Using OCL beyond specifications[1]

Dan Chiorean[2]

"Babes-Bolyai" University - Computer Science Research Laboratory
str. M. Kogalniceanu, 1
3400 Cluj-Napoca - Romania
chiorean@cs.ubbcluj.ro

Abstract: Despite of its important potential and role in defining a more rigorous modeling language and in designing and implementing safer applications, until today OCL was not used at its true value. Using the results obtained in implementing OCL support in ROCASE[3], and taking into account the conclusions obtained testing the tools mentioned at http://www.klasse.nl/ocl, the paper tries to analyze this state of facts. The requirements needed in order to obtain a strong OCL support are presented. The ROCASE OCL support beyond semantical analysis is presented by means of a real example. Finally the conclusions acquired in our activity are offered.

1 The state of the art

OCL is widely used both to define the well-formedness rules for the UML metamodel, and to express constraints in UML models. Unfortunately, until now the use of the constraint language was mainly restrained to the specification level in almost all cases. Even at this level, the existent OCL Tools should be improved. The errors found in the UML specification represent a strong argument in this respect. On the other hand, more and more positions and decisions sustain the use of OCL beyond specifications. The pUML proposal [MML'00] stated very clearly that, first of all, OCL should be used in the "definition" and validation of UML concepts. The adoption of XMI as transfer standard for the UML models opened the way for using OCL to the validation of UML models against UML well-formedness rules. (The above mentioned checks can be realized only if the tool has an OCL support and if the user has access to the UML metamodel level, in spite of the fact that in the 1.4 version, OCL doesn't have an "official" metamodel). The UML models' business rules cannot be taken into consideration if the model that includes them does not respect the UML syntax and semantic. It this context, it is not difficult to guess that the construction of safer applications modeled using UML, cannot be conceived without a strong OCL support beginning with the specification phase and finishing with the testing phase. Today, the support offered by the Case Tools is far from what it should be. This state of fact influenced in a negative way the performances obtained using UML and even the

[1] A part of the results presented in this paper, were obtained in the NEPTUNE IST 1999-20017 framework
[2] *Computer Science Research Laboratory, "Babes-Bolyai" University*
[3] ROCASE – UML CASE tool conceived, designed and implemented at the "Babes-Bolyai" University, Computer Science Research Laboratory http://lci.cs.ubbcluj.ro/rocase.htm

language development. The tools' support should be done beyond the syntactical and semantical analysis. The OCL expressions must be "executable". This implies that the OCL expressions have to be translated into a programming language, and more than that, used at run time. There are many benefits that can be obtained: the conformity between specification and implementation, the opportunity to test the results obtained after OCL expression evaluation, an increased trust in using OCL, etc.

Taking into account this state of the facts, two questions arise: What are the reasons for this situation ? What can we do ? Two questions which we will try to answer in this paper.

2 Requirements for reaching an effective OCL support

Before analyzing this subject, we remind you that, all the OCL Tools mentioned at http://www.klasse.com/ocl (excepting eventually USE) are in fact add-in "components" to different UML Tools. The teams designing and implementing the OCL Tools were different from those designing and implementing the UML Tools. In our point of view, this aspect is important because, despite the XMI adoption as transfer standard for the UML models, the Additional Operations, (essential in the semantical analysis of the OCL expressions) were not included in the repository interface. Therefore the OCL Tools don't have all the information required for the semantical analysis.

The OCL Tools support can be split in three levels. Taking into account their dependency relationship, these are: the semantical analysis of the OCL Expressions, the translation of the correct OCL Expressions in a programming language (C++, Java, Eiffel, etc.) and, finally, the use of the expressions obtained by translation at run-time. In this section, we will take into consideration the first level. The following two will be discussed in the section "ROCASE OCL support".

In brief, the first prerequisite for the semantic analysis is: "The UML model in its wholeness must be *complete* and *correct*". A model is complete only if each of its elements further referred by other model elements are accessible (meaning that the supplier is defined within the model or it was imported). More than that, each model element should contain all the information specified in its UML metamodel structure. The correctness of the model means respecting the Well-Formedness Rules (WFR) as they were defined in the UML specification [UML1.4].

Below we will try to exemplify some errors violating the above-mentioned requirements.

Many UML model errors appear after the deletion from the model of classes for example. In this case, the clients are not informed and consequently the types referred by the former class clients do not exists. The UML model obtained in this case is not complete.

Concerning the lack of information, a suggestive example encountered in all the tested tools was the nature of operations (*observer* or *modifier*). Although, in the UML metamodel this information is modeled through the `Boolean` attribute `isQuery` (defined in `BehaviouralFeature` class) the UML Tools don't take it into account. The type of the collections acquired by means of navigating associations is another very important aspect. Although as early as its first OCL version the collections' type was well defined, many modeling instruments do not provide this piece of information.

Therefore the OCL support cannot analyze correctly the expressions using the specific operations of each collection type. The operations `asSet`, `asBag` or `asSequence`, that the user can apply on collections obtained after navigating associations is not a solution. This is only a supposition that can be different from the modeler's intentions. In the same category (mistakes due to the lack of information), we include the error presented below and encountered in all the tested tools. Let it be a package `P`, two classes `A` and `B` defined in this package, a multiple (`1..*`) unidirectional association from `A` to `B`, named `b` and `o` an instance of `B`, defined in a collaboration diagram attached to the package `P`. Regardless of the collection type obtained navigating `b`, the following OCL expression is correct:

```
context class A inv:
b->includes(o)
```

Despite the expression's simplicity, the OCL Tools incorrectly report an error.

Relating to the WFR violation, we can find different situations. Strange, in our point of view is the fact that many UML and OCL Tools also, don't take into consideration the `visibility` values attached to `Classifiers` and `Features`. In the same category, we include the Classifier's WFR [3] and [4]:

- No opposite `AssociationEnds` may have the same name within a `Classifier`,
- The name of an `Attribute` may not be the same as the name of an opposite `AssociationEnd` or a `ModelElement` contained in the `Classifier`.

If these WFR are violated and we try to evaluate OCL expressions containing `AssociationEnds` or `Attributes` the result obtained is bizarre.

In this context, we whish to stress out that our firm belief is that the WFR compliance should be ensured by the UML Tools. Some OCL Tools (for example, ModelRun) do a few checks but of course not all.

The uniqueness of UML and OCL type systems represents another natural requirement that fortunately was taken into account in the OCL 2.0 proposal [].

As mentioned at the beginning of this section, the information provided by the UML Additional Operations is especially important for the semantical analysis of the OCL expressions. Accordingly the mistakes we found in the specification of some Additional Operations (see [UML 1.4]) must be corrected. In our opinion, the existence of these mistakes proves, at least for the moment, the lack of efficient OCL Tools.

3 Errors discovered in the UML 1.4 Additional Operations

One of the first goals of including OCL in the UML specification was specifying UML element's semantics. As it is well known, this semantics is defined using WFR. In order to obtain the required information in WFR, the so-called Additional Operations were defined. Unfortunately, many of these operations contain semantic errors. As the space of this paper doesn't allow us to exhaust this problem, we will refer only to some significant aspects.

The first two Additional Operations for `Namespace` are `contents` and `allContents`. Their specification as it is in [UML 1.4] is:

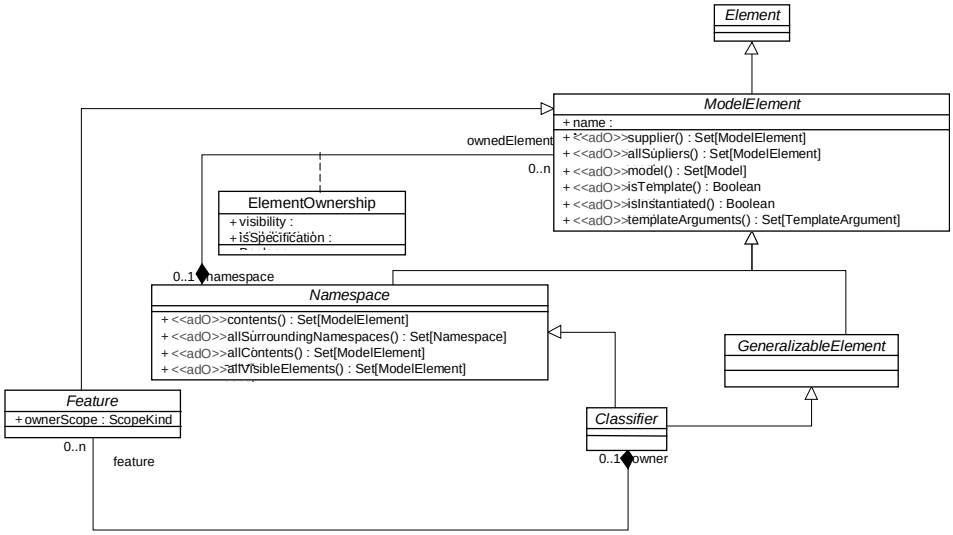

Figure 1: A part of the UML Backbone architecture

```
contents : Set(ModelElement)
contents = self.ownedElement -> union(self.namespace, contents)
```

respectively

```
allContents : Set(ModelElement);
allContents = self.contents
```

As far as `contents` is concerned, the solution is much simpler. If we follow the OCL specification, we should have:

```
Namespace::contents(): Set(ModelElement) post:
result = self.ownedElement
```

or even simpler, `result = ownedElement`.

This happens because in this case using `self` is not needed. This abusive usage of `self` can eventually lead to some unpleasant situations. Concerning `allContents`, even if we read the given specification:

`result=ownedElement->union(namespace.contents)`, it would still be semantically erroneous because in the process of its evaluation we would enter an infinite loop. If we analyze the diagram shown in Figure 1, having in mind the semantics of `allContents`, we should write:

```
result =
ownedElement->union(ownedElement->oclAsType(Set(Namespace))
->contents()
```

This specification will allow us to correctly evaluate the operation.

Let's take a look now at another very useful operation:

```
Classifier::allFeatures(): Set(Feature)
allFeatures=self.feature->union(self.parent.oclAsType(Classifier).
allFeatures)
```

Unfortunately, we also have some errors.

Even if, instead of `parent`, which is not visible in this context, we read `parents`, operation defined in `GeneralizableElement`, the expression would still be incorrect due to the fact that the cast `oclAsType(Classifier)` is wrong, because the type of parents is `Set(GeneralizableElement)`. In this case we suggest the following specification:

```
Context Classifier::allFeatures(): Set(Feature) post
result =
feature->union(allParents.oclAsType(Set(Classifier)).feature)
```

or its equivalent:

```
result =
feature->union(parents.oclAsType(Set(Classifier)).allFeature)
```

where the recursion doesn't occur while calculating `allParents` instead it occurs while calling `allFeature`.

In [Chiorean01] there are described some other semantical errors encountered in the specification of OCL expressions, the causes of these errors are explained and a few suggestions that could lead to more suggestive expressions are described. In this respect, one of our recommendations is that in situations such as the one concerning `allContents`, mentioned above, some examples to prove the results that need to be obtained would be very suggestive.

In order to catch errors like those mentioned above, a tool supporting all the three levels mentioned in the preview section is needed.

4 The ROCASE OCL support

As we stated in the second section, the OCL Tools support can be viewed at three levels. Concerning the first level, (the semantical analysis of the OCL Expressions), our tool, benefits from all the semantic information required. Moreover, ROCASE fully respects the UML WFR ensuring that the UML models produced using our tool are correct and complete. The type system is unique and the collection classes are implemented using C++ template classes. These templates can be used in the OCL support, to generate the code needed for the multiple associations management, and as parameterized classes anywhere in the application.

Regarding the second level (translation of the correct OCL specifications in programming language code) ROCASE completely translates the OCL expressions in C++. About the third level, (the use of the generated code at run-time) our tool offers a functional solution. In our opinion, this last level is the most complex. Many solutions can be taken into consideration and different improvements can be made.

Before presenting the support offered by ROCASE at the last two levels, some clarifications are needed:

Concerning the correctness of the executable models, OCL offers the possibility of expressing constraints in a clear and rigorous manner. Besides this, using OCL doesn't guarantee that the semantic of the OCL expressions is correct. This is the modeler's responsibility. Using ROCASE we have the guarantee that the C++ code, is fully

compliant with the OCL expressions. Our tool doesn't check if the OCL assertions specified in descendent classes are in contradiction with the assertions specified in the parent classes.

Another problem regards the potential of the OCL support in solving a problem after discovering errors. Our point of view, is that in this case, the modeler must find the cause of the errors (possibly by including new constraints) and correct them. After that he can do different tests. In order to enlarge the testing possibilities, the OCL constraints can be modified.

In order to present the way in which ROCASE takes advantage of OCL's potential, we will use a piece of a model. This model deals with a file (and folder) manager and it is simplified on purpose so that we can focus on the OCL issues.

The class diagram is taken and adapted from the model of the Navigator application. This application was developed entirely with ROCASE, and it is available at http://lci.cs.ubbcluj.ro. Navigator implements a small part of the functionalities of a command interpreter. Its real purpose consists in showing the utility and the efficiency of using OCL in real world applications.

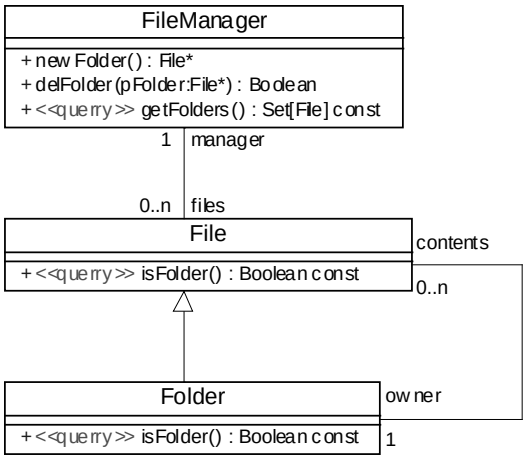

Figure 2: The architecture of a part of the model

In order to use the code generated for the OCL expressions, we must implement an OCL support at least in some of the classes involved in the OCL expressions. The set of classes needing OCL support can be computed. Consequently the elegant solution would be to generate the OCL support exclusively for these classes.

At present, ROCASE offers two alternatives for code generation: OCL support for all the classes and no OCL support. For the future we intend to implement the "elegant" solution mentioned above.

The code generated for one class, with OCL support, respects the following pattern, exemplified for the particular class `File`:

```
class File
{
       public:
              //constructor
```

```
            File() {
                  //Members initialization(1)

                  //User code  (2)

                  //update allInstances      (3)
                  allInstances.add( this);

                  //Class invariant   (4)
            }
            //destructor
            ~File()
            {
                  //User code
                  //update allInstances
                  allInstances.remove( this);
            }
      //methods
      public:
            File * FileManager::newFolder()
            //the rest of the methods …

      //internal methods
      public:
            File * FileManager::_internalnewFolder()
            //the rest of the methods …
//OclAny and OclType implementation
      private:
            static Set<File*> allInstances;
      public:
            static Set<File*> oclType_allInstances();
            //The rest of the oclType features (allAttributes,
            allOperations)
};
```

Figure 3: The pattern of code generation for the header of a class with OCL support

The methods corresponding to the classes `OclAny` and `OclType` are implemented straight into the target class, in our example the class `File`. This approach was preferred due to the fact that, in ROCASE, the target language chosen for implementation, C++, doesn't support run-time identification of a class' properties (such as its attributes, methods, or even its name). For this reason, specializing one class (and the best candidates would have been either `OclAny` or `OclType`) would be inefficient, as all their methods would have to be completely rewritten in each derived class.

To avoid confusing the user with choosing the name for the properties of problem-domain classes, all these methods are preceded by `OclType` respectively `OclAny`. This way of naming the properties adds some extra semantic information.

```
//_internalnewFolder operation
File * FileManager::_internalnewFolder()
{
      //User code
```

```
}
//newFolder operation
File * FileManager::newFolder()
{
       //Precondition (1)
       //auxiliary data used in post condition
       Set< File* > oclIsNew_0 = File::oclType_allInstances(); (2)
       // body:
       File* result = _internalnewFolder( );   (3)

       //Post condition: <the OCL form of the post condition>
       Boolean bPostcondition = FALSE;
       {
              bPostcondition = ((( NOT oclIsNew_0.includes( this))
              AND files.includes( result)) AND result.manager ==
              this);
       }
       assert(bPostcondition);(4)
       //ClassInvariant
       return result; (5)
}
```

Figure 4: The management of the `newFolder` method

When including the code associated with the assertions in the application code, we should consider again the feature of the target language. C++ allows multiple exit points from a function. For this reason, simply positioning the post condition or the class invariant at the end of the method doesn't certify their evaluation. Our solution was duplicating each modifier.

As it can be seen for the `newFolder()` method, specified by the user in the `FileManager` class, we would have the following structure:

- Evaluation of the precondition (excepting the constructors) (1)
- The definition and initialization of the auxiliary variables, which will be later, used by the post condition (when `@pre` and `oclIsNew` are used in the post condition) (2)
- The method body being, in fact, the call of the `_internalNewFolder` method. This method contains the implementation of the operation (the user code). For constructors, this duplication isn't required (3)
- Evaluation of the post condition (excepting the destructor) (4)
- Evaluation of the class invariant for modifiers (5)

The constructors and the destructors should accomplish some other tasks. More precisely, the constructor:

- initializes all the class members (using the information provided by the user, or some default values)
- appends the current instance to the list of the type's instances.

The destructor removes this instance from `allInstances`.

The implementation for `allInstances` offers some of the functionalities of a Garbage Collector. The user is given the possibility to find out at run-time whether an object was deleted from memory or if, on the contrary, it is still available. This is a real asset in the debugging and testing phases, especially if the target language is C++.

Numerous papers on this subject, even the OCL 1.4 specification [UML 1.4], persuade against using the `allInstances` operation. Without claiming that the adopted solution is the best, it makes its point by proving that this mechanism, even in this simple implementation, is useful and can be trusted. The problems that arise are linked to the use of libraries, where nothing certifies the existence of such a mechanism. Obviously, some solutions can be imagined for this problem also, one of them being the class adaptor. Another solution could be the use of derived classes. Both these solutions are far from being ideal, but we must not forget that the generated code for the OCL expressions is mainly used in the test – debug phases. As it is presented in [Meyer97], this auxiliary code can be removed for the final versions of the applications.

In this context the postconditions for the `FileManager::newFolder` and `FileManager::delFolder` operations are:

```
Context FileManager::newFolder post:
//The folder object was created
result.oclIsNew() and      //(a)
//it is kept by the FileManager
files->includes( result) and
//also has a link towards the file manager
result.manager = self //(b)

Context FileManager::delFolder post:
//The folder is no longer kept by the FileManager
not files->includes( pFolder) and
//the object was deleted from memory
not File.allInstances->includes( pFolder) //(c)
```

Figure 5: The postconditions for the `newFolder` and `delFolder` methods

As we can see in the implementation of the `newFolder` method (Figure 4) and from its post condition (Figure 5), in order to determine if an object was created during the method's execution, the following operations are performed:

- a copy of the set of `File` instances is created (2)
- the method body is executed (*_internalnewFolder*) (3)
- a check is made to determine whether *result* exists in the previously created set. If the result of the evaluation is true, it means that the object is new indeed (created by this method) (4), (b)

The utility of the last component of the `newFolder` post condition can be better understood at run-time.

Providing association integrity is a vital aspect of every OO application. In this sense, the ROCASE support is made of a library containing the `Collection` classes. These classes conform to the OCL specification.

Using OCL for specifying an operation can be extended to the point where the complete specification of the operation is achieved. Because OCL is a side effect-free language, in the case of a complete specification, the applicability domain limits itself to observers

(those operations for which the `isQuery` attribute is `true`). The task of providing the means for such a specification relies almost entirely on the modeling tool.

In order to have a correct image of this problem, we provide the code written by hand for implementing the `getFolders` method, and its complete specification by means of its post condition and the code automatically generated by ROCASE.

```
/* Manual implementation of getFolders operation*/
Set< File* > FileManager::_internalgetFolders() //(a)
{
	// body:
	Set< File* > _result;
	Set<File*>::iterator b = files.begin();
	Set<File*>::iterator e = files.end();
	while ( b != e)
	{
		Folder* f = (*b);
		if ( f->isFolder())
			result.add( f);

		++b;
	}
	return _result;
}
```

Figure 6: The "hand-written" code for the getFolder postcondition

```
Set< File* > FileManager::getFolders()
{
// body:
Set< File* > result = _internalgetFolders( );
//Postcondition
/*OCL expression: result = files->select( f : File |
f.isFolder())*/
Boolean bPostcondition = FALSE;  //(b)
{
	Set< File* > _vAux_0; //Operation: select
	for ( Set< File* >::iterator _itB_0 = files.begin(), _itE_0
		= files.end(); _itB_0 != _itE_0; ++_itB_0)
		{
			File *f = (*_itB_0); //The current collection
	element
			if ( f->isFolder( ))
				_vAux_0.add( f); //The condition holds
				for this element so we select it in the
				result.
		}
	bPostcondition = result == _vAux_0;
}
assert(bPostcondition);
return result;
}
```

Figure 7: The generated code for the getFolder postcondition

As one can see in the previous two code fragments, the difference between the two implementations, the first written "by hand" (a) and the second one generated automatically (b), resides mainly in one aspect: the intelligibility. Without disregarding this aspect, which is quite important in the development of any modern application, the lack of clearness is due to the names used for the temporal variables. The generated names do not offer the advantages of user given ones (the Hungarian notation for example). The most important advantages of using the generated code are the conformance among specification and implementation, as well as the reduction of the time needed for implementation.

The conformance among specification and implementation is, in our opinion, far more important compared to the inconvenience caused by the variable names used. The advantages are numerous, one of the less obvious ones being the possibility of checking the logic correctness of the specification. Using test cases (which can be achieved from an automatic test generator), it can be said not only if the implementation is correct but also if the model built for it solves the initial problem (the one behind all the modelling effort).

5 Conclusions and future work

The inclusion of OCL in the UML specification is an important moment in the evolution of the modeling language. Unfortunately, the opportunity offered by OCL has not been efficiently used yet. The lack of adequate tool support is one of the most important reasons which delayed this evolution. This conclusion is natural if we think that not even the thorough testing of a language can be conceived without specialized instruments: editors, compilers, debuggers and libraries. The OCL support can be efficient only if it gains from all the required semantical information and if the models that the OCL expressions refer are correct. Except for the compiler built at [DresdenOCL] which interchanges information in an efficient manner with the Novosoft repository, the other tools we've tested, [BoldSoft], [Key], [DresdenOCL], [USEEnvm], built as add-ins for well-known tools (Rose, Together etc) don't have all their required information. The errors existing in UML specification, errors which are not signaled by the various tools, are also an important impediment for the OCL support. No matter how well it is designed and implemented, the efficiency of the OCL support depends on the quality of the UML tool, as seen from the point of view of the model correctness. In this respect, the ROCASE experience is an advantage. The same team that built the case tool built the OCL support. This way we had access to all the required information and we were able to even modify some of the inner aspects of our tool, which were not right. The ROCASE OCL support also checks target values. Under the influence of Meyer's opinions regarding the "design by contract", we translated into source code (C++) the OCL specifications in our attempt to use them for maximum advantage. Our results proved that this approach is not only reachable but that it is also advantageous and it is well worth continuing it. In this regard we showed that implementing the support for the `allInstances` operation is not so very difficult. No doubt that the generated OCL source code can be further enhanced.

The different errors discovered in the `AdditionalOperations` specifications will allow us a better understanding of the true value of OCL and of its possibilities.

Within the framework of the NEPTUNE IST 1999-20017, our research targets the design and the implementation of a UML model checker. This checker will acquire its information from UML models kept in XMI format. The research made in this field allowed us to identify a number of WFR which are not validated by the large majority of the instruments. The NEPTUNE OCL checker, will give the user access both to the metamodel level and to the model level. The access to the metamodel level will provide the possibility to check some metrics rules. For example, to detect all the hierarchies having more than n levels etc.

In the future, we are interested in extending OCL horizontally, by using the constraints in an increasing number of contexts, and vertically, by extending the OCL support with new features for managing the actions, concurrency etc. We hope that the results of our activity can be useful to other teams interested in this topic, and also to support the further evolution of OCL and UML.

Acknowledgements

Many thanks to Dragos Cojocari for implementing a big part of the ROCASE OCL support, and to the anonymous reviewer for the observations and suggestion made.

Bibliography

[BoldSoft]	BoldSoft MDE AB - Model Run, http://www.boldsoft.com
[Chiorean01]	Chiorean D., Cojocari D. – Implementation of OCL Support in UML CASE Tools – the ROCASE Experience; Objectives, Proposals, Perspectives – The ISM'01 International Conference, http://www.fee.vutbr.cz/UIVT/ism/pdf/chiorean.pdf
[Clark00]	Clark T., Evans A., Kent S., Brodsky S., Cook S. - A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach, http://www.cs.york.ac.uk/puml/mmf/index.html
[Cook94]	Cook S., Daniels J. – "Designing Object Systems; Object-Oriented Modelling with Syntropy" – Prentice Hall, 1994
[Cook99]	Cook S, Kleppe A et all – Defining the Context of OCL Expression – in UML'99, Second International Conference, Fort Collins USA, page.372-383
[Daniels01]	Daniels J., Chessman J.,– UML Components – A simple Process for Specifying Component-Based Software – Addison Wesley 2001
[DresdenOCL]	http://dresden-ocl.sourceforge.net/index.html
[Kleppe'00]	Kleppe A, Warmer J. – Extending OCL to Include Actions – in UML'00, Third International Conference, York, England, page.440-450
[Meyer97]	Bertrand Meyer – "Object-Oriented Software Construction – second edition" – Prentice Hall 1997
[OCL-issues]	http://www.klasse.nl/ocl/ocl-issues.pdf
[UML 1.4]	UML 1.4 Draft Specification – February 2001 - 01-02-14.pdf – http://uml.sh.com
[Warmer99]	Warmer J, Kleppe A. – "The Object Constraint Language" – Addison Wesley 1999
[Jacobson01]	http://www.sdgnews.com/uml2001_002/uml2001_002.htm
[USEEnvm]	http://www.db.informatik.uni-bremen.de/projects/USE