A metric for measuring the complexity of OCL expressions

Jordi Cabot
Estudis d'Informàtica i Multimèdia
Universitat Oberta de Catalunya
Rambla del Poblenou, 156 E08018 Barcelona
jcabot@uoc.edu

Ernest Teniente
Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Campus Nord, Ed. Omega, 132, E08034 Barcelona
teniente@lsi.upc.edu

ABSTRACT

Despite the importance of OCL in the specification of UML models, few existing metrics are devoted to measure the complexity of OCL expressions. The proposed ones are based on the syntactic structure of the expressions (number of referred attributes, number of navigations,…) and on the constructs used in their definition (as the number of *forAll* and *select* iterators). Indeed, these metrics are helpful to determine, for instance, the understandability of the expressions but we believe they fall short when providing a precise measure of their complexity.

In this paper we propose a new metric to compute the complexity of OCL expressions. Our metric is based on the number of objects involved in the evaluation of the expression. An expression e_1 is more complex than an expression e_2 if the number of objects required to evaluate e_1 is greater than the number of objects required to evaluate e_2.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**] Requirements/Specifications;
D.2.2 [**Software Engineering**]: Design tools and techniques.

General Terms

Design, Languages, Algorithm

Keywords

OCL, metric, complexity

1. INTRODUCTION

The role of OCL in the specification of UML models is increasing every day. Mainly, OCL is used to specify invariants, derivation rules and operation contracts in class diagrams, although it is also used in state diagrams, collaboration diagrams, component diagrams, etc. Moreover, OCL plays an important part in the definition of relations and transformations in the recent QVT standard [3].

Even though its importance, OCL tends to be ignored when proposing metrics to measure the size and complexity of UML models. Instead, we believe that when measuring UML models, the complexity of the embedded OCL expressions should be taken into account since they have a direct effect on the complexity required to handle the UML model (for instance, for verification, validation and code-generation purposes)

We are only aware of a couple of approaches devoted to measure OCL expressions. [5] provides a set of metrics based on the syntactic structure of the OCL expression. The metrics take into account the number of model elements (attributes, associations and classes) referenced in the expressions and the kind of constructs used in its definition (number of *if* expressions, number of *oclIsTypeOf* operations, number of variables, and so on). [2] proposes to count the number of *forAll* iterators and the number of navigations appearing in the expression as complexity indicators.

According to both proposals, their main goal when developing the metrics was to help determining the *understandability* of the OCL expressions, i.e. to use the metrics to assist the designer when defining the expressions in order to, for instance, select (when possible) simpler alternatives. We would like to remark that such "*understandability factor"* for OCL expressions has not been proposed yet and it cannot be deduced from the proposed metrics[1]. More research and empirical experiments are still needed.

In this paper we propose an alternative approach. We define the complexity of an OCL expression as the number of objects that must be considered (i.e. accessed) to evaluate the expression. Note that at design-time we do not know the population of the classes and associations of the UML model, and thus, the result provided by our metric is not a number but an abstract formula[2].

We believe that this metric is worth to measure the complexity of the OCL part of an UML model, and specially, to compare, in the context of the MDA, the capabilities of the different MDA and CASE tools with respect to the (efficient) implementation of the OCL expressions in the final technology platform.

The rest of the paper is structured as follows. Next section formalizes the proposed metric. Then, section 3 presents an algorithm that given an OCL expression returns its complexity value according to our metric. Section 4 discusses some scenarios where this metric may be especially useful. Finally, section 5 presents some conclusions and further work.

[1] For instance, is it more complex an expression with two *forAll* iterators and two navigations than another one including four navigations but no *forAll* iterators?

[2] The designer could indicate the expected population for the classes and associations in order to get a numeric value.

2. COMPLEXITY OF AN OCL EXPRESSION

The complexity of an OCL expression *exp* is defined as the number of objects that must be accessed in order to evaluate *exp*. More formally:

Definition 2.1 Let B_{exp} be the bag of objects accessed during the evaluation of an expression *exp*. Given an object *o* of type *t*, $o \in B_{exp}$ if one of the following conditions holds:

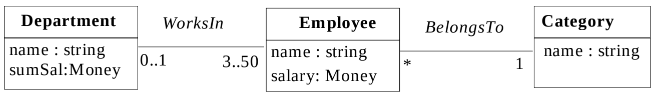
- *o* is the object referenced by a variable included in *exp* (except for iterator variables; objects referenced by these variables are already considered when computing the source expression over which the iterator is applied).
- The subexpression *t.allInstances()* or *super.allInstances()* (where *super* is a supertype of *t*) is included in *exp*
- For each subexpression $exp_i.r$ included in *exp* (where *r* is the name of a role) and for each entity $e_2 \in B_{expi}$, if $e \in e_2.r$ then $e \in B_{exp}$

We would like to remark that B_{exp} is a bag and not a set of objects. This implies that an object is repeatedly counted every time it is accessed during the evaluation of the OCL expression. We believe that the fact of accessing the same objects several times must also influence the complexity measure of the expression (two expressions may involve the same set of different objects but if one of them accesses those objects several times it must be regarded as more complex than the other).

Definition 2.2 Given an expression *exp*, its complexity is defined as $|B_{exp}|$, i.e. the number of objects in B_{exp}.

At design time, we do not know the population of the classes and associations appearing in the UML model, and thus, our complexity metric results in an abstract formula instead of a concrete numeric value. The designer may provide an expected value for every variable appearing in the formula to get a numeric result.

As an example, consider the simple class diagram of Figure 2.1. The body of *ValidSalary* constraint has a complexity value of 1, since its evaluation only involves the *employee* object referenced by *self*. The complexity of the *sumSal* derived attribute is $1 + 1xN_e$ (where N_e represents the number of employees of the department) because when asking for the value of the *sumSal* attribute in a department *d* we need to access *d* and all its related employees. Finally, the complexity of the body of the *RightCategory* constraint is $1 + 1xN_e + 1xN_ex1$ (it involves accessing the department plus its related employees plus the *category* object where each employee is associated to). Note that when navigating from *employee* to *category* we do not need to use an abstract variable to represent the number of related categories for each employee since the multiplicity constraint on the category role forces all employees to be related with exactly a single category.



context Employee **inv** ValidSalary: self.salary<100000
context Department::sumSal:Money **derive**: self.employee.salary->sum()
context Department **inv** RightCategory:
self.employee->select(e| e.category.name='Senior')->size()>=2

Figure 2.1. Example class diagram

Typically, for integrity constraints we are interested in computing the complexity of the whole constraint and not the complexity of its body expression when evaluated over a single instance of the context type of the constraint. We define the complexity of an OCL constraint as the complexity of evaluating its body expression over all instances of the context type of the constraint (for a constraint to be satisfied, all instances of the context type must verify the constraint body).

Definition 2.3. The complexity of an OCL constraint with a body *b* and specified using a context type *t* is equivalent to the complexity of the expression *t.allInstances()->forall(v1| t1.b')* where *b'* is equal to *b* except for that all occurrences of *self* are replaced with the variable *v1*.

According to this definition, the whole complexity of *ValidSalary* is P_ex1 (the result of evaluating the expression *Employee.allInstances()->forAll(e1| e1.salary<100000)*) where P_e is the population of the *Employee* class.

2.1 Maximum and minimum complexity of an OCL expression

From the previous metric we can easily deduce a couple of additional metrics, the maximum and the minimum complexity of an OCL expression. When computing these metrics, we are not interested in defining the exact complexity value but the complexity in the worst (best) possible scenario. Therefore, these metrics are less precise than the previous one but may be helpful when comparing the complexity of different expressions since they provide the complexity range of an OCL expression.

Definition 2.4. The *maximum complexity* of an expression *exp* is an upper limit for the complexity value of *exp* in any possible system state, i.e. independently of the system state at run-time the complexity of evaluating *exp* will never result in a value greater than this maximum complexity value.

Definition 2.5. The *minimum complexity* of an expression *exp* is a lower limit for the complexity of evaluating *exp* in any possible system state, i.e. independently of the system state at run-time the complexity of evaluating *exp* will never be lower than this minimum complexity value.

When computing the maximum complexity of an OCL expression we assume the worst possible scenario. This scenario is the one where all objects participate in as many links as permitted by the maximum multiplicity constraints defined in the associations referenced in the OCL expression. Therefore, the key idea when calculating this value is to replace from the formula computed in the previous section all variables representing the number of objects retrieved when navigating through a role (as the variable N_e in the examples above) with the maximum value specified in that role. When the maximum multiplicity is not specified, the population of the destination class is taken as a maximum value.

Similarly, when computing the minimum complexity we assume the best scenario, i.e. the one where all objects participate in as few links as permitted by the minimum multiplicity constraints of the associations. When a minimum multiplicity is not specified is considered to be zero.

As an example, consider again the class diagram of Figure 2.1. The maximum complexities of the previous OCL expressions are:

- Body of *ValidSalary:* 1 (no roles are traversed)
- Derivation of *sumSal*: *1 + 1x50 = 51* (N_e is replaced with 50 is the maximum multiplicity of the employee role in the *WorksIn* association)
- Body of *RightCategory*: *1 + 1x50+ 1x50 = 101*

and the minimum complexities are:

- Body of *ValidSalary*: 1
- Derivation of sumSal: *1 + 1x3* = 4
- Body of *RightCategory*: *1 + 1x3 + 1x3* = 7

3. AN ALGORITHM FOR DETERMING THE COMPLEXITY OF AN EXPRESSION

In this section we propose an algorithm to determine the complexity value of an OCL expression according to our metric. The algorithm copes with a subset of the full OCL language representative enough to show the feasibility of computing our metric. Collect operations [4] resulting in a collection of objects (as the *union* or *intersection* operations) are not handled. Other expressions (like *if-else* expressions) are assumed to be expressed using more basic constructs (e.g.: *if x then y* can be expressed as *x implies y*).

The algorithm assumes that the OCL expression is represented as an instance of the OCL metamodel [4]. This metamodel representation may be easily obtained by means of parsing the concrete (textual) representation of the OCL expression.

The basic structure of the OCL metamodel (Figure 3.1) consists of the metaclasses *OCLExpression* (abstract superclass of all possible OCL expressions), *VariableExp* (a reference to a variable, as, for example, the variable *self*), *IfExp* (an if-then-else expresion), *LiteralExp* (constant literals like the integer '1') and *PropertyCallExp* which is a supertype for the metaclasses *ModelPropertyCallExp* (expressions referring to model elements) and *LoopExp* (iterator expressions).

ModelPropertyCallExp (Figure 3.2) can be split in *AttributeCallExp* (a reference to an attribute), *NavigationCallExp* (a navigation through an association end or an association class) and *OperationCallExp*. This later class is of particular importance, because its instances are calls to operations defined in any class of the CS. This includes all the predefined operations of the types defined in the OCL Standard Library, such as the add operator ('+') or the 'and' operator.

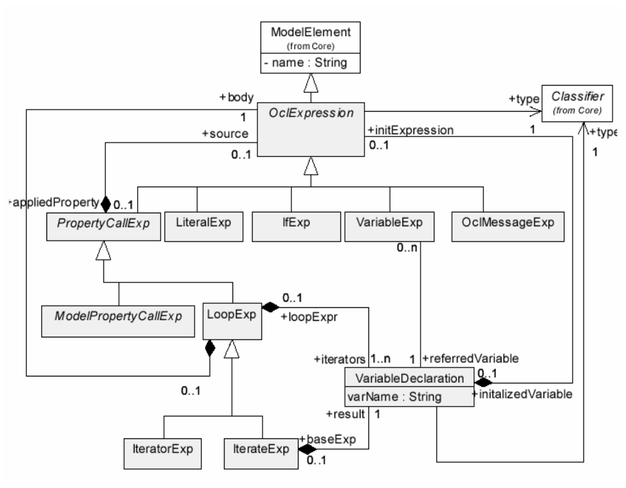


Figure 3.1. Basic structure of the OCL metamodel

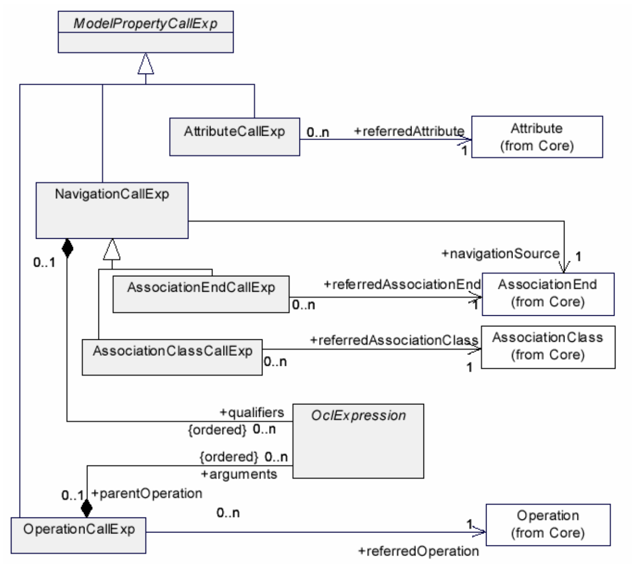


Figure 3.2. OCL metamodel fragment for *ModelPropertyCallExp*

When expressing a constraint as an instance of the OCL metamodel, the body of the constraint can be regarded as a binary tree where each node represents an atomic subset of the OCL expression defining the constraint body (an instance of any metaclass of the OCL metamodel: an operation, an access to an attribute or an association …).

The root of the tree is the most external operation of the OCL expression. The left child of a node is the source of the node (the part of the OCL expression previous to the node). The right child of a node is the body of an iterator expression if the node represents one of the predefined iterators defined in the OCL standard (a *forAll*, *select*…) or the argument of the operation if the node represents a binary operation (such as '>', union, '+',…). In this latter case, the source can be regarded as the first operand of the operation.

We show in Figure 3.3 the body of the *RightCategory* constraint as an instance of the OCL metamodel (for the sake of clarity we show a slightly simplified version of this representation, where we combine in the same tree node the kind of OCL subexpression and the name of the model element referenced by the node).

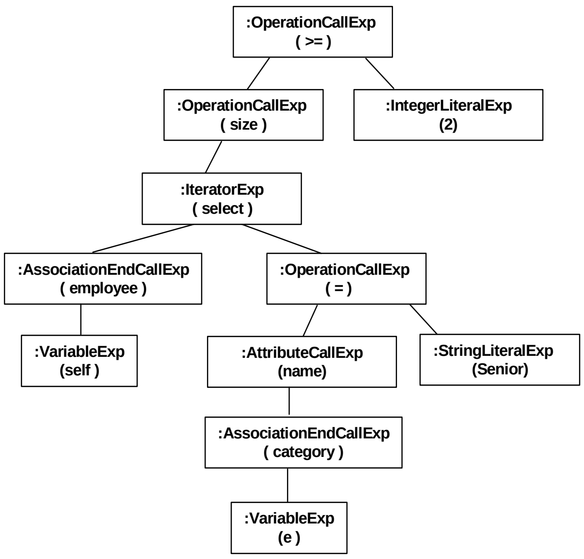


Figure 3.3. *RightCategory* as an instance of the OCL metamodel

According to this tree representation, we present an algorithm for computing the complexity of an OCL expression. First, the algorithm traverses the tree in postorder. Then, for each node, it takes into account the type of the node to compute the complexity up to that node and propagates this value to the parent node.

Algorithm: Complexity value of an OCL expression

```
Complexity(node: OCLExpression, aux:Value) : Value
    cL:=0; cR:=0;
    if (node.leftChild<>null) cL := complexity(node.leftChild);
    if (node.rightChild<>null) cR := complexity(node.rightChild);
    if (node instanceof LiteralExp) {aux:=0; compl:=0;}
    if (node instanceof VariableExp) { aux:=1;
        if (node.referredVariable.loopExpr<>null) compl:=1
      else compl:=0;}
   if (node instanceof AttributeCallExp) {
      aux:=node.leftChild.aux; compl:=cL;}
    if (node instanceof IteratorExp) {
       aux:= node.leftChild.aux;
      compl:= cL + node.leftChild.aux x cR; }
   if (node instanceof AssociationEndCallExp) {
      if(node.referredAssociationEnd.multiplicity='1') {
         aux:=node.leftChild.aux;
         compl:=cL+ node.leftChild.aux; }
      else{
       aux:= node.leftChild.aux x $N_{node.referredAssociationEnd.name}$;
      compl:= cL + node.leftChild.aux x $N_{node.referredAssociationEnd.name}$ ; }
   }
   if (node instanceof OperationCallExp)
         if (node.name='allInstances') {
            aux:= $P_{node.type}$ ; compl:=$P_{node.type}$}
         else  // operations like '+', '-', 'and', …
            aux:=0; compl:=cL + cR; }
   return compl;
```

The algorithm uses two main variables: *compl* and *aux*. *Compl* stores the accumulated complexity value. For a node, its *compl* value indicates the complexity of evaluating the subtree under that node. Instead *aux* is used to compute the number of objects returned when evaluating the expression corresponding to the subtree, necessary to determine the complexity value for some node types. Roughly, the *aux* value of an expression *exp* coincides with the result that would be returned by the expression *exp->size()* (for non-literal expressions).

For the sake of simplicity we assume that all kind of nodes have a *leftChild* and *rightChild* association with children nodes. Nodes with a single child have a null value in the *rightChild* association. These associations correspond to the *source* and *arguments* associations for nodes of type *PropertyCallExp*. For literal and variable expressions these associations return a null value.

As an example, when the node is a constant literal expression the *compl* and *aux* values of that node is zero (no objects are accessed). When the node is an *allInstances* operation, *compl* and *aux* take the formula P_x as a value, where *x* is the name of the type. When the node is an association end, the complexity value is defined as the complexity of the child node *ch* plus the added complexity induced by the current navigation, expressed as the set of objects accessed when applying the navigation over the bag of objects retrieved when evaluating *ch*. This later value is represented by the formula *node.leftChild.aux x* N_y where *y* is the name of the referenced association end. Note that, when *node.leftChild.aux* is greater than 1, N_y represents the average number of objects accessed when applying *y* over each object in *node.leftChild.aux*. The exact value would be *node.leftChild.aux x* N_{y1} + … + *node.leftChild.aux x* N_{yn}.

Figure 3.4 shows the application of this algorithm over the body of the constraint *RightCategory*. Next to each node we add its *compl* and *aux* values (we use N_e as a shorthand for $N_{employee}$). Obviously, the *compl* value of the root node corresponds to the complexity value of the whole OCL expression.

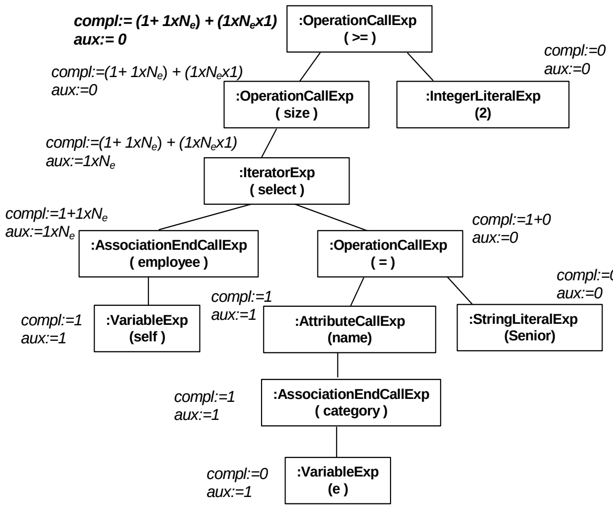


Figure 3.4. Calculating the complexity of *RightCategory*

4. APPLICATION SCENARIOS

We believe this metric is useful to measure the complexity of the OCL part of an UML model.

Besides, the metric is also useful to help designers take design decisions regarding the model. For instance, several refactoring operations have been proposed to improve the design and structure of UML models (see, for instance, [6]). Some of them induce a change in the definition of existing OCL expressions. For instance, moving an attribute *at* from a class *A* to a class *B* affects all expressions referencing *at*. The new expressions may be more (or less) complex than the original ones. This added (reduced) complexity should be taken into account when deciding whether to apply the refactoring operation.

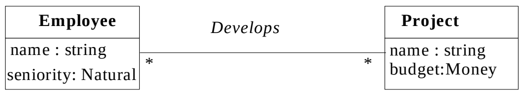
Another example is the design of derived elements. In general, there are two different approaches to deal with derived elements as far as the implementation of the system is concerned: they can be either computed by means of an operation whenever the value of the derived element it is requested or they can be materialized by explicitly storing its contents as part of the system state. A key aspect when deciding between both options is the complexity of the OCL expression used as derivation rule for the element. The more complex the rule, the better would be to materialize the derived element to reduce the response time when requesting its value.

As pointed out in the introduction, this metric is also especially useful in an MDA context to compare the efficiency of the code generated by the different MDA tools in order to implement the OCL expressions appearing in the UML model. As an example, consider the problem of verifying that the system state satisfies the integrity constraints defined in the model. MDA tools present different efficiency grades when evaluating the constraints after changes (inserts of new objects, update of attributes,…) on the system state [1].

Assuming the class diagram of Figure 4.1, after the creation of a new link between an employee *e* and a project *p*, some tools verify that all projects (and not only *p)* verify the constraint. This means that they verify the expression *Project.allInstances()->forAll(p|p.budget>1000 implies p.employee->forAll(e| e.seniority>2))*.

Other tools verify that all employees related with *p* (and not just *e)* satisfy the seniority condition (i.e. they check the expression *p.budget>1000 implies p.employee->forAll(e| e.seniority>2)*). Finally some tools just check that the new relationship between *e* and *p* is consistent with the constraint (i.e. they evaluate the expression *p.budget>1000 implies e.seniority>2*)

Measuring the complexity of the different OCL expressions actually evaluated by the system implementation generated by each tool permits to classify their efficiency with respect to the treatment of the OCL expressions included in the UML models. In this example, the complexity varies from $P_p+P_p x N_e$ (where P_p represents the number of instances of the *Project* class and N_e the average number of employees per project) to $1+ 1x\ N_e$ and to $1+1$.



context Project **inv** ValidEmployee:
self.budget>1000 implies self.employee->forAll(e| e.seniority>2)

Figure 4.1. Example class diagram

5. CONCLUSIONS

We have presented a new metric to measure the complexity of OCL expressions and discussed some possible applications for it. However, we believe that more research is still needed to complete a map of metrics for OCL expressions. Different metrics may be necessary depending on the designer's needs. Another open issue is to establish a correspondence between the values of the metrics and the effort required, for instance, to implement or understand them.

As a further work, we plan to complete the algorithm presented in section 3 and to integrate it with an existing CASE tool. Then we will be able to develop some empirical experiments to study the applicability and usefulness of our metric in real projects. We also plan to improve the precision of our maximum and minimum complexity metrics by taking the semantics of the expression into account during their computation. For instance, for expressions of type *X implies Y* the minimum value should be computed considering that *X* always evaluates to false (best scenario), and thus, that objects in *Y* does not influence the complexity value of the expression.

6. ACKNOWLEDGEMENTS

This work has been partly supported by the Ministerio de Educacion y Ciencia under project TIN 2005-06053.

7. REFERENCES

1. Cabot, J. and Teniente, E., Constraint Support in MDA tools: a Survey. in *2nd European Conference on Model Driven Architecture*, LNCS, 4066, (2006), 256-267.
2. Ledru, Y., Dupuy-Chessa, S. and Fadil, H. Towards computer-aided design of OCL constraints. in *CAiSE'04 Workshops Proceedings, Vol. 1*, Faculty of Computer Science and Information Technology, Riga Technical University, Riga, Latvia, 2004, 329-338.
3. OMG. Revised submission for MOF 2.0 Query/Views/Transformations RFP, 2005.
4. OMG. UML 2.0 OCL Specification. OMG Adopted Specification (ptc/03-10-14), 2003.
5. Reynoso, L., Genero, M. and Piattini, M. Towards a metric suite for OCL Expressions expressed within UML/OCL models. *Journal of Comptuer Science and Technology*, *4* (1). 38-44.
6. Sunyé, G., Pollet, D., Traon, Y.L. and Jézéquel, J.-M., Refactoring UML Models. in *4th Int. Conf. on the Unified Modeling Language (UML'01)*, LNCS, 2185, (2001), 134-148.