Electronic Communications of the EASST
Volume 5 (2006)

Proceedings of the Sixth OCL Workshop OCL for (Meta-)Models in Multiple Application Domains (OCLApps 2006)

Improving the OCL Semantics Definition by Applying Dynamic Meta Modeling and Design Patterns

Juan Martin Chiaradía and Claudia Pons

19 Pages

Guest Editors: Dan Chiorean, Birgit Demuth, Martin Gogolla, Jos Warmer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Improving the OCL Semantics Definition by Applying Dynamic Meta Modeling and Design Patterns

Juan Martín Chiaradía [1] **Claudia Pons** [1,2]

[1] LIFIA – Facultad de Informática, Universidad Nacional de La Plata

[2] CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)

La Plata, Buenos Aires, Argentina

{jmchiara,cpons}@sol.info.unlp.edu.ar

Abstract. OCL is a standard specification language, which will probably be supported by most software modeling tools in the near future. Hence, it is important for OCL to have a solid formal foundation, for its syntax and its semantic definition. Currently, OCL is being formalized by metamodels expressed in MOF, complemented by well formedness rules written in the own OCL. This recursive definition not only brings about formal problems, but also puts obstacles in language understanding. On the other hand, the OCL semantics metamodel presents quality weaknesses due to the fact that certain object-oriented design rules (patterns) were not obeyed in their construction. The aim of the proposal presented in this article is to improve the definition for the OCL semantics metamodel by applying GoF patterns and the dynamic metamodeling technique. Such proposal avoids circularity in OCL definition and increases its extensibility, legibility and accuracy.

Keywords: OCL; formal semantics; dynamic meta modeling; design patterns.

1 Introduction

OCL (Object Constraint Language) [1] is a formal specification language, accepted as a standard by the OMG (Object Management Group). OCL is a three-valued Kleene-Logic with equality that allows for specifying constraints on graphs of object instances whose structure is described by UML class diagrams[2], thus extending the expressive capacity of such notation. OCL is intended to be a practical formalism, addressing software developers who do not have a strong mathematical background. For that reason, OCL deliberately avoids mathematical notation; instead of symbols it uses a programming language oriented syntax and attempts to hide concepts such as logical quantifiers.

A fundamental requirement for any formalism is that it should count with a rigorous definition of both its syntax and its semantics. The recently adopted OCL 2.0 specification provides a formal definition of the OCL semantics (see official OCL semantics, appendix A in [1]) following the denotational approach. Such semantics is based on set theory with the notion of an object model, which is basically a formalization of UML Class Diagrams [4]. OCL expressions are interpreted by functions over environments, in the classical way [3]. Another approach to specify the OCL semantics that can be found in the literature consists in defining an embedding of OCL into other logics [5].

These two approaches have succeeded in describing the evaluation of OCL constraints in a formal, non-ambiguous manner providing established technologies for abstract reasoning, automatic verification, execution, or simulation of models; however they are not especially suited for explaining the semantics to people with modest mathematical background.

Due to the fact that the purpose of the semantics is to provide a common understanding of the formalism among its users, those mathematically rigorous definitions, which are not readable for a wide range of OCL users, are of little help. In the last years the academic community accepted that the semantics should be given in formalisms OCL users are familiar with, for example metamodeling. Adhering to this trend, the OCL 2.0 specification provides a semantics definition based on MOF metamodels (see chapter 10 of [1]) complemented by well formedness rules written in the own OCL. However, such circular definition not only gives rise to formal problems [6], but also puts obstacles in language understandings. Additionally, having into account the dynamic nature of semantics evaluation, it seems reasonable to think that dynamic meta-modeling techniques, rather than static meta-classes should be used to define the OCL semantics.

Working towards the solution to this problem, we propose to create a clearer and simpler alternative definition for the OCL semantics by giving a dynamic meta-model which is specified using a simple form of UML collaboration diagrams and applying well established design patterns [7].

The paper is organized as follows; in Section 2 we present a summary of the current OCL semantics [1] to provide an adequate context to the reading of this proposal. Then, in section 3, we propose a new definition for the OCL semantics, based on the Dynamic Meta Modelling technique (DMM) [8] [9]. In section 4 we apply the Visitor pattern [7] on the semantics metamodels. Finally, in Section 5, we present conclusions and future works.

2 OCL Specification Overview

An OCL expression is defined in [1] as "an expression that can be evaluated in a given environment" Additionally the specification in [1] states that “evaluation of the expression yields a value". Taking it into account, the ‘meaning’ (semantics) of an OCL expression can be defined as the value yielded by its evaluation in a given environment.

Figure 1 shows an overview of the UML based specification of the OCL syntax and semantics presented in [1].

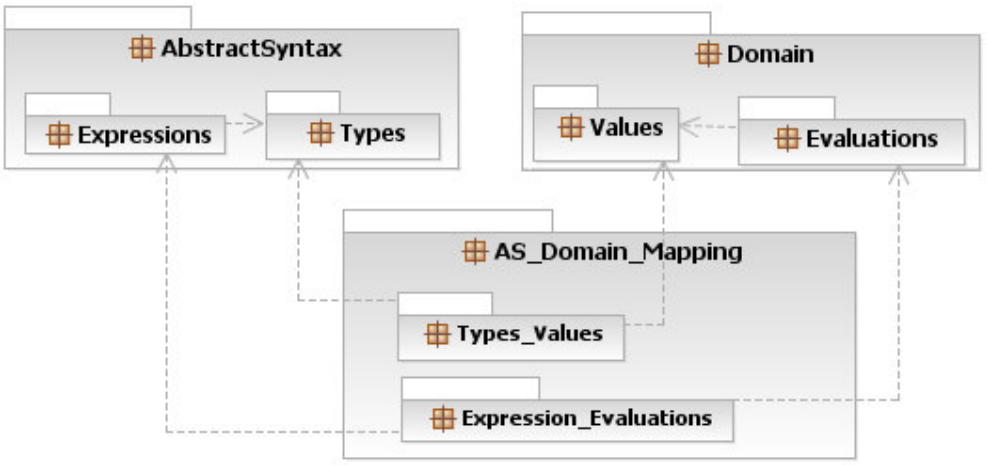

Figure 1: Overview of packages in the UML-based semantics

Figure 2 shows the overview of the AbstractSyntax package, which defines the abstract syntax of OCL as a hierarchy of meta classes. On the other hand, Evaluations package defines the semantics of these expressions using also a hierarchy of meta classes where each one represents an evaluation of a particular kind of expression (see figure 3). The idea behind this representation is that each evaluation yields a result in a given environment, therefore, the semantics evaluation of an expression in a specific environment is given by associating each evaluation instance to an expression model (see figure 4).

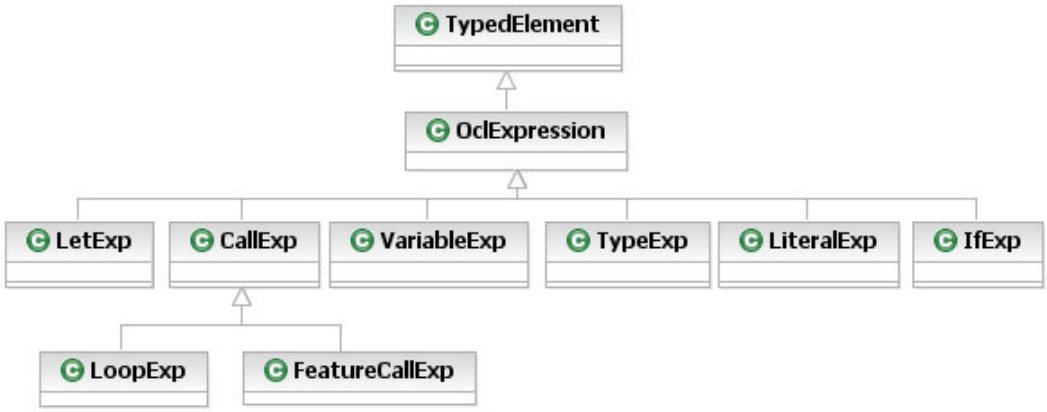

Figure 2: AbstractSyntax package overview

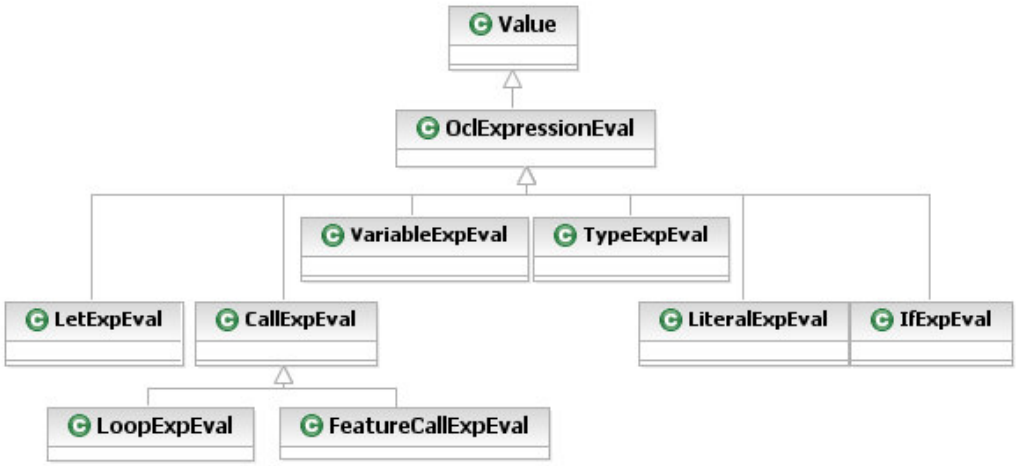

Figure 3: Evaluations package overview

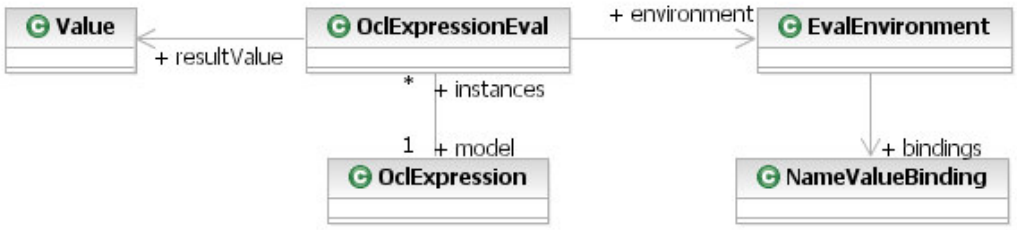

Figure 4: Semantics Evaluation of OCL expressions.

The Evaluations package replicates the hierarchy of the abstract syntax. We believe that such duplication should be avoided since it hinders the legibility of the meta model and the efficiency in the development of automatics tools based on this semantics. We will expand on this issue in the next section.

3 Semantics evaluation via Dynamic Meta Modeling

In this section we extend the Abstract Syntax structure by defining a new operation which will give semantic meaning to syntax expressions by associating them with their corresponding values.

The standard maths semantics of OCL ([1] Appendix A) states:

> *A context for evaluation is given by an environment* $\tau = (\sigma, \beta)$ *consisting of a system state* σ *and a variable assignment* β*:* $Var_t \rightarrow \mathbf{I}(t)$*. A system state* σ *provides access to the set of currently existing objects, their attribute values, and association links between objects. A variable assignment* β *maps variable names to values.*
>
> *Let Env be the set of environments* $\tau = (\sigma, \beta)$*. The semantics of an OclExp is a function* $\mathbf{I}[[e]] : Env \rightarrow I(t)$ *which binds each syntactic expression e with a value in* $\mathbf{I}(t)$*.*

Using this maths semantics as a foundation, we define an operation *evalOn()* which is expected to represent $\mathbf{I}[[e]]$*:Env* $\rightarrow \mathbf{I}(t)$*. EvalOn()* takes an EvalEnvironment as input parameter and returns a Value (i.e., **context** `OclExpression` **def:** `evalOn(env:EvalEnvironment):Value`). This operation acts like a bridge between AbstractSyntax and Values packages replacing the whole Evaluations package (see figure 5).

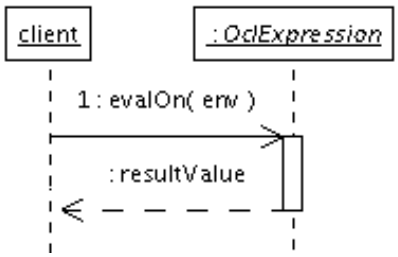

Figure 5: *evalOn()* over *OclExpression* structure.

In addition, we believe that the best way to understand the semantics evaluation is by showing the evaluation process itself. By using only class diagrams to reflect the semantics evaluation, it is hard to reveal the latter process, because of the static nature inherent to these diagrams. Because of that, to completely understand all the process it is necessary to pay attention to the constraints established on these diagrams. In [1], these constrains are written in OCL with two negatives outcomes:

- The expressibility and simplicity obtained by using UML in the semantics metamodel over the math one is lost because of the necessity of be aware of the constraints to fully understand the semantic.

- The constraints are written in OCL, so that the semantics of OCL is defined in terms of OCL itself! If someone didn't understand OCL, they would neither understand these constraints (see for example the *IterateExp* semantics in [1])

Consequently, with the aim of producing a simple, precise and clear explanation, in this section we use sequence diagrams to visualize the distinct steps throughout the semantics evaluation of expressions. Each meta-class belonging to the Domain package will be replaced with a sequence diagram which states the concrete semantics and evaluation process of the corresponding syntactic construction. This approach is known as Dynamic Meta Modelling (DMM) [8] [9], and has been used in the semantics specification of UML elements (such as State Machines and Collaborations), but its use in OCL specification has not been explored before.

Taking advantage of the classification proposed in [10] we categorize the OCLExpressions in: *Atomic Expressions, Navigation Expressions, OCL Predefined Operations* and *Iterator Expressions,* adding a "new" category to this list that we call OCL *Language Expressions*. In the following sections we show one or two particular expression for each category.

3.1 Semantics of Atomic Expressions

This category consists of expressions such as *LiteralExp* and *VariableExp* that do not have any subexpressions.

Because of the simplicity of these two kind of expressions, we only show our DMM semantics for *IntegerLiteralExp* ($\boldsymbol{I}[[Integer]] = Z \cup \perp$ in figure 6) and *VariableExp* ($\boldsymbol{I}[[v]](r) = \beta(v)$ in figure 7). A deeper discussion on these semantics can be found in [1].

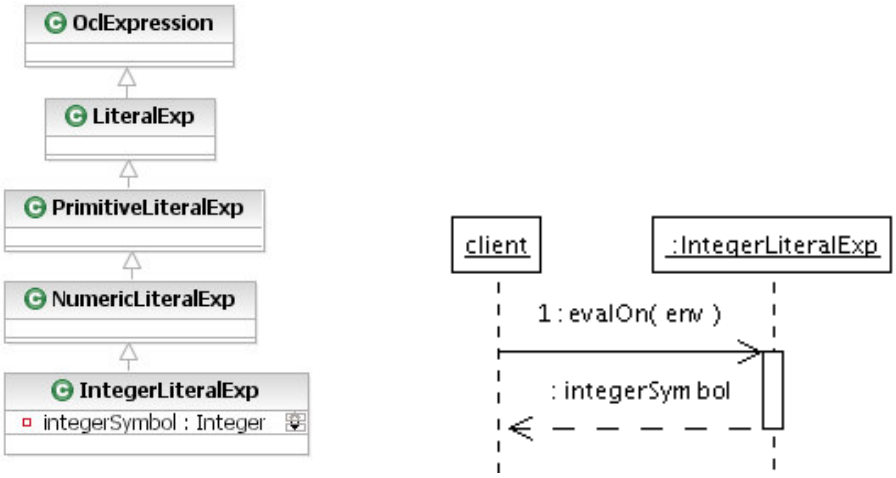

Figure 6: AS and *evalOn()* over *IntegerLiteralExp*.

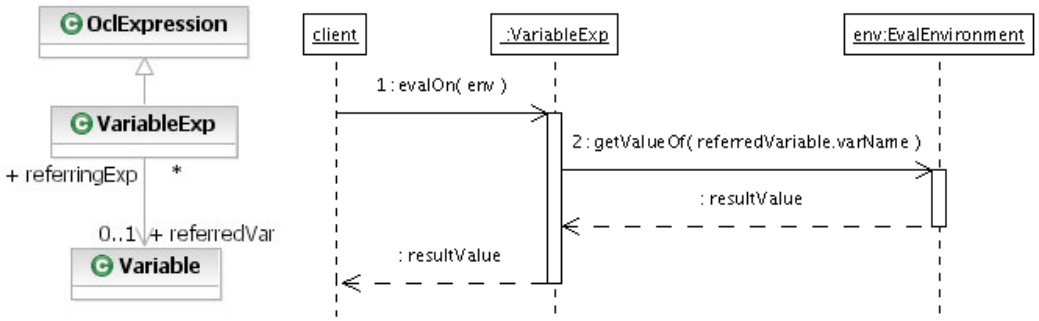

Figure 7: AS and *evalOn()* over *VariableExp*.

3.2 Semantics of OCL Language Expressions

This category consists of expressions such as *LetExp* and *IfExp* that are predefined constructions of the language. With the aim of highlighting the benefits obtained using this new "translation" of the maths semantics, we remark the differences between the standard UML based semantics and the semantics of a *LetExp* proposed in this paper. The standard UML based evaluation of a *LetExp* proposed in [1] is shown in figure 8. The diagram shows how the evaluation encapsulates the result value and the evaluation environment, although neither the evaluation method nor the structural constraints are specified on this diagram.

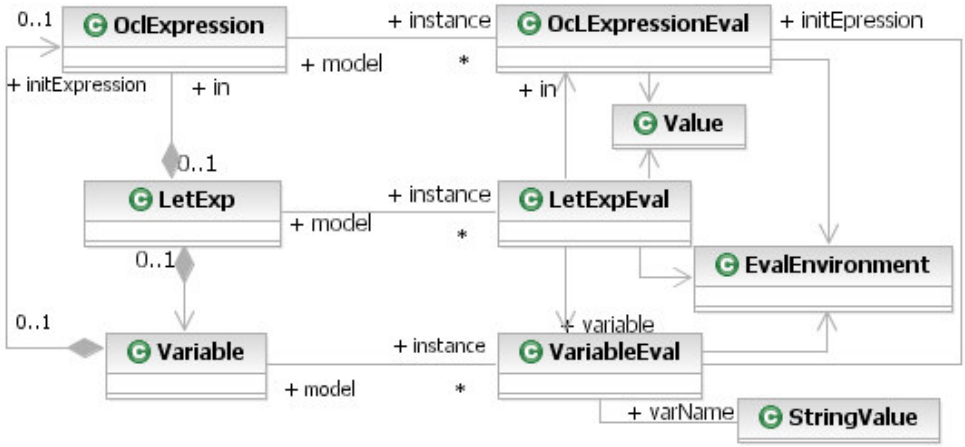

Figure 8: Standard UML based semantic evaluation of a *LetExp*.

A simple analysis of the last diagram does not give us enough information about the semantics of a *LetExp*; it only gives us information about the static structure of the elements implied in this evaluation. In order to fully understand the previous diagram, we must study its well formed rules expressed in OCL itself.

The Maths semantics of a *LetExp* as expressed in the Appendix A of [1] is shown in figure 9.

$$\boldsymbol{I}[[\text{let v} = e_1 \text{ in } e_2]](\tau) = \boldsymbol{I}[[e_2]](\sigma,\beta\{v/\boldsymbol{I}[[e_1]](\tau)\})$$

Figure 9: Standard Math based semantic of a *LetExp*.

At this moment, we can translate this algorithm under the applicative order reduction into a sequence diagram (see figure 10). As a first step of evaluation, we evaluate the init expression ($\boldsymbol{I}[[e_1]](\tau)$, signal 2) to get a new evaluation environment which extends the previous one with the latter evaluation ($\beta\{v/\boldsymbol{I}[[e_1]](\tau)\}$, signals 3 and 4). Then, we evaluate the in expression in the new environment, and the value returned by this later evaluation is the result of the whole *LetExp* evaluation ($\boldsymbol{I}[[e_2]](\sigma,\beta\{v/\boldsymbol{I}[[e_1]](\tau)\})$, signal 5).

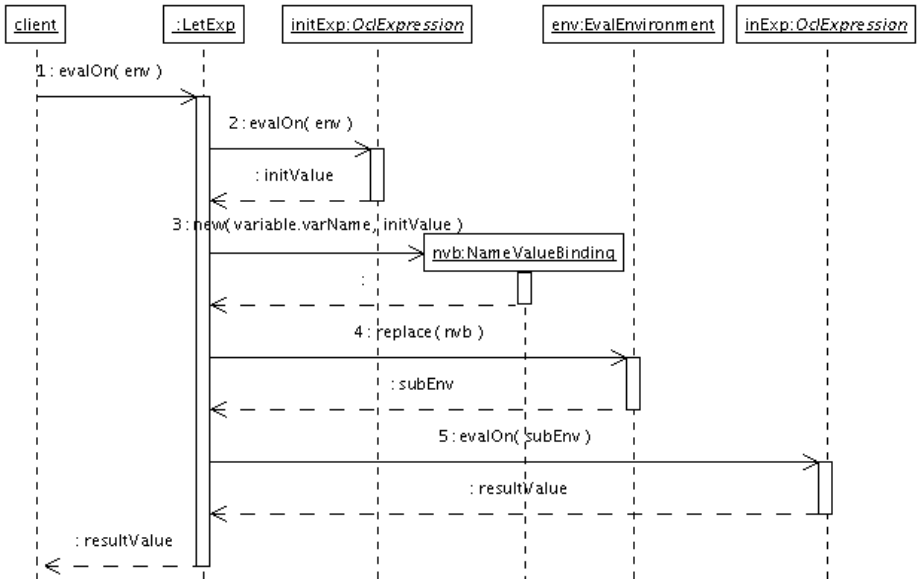

Figure 10: Sequence diagram of a *LetExp* evaluation.

3.3 Semantics of OCL Predefined Operations

As defined in [10], expressions from this category are instances of the metaclass *OperationCallExp* where the called operation is a predefined one, such as +, =.

Figure 11 shows the AS of these expressions. The semantics is given with a different scenario for each predefined operation. In particular we show the scenario corresponding to the maths semantics of equality operation in figures 12 and 13.

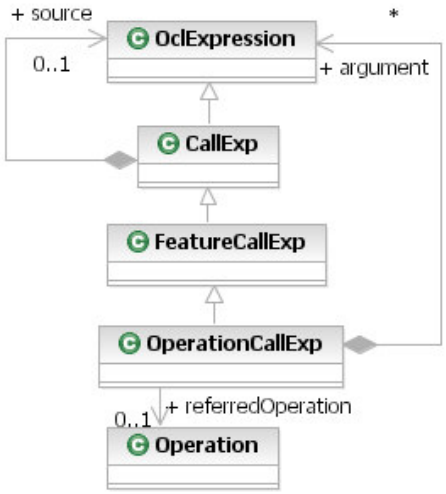

Figure 11. AS of *OperationCallExp*

$$I(=_t)(v_1, v_2) = \begin{cases} \text{true} & \text{if } v_1 = v_2, \text{ and } v_1 \neq \bot \text{ and } v_2 \neq \bot, \\ \bot & \text{if } v_1 = \bot \text{ or } v_2 = \bot, \\ \text{false} & \text{otherwise.} \end{cases}$$

Figure 12. Maths semantic of equality operation

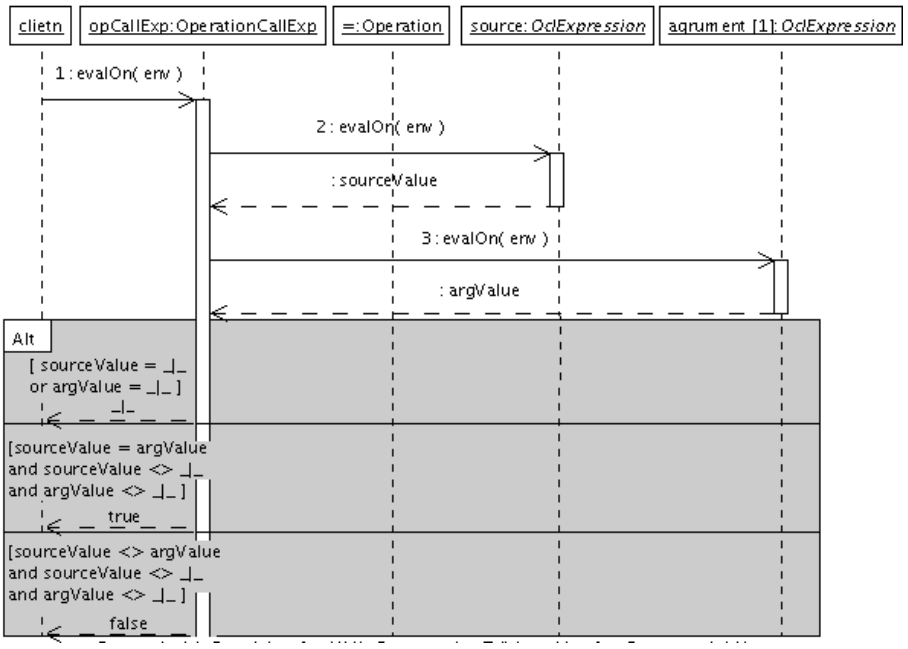

Figure 13. evalOn() over equality operation

3.4 Semantics of Navigation Expressions

OCL expressions of this category are instances of *PropertyCallExp* and *AssociationEndCallExp*. Such expressions are evaluated by 'navigating' from the object, to which the source expression is evaluated, to that element in the object diagram, which is referenced by the attribute or association end. We focus our example in the *PropertyCallExp*, named *AttributeCallExp* in the maths semantics stated in [1]; this semantics can be seen in figure 14.

$$I_{ATT}(a : t_c \to t)(\underline{c}) = \begin{cases} \sigma_{ATT}(a)(\underline{c}) & \text{if } \underline{c} \in \sigma_{CLASS}(c), \\ \bot & \text{otherwise} \end{cases}$$

Figure 14. Maths semantics of *AttributeCallExp*

Based on this semantics and using the AS of *PropertyCallExp* (figure 15) we construct the evalOn() as is shown in figure 16. The getCurrentValueOf() operation is a predefined operation over ObjectValue which returns either the value attached to the attribute name or OclUndefined if such value is not found.

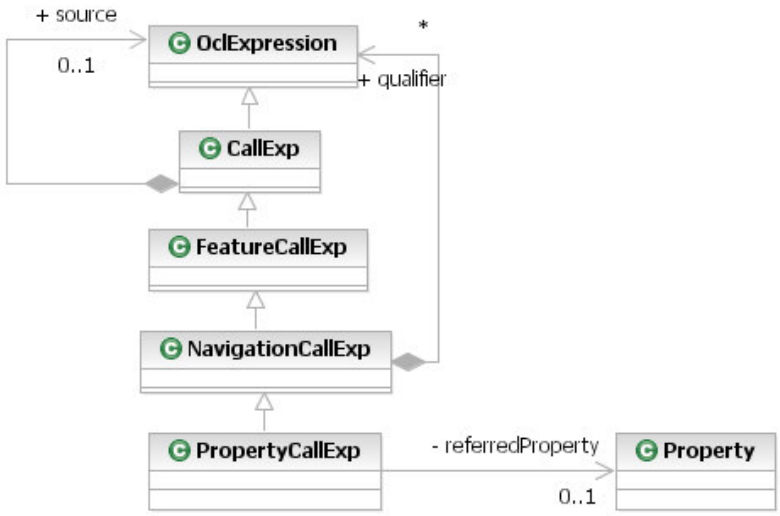

Figure 15. AS of *PropertyCallExp*

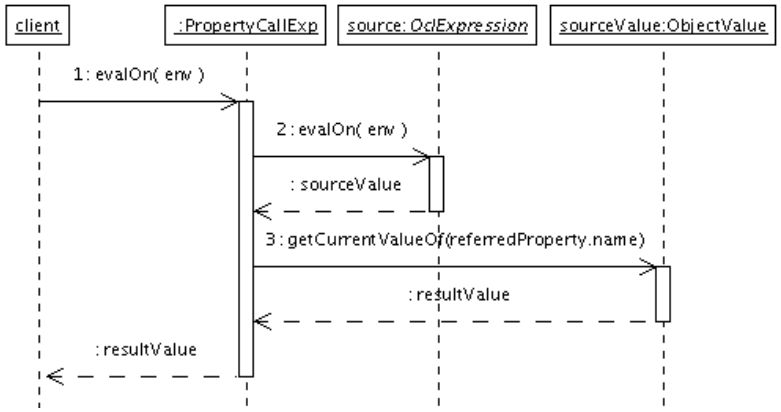

Figure 16. *evalOn()* over *AttributeCallExp*

3.5 Semantics of Iterator Expressions

Iterator expressions are the predefined operation over any Collection in OCL: *select(), reject(), forAll(), iterate(), exists(), collect()* or *isUnique()*. Since all these expressions can be expressed

by macros based on *iterate()*, it would be sufficient to refer for their semantics just to the semantics of iterate .

We will remark the differences between the standard UML based semantics ([1]) and our specific semantics of an *IterateExp*. The semantics evaluation of an *IterateExp* as is expressed in [1] is shown in figure 17. Once again we face the problem that such static diagram does not transmit enough information about the semantics evaluation process and we have to appeal to the well formedness rules established on this diagram [1]; without these constraints we would be unable to completely understand the semantic process of an *IterateExp*.

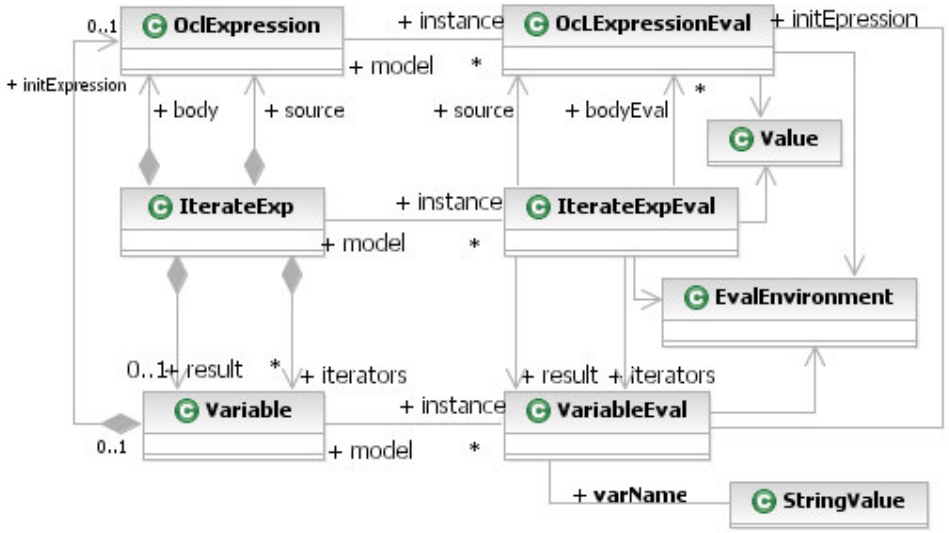

Figure 17: Standard UML based semantic evaluation of an *IterateExp*.

Even worse, such constraints try to explain how *IterateExp* works but lacks of correctness due to the fact that the *IterateExp* is defined in terms of a *ForAllExp* wich is itself defined in terms of *IterateExp*, as follows:

The environment of any sub evaluation is the same environment as the one from its previous sub evaluation, taking into account the bindings of the iterator variables, plus the result variable which is bound to the result value of the last sub evaluation.

```
context IterateExpEval inv:
let SS: Integer = source.value->size()
in if iterators->size() = 1
  then Sequence{2..SS}->forAll(i:Integer | bodyEvals-
  >at(i).environment = bodyEvals->at(i-1).environment-
  >replace(NameValueBinding(iterators->at(1).varName,
  source.value->asSequence()->at(i)))-
  >replace(NameValueBinding(result.varName,bodyEvals->at(i-
  1).resultValue )))
```

```
    else -- iterators->size() = 2
    Sequence{2..SS*SS}->forAll(i: Integer | bodyEvals-
    >at(i).environment = bodyEvals->at(i-1).environment->replace(
    NameValueBinding( iterators->at(1).varName,source->asSequence()-
    >at(i.div(SS) + 1)))->replace( NameValueBinding( iterators-
    >at(2).varName,source.value->asSequence()->at(i.mod(SS))))-
    >replace( NameValueBinding(result.varName,bodyEvals->at(i-
    1).resultValue )))
  Endif
```

If the reader has not a previous knowledge on OCL, it is clear that the previous constraint is almost impossible to understand, and with the proper knowledge of the language, the reading and comprehensiveness of these constraints is a hard task to do.

A summary of the math semantics is shown in figure 18 (see Appendix A of [1] for the full version), while figure 19 and figure 20 display the semantics expressed via sequence diagrams.

$$\boldsymbol{I}[[\, e_1 \rightarrow \text{iterate}\ (v_1; v_2 = e_2 \mid e_3)]]\ (\tau) = \boldsymbol{I}[[\, e_1 \rightarrow \text{iterate'}\ (v_1 \mid e_3)\ ]]\ (\tau')$$

Where $\tau' = (\sigma, \beta')$ and $\tau'' = (\sigma, \beta'')$

$\beta' := \beta\{\ v_2 / \boldsymbol{I}[[\, e_2]]\ (\tau)\ \}$

$\beta'' := \beta'\{\ v_2 / \boldsymbol{I}[[\, e_3]]\ (\sigma, \beta'\ \{v_1 / x_1\})\ \}$

and if $e_1 \in \text{Expr}_{\text{Sequence}(t1)}$, iterate' is defined as:

$$\boldsymbol{I}[[\, e_1 \rightarrow \text{iterate'}\ (v_1 \mid e_3)\ ]]\ (\tau') = \begin{cases} \boldsymbol{I}[[\, v_2\ ]]\ (\tau') \\ \qquad \text{if}\ \ \boldsymbol{I}[[\, e_1\ ]]\ (\tau') = \langle\rangle \\ \boldsymbol{I}[[\, \text{mkSequence}_{t1}\ (x_2, \ldots, x_n) \rightarrow \text{iterate'}\ (v_1 \mid e_3)\ ]]\ (\tau'') \\ \qquad \text{if}\ \ \boldsymbol{I}[[\, e_1\ ]]\ (\tau') = \langle x_1, \ldots, x_n \rangle \end{cases}$$

Figure 18: Maths semantics of *IterateExp*

The *IterateExp* evaluation is defined as follows:

The first sub evaluation will start with an environment in which the result variable is bound to the init expression (*initExp*) of the variable declaration in which it is defined ($\beta' := \beta\{\ v_2 / \boldsymbol{I}[[\, e_2]]\ (\tau)\ \}$, signals 2, 3 and 5 in figure 19); then we proceed to evaluate the body with all iterator variables bound to the different combinations of the source (figure 17). The iterators binding ($\beta'\{v_1/x_1\}$ in $\beta'' := \beta'\{v_2 / \boldsymbol{I}[[e_3]]\ (\sigma, \beta'\ \{v_1 / x_1\})\ \}$) is realized by CombinationGenerator (signals 7 and 8 in figure 20), under a 'depth first search' strategy. This strategy determines the number of sub evaluations over the body ($\boldsymbol{I}[[\, e_3]]\ (\sigma, \beta'\ \{v_1 / x_1\})$ in $\beta'' := \beta'\{\ v_2 / \boldsymbol{I}[[\, e_3]]\ (\sigma, \beta'\ \{v_1 / x_1\})\ \}$; signal 9 figure 20); as last step, these sub evaluations will update the result variable (signals 10 and 11 in figure 20) which is returned as the result value of the whole evaluation process ($\boldsymbol{I}[[\, v_2\ ]]\ (\tau')$, signal 12 in figure 19).

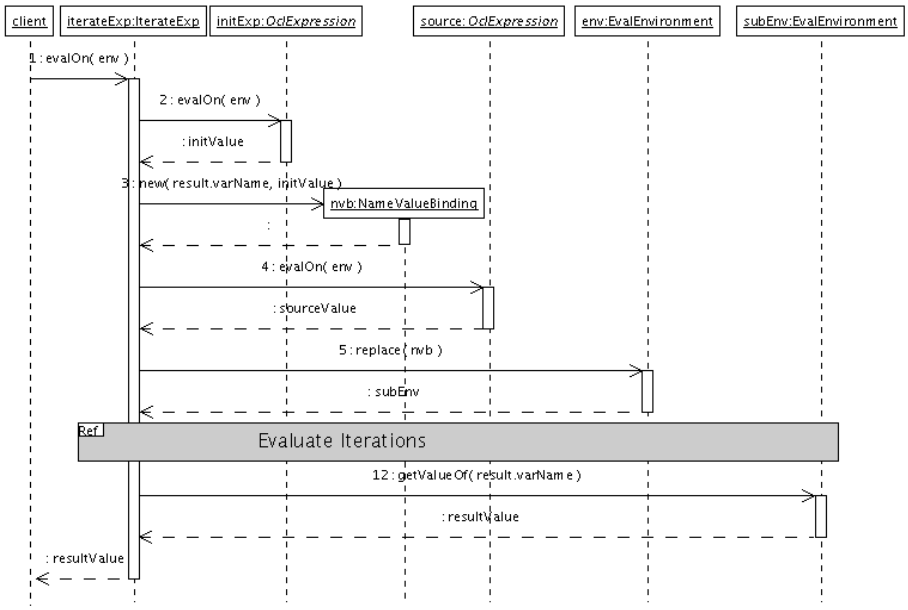

Figura 19: *IterateExp* semantics as sequence diagram.

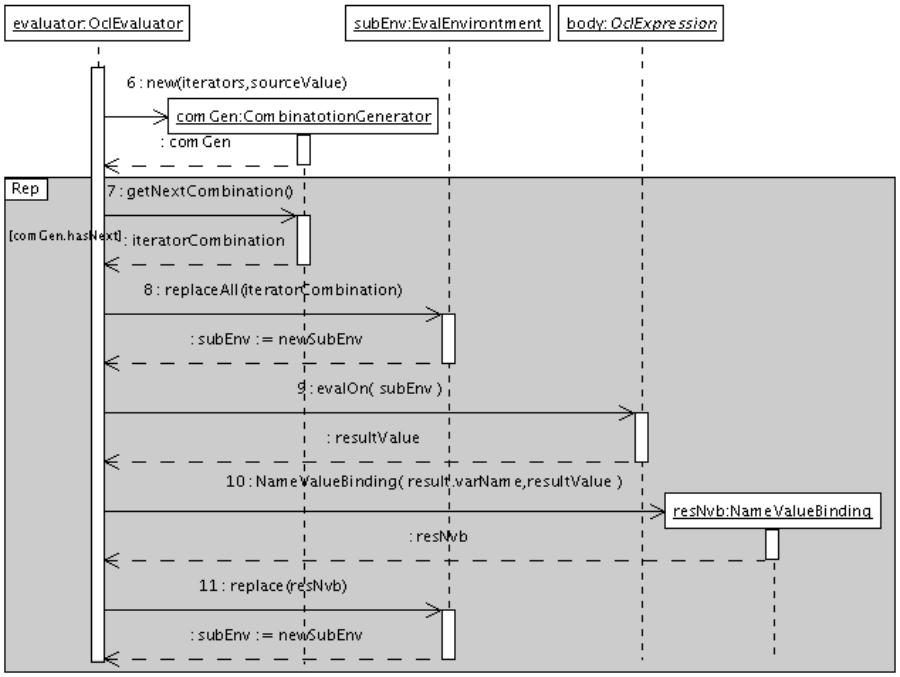

Figure 20: Body Evaluation of an *IterateExp*.

4 Applying the Visitor Pattern

The *OclExpression* structure is not likely to change, and several operations might be defined (e.g. refactoring operations, semantics evaluation, code generation operations, etc.). Consequently, we consider that it is more appropriate to avoid polluting the static structure with these operations and then to apply the Visitor pattern [7], in order to keep it simple and clear (See figure 21).

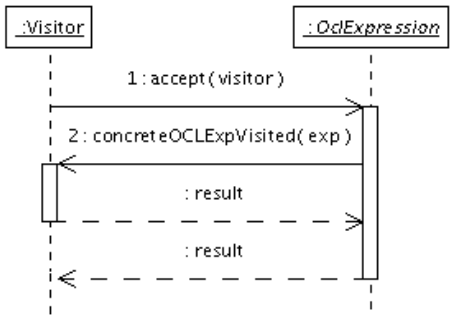

Figure 21: Visitor operation over *OclExpression* structure.

Now we define a meta class named *OclEvaluator* which will replace *evalOn()* maintaining its functionality in one separate *visit()* operation for each non abstract syntactic class of the AS package.

The evaluation environment now is known directly by the *OclEvaluator* and must be modified in a controlled way to avoid the side effect and maintain the query property (figure 22).

Figure 22: *OclEvaluator* meta-model.

With this approach, the semantics evaluation of an *OclExpression* is entirely carried out by *OclEvaluator* which associates AbstractSyntax package with Values packages (see figure 23).

As an example, in figure 24, we show how the operation *evalOn()* over a *LetExp* is modified to become adapted to the visitor pattern.

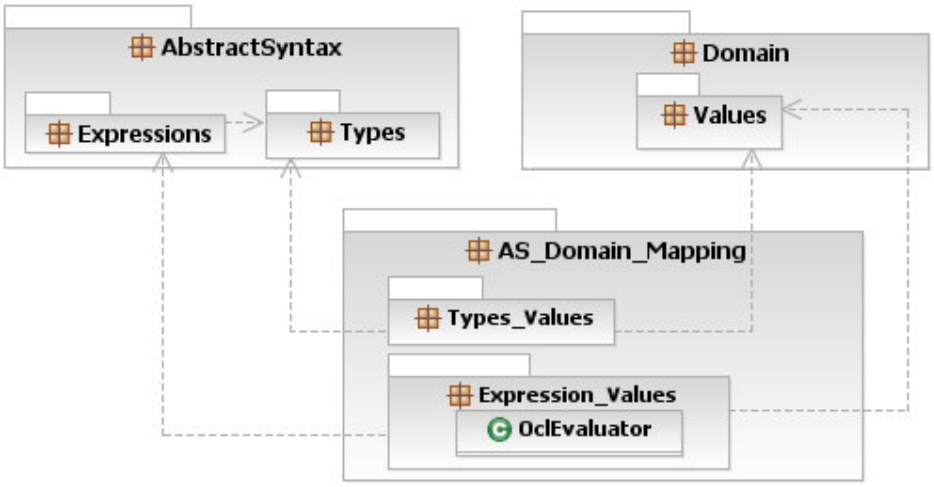

Figure 23: OCL meta model using *OclEvaluator*

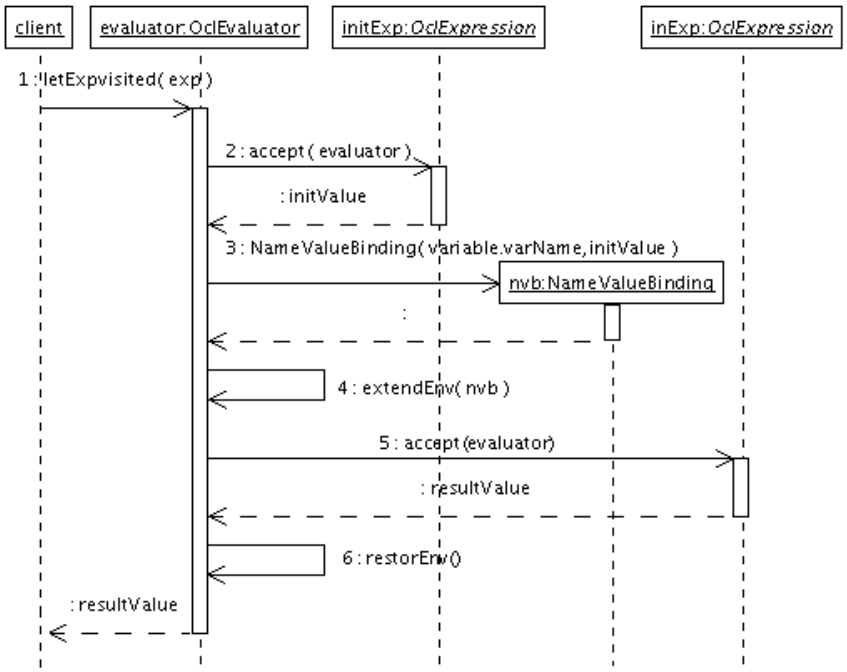

Figure 24: *LetExp* semantics using *OclEvaluator*

5 Conclusion and Future Works

OCL is an object property specification language, which is rigorous but simple and easy to use. Therefore, it becomes a very interesting option for the development of code verification and derivation tools.

OCL addresses people with modest mathematical background. Thus the OCL semantics should be given in a simple formalism OCL users are familiar with, for example metamodeling

In this article, we elaborate an alternative definition for the OCL semantics. This proposal re-uses the OCL syntax metamodel, defines the relation between syntax and semantics through UML collaboration diagrams adhering to the Dynamic Metamodeling (DMM) approach. In this way, circularity on the OCL definition is avoided, and intuitive communication is increased. Besides, the OCL math semantics was used as a foundation and guidance for the semantics definition. Although math semantics could be tedious and hard to understand, and demands users with more academic background, we showed that it could be translated into sequence diagrams offering a more readable and simple semantics metamodel.

On the other hand, the adequate performance of the tools supporting OCL [11] [12] strongly depends on the quality of language definition. To count on a well-defined syntax and semantics will result in benefits for such tools. Also, it is almost straightforward to translate this semantics into a programming language such as Java, because of the proximity between sequence diagrams and programming languages.

Finally, we re-designed the OCL semantics metamodel by applying the 'Visitor' design pattern, which makes it easier the creation of new functionality over the OCL syntax structure and its integration into CASE tools. For example, concerning model transformations, it is possible to define OCL constraints transformations by adding a new "visitor" for the OCL syntax hierarchy. In this sense, we are working on the redefinition of the ePlatero evaluator[13] following the proposal presented in this article in order to analyze the potential advantages regarding the different indicators, such as reliability, efficiency, modifiability, etc.

References

[1] Object Constraint Language OCL 2.0 OMG Adopted Specification. ptc/2005-06-06. URL:www.omg.org

[2] UML 2.0. The Unified Modeling Language Superstructure version 2.0 – OMG Final Adopted Specification.. http://www.omg.org. August 2003. URL:www.omg.org

[3] Hennessy, M. "The Semantics of Programming Languages: An elementary introduction using structural operational semantics". J Wiley&Sons. England. 1990.

[4] Richters, M., Gogolla, M. On Formalizing the UML Object Constraint Language OCL. In Ling, T.W., Ram, S., Lee, M.L., eds.: Proc. 17th Int. Conf. Conceptual Modeling (ER 98). Volume 1507 of LNCS., Springer (1998) 449-464

[5] Brucker, Achim D. and Wolff, Burkhart. A Proposal for a Formal OCL Semantics in Isabelle/HOL. C. Munoz, S. Tahar, V. Carreno (Eds.). TPHOLs 2002, LNCS 2410, pp. 99-114, 2003. pringer-Verlag Berlin Heidelberg 2003.

[6] Tchertchago Alexei. "Analysis of the Metamodel Semantics for OCL". URL: http://www.hwswworld.com/downloads/9_28_05_e/Cherchago-thesis.pdf

[7] Gamma, E. Helm, R. Johnson, R. and Vlissides, J. "Design Patterns, Elements of Reusable Object-Oriented Software". Addison-Wesley Publishing Company, 1995.

[8] Engels, G., Hausmann, J.H., Heckel, R., Sauer, S. "Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML". In Evans, A., Kent, S., Selic, B., eds.: UML 2000, York, UK, October 2-6, 2000, Proceedings. Volume 1939 of LNCS, Springer (2000) 323–337

[9] Hausmann, J.H. "Dynamic Meta Modeling. A Semantics Description Technique for Visual Modeling Languages" PhD thesis, Universit¨at Paderborn, Germany (2005)

[10] Markovic, Slavisa and Baar, Thomas. An OCL Semantics Specified with QVT. O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 645 - 659, 2006. © Springer-Verlag Berlin Heidelberg 2006.

[11] Richters Mark and Gogolla Martin. "OCL-Syntax, Semantics and Tools" in Advances in Object Modelling with the OCL Lecture Notes in Computer Science 2263. Springer. (2001).

[12] Akehurst D.H. and Patrascoiu, O. OCL. Implementing the Standard. in OCL2.0 - "Industry Standard or Scientific Playground?" - Proceedings of the UML'03 Workshop, Electronic Notes in Theoretical Computer Science (2003)

[13] Pons, Claudia and Garcia, Diego An OCL-based Technique for Specifying and Verifying Refinement-oriented Transformations in MDE.. In. Lecture Notes in Computer Science ISSN 0302-9743. O. Nierstrasz et al. (Eds.), volume 4199, pp. 645 – 659, 2006. © Springer-Verlag Berlin Heidelberg 2006. "Model Driven Engineering Languages and Systems, 9th International Conference, Genoa, Italy, October 2006".