EASST

Proceedings of the Workshop
The Pragmatics of OCL and Other Textual Specification Languages
at MoDELS 2009

Requirements Analysis for
an Integrated OCL Development Environment

Joanna Chimiak–Opoka, Birgit Demuth,
Darius Silingas, Nicolas F. Rouquette

15 pages

Requirements Analysis for
an Integrated OCL Development Environment

Joanna Chimiak–Opoka[1], Birgit Demuth[2],
Darius Silingas[3], Nicolas F. Rouquette[4]

[1] Institute of Computer Science, University of Innsbruck, Austria joanna.opoka@uibk.ac.at
[2] Department of Computer Science, Technische Universität Dresden, Germany
birgit.demuth@tu-dresden.de
[3] No Magic Europe, Savanoriu av. 363, 49425 Kaunas, Lithuania darius.silingas@nomagic.com
[4] Jet Propulsion Laboratory, Caltech, M/S 301–270, 4800 Oak Grove Drive Pasadena, CA
91109, USA nicolas.f.rouquette@jpl.nasa.gov

Abstract: An Integrated OCL Development Environment (IDE4OCL) can significantly improve the pragmatics and praxis of OCL. We present the domain concepts, tool–level interactions with OCL and the use cases we identified in a systematic analysis of requirements for an IDE4OCL. The domain concepts is an important contribution of our work as it attempts to clarify inconsistencies in the relevant specifications. Because OCL is not a stand–alone language, the OCL landscape includes several interacting tools including an IDE4OCL. The use cases describe our vision of the desired functionality unique to an IDE4OCL. The results of our analysis and the long term vision of our work should be relevant to developers of OCL tools as well as to the OMG Request for Information regarding the UML Futures[1]. Our work is relevant to the UML Futures Roadmap because providing OCL for the constraints in the UML specification has been a longstanding problem at the OMG.

Keywords: OCL concepts, OCL development, OCL pragmatics, OCL tool support, requirement specification

1 Introduction

The specification and implementation of the Object Constraint Language (OCL) involves three **language definition aspects**: syntax, semantics and pragmatics. For any language *syntax must be specified prior to semantics since meaning can be given only to correctly formed expressions in a language; semantics needs to be formulated before considering the issues of pragmatics since interaction with human users can be considered only for expressions whose meaning is understood* [SK95]. For OCL, the dependencies amongst these aspects are reflected in the chronological phasing of their maturity with pragmatics lagging behind semantics which is lagging behind syntax.

For OCL, the broad support for the syntactic and semantic aspects stand in sharp contrast with the dearth of support for **pragmatics**. Formalisations of OCL syntax and semantics are the basis for building tool support for automatic checking of syntactical correctness and formal reasoning

---

[1] http://www.omg.org/news/releases/pr2009/06-18-09.htm

about properties of OCL specifications. In contrast to syntax and semantics, pragmatics cannot be formalised [Bjo06]. However, pragmatics entices programmers to use a language. This implies the fact that pragmatics does not need theory, it needs practical solutions. Despite recent advances in tool support for OCL [B+05], much remains to be done conceptually and technically to encourage practitioners to work with OCL tools [CPP08] as defining OCL expressions is still *difficult, error–prone and a time–consuming task* [Ack01].

As a two–language hybrid artifact, a MOF–based model with OCL constraints is inherently more *difficult to understand and evolve* than an equivalent single–language artifact. For hybrid models, there is ample empirical evidence that the organization of the MOF–based model has a strong influence on the understandability of OCL constraints for that model [C+07]. This paper focuses on the pragmatics of OCL in the context of the life cycle of hybrid models. We consider here **internal pragmatics**, i.e. pragmatics within the OCL development process and one considering its impact on developers.

In [CPP08], it was mentioned that *tools' constituents (editors, compilers, browsers) must implement the functionalities established by integrated development environments* (IDEs). We want to go one step further with a systematic requirement analysis for an integrated OCL development environment, which we call IDE4OCL. Instead of targeting the ideal OCL tool, we focus on an IDE supporting the development of OCL specifications as part of an overall OCL tools landscape. In terms of an abstracted typical life cycle of an OCL specification to be *plan–do–check–act cycle* (Fig. 1), we focus on support of the second and the third steps where an OCL specification is the focus of the development and verification activities. In the life cycle we consider **external pragmatics**, i.e. how the OCL specifications are used outside an IDE4OCL.

Act—use the OCL specification to increase the quality of systems built with the conceptual model. It is related to the (external) **pragmatics**, as we consider usage of the specification.

Check—assess if the OCL specification meets the objectives/requirements. It is related to the **semantics**, as semantic properties of the specification are tested or verified, and (internal) pragmatics focusing on ease of assessment.

Plan—determine objectives/requirements of an OCL specification for a conceptual model.

Do—define the OCL specification or a part of it to operationalize the specification objectives. It is related to the **syntax**, as the syntactically correct specification is defined using error prevention mechanisms, and (internal) pragmatics focusing on ease of development.

Figure 1: A life cycle of an OCL specification seen as the Deming cycle [Dem86].

To flesh out the requirements for an IDE4OCL, we start with domain analysis and define the system context specifying what are the responsibilities of an IDE4OCL. Then we decompose

the identified use cases into tool features that are similar to the features implemented in modern integrated development environments like Eclipse platform. The applied requirements analysis approach is presented in more detail in [SB08, SB09]. On first approximation, we identified three classes of requirements: domain analysis, system context, and use case model. Domain analysis is based on a refined OCL metamodel, which is re–categorized from the pragmatical view and extended with additional concepts from programming, such as the notions of Project and Library. For defining the system context, we focus on the information flow between tools that either make use of OCL expressions or can help a developer specify or evaluate them.

The structure of the paper corresponds to the requirement analysis steps for an IDE4OCL. At first we analyse the domain of an IDE4OCL (Section 2). Next we describe the identified use cases and features (Section 3). The last section provides conclusion, and discusses relevance of our work and future steps.

2 Domain Specification

In the subsequent subsections we define domain concepts and give a context of an IDE4OCL.

2.1 Domain Concepts

Our proposal is based on our academic teaching and tool development experience [DW09, C$^+$08] and aims to clarify problems with understanding different concepts of OCL specification by students and developers. We also introduce axillary concepts [BD07, CO09] as means to improve OCL application to different metamodels and OCL development process. For better understanding we introduce domain concepts in three stages. At first we review the latest OCL standard specification [OMG06] (in the rest of the paper called *the standard* for short) and introduce our categorisation of related concepts (Fig. 2). Next we relate the OCL concepts with the modeling abstractions levels (Fig. 3). Finally, we introduce concepts necessary for the context and requirement specification for an IDE4OCL (Fig. 4).

2.1.1 OCL Concepts

We propose a **2–layer view** of the domain concept for OCL [OMG06]. It introduces a categorisation of these concepts considering a language definition [SK95] with syntax, semantics and pragmatics. Within the domain description we preserve the original syntax and semantics given by the standard and we add the third perspective, namely (external) pragmatics, to express how the concepts are used at a level of abstraction that matters for the IDE4OCL requirements. This model leaves out several aspects of pragmatics that are simply out of scope for the purposes of this paper. In Fig. 2 we give an overview of concepts and show separation between syntactical and pragmatic view.

The top row of the **syntactic context layer** (above the dashed line in Fig. 2) presents the top level concepts with their meaning and relations corresponding to [OMG06, Clause 12.12]: Package [(12.12.1)], Context Declaration [(12.12.2)] and Expressions [(9.3)]. The middle row introduces further categorisation of context declarations depending on the type of a contextual element [(12.12)], i.e. for Classifier, Operation and Attribute or Association. The bottom row corresponds to the

syntactical categories from [OMG06, Section 12] with their original meaning preserved: Definition [12.5,12.12.6], Invariant [12.6,12.12.6], Precondition [12.7,12.12.9], Postcondition [12.7.2,12.12.9], Operation Body Expression [12.10,12.12.8], Initial Value Expression [12.8,12.12.4], and Derived Value Expression [12.9,12.12.4].
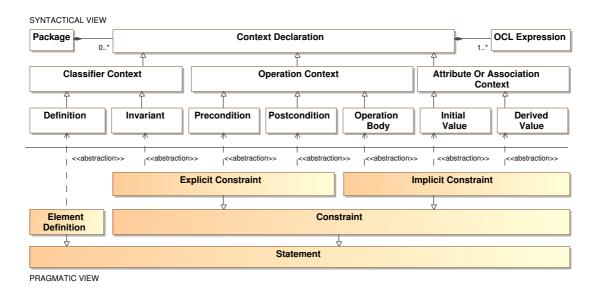

Figure 2: OCL concepts from syntactic and pragmatic point of view.

The leaf concepts of the syntactic context layer relate to the concepts of the **pragmatic domain layer** (below the dashed line in Fig. 2). A description of these concepts from left–to–right in the figure follows.

Element Definition is a new model element added by the OCL specification. It is a Definition [12.5,12.12.7] which can introduce an attribute, an association or an operation.

Constraint is any construct used to impose restrictions on a model instance. It can be defined explicitly or implicitly.

Explicit Constraint groups syntactical categories that are explicitly classified as a constraint in the standard and which consist of an OCL expression of Boolean type, i.e. invariant, post- and precondition. The standard explicitly introduces *guards* [12.11] as a semantic concept. We skipped guards in our domain concepts because a guard is a precondition from the syntactical view as well as from the pragmatic view.

Implicit Constraint groups syntactical categories that are not classified as constraints in the standard and consist of OCL expressions of arbitrary types, i.e. operation body, initial and derived value. This concept groups elements that are *used as constraints*, i.e. to impose restrictions on a model instance. They provide an expected value (e), e.g. as a derived value for an attribute, which is compared with an actual value obtained from a

model instance (a), e.g. of the attribute, and this comparison forms an equation, a Boolean expression (e=a), which is an implicitly–defined constraint.

Statement is the most general term in the pragmatic view. It is introduced to denote a single chunk of an OCL specification that can be developed within an IDE4OCL.

2.1.2 Modeling Abstraction Levels

As mentioned before, OCL is a language which always depends on another modeling language. Without another language used for modeling, it does not make any sense to define constraints because OCL is used for constraint specification but not for modeling itself. Thus, besides OCL, a modeling language is required to define a model on which OCL constraints shall be specified (Fig. 3). We assume the OMG MOF Four Layer Metadata Architecture which is used to arrange and structure the metamodel, the model, and its model instances into a layered architecture. Generally, four layers exist, the meta–metamodel layer (M3), the metamodel layer (M2), the model layer (M1), and the model instance layer (M0).
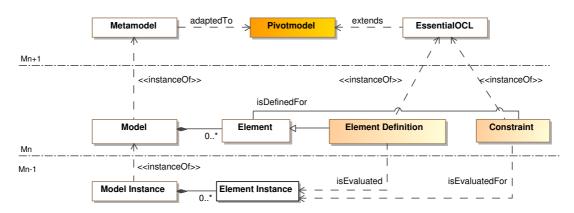

Figure 3: Generic Three Metadata Layer Architecture for OCL

OCL statements can be defined on both, metamodels or models and be evaluated on models or model instances, respectively. Thus, the four layer metadata architecture can be generalized to a **Generic Three Metadata Layer Architecture** [DW09]. On the Mn+1 layer lies the metamodel that is used to define the model that shall be constrained. The metamodel defines a modeling language. It is required that it is a MOF (or EMOF/Ecore)–based model, i.e. MOF/EMOF/Ecore itself or an instance of MOF/EMOF/Ecore, like UML or a DSL (domain specific language). The used metamodel has to be adapted to the so–called Pivot model [BD07] .

Pivot Model is an intermediate metamodel that allows the alignment of arbitrary metamodels with that of OCL. By directly supporting generics in this metamodel, modeling all of the template types and operations in the OCL standard library becomes possible. The pivot model is designed for any OCL tools and understood as a general concept or pattern.

The **Pivot model based architecture** provides therefore a flexible model repository adaptation mechanism and allows using OCL for any modeling languages which is an important feature

for an IDE4OCL. The Pivot model is designed on base of Essential OCL [OMG06]. Essential OCL plays the role of the OCL metamodel. However, it should be noted that Essential OCL is currently not very well–defined. In [RG99], a more founded metamodel was proposed for the first version of OCL. From a pragmatic point of view, Essential OCL is adequate for the implementation of the Pivot model, how it is proven in Dresden OCL2 for Eclipse. On the Mn layer lies the model which is an instance of the metamodel that is enriched by the specification of OCL constraints. Finally, on the Mn-1 layer lies the model instance on which the OCL constraints shall be verified. Please note, that in the context of such a generic layer architecture, a model instance can be both a model (like an UML class diagram) or an object (like a Java object or relational data).

2.1.3 OCL Development Concepts

In the last stage of the concepts definition we introduce concepts (Fig. 4) related to the OCL development process within an IDE4OCL and information exchange within the OCL tool landscape (Fig. 5).
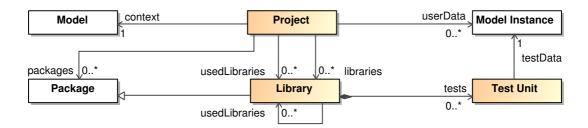

Figure 4: OCL Development Concepts and their interrelations.

Project is a collection of packages and libraries that are developed within an IDE4OCL. Libraries can be imported (used). A project refers to a contextual model for which OCL statements are defined and to model instances on which OCL statements are evaluated.

Library is a kind of package with intent to be reusable [CO09]. Libraries as reusable artifacts can be imported into a library and form a hierarchy of libraries. Additionally, a library contains test units.

Test Unit is used to test an element definition before it is reused in another library or in an OCL statement. It is related to a model instance which provides test data and is an instance of the library contextual model [CO09].

2.2 Context Specification

After a decade of several prototype implementations of OCL based tools and toolchains for multiple purposes, the OCL landscape is already manifold. This makes it difficult to classify these tools. We propose a simplified view on an **OCL tool landscape** (Fig. 5) required to define the

context of an IDE4OCL which is based on possible usage scenarios of OCL [DW09]. Based on our academic and industrial practice in OCL software development we identified tool–integration requirements for an IDE4OCL, which is responsible for development of correct OCL statements (corresponding to *do* and *check* in Fig. 1), whereas other tools consume OCL statements (corresponding to *act* in Fig. 1). The character of the overall architecture can be considered as a toolchain or a collection of plug–ins.

From a **toolchain** perspective, portability of OCL expressions across tools requires all tools to produce consistent OCL interpretations of the same OCL expressions. This approach requires a complete OCL specification to be respected by every tool vendor involved in the toolchain, including consideration of factors beyond the standard such as [CPP08, Section 3].

From a **plug–in architecture** perspective, there must be only one component responsible for OCL interpretation. In this paper we assume possibility of exchanging OCL expressions with the full preservation of their semantics. This assumption enables us to incorporate a feedback from usage of an OCL specification and thus impact its further development to enable continuous improvement in an OCL specification life cycle.
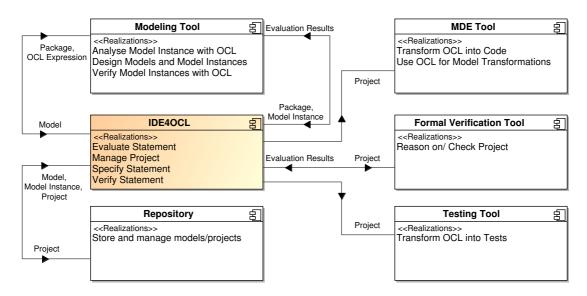

Figure 5: The OCL tools landscape: relations between tools.

Below we will discuss particular tools in the OCL tool landscape and **information exchange** between them and an IDE4OCL, which itself will be described in the next section. In the context of an IDE4OCL, the most tightly related tools with bidirectional communication are a modeling tool, a repository and a formal verification tool. The remaining two tools, namely a MDE tool and a testing tool, only consume OCL statements developed within an IDE4OCL. However, all tools in the landscape exchange different artifacts, in the diagram we only focus on communication with an IDE4OCL.

A **modeling tool** is typically an UML tool that allows specification of constraints in any language, most frequently just as strings. In this case the integration should be tight (e.g. via plug–in mechanism) as both tools provide services for one another and constitute a symbiosis required by

the hybrid nature of the models. On one side IDE4OCL provides OCL Expressions/Package for a given model designed in the modeling tool, next they can be evaluated within IDE4OCL. The evaluation results can be returned to the modeling tool, where model verification or analysis is performed. On the other side, model and model instances are required to create OCL statements. Then they can be designed within the modeling tool.

Another possibility to obtain models and model instances is to fetch them from a **repository**. In our architecture we consider a repository which plays a role similar to a version management system in software development or a data warehouse in a database system. The repository is a generic one, i.e. it is a kind of MOF repository whose structure is not determined by an underlying metamodel, thus it can store any MOF based metamodels, models, model instances, OCL expressions and projects. The artefacts from the repository can be loaded into an IDE4OCL as working copies and subsequently modified and archived. Specifying the complete functionality of a repository is a complex topic beyond the scope of this paper, thus we consider only it realizing storage and management of models/projects. Other tools can access the repository to obtain desired OCL expressions together with related models, but as mentioned before we focus on communication with an IDE4OCL only.

The last tool which is tightly related to an IDE4OCL is a **formal verification tool**. This tool has a producer and consumer role, as it can help to obtain semantically correct OCL specifications and use them to formally verify model instances. It is crucial to have any kind of formal reasoning supporting an IDE4OCL to be able, e.g. to determine a specification satisfiability or to detect contradicting constraints. However, formal verification is a too complex problem to be considered as an integral part of an IDE4OCL. Instead we consider an integration of existing approaches, such as HOL–OCL[2] [BW08] an interactive proof environment based on a semantic framework described in [Bru07], an interactive theorem prover [BHS07] (the transformation is a part of the KeY tool[3]) based on a translation of UML class diagrams with OCL constraints into first–order predicate logic described in [BKS02], PVS[4] a theorem prover together with a translation of UML class diagrams and a subset of OCL into its input language described in [Kya06].

Two approaches regarding the OCL evaluation can be considered, interpretation and code generation. The first one we consider to be in the scope of an IDE4OCL functionality, but the second one not; we outsource it like the formal verification. We consider a code generation to several target languages, such as Java and SQL, to be in the scope of an **MDE tool** (model driven engineering), which obtains OCL Statements and related information from an IDE4OCL. In case of a *code generator*, the execution of the target language code is done by a runtime system independently of an IDE4OCL. The generative approach includes two steps: (1) The code for an OCL statement must be generated, most frequently by using templates or transformation rules defined on the metamodel elements for the model and its constraints (Mn+1 and Mn). (2) The generated code must be woven with the model or application code. Another functionality of an MDE tool, where input from an IDE4OCL can be used, is a *transformation* of a model into another model defined on another metamodel. A model (Mn = M1) is transformed using transformation rules defined on its metamodel (M2). The OCL constraints specified on the model

---

[2] http://www.brucker.ch/projects/hol-ocl/

[3] http://www.key-project.org/

[4] http://pvs.csl.sri.com/

are transformed as well. Examples for model transformations are UML/OCL to SQL schema transformation [5], or UML/OCL to XML/XQuery transformation.

A similar approach is used for testing. A **testing tool** generates code based on OCL constraints to verify constraints for objects during software development. Typically, OCL constraints are defined on a model (Mn = M1) for which a code implementation shall be tested (M0).

As already mentioned, our view of the OCL tool landscape is simplified, but it can be **easily extended** with other tools like model execution tools, e.g. [JZM07] or model simulation tools, e.g. [KDH07] which would communicate with an IDE4OCL in mono- or bidirectional manner to complete alternative OCL development or OCL usage scenarios. It is important to precisely define scope of responsibilities, information flow and later on required interfaces.

3 Requirements

Based on the domain description provided in the previous section and our experience in tool development, we present the requirements for an IDE4OCL: use cases (Section 3.1) and features (Section 3.2). We based our selection on passive observation of improvement in OCL tool and IDE landscape as well as an active participation in OCL tool development. To increase the completeness of the selected features we discussed our selection with our developer teams.

3.1 Use Cases

We distinguished four main use cases (to be realised by IDE4OCL, compare Fig. 5), namely specification, evaluation and verification of statements and project management.

Specify Statement   This is the basic use case of an IDE4OCL, where an OCL developer specifies an OCL statement. We consider here the creation of a new statement from the scratch or modification of an existing one. Since OCL has a well–defined textual concrete syntax, the requirements for editing OCL expressions are similar to those for editing source code in textual languages. In this use case we consider also such functionality as refactoring, reuse, and debugging, which are described in the feature subsection.

Evaluate Statement   A specified statement can be evaluated by an OCL interpreter, which parses and executes the statement defined on the model for the model instance, working on the model and its objects (Mn and Mn-1). This use case can be performed on request from an OCL developer or from another tool in the OCL tool landscape. As mentioned in Section 2.2, we consider the evaluation in form of code generation as an outsourced functionality.

Verify Statement   We can consider formal and empirical attempts to verify an OCL specification. The former one, as mentioned in Section 2.2, we consider to be outsourced, due to its complexity. The latter one in form of statement testing, should be supported by an IDE4OCL. Testing is a complementary means to a formal verification. It enables dynamic analysis as opposed to formal verification enabling a statical analysis of hybrid models. However, testing of a

---

[5] http://dresden-ocl.sourceforge.net/

OCL statement is crucial [CO09], it is not so well–accepted as testing of programs where it is used as an evaluation and prevention mechanism [GH88]. There are many reasons for testing. In the context of OCL the most important ones are the facts that testing *reduces bugs in existing and new features, is good documentation, reduces the cost of change, enables refactoring, defends against other programmers and reduces fear* [BC03]. An IDE4OCL should at least support testing at the unit test level, i.e. testing of single OCL statements [CO09].

Manage Project For an efficient support of OCL development, especially if one has to deal with big projects, management of all artifacts within a project is required. This use case covers management issues within an IDE4OCL and related to communication with other tools. In respect of an IDE4OCL, the current status and dependencies between all artefacts as well as navigation between them should be supported. Concerning the communication with other tools, fetching and storing artifacts from and to other tools should be supported.

3.2 Features elicitation

In this subsection we list general and specific features of an IDE4OCL focusing on the statement specification use case. To collect **general features** we use experience with existing successful tools covering different types of textual languages, namely programming and formal ones. The general features are applicable to an IDE4OCL as OCL has a manifold character. On the one hand, regarding textual syntax OCL is similar to programming languages and the long term experience with tools supporting work of programmers can be inspiration for development of OCL tools. On the other hand, as it is more formal than programming languages, usually similar difficulties appear as in the formal specification domain, therefore this field can be a further inspiration. To collect *specific features* (in italics) we based on experience with OCL tool usage and development as well as our involvement in the standardisation process. As we do not want to prioritize features we list them in alphabetical order. The prioritizing of features and completion of the list should be a further discussion topic to become a reference list for development of new OCL tools or improvement of existing ones.

Association End Navigability OCL implementations should support association end navigability independently of the navigability of the underlying association in the model. Although navigability (as defined in UML) should not matter for OCL, the OCL specification is sufficiently vague on this point[6] that it creates significant problems for OCL implementations that provide such support. Proposed improvements for OCL2.1[7] are important for the pragmatics of OCL as long as the MOF metamodels are sufficiently well–formed to avoid ambiguities even if support for navigating non–navigable association ends is available[8].

Autocomplete enables predicting a word or phrase that the user wants to type in without the user actually typing it in completely. Not only OCL grammar but also an underlying metamodel has to be included in the autocomplete mechanism. For example, selection or classifiers

---

[6] http://www.omg.org/issues/ocl2-rtf.open.html#Issue10825

[7] See Clause 7.5.3 in http://www.omg.org/cgi-bin/doc?ptc/09-05-02

[8] e.g., see https://bugs.eclipse.org/bugs/show_bug.cgi?id=194245

after typing the context keyword or suggestions for dot and arrow navigations. The point to address accessibility of elements, i.e. developing a well–formed OCL requires a careful check that the references in an OCL expression resolve to accessible elements from the context of that OCL expression. This feature can improve efficiency and ease of editing and additionally provide an error prevention mechanism.

Auto Indentation helps to better convey the structure of code to human readers. In case of OCL, indentation can be used to show the relationship between nested structures.

Basic Editing is a set of features related to editing any kind of text documents, which can be useful when editing OCL statements. In this category the following features can be considered: spell checking, regular expression based find & replace (single or multiple line), encoding and newline conversion, multiple undo/redo, rectangular block selection. These features can improve ease of editing.

Code Folding enables user to selectively hide and display sections of an edited file, which is especially useful in case of editing large files.

Collaborative Editing allows several people to edit a file using different computers. This feature could be also realized as a repository functionality. The advantage of its implementation within an IDE4OCL is possibility of team/pair work to enable knowledge transfer (e.g. teacher–student) also in the case of geographically spread developers.

Debugging, especially a systematic debugging [Zel05] is unavoidable and a major economical factor, especially if a language is perceived as difficult to understand so bugs are not obvious. It should support developers in understanding a nature and case of a bug offering functions such as running a statement step by step, breaking a statement to examine the current state, and tracking the values of some variables. Additionally, it could enable to modify the state of variables while an OCL statement is interpreted and setting state guards. For traceability, automation and logging of all debugging activities is important. A support of test generation based on debugging activities is an optional functionality.

Document Interface is a set of features supporting editing of multiple documents and it covers support of: multiple instances, single and multiple document window splitting, multiple document overlappable windows, tabbed document interface. It is especially a useful feature while working with hybrid models and enables following relationships between textual and graphical notations.

Hybrid OCL/MOF View should provide an Abstract Syntax Tree and additionally highlight the context of any OCL expression in the MOF–based metamodel. The problem of hybrid OCL/MOF metamodel view is new and the recent discussions amongst experts at the OMG[9] indicates that experts could also benefit from better tool support for this hybrid view.

Macro mechanism enables short sequences of keystrokes and mouse actions to be transformed into other, usually more time–consuming, sequences of keystrokes and mouse actions.

---

[9] http://www.omg.org/issues/issue7364.txt

Name Resolution The environment of an OCL expression defines what model elements are visible and can be referred to an expression [OMG06, Clause 8.3]. Such references often take the form of simple or package–qualified names. However, adequate support for name resolution in OCL may require additional operations extending the metamodel of the domain for name resolution purposes as indicated in the OCL specification for the UML metamodel [OMG06, Clause 8.3.8]. In fact, the OCL specification is unnecessarily specific regarding the UML as the additional operations would be required of any MOF metamodel which includes the metaclasses extended in Clause 8.3.8. For example, adequate name resolution for foundational UML (fUML) models [10] would not require the additional operations for State or Transition since fUML does not merge the BehaviorStateMachines package of the UML superstructure.

Profiler enables performance analysis using information gathered when an OCL Statement/Specification is evaluated. In case of programs, it is typically used to determine for which sections of a program it is profitable to make optimization. Similarly, it can be used to determine which OCL statements are most frequently evaluated and focus on their optimization.

Refactoring Support for renaming and restructuring entities preserving the original semantics. For full support dependencies between statements must be analyzed to perform series of renaming activities. Also, extracting a definition or a template from a statement should be supported to avoid code duplications.

Reuse Support can be realized at different levels. At the same abstraction level OCL code can be reused by composition of statements, template and library import mechanisms. A specification from the upper abstraction level can be reused during development of a specification at a lower level to to ensure correctness of metamodel instantiation (compare [CPP08, Feature 2 in Section 5]).

Statement/Element Browser enables to browse, navigate, or visualize (e.g. as an outline) the structure of an OCL project, including OCL Statements, Elements and Element Instances (compare [CPP08, Feature 5 in Section 5]).

Statement Coverage is used to measure the degree to which an OCL specification has been tested. To implement this feature coverage criteria have to be defined.

Static Statement/Specification Analysis is the analysis conducted without evaluation of a Statement/Specification and provides highlighting possible coding errors and metrics in simple cases or proofs of program properties by applications of formal methods. The second option we will consider as to be outsourced.

Symbol Database enables quick and easy location of Statements, Elements, Element Instances and so on based on indexing.

---

[10] http://www.omg.org/spec/FUML/, OMG document ptc/2008-11-03

Syntax Highlighting enables displaying OCL code in different colors and fonts according to the category of terms. As OCL is not a stand alone language, additionally to the OCL grammar, an underlying metamodel should be considered. **Error Highlighting** can be considered as a special type of this features, where syntactical errors related to OCL or the metamodel (e.g. unknown classifiers used as a context) are stressed by special type of highlighting. **Brace Matching** is also a syntax highlighting feature, which show matching sets of braces to help to navigate through the code and spot any improper matching. Those mechanisms can improve *readability* and is an error prevention mechanism.

Template Support enables definition, usage and management of templates. It is related to refactoring and reuse features.

Visibility and Lexical Scoping MOF metamodels have complex visibility rules due to the semantics of Element/PackageImport and PackageMerge [11]. These rules are particularly confusing because the same package can have two distinct interpretations depending on its role as a source or as a target of a package merge or import relationship. The context–sensitive interpretation of a package has subtle implications for the name resolution of OCL constraints in the context of a package with two distinct interpretations about its extent.

4 Conclusions

In this paper we presented systematic analysis of requirements for an IDE4OCL, which in our opinion can significantly improve pragmatics of OCL. We identified domain concepts, interactions within OCL tools, use cases and features of an IDE4OCL from the academic, standardisation and industrial point of view represented by authors and their collaborators experience.

To improve results of our requirement analysis we want to discuss our proposal with members of the OCL community (questionnaires[12] and interviews). Based on their feedback we plan to realise future design steps of an IDE4OCL and compare existing tools in respect to the final list of features. Our work should also be considered as a first step to integrate the heterogeneous landscape of OCL tools. We hope it to be an inspiration for a cooperation between academic and industrial tool developers, which enables standardisation of exchange protocols between tools and in the long term will increase usage of OCL by practitioners.

---

[1]1 Clause 7.3.15,39,40 of http://www.omg.org/spec/UML/2.2/Superstructure/PDF, OMG document formal/2009-02-02

[1]2 on–line surveys are accessible at http://squam.info/ide4ocl/

Bibliography

[Ack01]   J. Ackermann. Fallstudie zur Spezifikation von Fachkomponenten. In *2. Workshop Modellierung und Spezifikation von Fachkomponenten*. Pp. 1–66. Bamberg, Deutschland, 2001. (In German).

[B⁺05]   T. Baar et al. Tool Support for OCL and Related Formalisms - Needs and Trends. In *MoDELS Satellite Events*. LNCS 3844, pp. 1–9. Springer, 2005.
http://lgl.epfl.ch/members/baar/oclwsAtModels05/reportOCLWSAtModels05.pdf

[BC03]   E. Burke, B. Coyner. Top 12 Reasons to Write Unit Tests. 2 2003.
http://www.onjava.com/pub/a/onjava/2003/04/02/javaxpckbk.html

[BD07]   M. Bräuer, B. Demuth. Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. Pp. 182–193 in [mod07].

[BHS07]   B. Beckert, R. Hähnle, P. H. Schmitt (eds.). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[Bjo06]   D. Bjorner. *Software Engineering 2: Specification of Systems and Languages (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[BKS02]   B. Beckert, U. Keller, P. H. Schmitt. Translating the Object Constraint Language into First–order Predicate Logic. In *In Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC)*. Pp. 113–123. 2002.

[Bru07]   A. D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD thesis, ETH Zurich, Mar. 2007. ETH Dissertation No. 17097.
http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007

[BW08]   A. D. Brucker, B. Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. In *FASE*. LNCS 4961, pp. 97–100. Springer, 2008.

[C⁺07]   A. L. Correa et al. An Empirical Study of the Impact of OCL Smells and Refactorings on the Understandability of OCL Specifications. Pp. 76–90 in [mod07].

[C⁺08]   J. Chimiak-Opoka et al. Advanced OCL Editor based on Eclipse OCL. Presentation in the OCL2008 Workshop collocated with MoDELS'2008, 9 2008.

[CO09]   J. Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In *Model Driven Engineering Languages and Systems, MODELS 2009, LNCS 5795*. Pp. 665–669. Springer Verlag, 2009.

[CPP08]   D. Chiorean, V. Petrascu, D. Petrascu. How my favorite tool supporting OCL must look like. *EC-EASST: OCL Concepts and Tools 2008* 15, 2008.

[Dem86]   W. Deming. *Out of the Crisis*. MIT, Center for Advanced Engineering, Cambridge, MA, USA, 1986.

[DW09]   B. Demuth, C. Wilke. Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: theory and practice*. Ufa, Russia, July 2009.

[GH88]   D. Gelperin, B. Hetzel. The growth of software testing. *Commun. ACM* 31(6):687–695, 1988.
           doi:http://doi.acm.org/10.1145/62959.62965

[JZM07]   K. Jiang, L. Zhang, S. Miyake. OCL4X: An Action Semantics Language for UML Model Execution. *Computer Software and Applications Conference, Annual International* 1:633–636, 2007.
           doi:http://doi.ieeecomputersociety.org/10.1109/COMPSAC.2007.158

[KDH07]   A. Kirshin, D. Dotan, A. Hartman. A UML Simulator Based on a Generic Model Execution Engine. Pp. 324–326. 2007.
           http://dx.doi.org/10.1007/978-3-540-69489-2_40

[Kya06]   M. Kyas. *Verifying OCL specifications of UML models : tool support and compositionality*. PhD thesis, Lehmanns Media; Faculty of Mathematics and Natural Sciences, Leiden University, 4 2006.
           https://openaccess.leidenuniv.nl/dspace/handle/1887/4362

[mod07]   *Model Driven Engineering Languages and Systems, 10th Int. Conf., MoDELS 2007, Nashville, USA, Proceedings*. LNCS 4735. Springer, 2007.

[OMG06]   OMG. Object Constraint Language. OMG Available Specification. Version 2.0. May 2006.
           http://www.omg.org/cgi-bin/doc?formal/2006-05-01

[RG99]   M. Richters, M. Gogolla. A Metamodel for OCL. In *UML*. LNCS 1723, pp. 156–171. Springer, 1999.

[SB08]   *UML-Intensive Framework for Modeling Software Requirements*. 2008.

[SB09]   D. Silingas, R. Butleris. Towards Implementing a Framework for Modeling Software Requirements in MagicDraw UML. *Information Technology And Control* 38(2):153 – 164, 2009.

[SK95]   K. Slonneger, B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[Zel05]   A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.