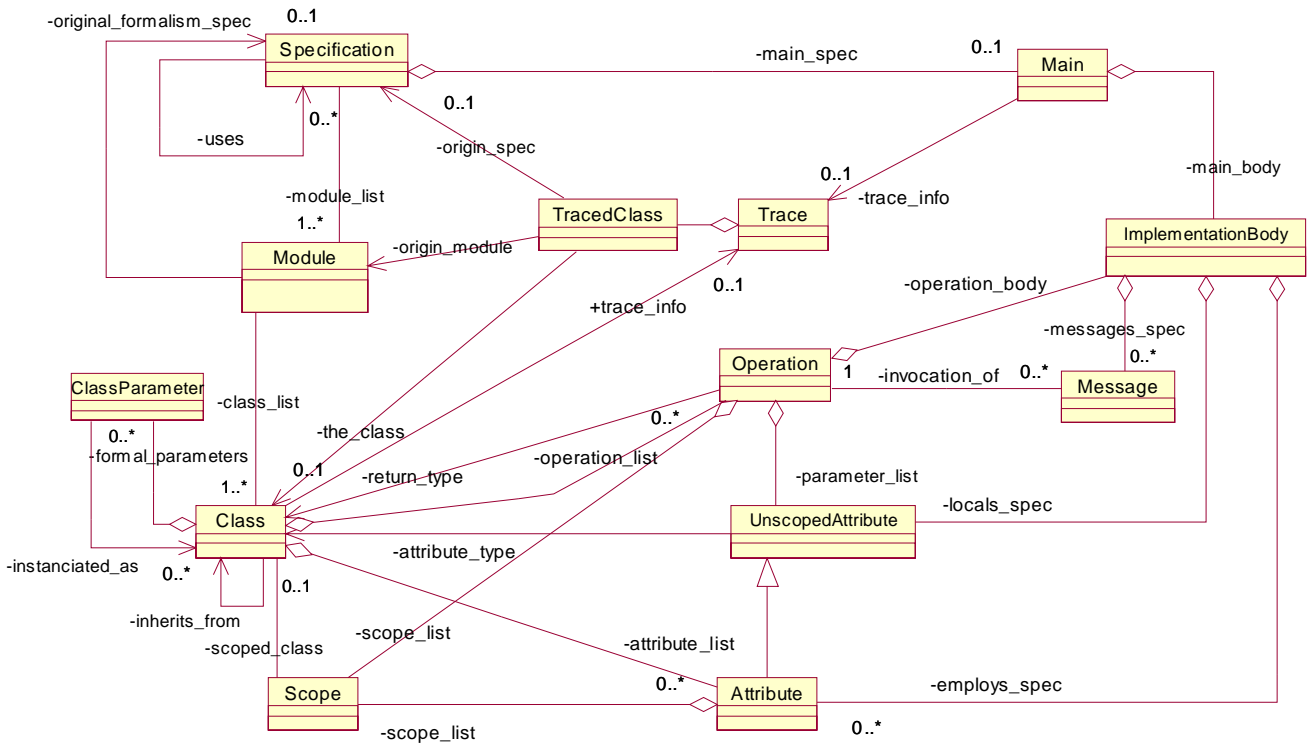


Figure 9 - The full version of the GOODLY meta-model

3. MOOD2 set

1.0 Introduction

The MOOD metrics set (Metrics for Object Oriented Design) was first introduced in [Abreu1994] and its use and validation was presented in several occasions such as in [Abreu1995, Abreu1996a, Abreu1996b, Harrison1998]. From the experience gathered during the corresponding experiments, it became evident that some important aspects of the design were not being measured in the initial set, namely the existence of different types of polymorphism and the amount of reuse. The initial MOOD set only considered metrics calculated within a given specification. However, actual executable systems (applications) are usually composed upon several specifications. This lead to a split in the MOOD2 set between intra-specification metrics, most of which inherited from the original set (Table 2) and inter-specification metrics (Table 3) [Abreu1998a]. Some metrics (inheritance and coupling ones) were split in two to reflect the internal (within the specification) design aspects and the external (among distinct specifications) ones. The new metrics are marked with a star. A few of the original MOOD metrics were renamed for naming consistency.

Acronym	Name	
m		
AIF	Attributes Inheritance Factor	
OIF	Operations Inheritance Factor ¹⁰	
IIF	Internal Inheritance Factor	*
AHF	Attributes Hiding Factor	
OHF	Operations Hiding Factor ¹¹	
AHEF	Attributes Hiding Effectiveness Factor	*
OHEF	Operations Hiding Effectiveness Factor	*
BPF	Behavioral Polymorphism Factor ¹²	
PPF	Parametric Polymorphism Factor	*
CCF	Class Coupling Factor ¹³	
ICF	Internal Coupling Factor	

Table 2 - MOOD2 Intra-Specification Level Metrics

Acronym	Name	
EIF (S)	External Inheritance Factor	*
ECF (S)	External Coupling Factor	*
PRF(S)	Potential Reuse Factor	*
ARF(S)	Actual Reuse Factor	*
REF(S)	Reuse Efficiency Factor	*

Table 3 - MOOD2 Inter-Specification Level Metrics

The MOOD2 metrics retain the main characteristics of the original set. All of them are defined as quotients where the numerator represents the actual value of the design characteristic being measured, while the denominator represents its theoretical maximum value. As a result, they take values in a percentual scale (real numbers in the interval [0,1]).

¹⁰ - originally called MIF - Methods Inheritance Factor

¹¹ - originally called MHF - Methods Hiding Factor

¹² - originally called POF – POLymorphism Factor (by then we did not consider parametric polymorphism)

¹³ - originally called COF – COupling Factor

2.0 Some auxiliary functions

Another improvement in the MOOD2 set was that their definition was made on a compositional way, based upon a set of auxiliary functions, at different levels of abstraction: attribute, operation, class and specification (Table 4 through Table 9). These functions were added as operations to the corresponding entities of the GOODLY meta-model. Their formal description using OCL is included in the appendix A.

Acronym	Name	Type
ACV(c)	Attribute to Class Visibility	Boolean
ASV(s)	Attribute to Specification Visibility	Percentage
AUN(s)	Attribute Use Number	Integer
AVN(s)	Attribute Visibility Number	Integer

Table 4 – Attribute-level functions

Acronym	Name	Type
OCV(c)	Operation to Class Visibility	Boolean
OSV(s)	Operation to Specification Visibility	Percentage
OUN(s)	Operation Use Number	Integer
OVN(s)	Operation Visibility Number	Integer

Table 5 – Operation-level functions

Acronym	Name	Type
IsInternal(s)	Internal class predicate	Boolean
IsRoot	Root class predicate	Boolean
IsLeaf	Leaf class predicate	Boolean

Table 6 – Class-level predicate functions

Acronym	Name	Type
Children()	Set of children classes	Set(Class)
Descendants()	Set of descendant classes	Set(Class)
Parents()	Set of parent classes	Set(Class)
Ascendants()	Set of ascendant classes	Set(Class)
CoupledClasses	Set of coupled classes	Set(Class)
NewOperations()	Set of class's new operations	Set(Operation)
InheritedOperations()	Set of class's inherited operations	Set(Operation)
OverriddenOperations()	Set of class's overridden operations	Set(Operation)
DefinedOperations()	Set of class's defined operations	Set(Operation)
AvailableOperations()	Set of class's available operations	Set(Operation)
NewAttributes()	Set of class's new attributes	Set(Attribute)
InheritedAttributes()	Set of class's inherited attributes	Set(Attribute)
OverriddenAttributes()	Set of class's overridden attributes	Set(Attribute)
DefinedAttributes()	Set of class's defined attributes	Set(Attribute)
AvailableAttributes()	Set of class's available attributes	Set(Attribute)

Table 7 – Class-level set functions

Acronym	Name	Type
CC	Children Count	Integer
DC	Descendants Count	Integer
PC	Parents Count	Integer
AC	Ascendants Count	Integer
ON	Operations New	Integer

OI	Operations Inherited	Integer
OO	Operations Overridden	Integer
OD	Operations Defined	Integer
OA	Operations Available	Integer
AN	Attributes New	Integer
AI	Attributes Inherited	Integer
AO	Attributes Overridden	Integer
AD	Attributes Defined	Integer
AA	Attributes Available	Integer

Table 8 – Class-level counting functions

Acronym	Name	Type
AllClasses	Set of all classes	Set(Class)
BaseClasses(s)	Set of base classes	Set(Class)
SupplierClasses(s)	Set of supplier classes	Set(Class)
RelatedClasses(s)	Set of related classes	Set(Class)

Table 9 – Specification-level set functions

Acronym	Name	Type
TC	Total number of Classes	Integer
TON	Total Operations New	Integer
TOO	Total Operations Overridden	Integer
TOD	Total Operations Defined	Integer
TOI	Total Operations Inherited	Integer
TOA	Total Operations Available	Integer
TAN	Total Attributes New	Integer
TAO	Total Attributes Overridden	Integer
TAD	Total Attributes Defined	Integer
TAI	Total Attributes Inherited	Integer
TAA	Total Attributes Available	Integer
IL(s)	Inheritance Links	Integer
TIL	Total Inheritance Links	Integer
CL(s)	Coupling Links	Integer
TCL	Total Coupling Links	Integer

Table 10 – Specification-level counting functions

3.0 Metrics definition using OCL

The MOOD2 set is divided in two subsets: intra-specification and inter-specification metrics. **Intra-specification metrics** are those that refer to the context specification only and whose definition relies upon information contained solely on it. Therefore, they are parameterless.

Inter-specification metrics are those that whose definition relates to the relationship between the context specification and the one that is passed as an argument.

1.0.0 Intra-Specification Level Metrics

Name	AIF – Attributes Inheritance Factor
Informal definition	Quotient between the number of inherited attributes in all classes of the specification and the number of available attributes (locally defined plus inherited) for all classes of the current specification.

Formal definition	<p><u>Specification::AIF(): Percentage</u></p> <pre>pre: self.TAA() > 0 -- the specification must have some attribute available ... post: result = self.TAI() / self.TAA()</pre>
Comments	AIF() ₀ means that there is no effective attribute inheritance (either there are no inheritance hierarchies or all inherited attributes are redefined).

Name	OIF – Operations Inheritance Factor
Informal definition	Quotient between the number of inherited operations in all classes of the specification and the number of available operations (locally defined plus inherited) for all classes of the current specification.
Formal definition	<p><u>Specification::OIF(): Percentage</u></p> <pre>pre: self.TOA() > 0 -- the specification must have some operation available ... post: result = self.TOI() / self.TOA()</pre>
Comments	This metric was called MIF (Methods Inheritance Factor) in the original MOOD set. OIF() ₀ means that there is no effective operation inheritance (either there are no inheritance hierarchies or all inherited operations are redefined).

Name	IIF – Internal Inheritance Factor
Informal definition	Quotient between the number of inheritance links where both the base and derived classes belong to the current specification and the total number of inheritance links originating in the current specification.
Formal definition	<p><u>Specification::IIF(): Percentage</u></p> <pre>pre: self.TIL() > 0 -- the specification must have some coupling defined on it ... post: result = self.IL(self) / self.TIL()</pre>
Comments	Inheritance links originating in the current specification are those where the derived class belongs to it. The inheritance link is directed from the derived class to the base one.

Name	AHF – Attributes Hiding Factor
Informal definition	Quotient between the sum of the invisibilities of all attributes defined in all classes in the current specification and the total number of attributes defined in the specification
Formal definition	<p><u>Specification::AHF(): Percentage</u></p> <pre>pre: self.TC() > 1 -- the specification must have some attribute defined on it ... pre: self.TAD() > 0 post: result = AllClasses().attribute_list-> iterate(elem:Attribute; acc:Real=0 acc + 1 - elem.ASV(self)) / self.TAD()</pre>
Comments	The invisibility of an attribute is the percentage of the total classes in the specification from which this attributes is not visible and is given by $1 - OSV(self)$, where <i>self</i> is the current specification.

	If all attributes are private the numerator is 0 and, as such AHF() \neq 0. The pre-condition regarding the number of classes is a requirement for calculating the attributes visibility.
--	--

Name	OHF – Operations Hiding Factor
Informal definition	Quotient between the sum of the invisibilities of all operations defined in all classes in the current specification and the total number of operations defined in the specification
Formal definition	<p><u>Specification::OHF(): Percentage</u></p> <pre>pre: self.TC() > 1 pre: self.TOD() > 0 post: result = (AllClasses().operation_list-> iterate(elem: Operation; acc: Real=0 acc + 1 - elem.OSV(self))/ self.TOD()</pre>
Comments	This metric replaces the MHF (Methods Hiding Factor) of the original MOOD set. The invisibility of an operation is the percentage of the total classes in the specification from which this operation is not visible and is given by $1-OSV(self)$, where <i>self</i> is the current specification. If all operations are public the numerator equals the denominator and then OHF() \neq 1. The pre-condition regarding the number of classes is a requirement for calculating the operations visibility.

Name	AHEF – Attributes Hiding Effectiveness Factor
Informal definition	Quotient between the cumulative number of the specification classes that <i>do</i> access the specification attributes and the cumulative number of the specification classes that <i>can</i> access the specification attributes.
Formal definition	<p><u>Specification::AHEF(): Percentage</u></p> <pre>pre: AllClasses().attribute_list-> iterate(elem: Attribute; acc: Integer=0 acc + elem.AVN(self)) > 0 post: result = AllClasses().attribute_list-> iterate(elem: Attribute; acc: Integer=0 acc + elem.AUN(self)) / AllClasses().attribute_list-> iterate(elem: Attribute; acc: Integer=0 acc + elem.AVN(self))</pre>
Comments	

Name	OHEF – Operations Hiding Effectiveness Factor
Informal definition	Quotient between the cumulative number of the specification classes that <i>do</i> access the specification operations and the cumulative number of the specification classes that <i>can</i> access the specification operations.
Formal definition	<p><u>Specification::OHEF(): Percentage</u></p> <pre>pre: AllClasses().operation_list-> iterate(elem: Operation; acc: Integer=0 acc + elem.OVN(self)) > 0 post: result = AllClasses().operation_list-> iterate(elem: Operation; acc: Integer=0 acc + elem.OUN(self)) / AllClasses().operation_list-> iterate(elem: Operation; acc: Integer=0 acc + elem.OVN(self))</pre>

Comments	

Name	BPF – Behavioral Polymorphism Factor
Informal definition	Quotient between the <i>actual</i> number of possible different polymorphic situations and the <i>maximum</i> number of possible distinct polymorphic situations (due to inheritance)
Formal definition	<pre><u>Specification::BPF(): Percentage</u> pre: TOA() > 0 post: result = TOO() / TOA()</pre>
Comments	<p>This metric was called POF (POLymorphism Factor) in the original MOOD set.</p> <p>A given message sent to class C_i can be bound, statically or dynamically, to a named operation implementation. The latter can have as many shapes (“morphos” in ancient Greek) as the number of times this same operation is overridden (in C_i descendants). This is what we call the <i>actual number of possible different polymorphic situations for that class</i>. We only consider the overriding of operations defined in the current specification.</p> <p>The <i>maximum number of possible distinct polymorphic situations</i> for class C_i occurs if all new operations defined in it are overridden in all of their derived classes.</p>

Name	PPF – Parametric Polymorphism Factor
Informal definition	Percentage of the specification classes that are parameterized
Formal definition	<pre><u>Specification::PPF(): Percentage</u> pre: self.TC() > 0 post: result = AllClasses()->select(formal_parameters->notEmpty()->size()/TC()</pre>
Comments	

Name	CCF – Class Coupling Factor
Informal definition	Quotient between the actual number of coupled class-pairs within the specification and the maximum possible number of class-pair couplings in the specification. This coupling is the one not imputable to inheritance.
Formal definition	<pre><u>Specification::CCF(): Percentage</u> pre: self.TC() > 1 -- with only one class there are no couplings within the specification ... post: result = sqrt (self.CL(self) / (sqr (self.TC()) - self.TC()))</pre>
Comments	<p>In a coupled class-pair one class is the <i>client</i> and the other is the <i>supplier</i>. These client-supplier relations can have several shapes; see the function <code>Class::CoupledClasses</code> for details.</p> <p>This metric is the square-root of COF (COUpling Factor) from the original MOOD set. The square-root counteracts for the fact that the couplings grow quadratically with the number of classes. The square and square-root functions <code>sqr()</code> and <code>sqrt()</code> were added to the OCL Real type.</p>

Name	ICF – Internal Coupling Factor
Informal definition	Quotient between the number of coupling links where both the client and supplier classes belong to the current specification and the total number of coupling links originating in the current specification.

Formal definition	<p><u>Specification::ICF(): Percentage</u></p> <pre>pre: self.TCL() > 0 post: result = self.CL(self) / self.TCL()</pre>
Comments	Coupling links originating in the current specification are those where the client class belongs to it. The coupling link is directed from the client class to the supplier one.

2.0.0 Inter-Specification Level Metrics

Name	EIF – External Inheritance Factor
Informal definition	Quotient between the number of external inheritance links to specification “s” and the total number of inheritance links originating in the current specification.
Formal definition	<p><u>Specification::EIF(s: Specification): Percentage</u></p> <pre>pre: self.uses->includes(s) pre: self.TIL() > 0 post: result = self.IL(s) / self.TIL()</pre>
Comments	External inheritance links are those originating in the current specification, but where the base class lies outside of it. By other words, they correspond to local derivations of external classes (defined in external specification “s”).

Name	ECF – External Coupling Factor
Informal definition	Quotient between the number of external coupling links to specification “s” and the total number of coupling links originating in the current specification.
Formal definition	<p><u>Specification::ECF(s: Specification): Percentage</u></p> <pre>pre: self.uses->includes(s) pre: self.TCL() > 0 post: result = self.CL(s) / self.TCL()</pre>
Comments	External coupling links are those originating in the current specification, but where the supplier class is defined outside of it (in external specification “s”).

Name	PRF – Potential Reuse Factor
Informal definition	Percentage of the available operations in the current specification that were imported from the “s” specification.
Formal definition	<p><u>Specification::PRF(s: Specification): Percentage</u></p> <pre>pre: self.uses->includes(s) pre: (AllClasses().AvailableOperations() union RelatedClasses(s).AvailableOperations()-> iterate(elem: Operation; acc: Real = 0 acc + elem.OSV(self)) > 0 post: result = RelatedClasses(s).AvailableOperations()-> iterate(elem: Operation; acc: Real = 0 acc + elem.OSV(self)) / (AllClasses().AvailableOperations() union RelatedClasses(s).AvailableOperations()-></pre>

	<code>iterate(elem: Operation; acc: Real = 0 acc + elem.OSV(self))</code>
Comments	The operations imported from the external specification ‘s’ correspond to those inherited from the classes from which current specification classes derive, plus the ones from ‘s’ which are coupled to internal classes.

Name	ARF – Actual Reuse Factor
Informal definition	Percentage of the available operations in the current specification that corresponds to effectively used operations imported from the “s” specification
Formal definition	<p><u>Specification::ARF(s: Specification): Percentage</u></p> <pre>pre: self.uses->includes(s) pre: (AllClasses().AvailableOperations() union RelatedClasses(s).AvailableOperations()-> iterate(elem: Operation; acc: Real=0 acc + elem.OSV(self)) > 0 post: result = RelatedClasses(s).AvailableOperations()->select(OUN(self)>0)-> iterate(elem: Operation; acc: Real=0 acc + elem.OSV(self)) / (AllClasses().AvailableOperations() union RelatedClasses(s).AvailableOperations()-> iterate(elem: Operation; acc: Real=0 acc + elem.OSV(self))</pre>
Comments	<p>The <i>elem</i> variable is called the iterator, here of type Class since we are going to iterate over the collection returned by <i>SupplierClasses(s)</i>, which is a Set{Class}. The <i>acc</i> variable is called the accumulator variable, here of type Integer, which is initiated with zero. In each iteration the accumulator gets assigned the value after the “ ” character.</p> <p>Attention: In the numerator we have the number of external supplier class operations that are used internally. It is still missing the inherited external operations that are effectively used.</p>

Name	REF – Reuse Efficiency Factor
Informal definition	Percentage of the imported operations (from the “s” specification) that are effectively used
Formal definition	<p><u>Specification::REF(s: Specification): Percentage</u></p> <pre>pre: self.uses->includes(s) pre: self.PRF() > 0 post: result =self.ARF(s) / self.PRF(s)</pre>
Comments	

4. Related work

Our work is similar in aim to the one of Scott Whitmire where he proposes a formal object model based on category theory, upon which he puts, as we do, the metrics on context [Whitmire1996, Whitmire1997]. Category theory is a major tool in the conception and study of programming languages [Pierce1991, Mac Lane1994, Barr1995]. A category is a structure that generalizes and unifies algebraic structures of discrete mathematics. Whitmire uses three categories (Class, DesignState and Design) to represent the static aspects of object-oriented designs and two additional categories (Message and Method) for their dynamic aspects. He uses simple relational algebra for describing the operations that alter the design structure. In dynamic modeling he adapted Object-Z predicate calculus [Duke1991, Stepney1992] for describing states and operation outcomes. He also borrowed from OPUS [Mens1994b, Mens1994a] an object calculus originally developed to support the design of object-oriented programming languages, to represent the mechanisms of operation selection and binding. We remark the applicability potential of his approach for supporting the development of tools such as theory-based

static analyzers for obtaining static measures, as well as for dynamic analysis where the impact of design changes and the results of stimuli (event sequences), applied at the design level, could be observed.

Also similar in aim is the dissertation research of Geert Poels where he proposes a measurement theory-based approach to software measure construction, using a combination of set theory and process algebra as the modeling language [Poels1999]. He makes a clear distinction between attribute definition and measure definition. Using his distance-based measurement approach each software artifact attribute is defined in terms of a segmentally additive proximity structure, i.e. a formal representation of the concept of 'distance' that allows defining a unit of distance and preparing the path to a ratio measurement scale. Analogously, each software measure is defined in terms of a metric space with additive segments, i.e. a measurement theoretic interpretation of the mathematical function that is called a 'metric'. The distance-based approach links both types of definition by using proximity structure representation and uniqueness theorems that can be found in advanced textbooks on measurement theory, such as [Suppes1989]. According to his author, all software metrics that are constructed using this approach have a proven construct validity and are defined precisely and unambiguously.

Both of the previous proposals are mathematically elegant and sound. However, from our experience of more than 15 years in the software field, the vast majority of software designers lack the appropriate background (or have lost it) to understand the intricacies of measurement theory, category theory or of proximity structures. Therefore, we suspect that we will more likely find followers of Whitmire's and Poels's approaches within the metrics researchers community rather than among design practitioners.

5. Conclusions and further work

We have shown that the Object Constraint Language, a part of the UML standard, can be used to define object-oriented design metrics in a very natural and understandable way. The precision granted by the formality of OCL comes at a much lower cost, for both practitioners and tool builders, than when using other formal specification constructs. Since UML has also become a *de facto* standard both in academia and industry, more and more people are expected to master OCL and use it currently in their designs. In a simple search in the WWW we have found that several OCL supporting tools (or add-ins to existing design tools) have started to emerge.

We are now working on the application of the same approach, but using the UML meta-model that is part of the standard [OMG1997] instead of the GOODLY one used in this report. We also plan to apply our approach to the **OML** (*OPEN Modeling Language*) meta-model. OML emerged from the OPEN (*Object-oriented Process, Environment and Notation*) consortium [Firesmith1996, Henderson-Sellers1996b, Henderson-Sellers1996c, Henderson-Sellers1998]. The latter is supported by a large group of well-known methodologists such as *Brian Henderson-Sellers* (author of the *MOSES* method [Henderson-Sellers1991, Henderson-Sellers1994]), *Ian Graham* (author of *SOMA – Semantic Object Modelling Approach* [Graham]), *Donald Firesmith* or *Jim Odell*. We also have provisions to further show the applicability of our approach by trying to model other object-oriented metric sets published in the literature.

We believe the time has come for object-oriented metrics research community to standardize the way we define the metrics, as it happened with the object-oriented analysis and design notations. Although we are strong believers that diversity and innovation should not be constrained, we owe that standardization effort to those that, after all, are our final users – the design practitioners and those that support and train them, such as tool manufacturers, consultants, professional trainers or academic teachers. We think that such a standardization effort will not reach widespread acceptance if it is not integrated with the current state-of-the-practice object-oriented design technology. We hope to have shown here that that is possible. We will be happy if this document will generate discussion and feedback around this topic.

Bibliography

- [Abreu1998a] Fernando Brito Abreu : “The MOOD2 Metrics Set (in Portuguese)”, *Grupo de Engenharia de Software, INESC*, technical report R7/98, April, 1998.
- [Abreu1994] Fernando Brito Abreu & Rogério Carapuça : “Object-Oriented Software Engineering: Measuring and Controlling the Development Process”, *proceedings of the 4th International Conference on Software Quality*, McLean, Virginia, USA, October 3-5, 1994.

- [Abreu1996a] Fernando Brito Abreu, Rita Esteves & Miguel Afonso Goulão : “The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics”, *proceedings of the TOOLS'96 (Technology of Object Oriented Languages and Systems)*, Santa Barbara, CA, USA, July, 1996.
- [Abreu2001a] Fernando Brito Abreu & Miguel Goulão : “Coupling and Cohesion as Modularization Drivers: Are we being over-persuaded?”, *proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR'2001)*, Lisbon, Portugal, March 2001, 2001.
- [Abreu1995] Fernando Brito Abreu, Miguel Afonso Goulão & Rita Esteves : “Toward the Design Quality Evaluation of Object-Oriented Software Systems”, *proceedings of the 5th International Conference on Software Quality*, pp.44-57, Austin, Texas, USA, October, 1995.
- [Abreu1998b] Fernando Brito Abreu & Jean Sebastien Cuche (École de Mines de Nantes) : “Collecting and Analyzing the MOOD2 Metrics”, *proceedings of the ECOOP'98 Workshop - Object-Oriented Product Metrics for Software Quality Assessment*, pp.258-260, Brussels, Belgium, 21st July, 1998.
- [Abreu1999] Fernando Brito Abreu, Luís Miguel Ochoa & Miguel Afonso Goulão : “The GOODLY Design Language for MOOD2 Metrics Collection”, *proceedings of the ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Lisboa, Portugal, 15 June, 1999.
- [Abreu2000] Fernando Brito Abreu, Gonçalo Pereira & Pedro Sousa : “A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems”, *proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'2000)*, pp.13-22, Zurich, Switzerland, 29 February to 3 March 2000, 2000.
- [Abreu1996b] Fernando Brito e Abreu & Walcelio Melo (University of Maryland) : “Evaluating the Impact of Object-Oriented Design on Software Quality”, *proceedings of the 3rd International Software Metrics Symposium (Metrics'96)*, Berlin, Germany, March, 1996.
- [Araújo1998] João Araújo & P. Sawyer : “Integrating Object-Oriented Analysis and Formal Specification”, *Journal of Brazilian Computer Society*, July, 1998.
- [Barr1995] M. Barr & C. Wells : *Category Theory for Computing Science*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1995.
- [Cherniavsky1991] John C. Cherniavsky & Carl H. Smith : “On Weyuker's Axioms For Software Complexity Measures”, *IEEE Transactions on Software Engineering*, vol.17, n°6, pp.636-638, IEEE Computer Society Press, June, 1991.
- [Chidamber1994] Shyam R. Chidamber & Chris F. Kemerer : “A Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, vol.20, n°6, pp.476-493, IEEE, June, 1994, publicado também em: Center of Information Systems Research, MIT, USA, WP No. 249, July 1993.
- [Churcher1995] Neville I. Churcher & Martin J. Shepperd : “Comments on a Metrics Suite for Object Oriented Design”, *IEEE Transactions on Software Engineering*, vol.21, n°3, pp.263-265, IEEE, March, 1995.
- [Cook1994] Steve Cook & John Daniels : *Designing Object Systems: Object Oriented Modeling with Syntropy*, Prentice Hall, Hemel Hempstead (U.K.), 1994.
- [D'Souza1998] Desmond Francis D'Souza & Alan Cameron Wills : *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison Wesley Longman, Reading, Massachusetts, ISBN 0-201-31012-0, 1998.
- [Duke1991] D. Duke, P. King, G. A. Rose & G. Smith : “The Object-Z Specification Language”, *Department of Computing Science, University of Queensland, Australia*, technical report 91-1, 1991.
- [Fenton1991] Norman E. Fenton (ed.) : *Software Metrics: A Rigorous Approach*, Chapman & Hall (UK) / Van Nostrand Reinhold (USA), ISBN 0-412-40440-0, 1991.
- [Firesmith1996] Donald Firesmith, Brian Henderson-Sellers & Ian Graham : “OPEN Modeling Language (OML) - Core Notation Specification”, *OPEN Consortium*, technical report Version 0.1, July, 1996.

- [Graham1995] I. M. Graham : “A Non-Procedural Process Model for Object-Oriented Software Development”, *Report on Object Oriented Analyses and Design*, vol.5, n°1, 1995.
- [Harrison1998] Rachel Harrison, Steve J. Counsell & Reuben V. Nithi : “An Evaluation of the MOOD Set of Object-Oriented Software Metrics”, *IEEE Transactions on Software Engineering*, vol.24, n°6, pp.491-496, IEEE Computer Society, June, 1998.
- [Henderson-Sellers1991] Brian Henderson-Sellers : *A BOOK of Object-Oriented Knowledge*, Prentice Hall PTR, Sydney, Australia, 1991.
- [Henderson-Sellers1996a] Brian Henderson-Sellers : *Object-Oriented Metrics - Measures of Complexity*, The Object-Oriented Series, Prentice Hall PTR, Upper Saddle River, NJ, USA, ISBN 0-13-239872, 1996.
- [Henderson-Sellers1996b] Brian Henderson-Sellers : “The OPEN Methodology”, *Object Magazine*, vol.6, n°9, pp.56-59, November, 1996.
- [Henderson-Sellers1994] Brian Henderson-Sellers & Julian M. Edwards : *BOOK TWO of Object-Oriented Knowledge: the Working Object*, Prentice Hall, Sydney, Australia, 1994.
- [Henderson-Sellers1996c] Brian Henderson-Sellers & Ian Graham : “OPEN: Towards Method Convergence?”, *IEEE Computer*, vol.29, n°4, pp.86-89, IEEE / Object Technology Department, April, 1996.
- [Henderson-Sellers1998] Brian Henderson-Sellers, Tony Simons & Houman Younessi : *The OPEN Toolbox of Techniques*, Addison-Wesley Publishing Company, ISBN 0-201-33134-9, 1998.
- [Jones1990] Cliff B. Jones : *Systematic Software Development Using VDM*, 2nd edition, Prentice-Hall International, Hemel Hempstead (U.K.), 1990.
- [Lorenz1994] Mark Lorenz & Jeff Kidd : *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Englewood Cliffs, NJ, USA, ISBN 0-13-179292-X, 1994.
- [Mac Lane1994] S. Mac Lane : *Categories for the Working Mathematician*, 2nd edition, Springer-Verlag, New York, EUS, 1994.
- [Mens1994a] T. Mens, K. Mens & P. Steyaert : “OPUS: A Calculus for Modeling Object-Oriented Concepts”, *Department of Mathematics and Computer Sciences, Brussels Free University (VUB)*, technical report vub-tinf-tr-94-04, 1994.
- [Mens1994b] T. Mens, K. Mens & P. Steyaert : “OPUS: A Formal Approach to Object Orientation”, *Department of Mathematics and Computer Sciences, Brussels Free University (VUB)*, technical report vub-tinf-tr-94-02, 1994.
- [Meyer1985] Bertrand Meyer : “On Formalism in Specifications”, *IEEE Software*, vol.2, n°1, pp.6-26, January, 1985, also published in: T. Colburn, J. Fetzer, and T. Rankin (eds.), *Program Verification: Fundamental Problems in Computer Science*, Kluwer Academic Publishers, Dordrecht (The Netherlands), 1993.
- [Meyer1995] Bertrand Meyer : “Beyond Design by Contract: Putting More Formality into Object-Oriented Development”, *proceedings of the TOOLS EUROPE*, Versailles, France, 1995.
- [Meyer1997] Bertrand Meyer : *Object-Oriented Software Construction*, 2nd edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, ISBN 0-13-629155-4, 1997.
- [Moreira1996] Ana Moreira & R. Clark : “Adding Rigour to Object-Oriented Analysis”, *Software Engineering Journal*, vol.11, n°5, pp.270-280, July, 1996.
- [Myers1997] Andrew C. Myers, Joseph A. Bank & Barbara Liskov : “Parameterized types for Java”, *proceedings of the Symposium on Principles of Programming Languages*, pp.132-145, 1997.
- [OMG1997] OMG : *Object Constraint Language Specification (version 1.1)*, Rational et al. (ed.), ad97-08-08, Object Management Group, 1997.
- [Pierce1991] B. Pierce : *Basic Category Theory for Computer Scientists*, MIT Press, Cambridge, MA, USA, 1991.
- [Poels1999] Geert Poels : *On the Formal Aspects of the Measurement of Object-Oriented Software Specifications*, dissertação de Ph.D., Faculty of Economic and Applied Economic Sciences, Katholieke Universiteit Leuven, Leuven, Belgium, 1999.
- [Rumbaugh1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy & William Lorenzen :

Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

- [Spivey1992] J. Michael Spivey : *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall, Hemel Hempstead, United Kingdom, 1992.
- [Stepney1992] S. Stepney, R. Barden & D. Cooper (eds.) : *Object-Orientation in Z*, Springer-Verlag, London, U.K., 1992.
- [Suppes1989] Patrick Suppes, David H. Krantz, R. Duncan Luce & Amos Tversky : *Foundations of Measurement: Geometrical, Threshold and Probabilistic Representations*, vol.2, Academic Press, San Diego, CA, USA, 1989.
- [Waldén1995] Kim Waldén & Jean-Marc Nerson : *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*, Prentice Hall, Hemel Hempstead, United Kingdom, 1995.
- [Wand1989] Yair Wand : “A Proposal for a Formal Model of Objects”, em Kim WoneFrederick H. Lochovsky (eds.): *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, MA, USA, 1989.
- [Warmer1999] Jos Warmer & Anneke Kleppe : *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Publishing Company, ISBN 0-201-37940-6, 1999.
- [Weyuker1988] Elaine J. Weyuker : “Evaluating Software Complexity Measures”, *IEEE Transactions on Software Engineering*, vol.14, n°9, pp.1357-1365, IEEE, September, 1988.
- [Whitmire1996] Scott A. Whitmire : “A Theory of Objects”, *proceedings of the Object-Oriented Product Metrics Workshop (OOPSLA'96)*, San José, California, USA, 1996.
- [Whitmire1997] Scott A. Whitmire : *Object-Oriented Design Measurement*, John Wiley & Sons, Inc., New York, USA, ISBN 0-471-13417-1, 1997.
- [Zuse1991] Horst Zuse : *Software Complexity: Measures and Methods*, Walter DeGruyter Publisher, Berlin / New York, 1991.
- [Zuse1993] Horst Zuse : “Discussion of the Weyuker Properties from a Measurement Theoretic View”, *IEEE Transactions on Software Engineering*, IEEE, Zuse93a, 1993.
- [Zuse1989] Horst Zuse & Peter Bollmann-Sdorra : “Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics”, *SIGPLAN Notices*, vol.24, n°8, pp.22-33, August, 1989.

Appendix A – Auxiliary meta-model OCL functions

1.0 Attribute-level functions

Name	ACV – Attribute to Class Visibility
Informal definition	Predicate that indicates if a given class can access the attribute
Formal definition	<p><u>Attribute::ACV(c: Class): Boolean</u></p> <pre> post: result = scope_list->exists((class = c) or (scope_type = #PUBLIC) or (scope_type=#SPEC) and (class.module.specification=c.module.specification) or (scope_type = #MODULE) and (class.module = c.module) or (scope_type = #CLASS_TREE) and scoped_class.Descendants()->includes(c) or (scope_type = #PROTECTED) and class.Descendants()->includes(c) or (scope_type = #DISCRIMINATED) and (scoped_class = c)) </pre>
Comments	

Name	AVN – Attribute Visibility Number
Informal definition	Number of classes in the considered specification where the attribute can be accessed
Formal definition	<p><u>Attribute::AVN(s: Specification): Integer</u></p> <pre> post: result = s.AllClasses()->iterate(elem: Class; acc: Integer = 0 if self.ACV(elem) then acc + 1 else acc endif) </pre>
Comments	

Name	ASV – Attribute to Specification Visibility
Informal definition	Percentage of classes in the considered specification where the attribute can be accessed (excludes the class where the attribute is declared)
Formal definition	<p><u>Attribute::ASV(s: Specification): Percentage</u></p> <pre> pre: s.TC() > 1 post: result = (AVN(s) - 1) / (s.TC() - 1) </pre>
Comments	The pre-condition states that at least another class besides the one where the attribute is defined must exist. This would hardly be a true restriction.

Name	AUN – Attribute Use Number
Informal definition	Number of classes in the considered specification where the attribute is used (excludes the class where the attribute is declared)
Formal definition	<p><u>Attribute::AUN(s: Specification): Integer</u></p>

	<pre> post: result = s.AllClasses()-> select(operation_list.operation_body.employs_spec->includes(self)) ->asSet() ->size() </pre>
Comments	The asSet() operation is required because the select returns a Bag

2.0 Operation-level functions

Name	OCV – Operation to Class Visibility
Informal definition	Predicate that indicates if a given class can access the operation
Formal definition	<p><u>Operation::OCV(c: Class): Boolean</u></p> <pre> post: result = scope_list->exists((class = c) or (scope_type = #PUBLIC) or (scope_type = #SPEC) and (class.module.specification=c.module.specification)or (scope_type = #MODULE) and (class.module = c.module) or (scope_type = #CLASS_TREE) and scoped_class.Descendants()->includes(c) or (scope_type = #PROTECTED) and class.Descendants()->includes(c) or (scope_type = #DISCRIMINATED) and (scoped_class = c)) </pre>
Comments	

Name	OVN – Operation Visibility Number
Informal definition	Number of classes in the considered specification where the operation can be accessed
Formal definition	<p><u>Operation::OVN(s: Specification): Integer</u></p> <pre> post: result = s.AllClasses()->iterate(elem: Class; acc: Integer = 0 if self.OCV(elem) then acc + 1 else acc endif) </pre>
Comments	

Name	OSV – Operation to Specification Visibility
Informal definition	Percentage of classes in the considered specification where the operation can be accessed (excludes the class where the operation is declared)
Formal definition	<p><u>Operation::OSV(s: Specification): Percentage</u></p> <pre> pre: s.TC() > 1 post: result = (OVN(s) - 1) / (s.TC() - 1) </pre>
Comments	The pre-condition states that at least another class besides the one where the attribute is defined must exist. This would hardly be a true restriction.

Name	OUN – Operation Use Number
Informal definition	Number of classes in the considered specification where the operation is used
Formal definition	<pre><u>Operation::OUN(s: Specification): Integer</u> post: result = s.AllClasses()-> select(operation_list.operation_body.messages_spec.operation-> includes(self))->asSet()->size()</pre>
Comments	The <i>asSet()</i> operation is required because the select returns a <i>Bag</i>

3.0 Class-level predicate functions

Name	IsInternal
Informal definition	Internal class predicate – indicates if the class belongs to the named specification “s”
Formal definition	<pre><u>Class::IsInternal(s: Specification): Boolean</u> post: result = self.module.specification = s</pre>
Comments	

Name	IsRoot
Informal definition	Root class predicate – indicates that it has no ascendants
Formal definition	<pre><u>Class::IsRoot(): Boolean</u> post: result = Parents()->isEmpty()</pre>
Comments	

Name	IsLeaf
Informal definition	Leaf class predicate – indicates that it has no descendants
Formal definition	<pre><u>Class::IsLeaf(): Boolean</u> post: result = Children()->isEmpty()</pre>
Comments	

4.0 Class-level set functions

Name	Children
Informal definition	Set of directly derived classes
Formal definition	<pre><u>Class::Children(): Set(Class)</u> post: result = Class.allInstances->select(inherits_from->includes(self))</pre>

Comments	
----------	--

Name	Descendants
Informal definition	Set of all derived classes (either directly or indirectly)
Formal definition	<pre><u>Class::Descendants(): Set(Class)</u> post: result = Children()-> iterate(elem: Class; acc: Set(Class)=Children() acc-> union (elem.Descendants())</pre>
Comments	This operation is recursive. Notice that even with multiple inheritance the result is a set (no repeated classes)

Name	Parents
Informal definition	Set of classes from which the current class derives directly
Formal definition	<pre><u>Class::Parents(): Set(Class)</u> post: result = inherits_from</pre>
Comments	

Name	Ascendants
Informal definition	Set of classes from which the current class derives directly or indirectly
Formal definition	<pre><u>Class::Ascendants(): Set(Class)</u> post: result = Parents()-> iterate(elem: Class; acc: Set(Class)=Parents() acc-> union(elem.Ascendants())</pre>
Comments	This operation is recursive. Notice that even with common ancestors due to multiple inheritance the result is a set (no repeated classes)

Name	CoupledClasses
Informal definition	Set of classes to which the current class is coupled (excluding inheritance)
Formal definition	<pre><u>Class::CoupledClasses(): Set(Class)</u> post: result = formal_parameters.instanced_as union(attribute_list.attribute_type union(operation_list.parameter_list.attribute_type union(operation_list.return_type union(operation_list.operation_body.locals_spec.attribute_type union(operation_list.operation_body.employs_spec.attribute_type union(operation_list.operation_body.messages_spec.invocation_of.class))))</pre>
Comments	<p>This function includes the coupled classes corresponding to:</p> <ul style="list-style-type: none"> - instantiation of class parameters - class attributes - parameters of class operations - return type of class operations

	<ul style="list-style-type: none"> - local attributes of class operations - attributes of other classes employed by class operations - recipients of messages sent in the class operations implementation body
--	---

Name	NewOperations
Informal definition	Operations defined in the class that are not overriding inherited ones
Formal definition	<pre><u>Class::NewOperations(): Set(Operation)</u> post: result = DefinedOperations() - InheritedOperations()</pre>
Comments	

Name	InheritedOperations
Informal definition	Number of inherited operations that are not overridden by locally defined ones
Formal definition	<pre><u>Class::InheritedOperations(): Set(Operation)</u> post: result = Ascendants()-> iterate(elem: Class; acc: Set(Operation)=Set{} acc->union(elem.operation_list))</pre>
Comments	

Name	OverriddenOperations
Informal definition	Number of operations defined in the class that override inherited ones = number of inherited operations that are overridden by locally defined ones
Formal definition	<pre><u>Class::OverriddenOperations(): Set(Operation)</u> post: result = DefinedOperations()->intersection(InheritedOperations())</pre>
Comments	

Name	DefinedOperations
Informal definition	Number of operations defined in the class
Formal definition	<pre><u>Class::DefinedOperations(): Set(Operation)</u> post: result = operation_list</pre>
Comments	

Name	AvailableOperations
Informal definition	Number of operations that may be applied to instances of the class
Formal definition	<pre><u>Class::AvailableOperations(): Set(Operation)</u> post: result = NewOperations()-> union (InheritedOperations())</pre>
Comments	The following invariant could be stated in alternative: <pre>post: result = DefinedOperations()-> union (InheritedOperations())</pre>

Name	NewAttributes
Informal definition	Attributes defined in the class that are not overriding inherited ones
Formal definition	<u>Class::NewAttributes(): Set(Attribute)</u> <i>post: result = DefinedAttributes() - InheritedAttributes()</i>
Comments	

Name	InheritedAttributes
Informal definition	Number of inherited attributes that are not overridden by locally defined ones
Formal definition	<u>Class::InheritedAttributes(): Set(Attribute)</u> <i>post: result = Ascendants()->iterate(elem: Class; acc: Set(Attribute)= Set{} acc-> union (elem.attribute_list))</i>
Comments	

Name	OverriddenAttributes
Informal definition	Number of attributes defined in the class that override inherited ones = number of inherited attributes that are overridden by locally defined ones
Formal definition	<u>Class::OverriddenAttributes(): Set(Attribute)</u> <i>post: result = DefinedAttributes()->intersection(InheritedAttributes())</i>
Comments	

Name	DefinedAttributes
Informal definition	Number of attributes defined in the class
Formal definition	<u>Class::DefinedAttributes(): Set(Attribute)</u> <i>post: result = attribute_list</i>
Comments	

Name	AvailableAttributes
Informal definition	Number of attributes that may be applied to instances of the class
Formal definition	<u>Class::AvailableAttributes(): Set(Attribute)</u> <i>post: result = NewAttributes()-> union (InheritedAttributes())</i>
Comments	The following invariant could be stated in alternative: <i>post: result = DefinedAttributes()-> union (InheritedAttributes())</i>

5.0 Class-level counting functions

Name	CC – Children Count
Informal definition	Number of directly derived classes
Formal definition	<pre><u>Class::CC(): Integer</u> post: result = Children()->size()</pre>
Comments	If CC() = 0 then the class is a leaf class

Name	DC – Descendants Count
Informal definition	Number of all derived classes (either directly or indirectly)
Formal definition	<pre><u>Class::DC(): Integer</u> post: result = Descendants()->size()</pre>
Comments	

Name	PC – Parents Count
Informal definition	Number of classes from which the current class derives directly
Formal definition	<pre><u>Class::PC(): Integer</u> post: result = Parents()->size()</pre>
Comments	If PC() = 0 then the class is a base class; if PC() > 1 we have multiple inheritance

Name	AC – Ascendants Count
Informal definition	Number of classes from which the current class derives directly or indirectly
Formal definition	<pre><u>Class::AC(): Integer</u> post: result = Ascendants()->size()</pre>
Comments	

Name	ON – Operations New
Informal definition	Number of operations defined in the class that are not overriding inherited ones
Formal definition	<pre><u>Class::ON(): Integer</u> post: result = NewOperations()->size()</pre>
Comments	

Name	OI – Operations Inherited
------	----------------------------------

Informal definition	Number of inherited operations that are not overridden by locally defined ones
Formal definition	<u>Class::OI(): Integer</u> <i>post: result = InheritedOperations()->size()</i>
Comments	

Name	OO – Operations Overridden
Informal definition	Number of inherited operations that are overridden by locally defined ones = Number of operations defined in the class that override inherited ones
Formal definition	<u>Class::OO(): Integer</u> <i>post: result = OverriddenOperations()->size()</i>
Comments	

Name	OD – Operations Defined
Informal definition	Number of operations defined in the class
Formal definition	<u>Class::OD(): Integer</u> <i>post: result = DefinedOperations()->size()</i>
Comments	

Name	OA – Operations Available
Informal definition	Number of operations that may be applied to instances of the class
Formal definition	<u>Class::OA(): Integer</u> <i>post: result = AvailableOperations()->size()</i>
Comments	

Name	AN – Attributes New
Informal definition	Number of attributes defined in the class that are not overriding inherited ones
Formal definition	<u>Class::AN(): Integer</u> <i>post: result = NewAttributes()->size()</i>
Comments	

Name	AI – Attributes Inherited
Informal definition	Number of inherited attributes that are not overridden by locally defined ones
Formal definition	<u>Class::AI(): Integer</u>

	<code>post: result = InheritedAttributes()->size()</code>
Comments	

Name	AO – Attributes Overridden
Informal definition	Number of attributes defined in the class that override inherited ones = number of inherited attributes that are overridden by locally defined ones
Formal definition	<u>Class::AO(): Integer</u> <code>post: result = OverriddenAttributes()->size()</code>
Comments	

Name	AD – Attributes Defined
Informal definition	Number of attributes defined in the class
Formal definition	<u>Class::AD(): Integer</u> <code>post: result = DefinedAttributes()->size()</code>
Comments	

Name	AA – Attributes Available
Informal definition	Number of attributes that may be associated to instances of the class.
Formal definition	<u>Class::AA(): Integer</u> <code>post: result = AvailableAttributes()->size()</code>
Comments	

6.0 Specification-level set functions

Name	AllClasses
Informal definition	Set of all classes belonging to the current specification
Formal definition	<u>Specification::AllClasses(): Set(Class)</u> <code>post: result= module_list.class_list</code>
Comments	

Name	BaseClasses
Informal definition	Set of base classes of classes from the current specification that belong to the given “s” specification
Formal definition	<u>Specification::BaseClasses(s: Specification): Set(Class)</u>

	<code>post: result= AllClasses().inherits_from->select(IsInternal(s))->asSet()</code>
Comments	The <i>asSet()</i> operation is required because the <i>select</i> returns a <i>Bag</i>

Name	SupplierClasses
Informal definition	Set of supplier classes of classes from the current specification that belong to the given “s” specification (excludes inheritance)
Formal definition	<p><u>Specification::SupplierClasses(s: Specification): Set(Class)</u></p> <pre>post: result = AllClasses()->iterate(elem:Class; acc: Set(Class)=Set{} acc union (elem.CoupledClasses()-> select(IsInternal(s))))</pre>
Comments	

Name	RelatedClasses
Informal definition	Set of classes from the “s” specification that are either base or supplier classes from the ones of the current specification
Formal definition	<p><u>Specification::RelatedClasses(s: Specification): Set(Class)</u></p> <pre>post: result = BaseClasses(s) union SupplierClasses(s)</pre>
Comments	

7.0 Specification-level counting functions

Name	TC – Total Classes
Informal definition	Total number of classes in the specification
Formal definition	<p><u>Specification::TC(): Integer</u></p> <pre>post: result = AllClasses()->size()</pre>
Comments	Although, in the general case, the result of navigating two associations is a <i>Bag</i> , here we can guarantee that the result is like a <i>Set</i> since the same class cannot belong to distinct modules.

Name	TON – Total Operations New
Informal definition	Total number of new operations in the specification
Formal definition	<p><u>Specification::TON(): Integer</u></p> <pre>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.ON())</pre>
Comments	

Name	TOO – Total Operations Overridden
Informal	Total number of overridden operations in the specification

definition	
Formal definition	<p><u>Specification::TOO(): Integer</u></p> <p><i>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.OO())</i></p>
Comments	

Name	TOD - Total Operations Defined
Informal definition	Total number of defined operations in the specification
Formal definition	<p><u>Specification::TOD(): Integer</u></p> <p><i>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.OD())</i></p>
Comments	

Name	TOI – Total Operations Inherited
Informal definition	Total number of inherited operations in the specification
Formal definition	<p><u>Specification::TOI(): Integer</u></p> <p><i>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.OI())</i></p>
Comments	

Name	TOA – Total Operations Available
Informal definition	Total number of available operations in the specification
Formal definition	<p><u>Specification::TOA(): Integer</u></p> <p><i>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.OA())</i></p>
Comments	

Name	TAN – Total Attributes New
Informal definition	Total number of new attributes in the specification
Formal definition	<p><u>Specification::TAN(): Integer</u></p> <p><i>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.AN())</i></p>
Comments	

Name	TAO – Total Attributes Overridden
Informal	Total number of overridden attributes in the specification

definition	
Formal definition	<u>Specification::TAO(): Integer</u> <code>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.AO())</code>
Comments	

Name	TAD – Total Attributes Defined
Informal definition	Total number of defined attributes in the specification
Formal definition	<u>Specification::TAD(): Integer</u> <code>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.AD())</code>
Comments	

Name	TAI – Total Attributes Inherited
Informal definition	Total number of attributes inherited in the specification
Formal definition	<u>Specification::TAI(): Integer</u> <code>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.AI())</code>
Comments	

Name	TAA – Total Attributes Available
Informal definition	Total number of available attributes in the specification
Formal definition	<u>Specification::TAA(): Integer</u> <code>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.AA())</code>
Comments	

Name	IL – Inheritance Links
Informal definition	Total number of inheritance relations where the derived classes belongs to the current specification and the base one belongs to the given “s” specification
Formal definition	<u>Specification::IL(s:Specification): Integer</u> <code>post: result= AllClasses().Parents()-> select(IsInternal(s))->size()</code>
Comments	Notice that IL(s) <= TIL()

Name	TIL – Total Inheritance Links
Informal	Total number of inheritance relations where the derived classes belongs to the current specification

definition	
Formal definition	<p><u>Specification::TIL(): Integer</u></p> <pre>post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 acc + elem.PC())</pre>
Comments	Alternative post-condition: <i>result = AllClasses().inherits_from->size()</i>

Name	CL – Coupling Links
Informal definition	Total number of coupling relations where the client class belongs to the current specification and the supplier class belongs to the given “s” specification (excludes inheritance)
Formal definition	<p><u>Specification::CL(s: Specification): Integer</u></p> <pre>post: result =self.SupplierClasses(s)->size()</pre>
Comments	

Name	TCL – Total Coupling Links
Informal definition	Total number of distinct coupling relations where the client class belongs to the current specification (excludes inheritance)
Formal definition	<p><u>Specification::TCL(): Integer</u></p> <pre>post: result = AllClasses()-> iterate(elem: Class; acc: Integer = 0 acc + elem.CoupledClasses()->size())</pre>
Comments	

Appendix B – GOODLY code extract

```
SPECIFICATION An_example
  TYPE APPLICATION
  VERSION "2.1"
  DESCRIPTION "A rally competition"
  OWNER "MOOD Project"
  USES
    xpto
  END_USES
  MODULES
    Race
  END_MODULES

  MAIN
    LOCALS
      mikkola: Pilot;
      chuck: Mechanic;
    END_LOCALS

    MESSAGES
      Pilot.create;
      Mechanic.create;
      Pilot.qualification;
      Pilot.talks_to {Mechanic};
      Mechanic.repairs;
      Pilot.qualification;
      Pilot.talks_to {Mechanic};
      Mechanic.repairs;
      Pilot.qualification;
      Pilot.destroys;
      Mechanic.destroys
    END_MESSAGES

    CALLS ... END_CALLS
  END_MAIN
END_SPECIFICATION
```

```
MODULE Race
  DESCRIPTION "Example of a module of an application in the race car world"
  AUTHORS "Fernando Brito e Abreu et al."
  PRODUCED "18/7/98_12:00:03"
  CONVERTED "20/8/98_15:34:45"
  FORMALISM xpto

  CLASS Document[Objective] END_CLASS

  CLASS Sport END_CLASS

  CLASS Bank END_CLASS

  CLASS Person
    STATE
      name: String SCOPE PUBLIC;
      birth_date: Date SCOPE PRIVATE;
      weight: Integer SCOPE PRIVATE;
      address: String SCOPE PRIVATE;
```

```

        bankcard: Document[Myers] SCOPE PRIVATE
    END_STATE

    BEHAVIOR
        OPERATION creates( ) SCOPE PUBLIC END_OPERATION
        OPERATION destroys( ) SCOPE PUBLIC END_OPERATION
    END_BEHAVIOR
END_CLASS { Person }

CLASS Pilot ISA Person
    STATE
        driving_permit: Document[Sport] SCOPE PRIVATE;
        helmet: Protection SCOPE PROTECTED
    END_STATE

    BEHAVIOR
        OPERATION qualification(start: Time; from: String; to: String) SCOPE PUBLIC
        MESSAGES
            Protection.creates;
            Protection.put_on;
            Document[Sport].shows;
            RaceCar.close_window;
            RaceCar.starts;
            drives;
            RaceCar.stops;
            RaceCar.open_window;
            Document[Sport].shows;
            Protection.take_off;
            Protection.destroys
        END_MESSAGES
    END_OPERATION

        OPERATION drives(vehicle : RaceCar): Boolean SCOPE PUBLIC
        MESSAGES
            Co-pilot.read_notes;
            RaceCar.accelerates;
            Co-pilot.read_notes;
            RaceCar.breaks;
            RaceCar.turns
        END_MESSAGES
    END_OPERATION

        OPERATION talks_to(whom: Person): Boolean SCOPE Co-pilot, Mechanic
    END_OPERATION
    END_BEHAVIOR
END_CLASS { Pilot }

CLASS Co-pilot ISA Person
    BEHAVIOR
        OPERATION read_notes( ) SCOPE PUBLIC END_OPERATION
    END_BEHAVIOR
END_CLASS { Co-pilot }

CLASS Mechanic ISA Person
    BEHAVIOR
        OPERATION repairs (vehicle : RaceCar) SCOPE PUBLIC END_OPERATION
    END_BEHAVIOR
END_CLASS { Mechanic }

```

```

CLASS Artifact
  STATE
    brand: String SCOPE PUBLIC;
    model: String SCOPE PROTECTED;
    production_date: Date SCOPE PRIVATE
  END_STATE

  BEHAVIOR
    OPERATION creates( ) SCOPE PUBLIC END_OPERATION;
    OPERATION destroys( ) SCOPE PUBLIC END_OPERATION
  END_BEHAVIOR
END_CLASS { Artifact }

CLASS Protection ISA Artifact
  BEHAVIOR
    OPERATION put_on( ): Boolean SCOPE PUBLIC END_OPERATION
    OPERATION take_off( ): Boolean SCOPE PUBLIC END_OPERATION
  END_BEHAVIOR
END_CLASS { Protection }

CLASS RaceCar ISA Artifact
  STATE
    zero_to_hundred: Time SCOPE PUBLIC;
    horse_power: Integer SCOPE PUBLIC;
    oil_level: Real SCOPE Mechanic.*, Pilot, Co-Pilot;
    actual_speed: Integer SCOPE PROTECTED, Pilot, Co-Pilot
  END_STATE
  BEHAVIOR
    OPERATION starts( ): Boolean SCOPE PUBLIC END_OPERATION
    OPERATION accelerates(duration:Time; acceleration:Integer) SCOPE PUBLIC
    END_OPERATION
    OPERATION breaks(duration: Time; desacceleration: Integer) SCOPE PUBLIC
    END_OPERATION
    OPERATION turns(angle: Integer ) SCOPE PUBLIC END_OPERATION
    OPERATION stops( distance: Integer ): Boolean SCOPE PUBLIC
    END_OPERATION
    OPERATION open_window( ): Boolean SCOPE PUBLIC END_OPERATION
    OPERATION close_window( ): Boolean SCOPE PUBLIC END_OPERATION
  END_BEHAVIOR
END_CLASS { RaceCar }

END_MODULE { Race }

```