How My Favorite Tool Supporting OCL Must Look Like

Dan Chiorean, Vladiela Petraşcu, and Dragoş Petraşcu

Computer Science Research Laboratory
Babeş-Bolyai University of Cluj-Napoca
Mihail Kogălniceanu nr. 1, Cluj-Napoca, Romania
{chiorean,vladi,petrascu}@cs.ubbcluj.ro

Abstract. This paper presents its authors' viewpoint on the assessment of tools that support the use of OCL. At this time, deciding on which such tool to use is not an easy task. This is influenced by a number of objective factors, including: the user's needs, knowledge of existing tools, knowledge of the language and of the various possibilities of using it. Undoubtedly, the choice of a particular tool does also include subjective aspects. The paper is limited to the presentation of objective criteria. In this context are examined: the features that distinguish OCL within the modeling languages' family, some aspects that are either incompletely or ambiguously described in the OCL specification, the main functionalities that an OCL supporting tool should implement, the universe of tools supporting OCL. In the end, the tools listed in Section 6 are characterized with respect to the functionalities of an ideal tool and the obtained conclusions are presented.

Key words: OCL, OCL like languages, modeling languages, OCL tools, MDA, MDE, `undefined` values

1 Introduction

There is an interdependency relationship between languages and tools supporting their use. On the one hand, the languages preced the tools, on the other hand, they can not be used in the absence of appropriate tools. Such tools facilitate the development of software applications, creating thus a proper framework for obtaining the feedback needed in order to validate and refine the languages. Without tools, such activities can not be conceived!

At less than a decade since the appearance of the first tools offering OCL support, the universe of these tools is diversified, counting more than fifteen members. Nevertheless, compared to the overal number of UML or MOF-based tools, the number of those which also provide OCL support is noticeably smaller. The above mentioned diversity of OCL supporting tools' universe can be justified by the fact that, with an acceptable approximation, it can be said that there are no two tools implementing the same set of features. This is just one of the reasons that makes a rigorous comparison between them difficult. The great majority of

tool implementors justify the decision of building a new tool by the desire (need) to include features not still implemented within the existing ones.

In this context, our first goal is to achieve a set of objective criteria which will allow users to evaluate the existing OCL supporting tools and even those that may appear. Through the approached issues, as well as by presenting the viewpoint of a tool realizer, we hope that the work will be useful for tool implementors and even that it will contribute to the language's evolution.

Several aspects of the topic that we address have already been presented in [9] and [6].

In order to understand and use effectively the tool assessment criteria introduced within this paper, it is required to know the fundamentals that underlie them. Therefore, the rest of the paper is structured as follows: Section 2 presents the main OCL features, insisting upon those which have a significant impact on tools' design and implementation. Section 3 examines some ambiguous or insufficiently detailed aspects within the language specification. In Section 4, a tool functionality that allows for an efficient use of specifications is detailed, preceded by a description of how to write effective specifications. Section 5 introduces and justifies the proposed tool assessment criteria. The universe of tools supporting OCL is summarized in Section 6. A conclusions section ends the paper.

2 OCL features

Within this section, we summarize those features of the language that have a major influence on tools supporting OCL.

- OCL is a complementary modeling language, which uses a textual formalism. The language is used for navigation (query models), specification of assertions, specification of behaviour (all existing versions, including 2.0, only allow observers' specification). The fact that the OCL specifications complement an existing model description has a great impact on tools, whereas they should support both navigation of models (which are represented in most cases using a graphical formalism) and navigation of OCL specifications. These navigations must be carried out in a synchronous manner.
- Any OCL specification is written in a well-defined context of the model complemented by that specification. In order for the OCL specifications to make sense, the model must conform to the well formedness rules of its modeling language (WFR), rules specified at the metamodel level. Therefore, verifying this conformance should precede the compilation of OCL specifications, and, implicitly, their evaluation.
- The language specification is not sufficiently detailed in the following cases:
  1. assessment of expressions containing undefined values,
  2. managing exceptions that may occur in evaluating OCL expressions,
  3. assessment of undeterministic functions (`any`, `asSequence`, `asOrderedSet`),
  4. accessing properties from ascendants.

- Validating OCL specifications involves evaluating them on significant models. The evaluation can be done in a direct / static manner (by passing through the evaluated expression associated AST), or a dynamic one (by evaluating the application obtained through automatic code generation). In the static case, the AST contains all the values required by the evaluation process. Comparing the results obtained by direct evaluation to those obtained by evaluating the corresponding code (obtained by converting the OCL expressions in a programming language) is a test required in order to validate the OCL specifications.
- Using two formalism, one textual and the other one graphical, implies the existence of two types of corresponding tools, one for each formalism.
- All the standard versions emerged so far treat OCL as a cohesive language. Nevertheless, many tools have implemented language extensions, useful for various application domains. Furthermore, within Model Driven Engineering (MDE), languages referenced as "OCL-based languages" have been introduced. For these reasons, and by also taking into account the language characteristics, OCL can be considered as a family of typed languages [2]. This approach is advantageous for both users and for those who manage the language evolution. Users must only know those components they use. Changes made at any language in the family, except for the core part, does not affect the other components [2].
- The language employed at the M3 (MOF) level, as well as the OCL-based languages, use the navigation component and a part of the component required for specifying assertions (which supports the specification of invariant constraints). The navigation language used in this case has some restrictions. They are due to the fact that within MOF or other meta-metamodels, the concepts of *qualified association* and *association class* are missing (compared to the language defined for UML) and all associations are binary (there are no n-ary associations with $n > 2$).

3 Ambiguous or not enough detailed aspects in the OCL specification

3.1 Evaluating expressions containing `undefined` values and managing evaluation exceptions

OCL is meant to complement model description in order to support a more complete and rigorous model specification. Therefore, the results obtained when evaluating OCL expressions have to comply with the above mentioned goal. On the other hand, working with `undefined` values is really important when using models. At different stages of software development, it happens that not all decisions are taken or not all the information is known. However, even in these situations, for modelers, it is crucial to work with models, and, by consequence, to evaluate OCL expressions that may contain `undefined` values. Of course, in such situations it is expected to obtain results as accurate as possible.

In the OCL specification [26], `undefined` is used both for mentioning that some information is missing and in case of some runtime exceptions such as division by 0 or accessing the elements of an empty collection. As we will discuss in the following, this can cause unpleasant situations.

Suppose that, by navigation, we obtain a collection of `Integer`s, representing some persons' ages. If we are interested in knowing whether there is at least a person aged more than 80, using the following OCL expression:

```
Bag{89,15,23,undefined}->exists(a|a>80),
```

in accordance with the OCL specification, we will obtain `undefined`, even if such a person exists. This situation is unacceptable, therefore, in [2], we have proposed a more detailed evaluation strategy for expressions containing `undefined` values. In accordance with our proposal, we will obtain `true` in each case when the collection iterated by the `exists` operation contains at least one element complying with the rule specified in the body of `exists`.

Evaluating expressions containing `undefined` values is included in the benchmark test addressed in [6]. Analyzing the results obtained when evaluating the following three OCL expressions, we have noticed that the obtained results depend on the tool used. By using the OCLE [28] tool, the results obtained are those mentioned after each OCL expression.

```
Sequence{1..3}->iterate(i; c:Sequence(Boolean)=Sequence{} |
  c->including( Sequence{1,8,7}->at(i) > Sequence{2,3,8}->at(i) ))
result obtained: Sequence{false, true, false}
```

```
Sequence{1..3}->iterate(i; c:Sequence(Boolean)=Sequence{} |
  c->including( Sequence{1,8,7}->at(i) > Sequence{2,3}->at(i) ))
result obtained: sequence index out of range
```

```
Sequence{1..3}->iterate(i; c:Sequence(Boolean)=Sequence{} |
  c->including( Sequence{1,8,7}->at(i) >
    Sequence{2,3,oclUndefined(Integer)}->at(i) ))
result obtained: Sequence{false, true, Undefined}
```

Using other tools, like USE [34] or MDT-OCL [19], we will obtain the same result for all three different OCL expressions, namely:

```
Sequence{false, true, false}.
```

We consider that the results obtained by using OCLE are more accurate than those obtained with the other two tools. The differences in evaluation are due to the followings reasons:

- In OCLE,

  `Sequence{1,8,7}->at(i) > Sequence{2,3}->at(i)}`

  raises the exception "sequence index out of range" when `i=3`, because `Sequence{2,3}` has only two elements. USE and MDT-OCL evaluate the same expression to `false`.

- The expression `7 > oclUndefined(Integer)` is evaluated to `undefined` in OCLE and to `false` in both USE and MDT-OCL.

The results obtained using OCLE are the same with the results obtained in case of a dynamic evaluation, at run time, when the expressions specified in Java would be translated from the above OCL expressions. In the context of the new Software Engineering paradigms (MDA, MDE, LDD), the full conformance of static and dynamic evaluations is essential.

We suggest introducing a new class, `Exception`, in the OCL metamodel, in order to model the exceptions that can be raised when evaluating OCL operations. In this manner, the results obtained will be more accurate and the framework required for obtaining the same results at both static and dynamic evaluation will be ensured.

In cases when is expected to obtain `undefined` values, we recommend taking a more secure approach, namely using the standard `oclIsUndefined()` operation. This because, in our oppinion, it is not secure to say that the value of the expression `oclUndefined(T) = oclUndefined(T)`, is `true`. In this case, we agree with the standard. The above mentioned value is `undefined`.

The OCL benchmark proposed in [6] contains the following test:

```
(1) Person.allInstances->select(husband =
      Person.allInstances->any(wife->isEmpty).wife).
```

Analyzing this test, we notice that any evaluations of

```
Person.allInstances->any(wife->isEmpty).wife,
```

will result in `undefined`. So, we conclude that, in fact, the intent is to compute the set of persons without husband. In our opinion, a simpler and safer expression equivalent to (1) is

```
(2) Person.allInstances->select(p|p.husband.oclIsUndefined()).
```

In OCLE, for example, generating the Java code corresponding to the expression (1) will provide

```
Set setAllInstances =
  Ocl.getType(new Class[]{Person.class}).allInstances();
//evaluate
//'select(husband=Person.allInstances->any(wife->isEmpty).wife)':
Set setSelect = CollectionUtilities.newSet();
final Iterator iter = setAllInstances.iterator();
while (iter.hasNext())
{
  final Person iter1 = (Person)iter.next();
  Person personHusband = iter1.getHusband();
  Set setAllInstances0 =
    Ocl.getType(new Class[]{Person.class}).allInstances();
  //evaluate 'any(wife->isEmpty)':
```

```
  Object temp = null;
  final Iterator iter0 = setAllInstances0.iterator();
  while (temp == null && iter0.hasNext())
  {
    Object temp0 = iter0.next();
    Person iter2 = (Person)temp0;
    Person personWife = iter2.getWife();
    boolean bIsEmpty = CollectionUtilities.isEmpty(personWife);
    if (bIsEmpty) temp = temp0;
  }
  Person personAny;
  if (temp == null) personAny = null;
  else personAny = (Person)temp;
  Person personWife0 = personAny.getWife();
  boolean bEquals = personHusband.equals(personWife0);
  if (bEquals) CollectionUtilities.add(setSelect, iter1);
},
```

`setSelect` being the evaluation result. Executing this code, we will get

```
Exception in thread "main" java.lang.NullPointerException,
```

since the `equals` function would be called on a `null personHusband`. On the contrary, executing the Java code generated by OCLE for the expression (2), the result returned for the same snapshot will be identical to the result obtained through an OCLE static evaluation.

In USE, evaluating the expression

```
Set{oclUndefined(Integer)}->exists(e | e <> i)},
```

the result obtained is `true`, irrespective of the value of `i:Integer`. So, by induction, we can conclude that the value of `oclUndefined(Integer)` is not an `Integer`, which is false. Therefore, we consider that `undefined` is the correct result obtained when comparing two values from which at least one is `undefined`.

3.2 Evaluating undeterministic operations - `any`, `asSequence`, `asOrderedSet`

From a tool maker perspective, the OCL standard [26] gives an incomplete specification for the `any(iterator | <boolean expression>)` operation defined on `Collection`s. Therefore, the results obtained when evaluating OCL expressions that include `any` depend on the tool. For example, in USE

```
Set{9,1,7,oclUndefined(Integer)}->any(true) = oclUndefined(Integer),
```

while in OCLE

```
Set{9,1,7,oclUndefined(Integer)}->any(true) = 1.
```

The results will be the same, irrespective of the number of times the evaluation is performed. As both outputs comply with the standard, we consider that, in this case, the algorithm for evaluating the `any` operation must be included in the OCL specification. Moreover, modelers must use this operation carefully, taking into account the different results that can be obtained by evaluation.

In the standard, the collection operations `asSequence` and `asOrderedSet` are specified as undeterministic operations. Therefore, the results obtained when evaluating these operations depend on the used tool. For example, in USE

```
Set{9,1,7,oclUndefined(Integer)}->asSequence() =
  Sequence{oclUndefined(Integer),1,7,9},
```

while in OCLE

```
Set{9,1,7,oclUndefined(Integer)}->asSequence() =
  Sequence{9,1,7,oclUndefined(Integer)}
```

It is enough clear that the evaluation strategies implemented in these tools are different. In OCLE, both `asSequence` and `asOrderedSet` operations do not change the "order" in which the set elements are listed. In USE, `asSequence` is equivalent to `sortedBy(i | i)`. That is why in USE we obtain:

```
Set{9,1,7,oclUndefined(Integer)}->sortedBy(i | i) =
  Sequence{oclUndefined(Integer),1,7,9}.
```

The results obtained with USE show that, in the 2.4.0 version of this tool, `OrderedSet` was not yet implemented, and that `oclUndefined(Integer) < 1`. However, when we evaluated the expression `oclUndefined(Integer) < 1` separately, the result obtained was `false`. In our opinion, the solution adopted by OCLE gives the opportunity to keep the order of terms when including new terms in the collection.

3.3 Accessing features with the same name, from ascendants

Accessing features defined in parents is an usual operation in object-oriented applications. Suppose that we are in the `C` context and that we need to redefine the operation `o1():Integer`, initially defined in `A`. In order to do this, normally there are at least two possibilities: (1) using an upcast, as in:

```
context C::o1():Integer
  body: self.oclAsType(A).o1() + 1
```

or (2) using an explicit notation, like in C++:

```
context C::o1():Integer
  body: self.A::o1() + 1
```

A similar situation happens if, in the `C` context, we need an invariant such as:

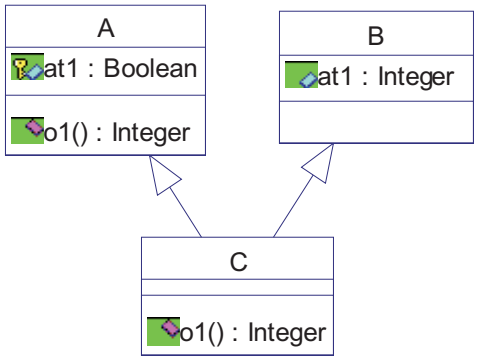

Fig. 1. Accessing features with the same name, from ascendents

```
context C
  inv: self.oclAsType(A).at1 implies self.oclAsType(B).at1 > 0
```

The OCL standard specification ([26], page 25) mentions only the downcast, but not the upcast. We consider that this is a mistake, at least for two reasons: (1) upcasts (to an ascendent) are always safe and (2) sometimes it is really necessary to be able to access features defined in ascendents, as we have discussed before.

4 Efficient OCL specifications and the required tool support

By *efficient OCL specification*, we denote a specification that, when evaluated, provides the modeler with the information needed in order to take the best decision. As we will show in the following, many times it happens that the most efficient OCL specification is not the shortest or the easiest to understand. However, the obtained benefits justify the effort. Beyond the specification, the OCL supporting tool must implement partial evaluation and a natural navigation between different views of the same element.

Usually, when evaluating an invariant, users are only warned that it fails in a certain context. Most of the times, a simple failure message is not enough, requiring deeper investigations. The example presented in the following is mentioned in [5].

It involves libraries that lend book copies to their members. The first invariant, found in [5], checks that all copies currently lent are from libraries whose members are the loaners. This first invariant, named `usual`, is similar with almost all the invariants currently specified for such a constraint. The only information obtained when evaluating it is represented by the members having wrong loans. When the evaluation fails, the second invariant, named `moreUsefull`, may provide, by partial evaluation, a member's list of wrong loans. More than that,

the third invariant gives the list of wrong loans, and for each such loan the list of copies and libraries owning these copies. If in [1] the `moreUsefull` invariant could be obtained automatically, the last invariant (oferring the richest information) cannot. In case of real applications, containing an important volume of data, it is highly important to be able to identify as soon as possible and with a minimum effort the reasons of a possible failure. The last specification is the most helpful in this purpose.

```
context Member

inv usual:
  self.currentLoans->forAll(l|l.copies.library->forAll(li|
    li=self.memberOf))

inv moreUsefull:
  self.currentLoans->select(l|l.copies.library->exists(li|
    li<>self.memberOf))->isEmpty

inv moreUsefullThanPrec:
  let wrongLoans:Set(Loan)=self.currentLoans->select(l|
    l.copies.library->exists(li|li<>self.memberOf)) in
  (wrongLoans->iterate(l;
    sLC:Set(TupleType(a:Loan,b:Set(Copy),c:Set(Library)))=Set{} |
      if l.copies.library->select(li|li<>self.memberOf)->notEmpty
      then sLC->including((Tuple{a=l,b=l.copies,
        c=l.copies.library->asSet}))
      else sLC
      endif))->isEmpty
```

As shown in Figure 2, the result of iterating the `wrongLoans` collection is posted on the output pane, the lowest located in the tool window. The expression evaluated is highlighted in the OCL editor window, located on the top right of the tool window. Just clicking on the first element of the first tuple printed on the output pane, l1, the corresponding model element is automatically selected on the model browser and on the snapshot. In a similar manner, the other two elements of the first tuple, the copies and the libraries, can be navigated. So, when the invariant fails, by means of partial evaluations and navigations, the modeler can easy identify the loans, the corresponding copies, and the libraries that do not comply with the invariant requirement.

5 Functionalities that our ideal OCL tool should implement

The functionalities discussed in this section have been suggested by the experience we acquired while conceiving, implementing, and using the OCLE tool, as well as while working with various other tools, such as USE, OCL Dreseden

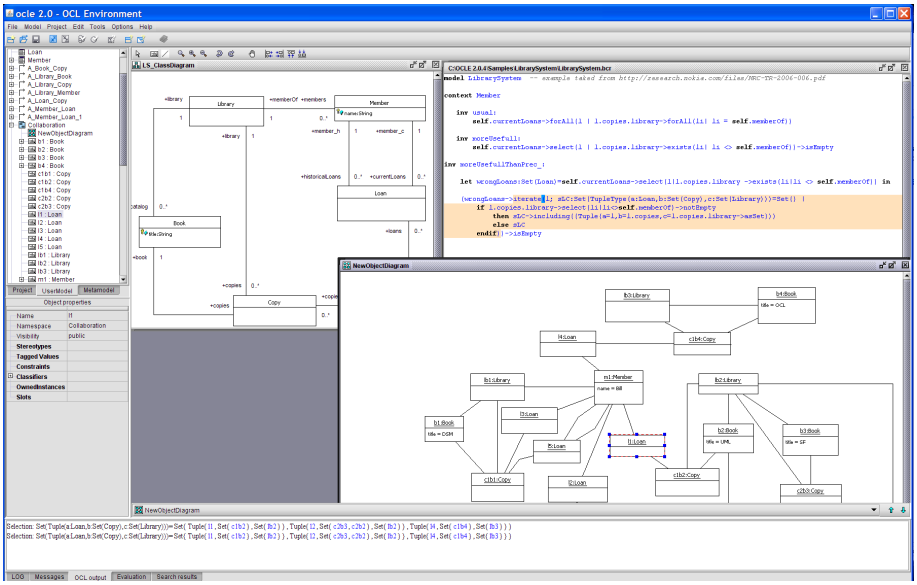

Fig. 2. Navigating the evaluation snapshot in OCLE, in order to identify the failure reasons

Toolkit, Octopus, MDT-OCL, Together, or XMF, to mention just the most used by us. Each of these tools (including OCLE) has offered us both satisfactions and frustrations. Also, these requirements are influenced, of course, by the language characteristics discussed in Section 2. We are deeply convinced that a tool expresses more or less the vision that its realizers have on OCL and on different possibilities of using OCL specifications. Our experience proved that, at the moment, there are no tools implementing ad literam the standard. Moreover, there are significant differences between the current tools, both in the way they implement the standard, and in their utility. Therefore, comparing the existent OCL tools is, in our point of view, on the one hand, a very difficult and risky task, and, on the other hand, a not very useful activity. We spoked about "ideal" OCL tool, because ideal is something not very easy to obtain. Moreover, the ideal differs from one person (tool user) to another. However, we hope that the functionalities discussed will provide a set of useful evaluation criteria for both tool makers and existing/potential users.

Tools must allow working with models and specifications irrespective of their size. It is extremely important that they provide strong support for managing big models and large specifications.

Tools must adhere to the standards. In case there is no full conformance, all the differences have to be explicitly and clearly described. Standards cannot be tested, validated, and improved without appropriate tools. Moreover, it would be very useful if tools supported automated transformation of models/specifications having differences from the standards in standards-compliant

models/specifications. A consequence would be an unrestricted exchange of models and specifications between existing tools.

Tools' constituents (editors, compilers, browsers) must implement the functionalities established by IDEs. This implies the existence of text beautifiers, auto-completion, diagram filtering facilities a.s.o. Tools must promote automation whenever possible and suitable. To give an example, this may include support for snapshot generation, as implemented in USE.

Any tool's documentation must state explicitly the main objective for which the tool was conceived and implemented. Also, suitable examples of its use are recommended.

Tools must support reuse. This includes the possibility of attaching and detaching in a very simple manner different kind of OCL rules specified at M2 or M3 levels, as allowed by OCLE.

Our "ideal" OCL tool should:

1. **Support users in managing models by means of both graphical and textual formalisms**.
   Justification: A graphical formalism is easier to understand. Models' persistency is accomplished by the XMI standard. After the first UML standard was released, in 1997, two opposite modeling tendencies were proposed: one suggesting to represent even constraints as diagrams [8], the other [7] allowing model specification only in a textual form. Having both graphical and textual editors, users may take advantage of both or use the preferred one. Managing both graphical and textual formalisms means that users are able to import, export, update, and to browse models. All these functionalities are mandatory.
2. **Enable working with OCL expressions at three abstraction levels (meta-metamodel, metamodel and model) in case of MDA or DSML tools, and at least the last two levels in case of UML tools.**
   Justification: In case of MDA or DSML tools, the modeling languages specified using metalanguages such as MOF, Ecore, or KM3, have to be checked against the WFRs specified at the highest level. The same requirement applies for models, that must be verified against the WFRs specified at the metamodel level. The correctness of OCL expressions specified at the model level must be checked against the OCL specification. Only if the model has been proved to be consistent with respect to the metamodel WFRs, the last mentioned operation can be realized in good conditions. Therefore, model's consistency must be ensured before verifying the OCL expressions specified on it.
3. **Allow compilation and statical evaluation of all kind of OCL expressions (pre & postconditions, invariants, observers, guards or modifiers - for OCL extended specifications as in XMF [3]).**
   Justification: Statical evaluation of the above mentioned OCL expressions is important since it is very intuitive. It provides users with the possibility of comparing the values obtained by evaluating different OCL sub-expressions with the states of the corresponding parts of the evaluated snapshot. Therefore, statical evaluation is the best way of testing OCL specifications.

4. **Include code generation facilities, for different target programming languages.**
   Justification: First of all, comparing the results otained in case of static evaluation with those obtained at dynamic evaluation represents a way for increasing the confidence of modelers in the OCL specifications. Moreover, MDA, MDE, and LDD promote M2T transformations. Therefore, it is important that OCL tools implement this functionality. Fortunately, important advances have been achieved in this area [23]. As a consequence, powerful and flexible code generators can be implemented with a minimum efort. The code corresponding to OCL specifications has to be injected within the application code. In this context, two approaches may be distinguished. The first one covers the situation in which the model code was either produced by another tool or was manually written. In this case, examples on how the assertion code could be injected must be provided. The most time consuming operation is the management of invariants' evaluation. The second approach treats the case when the entire code is generated by the OCL tool. This one offers a stronger automation support. Even the code containing invariants' calls can be generated, if users specify in advance the strategy. The drawback of this approach concerns OCL tools makers, which must provide users enough flexibility in order to customize code generation in accordance with their requirements. In both mentioned approaches, the management of execution exceptions (assertion failures) must be considered.
5. **Offer browsing facilities: from the textual editor to the output pane, from the output pane to the (meta)model browser, from (meta)model browser to different diagrams representing the model element selected in the browser.**
   Justification: This functionality is probably less important compared with the others. However, we have included it since it enables users to understand quickly the results of evaluating OCL specifications on different model instantiations and to take advantage of efficient specifications.

In recent years, different research groups have studied various aspects related to OCL specifications such as: refactoring OCL specifications, patterns in OCL specifications, translating OCL specifications in other formalisms (HOL, CSP) in order to prove their correctness. We have not yet included these functionalities in our requirements list since we consider that implementing those from the list should be a precondition for taking a real advantage of the latters. Before translating an OCL specification into any other formalism, we must confide that this specification is correct and complies with our intent.

6 An overwiev of the existing tools supporting OCL

There are two different positions regarding tools supporting OCL. One is represented by modular tools especially conceived to be integrated with other modeling tools and Integrated Development Environments (IDEs). The most known is the Dresden OCL Toolkit. Octopus, MDT-OCL, KMF, and most of the existing

tools are included in this category. The second is represented by stand alone tools. Together, USE, XMF and OCLE are included in this group. Each group offers a set of advantages and has also some drawbacks.

The tools from the first group are well integrated at least in a host modeling tool, or better, in different other tools like in case of Dresden OCL Toolkit. Their weakness is mainly due to the strong dependency on the host tool. The tools of the second group have the advantage of independency, but are probably less integrated with other tools.

Analyzing the latest versions of the most used commercial UML tools, we can notice that, except for Together [22], the OCL support offered by Rose [32], MagicDraw [18], Poseidon [29], Telelogic Tau [33] and Telelogic Rhapsody [30] lacks or is insignificant. Even more strange, on the Objects by Design web page [24], among the 16 criteria proposed for choosing a tool, there are neither explicit nor implicit references to OCL.

An incontestable criterion for classifying tools supporting OCL groups these in commercial and non-commercial (free or open source) tools. In the first group, Together is from far the leader with respect to the functionalities offered, since it supports OCL both at the metamodel and model level. Together implements also model transformation support by means of QVT. However, not even Together is very closed to the ideal tool that we would like to work with. The UML metamodel it uses is not standard-compliant and its metamodel's documentation is not fully available. Therefore, working with OCL specifications at the metamodel level is awkward. Static evaluation of OCL expressions specified at the M1 level is not possible. The Java code obtained by transforming the OCL specification is large and, by consequence, understanding it requires significant effort.

Other tools, such as Oclarity - the Empowertech Rose AddIn [27], only offer compilation support for OCL specifications. For this reason, their analysis was not included in this paper.

Within the second group, that of non-commercial tools, the great majority have emerged from academia. ATL [10], Dresden OCL Toolkit [11], E-Platero (formerly Pampero) [12], Epsilon Object Language (EON) [13], GME [14], HOL-OCL [15], Key [16], KMF [17], MOVA [20], Naomi [21], OCLE [28], RoclET [31], USE [34] - listed in alphabetical order, have been all developed in academia. Each of the above mentioned tools is focused on one or more aspects of interest for the teams involved. In our point of view, Dresden OCL Toolkit and USE must both be considered for their influence on the other tools, for the functionalities they were the first to implement (some of them still unique in the universe of tools supporting OCL), and for their continuous improvement end extension. Also, HOL-OCL, ATL, GME, Key, RoclET, MOVA provide different useful functionalities, not yet implemented by other tools. KMF must be mentioned as well, since its OCL library is used by different tools supporting OCL. EMF MDT-OCL [19], oAW [23], and XMF [35] have a distinct place within the non-commercial tools. EMF MDT-OCL is an open source tool using the KMF library that offers OCL support within the EMF framework. oAW, developed by the oAW consor-

tium, supports an OCL like language, named Check. We consider that, among the three, XMF has a priviledged position since it uses an extended OCL version (supporting model transformation and modifiers' specification). This is the first tool offering an unconditioned support for LDD.

At this time, the non-commercial tools are from far more numerous compared to commercial ones. The functionalities implemented in free or open source tools are more varied and interesting. These are mainly research tools. As consequence, their available documentation does not contain all the needed information. HOL-OCL and USE are two notable positive examples. Therefore, we think that, for now, the non-commercial tools are not in the position of being widespread used in modeling, at an industrial level. They can be successfully used, but only by the OCL community researchers or by mixed teams (from research and industry), as acknowledged by different reports.

Are the tools supporting OCL the only barrier against a widespread use of OCL or OCL like languages now? Our answer is no. How many modelers are convinced about the need of checking models' consistency against WFRs? How many developers use Design By Contract? Trying to answer to the last two questions, we assume that these numbers are not as significant as they should be. Moreover, as discussed in Section 3, some OCL-specification aspects are still waiting for a better answer. As mentioned by this workshop's organizers in the Call for Papers, it seems that the time for a new generation of tools supporting OCL has arrived. We totally agree with that.

7 Related works & conclusions

The topics addressed in this paper are also analyzed in [9], [6] and [4].

In [9], Toval and others mention that in 2003, almost all the analyzed tools were in the development stage, and "that most of the existing tools have progressed slowly and have been available only as beta versions". They also mention that "The future regarding OCL tools is rather uncertain". Now, five years after Toval's analysis, we can notice that tools have improved their functionality (especially Dresden OCL Toolkit and USE) and many new other tools hav been released, including commercial tools like Together. From the five groups of criteria used in [9] to evaluate tools supporting OCL, excepting the first group - "Static Analyze and Type checking" and the last one - "The version of the OCL standard", all the other criteria have been included in our requirements in different forms. Normally, in order to be synchronized with tools' evolution, the criteria have been updated. In our requirements, new criteria such as: comparing the results of static and dynamic evaluations, browsing between different views of the same model element, working with large models and so on were included.

In [6], Gogolla and collaborators propose a first OCL benchmark for evaluating OCL engines. For sure, the proposed tests are useful. In case of OCLE for example, the benchmark helped us in identifying an evaluation bug related to exceptions' management. However, other tests, especially those related to

the evaluation of expressions containing `undefined` values, need a more careful analysis, as mentioned in Section 3.

An interesting approach within the universe of tools supporting OCL, that of the Dresden OCL Toolkit, is described in [4].

The results presented in this paper highlight some aspects related to the improvement of OCL specifications. Moreover, by means of the presented examples, we hope to contribute to the improvement of OCL benchmarks, such as the one proposed in [6]. Our experience proved that validating models before compiling and evaluating OCL specifications is mandatory. As WFRs are specified in OCL, the model conformance with its modeling language's rules is a required functionality for all tools supporting OCL. By means of examples illustrating evaluation exceptions, we have shown that both static and dynamic evaluations are required in validating specifications. The model description is realized by weaving the specification made in the modeling language with the specification made in OCL. Therefore, the tools must enable a natural navigation between the evaluation result and the model elements represented into the browsers and appropriate diagrams. Moreover, tools must implement functionalities related to the support for specifications' reuse at M2 and M3 levels. The above mentioned functionalities are required irrespective of the kind of applications considered. Complying with standards ensures a greater number of users and feedbacks. By contrast to Domain Specific Modeling Languages (DSMLs), which are different one from another, being conceived to better manage diversity, the expression language (OCL) is the same for all DSMLs. Therefore, the language evolution and tool makers must take advantage of the experience acquired in different domains.

Hoping that knowledge resulting from different applications of OCL specifications attained the critic point enabling the raise of new more useful tools supporting OCL.

References

1. Cabot, J., Teniente, E.: Transformation Techniques for OCL Constraints. Science of Computer Programming Journal, vol. 68/3, 179–195. Elsevier (2007)
2. Chiorean, D., Borteş, M., Coruţiu, D.: Proposals for a Widespread Use of OCL. In: Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica (2005). Technical report LGL-REPORT-2005-001. Online at: `http://lgl.epfl.ch/members/baar/oclwsAtModels05/technicalReport.pdf`
3. Clark, T., Sammut, P., Willans, J.: Applied Metamodeling - a Foundation for Language Driven Development, Second Edition. Ceteva (2008)
4. Demuth, B.: The Dresden OCL Toolkit And Its Role In Information Systems Development. In Proceedings of 13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice (ISD'2004). Vilnius, Lithuania (2004)
5. Oliver, I.: Using Class Relationships for Identifying Invariants. Technical report NRC-TR-2006-006, Nokia Research Center (2006). Online at: `http://research.nokia.com/files/NRC-TR-2006-006.pdf`

6. Gogolla, M., Kuhlmann, M., Buttner, F.: A benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In Krzysztof Czarnecki, editor, Proc. 11th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2008), LNCS 5301, 446–459. Springer, Berlin (2008)
7. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Vlkel, S.: MontiCore: a framework for the development of textual domain specific languages. ICSE Companion (2008), 925–926
8. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Consistency Checking and Visualization of OCL Constraints. In LNCS, vol. 1939, pp. 294–308. Springer, Heidelberg (2006)
9. Toval, A., Requena, V., Fernandez, J.L.: Emerging OCL tools. Software and Systems Modeling, vol. 2, nr. 4, 248–261. Springer Berlin / Heidelberg (2003)
10. ATL web page, http://modelware.inria.fr/rubrique12.html
11. Dresden OCL Toolkit, http://dresden-ocl.sourceforge.net/index.php
12. E-Platero web page, http://www.oneclipse.com/plugins/education/eplatero/view
13. Epsilon web page, http://www.eclipse.org/gmt/epsilon/about.php
14. GME web page, http://www.isis.vanderbilt.edu/projects/gme/
15. HOL-OCL web page, http://www.brucker.ch/projects/hol-ocl/
16. Key web page, http://www.key-project.org/
17. KMF web page, http://www.cs.kent.ac.uk/projects/kmf/
18. MagicDraw web page, http://www.magicdraw.com/
19. MDT-OCL web page, http://www.eclipse.org/modeling/mdt/?project=ocl
20. MOVA web page, http://maude.sip.ucm.es/mova/
21. Naomi web page, http://mocl.sourceforge.net/
22. Borland Together web page, http://www.borland.com/us/products/together/index.html
23. openArchitectureWare web page, http://www.openarchitectureware.org/
24. Objects by Design web page, http://www.objectsbydesign.com/tools/modeling_tools.html
25. OCL benchmark, https://muse.informatik.uni-bremen.de/wiki/index.php/OCL_Benchmark_0_-_CivilStatus
26. OCL Specification v2.0, http://www.omg.org/docs/formal/06-05-01.pdf
27. Oclarity web page, http://www.empowertec.de/products/rational-rose-ocl.htm
28. OCLE web page, http://lci.cs.ubbcluj.ro/ocle/index.htm
29. Poseidon web page, http://www.gentleware.com/products.html
30. Telelogic Rhapsody web page, http://modeling.telelogic.com/products/rhapsody/index.cfm
31. RoclET web page, http://www.roclet.org/
32. Rational Rose web page, http://www-306.ibm.com/software/awdtools/developer/rose/index.html
33. Telelogic Tau web page, http://www.telelogic.com/products/tau/index.cfm
34. USE web page, http://www.db.informatik.uni-bremen.de/projects/USE/
35. XMF web page, http://www.ceteva.com/ceteva.html