Electronic Communications of the EASST
Volume 5 (2006)

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

Use of OCL in a Model Assessment Framework: An experience report

Joanna Chimiak–Opoka, Chris Lenz

17 pages

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

Use of OCL in a Model Assessment Framework: An experience report

Joanna Chimiak–Opoka, Chris Lenz

Quality Engineering Research Group
Institute of Computer Science, University of Innsbruck
Technikerstrasse 21a, A–6020 Innsbruck
joanna.opoka@uibk.ac.at

Abstract: In a model assessment framework different quality aspects can be examined. In our approach we consider consistency and perceived semantic quality. The former can be supported by constraints and the later by queries. Consistency can be checked automatically, while for the semantic quality the human judgement is necessary. For constraint and query definitions the utilisation of a query language was necessary. We present a case study that evaluates the expressiveness of the Object Constraint Language (OCL) in the context of our approach. We focus on typical queries required by our methodology and we showed how they can be formulated in OCL. To take full advantage of the language's expressiveness, we utilise new features of OCL 2.0. Based on our examination we decided to use OCL in our analysis tool and we designed an architecture based on Eclipse Modeling Framework Technology.

Keywords: model assessment, semantical model quality, model integration, model consistency, information retrieval

1 Introduction

The necessity of model maintenance is growing together with the increasing utilisation of models in real applications. The importance of **integration** grows with the size and the number of designed models. The aspect of integration becomes crucial if the modelling environment is not homogeneous, i.e., it has to be dealt with diverse modelling tools and even with diverse notations. Such a situation is common if various aspects of the same system have to be described. For example in the domain of enterprise architecture modelling, for the description of business processes and technical infrastructure different tools and notations can be used.

If additionally the models are large scale models with hundreds or thousands of elements they might very likely contain inconsistencies and gaps. Quality assurance of these models can not be done by pure manual inspection or review but requires tool assistance to support **model assessment**.

We have developed a **framework** that is dedicated to both the integration and the assessment of models. To support the former we designed a modular architecture with a generic repository as a central point, with a common meta model and consistency checks. For the latter we defined a mechanism for information retrieval, namely queries of different types. In our entire approach we focus on the static analysis of models.

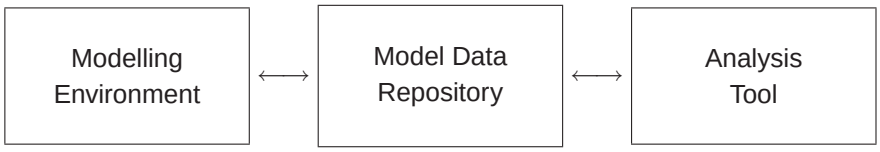


Figure 1: Base components of our framework

The language for expressing **constraints and queries** over models is an important part of the model assessment process. Depending on the language expressiveness it is possible to cover a wider or a narrower range of constraints and to retrieve more or less information from models.

One of the components of our heterogeneous tool environment for model assessment [CGIT06] is a generic analysis tool supporting queries over the model repository. Therefore, we started our case study with the Object Constraint Language (OCL, [OMG05, WK99]). In our study we want to examine all types of queries required by our methodology [BC05]. The OCL 2.0 provides a new definition and a querying mechanism which extend the expressiveness of this language. As described in [AB01, MC99], previous versions of OCL (1.x) were not expressive enough to define all of the operations required by relational algebra (RA) and were not adequate query languages (QLs). The main deficiency of previous versions was the absence of the tuples concept. In the current version of OCL, tuples are already supported. Thus all primitive operators [Cod72] needed to obtain full expressiveness of a QL, namely *Union, Difference, Product, Select* and *Project*, can be expressed. This fact encouraged us to use OCL within our framework.

The remainder of the paper is structured as follows. In the next section we give a brief introduction to our methodology. Then, we present exemplary models (section 3.1) on which the case study (section 3.2) relies. In section 4 we present a design of our analysis module and finally, in the last section we draw a conclusion.

2 Model Assessment Framework

In this section, the methodology developed within the *MedFlow* project [BC05, CGIT06] is briefly described. A broader description of the methodology developed for systematic model assessment can be found in [BC05]. The architecture of the first prototype and the technologies and standards used for its implementation were described in [CGIT06]. The design of the second prototype based on the Eclipse Modeling Framework with a generic analysis tool is described in section 4. We decided to change utilised technologies to be up to date with the current projects and to take advantages from integration with the open source community. In the later phase we plan to release our tool on a public licence.

In this section, only the main ideas related to OCL application within our framework, which are necessary to understand the examples presented in section 3.2, are described.

2.1 Structure of the framework

As depicted in Figure 1 at the topmost level of our architecture three components can be distinguished: a modelling environment, a model data repository, and an analysis tool.

The main assumption in our framework is that all designed models are based on a common **meta model**. Based on the meta model, the constraints for modelling tools are provided and the structure of the common central repository of model elements is generated. **User models** can be imported into the repository from modelling tools via adapters (c.f. Figure 2). The usage of a common meta model is crucial for model integration in a heterogeneous modelling environment with diverse notations and modelling tools.

The analysis tool allows the definition of constraints and queries, and as described in the subsequent section, we plan to integrate metrics and regression tests into the analysis module. The metrics are thought to be a mechanism to evaluate models in a quantitative way. The regression tests provide a mechanism for continues quality checks. All expressions are also saved in the repository thus they can be shared among different clients. The analysis tool includes also an interpreter for constraints and queries, an interpreter for test suites (for regression tests) is planned. The interpreter can work on the user models saved in the repository. There are two manners of using the interpreter, it can be used as a stand–alone application as well as part of the modelling tools (c.f. Figure 2).

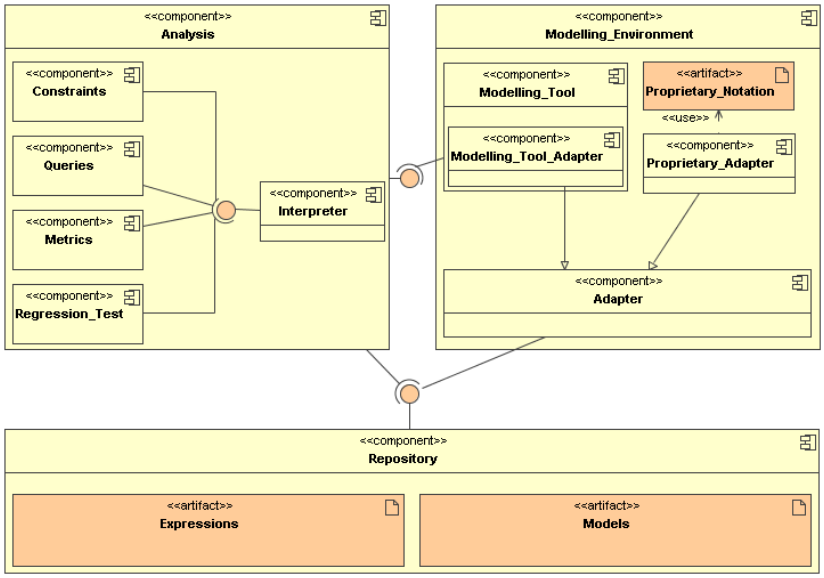


Figure 2: The architecture design of the framework

2.2 The analysis module and OCL expressions

An analysis module for our methodology should be generic enough to enable both the definition and interpretation of arbitrary queries. In this section we present the purpose and application of constraints and queries expressed in OCL. We give a detailed classification of expressions in context of our framework and at the end we describe how the expression can be initialised and evaluated. In section 3.2, we examine the expressiveness of OCL and evaluate the possibility of its usage in the analysis module.

2.2.1 Constraints and queries

Within our framework we consider two types of OCL expressions: constraints and queries, both defined at the meta model level and evaluated over user models.

Constraints extend the specification of models. The aim of using constraints is to support *model consistency* in an early modelling phase. They can be checked automatically each time model elements are saved to the repository or on demand. The expressions used for ensuring syntactical correctness are called *checks* (compare section 3.2).

Queries provide aggregated information on sets of model elements. The analysis by means of queries supports *semantic quality* of models. As stated in [KS00], semantic quality belongs to the social layer and needs to be judged by humans. Our framework supports the user in the judgement process by providing mechanisms for information retrieval. Moreover, we can only evaluate the perceived semantic quality comparing user knowledge of the considered domain with his interpretation of models [KS00] or in our case the results obtained by query evaluations. Both aspects of semantic quality examination — *validity* and *completeness* — can be supported by queries. In the first case we check if all model elements are relevant to the domain. This can be achieved by listing all instances of a given meta model element and human inspection of their relevance. In the second case we look for elements from the domain in the model data repository.

2.2.2 Detailed classification

We classify the constraints and queries in four categories (see examples in section 3.2). The dependencies between categories are depicted in Figure 3.

Primitive query is the simplest query, which takes as arguments OCL Primitive Types or MOF Classes.

Check is a special kind of primitive query which returns a Boolean value. It is considered as a constraint for a model or, in particular case, as an invariant for a classifier.

Compound query is a query which aggregates results of primitive queries. The arguments of the query are collections. For a given collection the Cartesian Product is built and for each of its element a given primitive query is evaluated. The result is of Set(TupleType) type.

Complementary query is a query evaluated over the result of a given compound query. All other queries are evaluated over a set of model elements. The query can use checks and primitive queries for result calculation.

2.2.3 Initiation and evaluation scopes

All types of queries and checks can be evaluated on demand in different scopes selected by a user. We distinguish two types of scopes, namely an evaluation and an initiation scope. The evaluation scope (Figure 4.a) determines, over what content the query will be interpreted, and the initiation

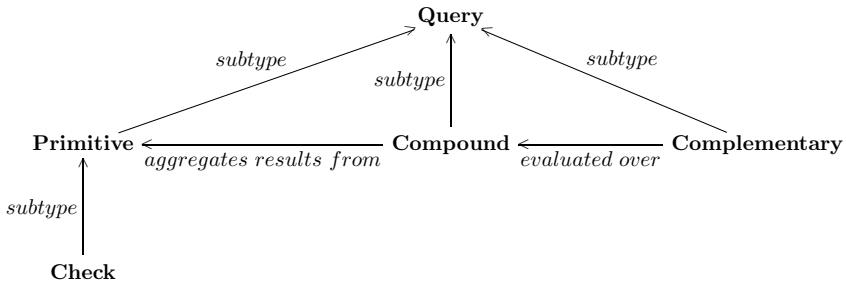


Figure 3: The queries hierarchy

Figure 4: The classification of scopes

scope (Figure 4.b), how the query is called. Furthermore in both scopes we distinguish modes. In the evaluation scope we distinguish following three modes.

Repository mode — the complete set of model elements is considered, i.e., a given expression is evaluated over the content of the repository.

Model mode — only a single user model is considered, e.g. in a running modelling tool on a local machine or model of a predefined type (filtered from the repository). This mode can be used if the queries do not need to be evaluated in the context of the complete set of elements (e.g. checks).

Diagram mode — only one diagram is considered. The usage of this mode is similar to the model mode.

The repository mode is typical for queries in the analysis phase. The advantage of the model and the diagram mode is the possibility of making fast evaluation and ongoing corrections during the modelling phase. In the initiation scope consider two different modes, both can be evaluated in any evaluation scope.

Single (element) mode — only queries related to a given element can be activated. This mode enables fine granular analysis of models.

Global mode — all queries can be activated. This mode enables global analysis of models.

3 Case study

In this section the case study from the *MedFlow* project is presented. At first (section 3.1) the excerpt of the domain in question, in form of meta and user models, is presented. Then exam-

ples of queries are presented (section 3.2) and their analysis within our framework is conducted (section 3.3).

3.1 Modelling of clinical processes

In the subsequent sections, parts of a meta model designed within the *MedFlow* project and exemplary user models are presented. The meta model is used as a base for check and query definitions, the user models as a base for check and query evaluations (section 3.2). For our study we used a tool dedicated for OCL compilation, namely the OCL Environment (OCLE, [LCI05]). In this tool, OCL expressions can be compiled and evaluated for single instances or for an entire project. The models and all queries were implemented in the OCLE version 2.0. We stress the fact that the used OCL syntax is the one implemented in OCLE. Currently it is also possible to evaluate all the queries presented below within our analysis module.

3.1.1 Meta model

The aim of the *MedFlow* project was the optimisation of clinical processes. Within the project, we developed a meta model of the clinical processes domain. Figure 5 shows a fragment of the meta model (the complete meta model can be found in [BC05]). For technical reasons our meta model was designed in OCLE as a model (at the M1 level) and user models as object models (at the M0 level). In our framework we are working at M2 and M1, respectively.

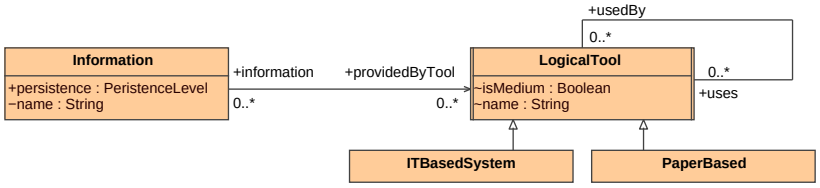


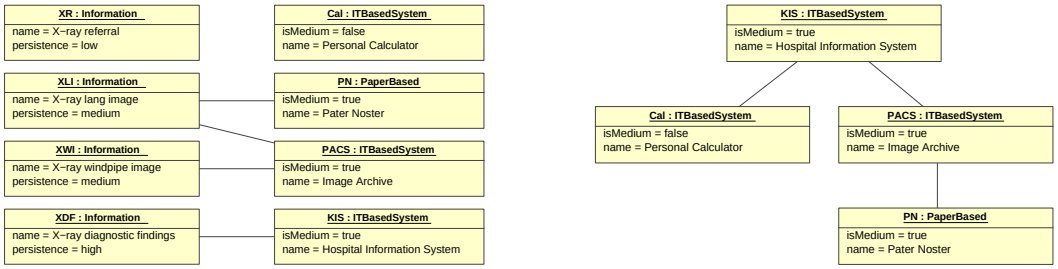
Figure 5: Part of the *MedFlow*'s meta model

In the meta model excerpt we can distinguish two main classes: Information and LogicalTool. LogicalTool is an abstract class with two subclasses: ITBasedSystem and PaperBasedSystem. Information can be saved in LogicalTool, expressed by an association providedByTool. LogicalTool can use another LogicalTool, what is expressed by the association uses. This simplified meta model is used as a base for the check and query definitions in section 3.2.

3.1.2 User models

Based on an meta model (c.f. the previous section) user models are created. In our case study, we used the simplified meta model and two exemplary user models presented in Figure 6. Models presented in the Figure are objects models, whereas within our framework they are at the M1 level (compare the explanation in the previous section).

In the first user model (Figure 6.1) four instances of Information and four instances of LogicalTool are defined. The instances of Information have diverse persistence levels (low, medium, high) and



(1) (2)

Figure 6: Exemplary user models: (1) instances of classes Information, LogicalTool and associations between them, (2) hierarchy of instances of LogicalTool

instances of LogicalTool are of diverse type (IT– and paper–based). An Information can be saved in a LogicalTool if the LogicalTool is a medium (c.f. Example 2). There are four association links between instances of Information and instances of LogicalTool. In the second user model (Figure 6.2) the hierarchical dependencies between four instances of LogicalTool are defined. These simplified user models are used as a base for the check and query evaluations in the next section.

3.2 Definition and evaluation of checks and queries

In this section, we present typical checks and queries. All definitions conform to the *MedFlow* meta model (Figure 5) and their results are evaluated over the exemplary user models (Figure 6). The examples are based on a representative selection of all types of checks and queries used within our framework for model assessment.

In the examples the checks and queries are defined in natural language and inspected manually. The corresponding listings are expressed in OCL 2.0 and automatically evaluated in OCLE version 2.0.

If not stated otherwise, definitions (**`def`**) and invariants (**`inv`**) are defined and evaluated in the context of Information (**`context Information`**) and based on the diagram depicted in Figure 6.1. This context is added for technical reasons to enable easier compilation of OCL expressions. The definitions themselves are not context dependent (no reference of self is used within them).

3.2.1 Primitive query

A primitive query can return a value of primitive type (except the Boolean type), class type or collection type. The construction of a primitive query is similar to the below defined examples for checks, thus we do not provide additional examples.

3.2.2 Check

The simplest concept for information retrieval is a check. It is a function with a set of objects as a domain. In Example 1 and Listing 1, the check is defined and evaluated. It checks if there

exists an association between a given Information and a given LogicalTool.

Example 1 *(theCheck)*
Definition*:* Is a given information saved in a given logical tool?
Evaluation *for* XLI *and* KIS*: no.*

Listing 1 (theCheck)
Definition:

```
1 def: let
    theCheck(i:Information, lt: LogicalTool)
3       = i.providedByTool->intersection(Set{lt})->notEmpty()
```

Evaluation:

```
1 def:
    let objInfo         = Information.allInstances
3                         ->select(name="X-ray lung image")
                          ->any(true)
5   let objLTool        = LogicalTool.allInstances
                          ->select(name="Hospital Information System")
7                         ->any(true)
    let theCheckResult = theCheck(objInfo, objLTool)
9 -- Selection: Boolean = false
```

Predefined check Checks can be used to express some well–formedness rules. Such checks should be defined during the meta modelling phase and are called predefined checks. In Example 2 and Listing 2 a predefined check is defined and evaluated.

Example 2 *(thePredefinedCheck)*
Definition*:* An information can be saved only in logical tools which are mediums.
Evaluation*: is fulfilled for all instances.*

Predefined checks can be expressed in the form of invariants and checked for all instances of the context class by calling the function *check UML models for errors* in the OCLE tool.

Listing 2 (thePredefinedCheck)
Definition:

```
1 inv: self.providedByTool->forAll(lt | lt.isMedium=true)
```

Evaluation *check UML models for errors*:

```
Model appears to be correct according to the selected rules.
```

3.2.3 Compound query

In order to aggregate information collected with single queries, we can build a compound query. The collections of elements, used as arguments, can be built in different manners, we can use all instances or a subset of them. Example 3 and Listing 3 depict the results of the compound query with the check defined in Example 1 and Listing 1, applied for all instances of Information and

LogicalTool. In Example 3, the result is presented in form of a table while in the Listing 3 it is presented as a set of tuples.

Example 3 *(theCompoundQuery)*

***Definition**:*

Evaluate `theCheck` for all instances of Information and LogicalTool classes.

***Evaluation**:*

Information \ Logical Tool	*KIS*	*PACS*	*PN*	*Cal*
XR	*no*	*no*	*no*	*no*
XLI	*no*	*yes*	*yes*	*no*
XWI	*no*	*yes*	*no*	*no*
XDF	*yes*	*no*	*no*	*no*

Listing 3 (theCompoundQuery)
Definition:

```
1  def: let
     theCompoundQuery(InfC:Set(Information), LToolC:Set(LogicalTool)) :
3      Set(TupleType(
         i : Information,
5        lt: LogicalTool,
         r : Boolean )) =
7          InfC->collect( info | LToolC->collect( ltool |
             Tuple {
9              i : Information = info,
               lt: LogicalTool = ltool,
11             r : Boolean     = theCheck(info, ltool)
             }))->asSet()
```

Evaluation:

```
   def:
2    let theCompoundQueryResult =
       theCompoundQuery(Information.allInstances,LogicalTool.allInstances)
4  /*
   Selection: Set(Tuple(i:Information,lt:LogicalTool,r:Boolean)) = Set{
6    Tuple{ XDF , PN , false } , Tuple{ XDF , PACS , false } ,
     Tuple{ XDF , Cal , false } , Tuple{ XDF , KIS , true } ,
8    Tuple{ XR , PN , false } , Tuple{ XR , PACS , false } ,
     Tuple{ XR , Cal , false } , Tuple{ XR , KIS , false } ,
10   Tuple{ XWI , PN , false } , Tuple{ XWI , PACS , true } ,
     Tuple{ XWI , Cal , false } , Tuple{ XWI , KIS , false } ,
12   Tuple{ XLI , PN , true } , Tuple{ XLI , PACS , true } ,
     Tuple{ XLI , Cal , false } , Tuple{ XLI , KIS , false }
14 } */
```

Filtering We can additionally apply filters before or after evaluating the result of a given compound query. The filtered compound query presented in Example 4 and Listing 4 is evaluated only for instances of Information and LogicalTool classes, which fulfil additional constraints.

Example 4 *(theFilteredCompoundQuery)*

***Definition**:*

Evaluate `theCheck` for instances of Information, which have the persistence attribute set to medium or high and instances of LogicalTool, which have the attribute isMedium equal to true.

Evaluation:

Information \ Logical Tool	*KIS*	*PACS*	*PN*
XLI	*no*	*yes*	*yes*
XWI	*no*	*yes*	*no*
XDF	*yes*	*no*	*no*

The definition of `theFilteredCompoundQuery` presented in Listing 4 uses the result `theCompoundQuery` from Listing 3. Like in the previous section, the result (`theFiltered-CompoundQueryResult`) is presented as a set of tuples.

Listing 4 (theFilteredCompoundQuery)
Definition:

```
def: let
  theFilteredCompoundQuery() = theCompoundQueryResult->select(t |
          (t.i.persistence=#medium or t.i.persistence=#high)
          and t.lt.isMedium = true )
```

Evaluation:

```
def:
  let theFilteredCompoundQueryResult = theFilteredCompoundQuery()
/*
Selection: Set(Tuple(i:Information,lt:LogicalTool,r:Boolean))= Set{
  Tuple{ XDF , PN , false } , Tuple{ XDF , KIS , true } ,
  Tuple{ XDF , PACS , false } , Tuple{ XWI , PN , false } ,
  Tuple{ XWI , KIS , false } , Tuple{ XWI , PACS , true } ,
  Tuple{ XLI , PN , true } , Tuple{ XLI , KIS , false } ,
  Tuple{ XLI , PACS , true } } */
```

Collecting Elements can be collected according to specific properties (e.g. values of slots, existing links). In the example below we collect elements according to the element hierarchy (c.f. Figure 6.2). We do not construct a complete definition of a compound query, we only demonstrate how to create a collection using a recursive OCL function.

Example 5 *(theCollection)*
Definition:
Collect all LogicalTools used by a given LogicalTool.
Evaluation *for* KIS*:* {PN, Cal, PACS}

Listing 5 (theCollection)
Definition:

```
1 context LogicalTool
  def: let
3   getUsedTools(t: LogicalTool) : Set(LogicalTool)
        = t.uses->collect(x|getUsedTools(x))->asSet()->union(t.uses)
```

Evaluation:

```
def:
2 let objLTool = LogicalTool.allInstances
                  ->select(name="Hospital Information System")
4                 ->any(true)
  let LToolC    = getUsedTools(objLTool)
6 -- Selection: Set(LogicalTool) = Set{ PN , Cal , PACS }
```

3.2.4 Complementary query

After the evaluation of a compound query, complementary queries can be evaluated over the obtained result. In Example 6, a complementary query is defined and evaluated.

Example 6 *(theComplementaryQuery)*
Definition:
Which instances of LogicalTool are used to save Information objects with persistence level medium.
Evaluation:
$\{PACS, PN\}$

The OCL expression presented below depicts one of the possible ways to express this complementary query. The condition in line 4 corresponds to the filtering condition and the remaining conditions correspond to the iteration over the result of the compound query.

Listing 6 (theComplementaryQuery)
Definition:

```
def: let
2 theComplementaryQuery: Collection( LogicalTool ) =
    LogicalTool.allInstances()
4     ->select( ltool | theCompoundQueryResult
      ->select( t | (t.i.persistence = #medium) and
6                  (t.lt = ltool and t.r = true))->notEmpty()  )
```

Evaluation:

```
def:
2 let theComplementaryQueryResult = theComplementaryQuery
-- Selection: Collection(LogicalTool)= Set{ PACS , PN }
```

One can notice that the usage of compound queries does not simplify OCL expressions for complementary queries. The complementary query defined in Example 6 can be expressed based on the result of the previously defined compound query (`theCompoundQueryResult`) as in Listing 6 or without any definition as in Listing 7. The results in both listings, 6 and 7, are equal. The expression in Listing 7 seems to be easier and does not depend on any other definitions.

Listing 7 (theComplementaryQueryBis)
Definition:

```
1 LogicalTool.allInstances
    ->collect(ltool | Information.allInstances
3   ->select(i | i.persistence=#medium).providedByTool)->asSet()
```

At this point, the question why compound queries are useful for complementary queries may arise. Let us explain our motivation for the usage of the first variant. In our prototype for the *MedFlow* project we have a common repository for all models. To evaluate a compound query we have to gather information from the repository, which can be located on a remote server. If we define a complementary query based on the result of the compound query, then the evaluation is faster, otherwise for the evaluation of a complementary query we again need to gather information from the repository. Moreover, we can evaluate more complementary queries over the same compound query without further connection to the repository. The second reason for using the variant with compound queries is the modified presentation of the results of complementary queries. With some additional effort the result can be presented as a set of elements in form of highlighted elements in the result of a compound query (c.f. Example 7).

Example 7 (theComplementaryQuery)
Evaluation: $\{PACS, PN\}$

Persistence \ Logical Tool	*KIS*	***PACS***	***PN***	*Cal*
low	*0*	*0*	*0*	*0*
medium	*0*	***2***	***1***	*0*
high	*1*	*0*	*0*	*0*

3.3 Summary

We showed how to construct all types of checks and queries used in our framework. The OCL 2.0 is expressive enough to be applied in our framework for model assessment.

The models created in our framework are MOF compliant and as the OCL supports the object oriented paradigm, it is easy to navigate through the object structures and create checks (Example 1) and queries. The invariants can be used as consistency checks before saving models to the repository (Example 2). Tuples provide useful mechanisms for the aggregation of information of different types. Using tuples it is possible to evaluate the Cartesian Product of given sets, what was used within our compound queries concept (Example 3). Using the select operation it is possible to filter collections. The select operation can be applied either to the result of a compound query (Example 4) or to a domain of it (for each argument separately). The first manner enables the expression of more complex conditions. OCL does not have a built–in operator for transitive closure, but it allows definitions of recursive functions. In Listing 5 used tools are recursively collected in order to represent the transitive closure of the relation defined by uses. Complementary queries can be expressed in OCL in two different manners. The first is based on a previously defined compound query and the second is a definition from scratch. The first one seems to provide an easier manner to automate query definition and results presentation.

4 Technical aspects

In the *SQUAM* project we continue development of the system for quality assessment of models started in the *MedFlow* project [CGIT06]. In this section we present redesigned architecture of our system which utilises the newest components developed within the Eclipse Modeling Framework (EMF[1]). The architecture presented below integrates three components of Eclipse Modeling Framework Technology (EMFT[2]), namely Connected Data Objects (CDO[3]), Object Constraint Language (OCL[4]) and Query (QUERY[5]), to create a system with a central model repository and a generic analysis tool. The architecture of the repository and the management of checks and queries are described in subsequent sections.

4.1 Architecture

As mentioned above the design of the model data repository is based on the EMF and some of the EMFT projects. *EMF is a modelling framework and code generation facility for building tools and other applications based on a structured data model* [Ecl06]. The model data repository uses EMF as the meta model, it can save model instances of different EMF meta models (c.f. Figure 7).

The architecture of the model data repository is based on the client–server paradigm. The repository clients can connect to a relational database management system via CDO, or they connect to a version control system like subversion (SVN[6]). The connection via CDO provides multi user support. The connected clients can search, load, save or create new EMF model instances of an arbitrary EMF meta model. Moreover CDO provides a notification mechanism to keep connected clients up to date on model changes. SVN provides versioning, change histories, merging and also locking mechanisms. The clients can save the EMF model instances to SVN which makes multi user support implicit available because of the merging and locking functionalities.

The repository client integrates the EMFT projects, OCL and QUERY, to specify and execute queries on EMF model elements. OCL component provides an Application Programming Interface (API) for OCL expression syntax which can be used to implement OCL queries and constraints. The QUERY component facilitates the process of search, retrieval and update of model elements; it provides an SQL like syntax.

The *SQUAM* tool family is based on the above described core functionalities out of the EMF and EMFT projects. The repository client API (CDO, EMF, OCL and QUERY) provides an access mechanism for other tools, mostly modelling tools. The tree–based editors can be generated out of EMF meta model definitions. The native editors are especially useful for the prototyping phase, later on we plan to integrate some graphical editors to create model instances. In the

1 http://www.eclipse.org/emf/
2 http://www.eclipse.org/emft/
3 http://www.eclipse.org/emft/projects/cdo/
4 http://www.eclipse.org/emft/projects/ocl/
5 http://www.eclipse.org/emft/projects/query/
6 http://subversion.tigris.org/

MedFlow prototype we integrated the MS Visio[7] and MagicDraw[8] modelling tools. We plan to integrate these two modelling tools as well as editors developed within the Graphical Modeling Framework (GMF[9]) with the *SQUAM* tool family.

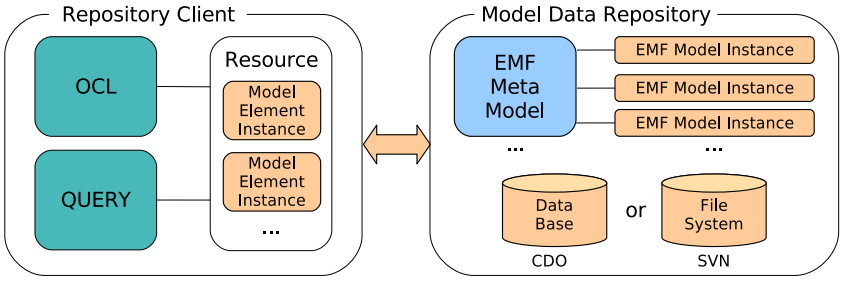


Figure 7: The model data repository architecture design

For analysis purposes we use the repository client which uses the OCL component to make queries on the model instances. The management of checks and queries is described in the subsequent section.

4.2 Checks and queries management

The OCL component provides mechanisms for check and query definitions and evaluations. In our framework it should be possible to evaluate checks and queries on demand, thus we need an OCL management system to store OCL expressions. For this purpose we implement a checks and queries catalogue. The catalogue enables users to evaluate OCL expressions in different modes (c.f. Figure 4 in section 2).

The meta model of the OCL management system is also modelled in EMF, therefore the OCL expressions can also be saved in the model data repository in the same manner as other model instances.

Figure 8 illustrates the simplified meta model for the OCL management system. The model data repository supports the storage of several meta models. To differentiate between queries specific to a given meta model we assign OCL expressions to a specific Bundle. The Bundle defines the type of the model instances by specifying the meta model they have to conform to.

Further we consider queries, the Query element contains one OCLExpression. We distinguish between a definition (Definition) and an evaluation (Evaluation) of queries. Within one Definition the prior definitions can be used, e.g. a compound query can use a primitive query (compare section 3.2). An OCL expression in the Evaluation also uses definitions. The Definition is split into the Check, Primitive, Compound and Complementary expressions. The Definition elements are elements which can be used as subroutines in other expressions and the Evaluation elements are evaluated over an explicit data model. Each Evaluation element is placed in a particular OCLContext. The context of the OCL expression enables the usage of the `self` element. The context can

[7] http://office.microsoft.com/visio/
[8] http://www.magicdraw.com/
[9] GMF is a combination of the EMF and GEF (Graphical Editing Framework) projects, http://www.eclipse.org/gmf/

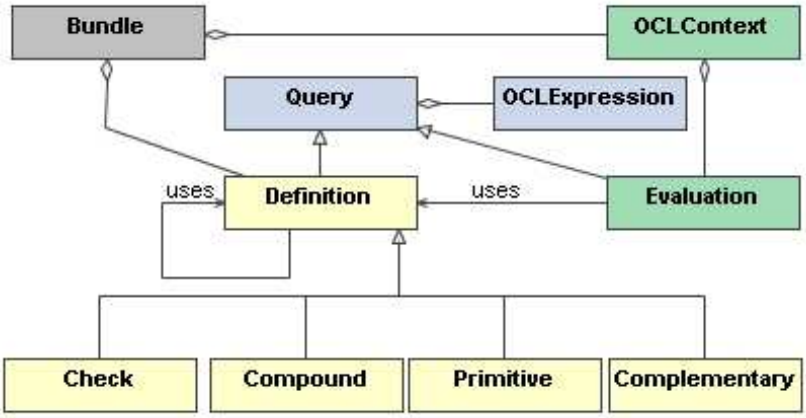


Figure 8: The meta model of the OCL management system

also be `NULL`, it is useful for expressions without any particular contexts. In the example listings presented in section 3.2, for all listings except Listing 2 and Listing 5, `NULL` context can be used (these listings do not use the `self` keyword and the definitions are not related to a particular classifier). For interpretation of a Evaluation element the OCLContext has to be set to an explicit instance of a model element.

The presented design is a proof of concept for the model data repository. Used technologies and design allow easy extensions with additional features such as dynamic load of new meta models, or an extended editor for the OCL management system with OCL syntax check and compilation at design time.

5 Conclusion

Our examination shows that the OCL is expressive enough to be applied as a query language for model analysis. It is possible to define all types of checks and queries required by our model assessment framework (section 3.2). There are two other reasons for OCL usage within our framework. Firstly, there are more and more tools supporting the OCL notation, also non–commercial tools (e.g. OCL project within EMFT described in section 4 or tools presented in [BCC+05]). The second reason follows from the first: the knowledge of the notation is getting broader among scientists and pragmatic modellers.

We presented a proof of concept for the model data repository created within EMF and EMFT technologies. In the presented architecture OCL queries for assessment of models can be saved in the repository (section 4.2) and evaluated on demand. Currently we are developing full support for the OCL management system (section 4.2). We plan to carry out more case studies to determine more requirements for model assessments queries and define patterns for query definitions.

Acknowledgement

We would like thank *Dan Chiorean* for the presentation of the OCLE tool at our University and the later helpful tips for OCL expression implementation in the OCLE. We would like to thank all of the people who reviewed our paper and gave us constructive input, especially *Frank Innerhofer–Oberperfler* and *reviewers*. And at last but not least *Ruth Breu*, who supported us in our work.

Bibliography

[AB01] D. H. Akehurst, B. Bordbar. On Querying UML Data Models with OCL. In Gogolla and Kobryn (eds.), *UML*. Lecture Notes in Computer Science 2185, pp. 91–103. Springer, 2001.
http://link.springer.de/link/service/series/0558/bibs/2185/21850091.htm

[BC05] R. Breu, J. Chimiak-Opoka. Towards Systematic Model Assessment. In Akoka and al. (eds.), *Perspectives in Conceptual Modeling: ER 2005 Workshops CAOIS, BP-UML, CoMoGIS, eCOMO, and QoIS, Klagenfurt, Austria, October 24-28*. Lecture Notes in Computer Science 3770, pp. 398–409. Springer-Verlag, October 2005.
doi:http://dx.doi.org/10.1007/11568346_43

[BCC+05] T. Baar, D. Chiorean, A. L. Correa, M. Gogolla, H. Hußmann, O. Patrascoiu, P. H. Schmitt, J. Warmer. Tool Support for OCL and Related Formalisms - Needs and Trends. In Bruel (ed.), *MoDELS Satellite Events*. Lecture Notes in Computer Science 3844, pp. 1–9. Springer, 2005.
http://lgl.epfl.ch/members/baar/oclwsAtModels05/reportOCLWSAtModels05.pdf

[CGIT06] J. Chimiak-Opoka, G. Giesinger, F. Innerhofer-Oberperfler, B. Tilg. Tool–Supported Systematic Model Assessment. In Mayr and Breu (eds.). Lecture Notes in Informatics (LNI)—Proceedings P–82, pp. 183–192. Gesellschaft fuer Informatik, 2006.

[Cod72] E. F. Codd. Relational Completeness of Data Base Sub-Languages. *Data Base Systems, Rustin(ed), Prentice-Hall publishers*, 1972.

[Ecl06] Eclipse Foundation Inc. Eclipse Modeling Framework homepage. 2006.
http://www.eclipse.org/emf/

[KS00] J. Krogstie, A. Solvberg. Quality of conceptual models. In *Information systems engineering: Conceptual modeling in a quality perspective*. Chapter 3, pp. 91–120. Kompendiumforlaget, Trondheim, Norway, 2000.
http://www.idi.ntnu.no/~krogstie/publications/2003/quality-book/b3-quality.pdf

[LCI05] LCI team. Object Constraint Language Environment. 2005. Computer Science Research Laboratory, ”BABES–BOLYAI” University, Romania.

[MC99] L. Mandel, M. V. Cengarle. On the Expressive Power of OCL. In Wing et al. (eds.), *World Congress on Formal Methods*. Lecture Notes in Computer Science 1708, pp. 854–874. Springer, 1999.
http://link.springer.de/link/service/series/0558/bibs/1708/17080854.htm

[OMG05] OMG. Object Constraint Language Specification, version 2.0. May 2005.
http://www.omg.org/cgi-bin/doc?formal/2006-05-01

[WK99] J. Warmer, A. G. Kleppe. *The Object Constraint Language—Precise Modeling with UML*. first edition, 1999.