An OCL extension for checking and transforming UML models

Thierry MILLAN¹, Laurent SABATIER², Thanh-Thanh LE THI¹, Pierre BAZEX¹, Christian PERCEBOIS¹

(1) Institut de Recherche en Informatique de Toulouse - Université Paul Sabatier
118, route de Narbonne
31062 Toulouse Cedex 9 - FRANCE

(2) Sodifrance
15, Chemin de la Crabe
31300 Toulouse - FRANCE

millan@irit.fr, LSabatier@sodifrance.fr, lethi@irit.fr, bazex@irit.fr, percebois@irit.fr
http://neptune.irit.fr

Abstract: The increasing use of models in specification and design phases makes them first class citizens. Models which had, till now, been used to gain a better grasp of the software to be designed, have been attributed a productive role that has become central to development. This new status of models and meta-models poses the problem of how they are established and especially, how they are validated. So, each activity in a process can be supported by MDE (Model-Driven Engineering) components realizing transformations and verifications. These operations require redefining the OCL type system for taking into account multi-model handling and side-effects. Checking the transformation consists of ensuring that the source models and the targets respect a set of properties that can be verified through transformation contracts. The NEPTUNE platform was designed having this logic as a driving force – allowing the verification of models and of their transformation.

Key-Words: OCL, process development, multi-model access, type synthesizer, validation, transformation

1 OCL and the NEPTUNE platform

The checking and transformation of models are justified in numerous applications. For instance, they help in the thought processes during the specification and design phases, they help to adapt the model to its projection onto different target platforms under the supervision of the designers/developers and even facilitate studies performed under collaboration. During a fusion operation, for instance, it is necessary to be able to handle at least two models at the input and two at the output. With reference to the “View” component in the QVT standard [1] (Query / View / Transformation), the creation of views is also covered by our study.

The NEPTUNE platform fits into this logic involving the checking and transformation of models. It uses an extension of OCL language [2] (Object Constraint Language), which enables the OMG (Object Management Group) standard to be strictly respected during verifications but less so for transformations.

We first present some related works and the OCL extensions to illustrate how they are used to transform models. Then, we show how the extension requires the coherence of navigation expression typing to be checked by means of several meta-models and how it allows transformation contracts to be expressed. We finish with an overview of the first experiments and the resulting future of the platform.

2 Related works on OCL

Several OCL tools exist which are compared in [3]. Some of them are integrated with other modelling tools, while others are stand alone tools. In the following we discuss about ATL, USE and KerMeta. At the end of the section we analyse imperativeOCL proposed by QVT with respect to OCL.

The ATL language [4] is developed by the ATLAS Group (INRIA & LINA). It is a declarative language of transformation. An ATL transformation is based on rules “rules”. A rule defines a transformation from a source model to a target model. It is possible to define specific methods called “helper” to make easier the treatment of the transformation rules. The USE platform [5] is developed in the Bremen University. It is an environment dedicated to OCL expressions evaluation. The end-user can dynamically generate object diagrams from UML class diagrams containing OCL invariants and operations written using an action language specified using OCL pre- and post-conditions.

At any time, the user can verify properties and executes the defined operations insuring that pre- and post-conditions are verified. Such a runtime environment is used, in particular, for metamodelling programming languages. However, the KerMeta platform [6], more
The KerMeta language is developed by the Triskell team from IRISA. This language is created as a metamodelling kernel. It is based on MOF and OCL. It is considered as an executable metamodelling language. KerMeta is an Eclipse plug-in and, so, used the Ecore metamodelling language. QVT [1] introduces ImperativeOCL as an imperative language to define transformations using an operational language for operational mappings. ImperativeOCL extends the OCL initial definition with facilities to manipulate system states through variables and includes constructs from imperative programming languages such as loops or conditionals. The ImperativeOCL abstract syntax allows mixtures of imperative expressions and OCL functional expressions because all imperative expression classes inherit from OCLExpression at the top of the abstract syntax model. As an ImperativeOCL expression can occur anywhere an OCL expression is needed, ImperativeOCL must produce for its interpretation an environment dealing with variable bindings and not only a value as in conventional OCL. This approach does not leave OCL conformant to its original definition [7]. On the contrary, we claim reusing OCL in a non-intrusive way. This means that side-effects have to be formulated within an OCL expression and that only some reduced transformation primitives have to be defined. In addition, the evaluation of all new authorized constructions must return a value in order to be easily integrated with the OCL semantics.

### 3 OCL extensions

Our approach aimed to extend the use of OCL language towards the simultaneous manipulation of several models and towards the transformation of the models. The choice of OCL mainly results from its standard inclusion in UML (Unified Modelling Language) and from the ease with which it navigates within the meta-models that generate the models. We can thus imagine using extended OCL to design operations such as the fusion or the comparison of modelling elements, justifying the existence of contracts after a transformation stage. The extended OCL language, which we will call pOCL (procedural OCL), has two parts: one for the simultaneous access and manipulation of several models, each conforming to its meta-models and the other for the introduction of transformation primitives.

#### 3.1 Multi-model access and manipulation

The first extension proposed by pOCL consists of explicitly naming – using pOCL expressions – the models and meta-models that will be simultaneously manipulated by the rules. The aim of this extension is to reference the model and the meta-model that it depends on. A form of syntax close to the association classes was used in the pOCL scripts. The syntax associates a model to its meta-model via a name of type “name of meta-model, name of model”. The following text shows that the models $m, m_1, m_2, ..., m_n$ respect the meta-model named $mm$.

```
mm [m]  
mm [m_1, m_2, ..., m_n]
```

We must also be able to declare the set of models that will be taken into account during the evaluation of a pOCL rule. The access paths to a given meta-class are thus prefixed by the names of the corresponding models and meta-models, as suggested above. It then becomes possible to declare the access path to a meta-class $mc$ by using the prefixes $ch_1::ch_2::...::ch_m$.

```
mm [m]::ch_1::ch_2::...::ch_m
mm [m_1, m_2, ..., m_n]::ch_1::ch_2::...::ch_m::mc
```

Referencing a model using OCL implies the availability of a variable to name it and a type to characterise it. The variable designates a path in a model. In the rule below, the variable $aux$ references the model $m'$ which respect meta-model UML and OCLModel represents the type of the variable $aux$.

```
context UML[m]::Foundation::Core::Class
def : aux : OclModel = UML[m']
```

In addition to syntax, the use of pOCL contributed to the development of new evaluation functions and to the creation of the new type OclModel allowing a type hierarchy to be defined. It is the concept of the OclModel that enables the interpreter to navigate in the current model during the evaluation of an expression.

#### 3.2 Type synthesizer

The implementation of inter-model transformations and checks is only possible using a type synthesizer, which enables the different expressions manipulated and to be typed and checked. The structure of pOCL types presented in Figure 1 is organised according to three type lattices plus one common bottom type: OclVoid. The three lattices are OCL atomic type lattice (primitive types of OCL and the meta-classes of the meta-models transformed); the lattice of types associated to homogeneous collections (set, bag, sequence for instance); and the lattice of heterogeneous collections (tuples). The type OclModel, mentioned above to type a model, occurs in the first lattice.
Type structuring makes it possible to dynamically check that the type of the data manipulated and generated respects the meta-model that they originated from. If two sets of data have a type that belongs to two different lattices, they are not compatible and this leads to an error of typing.

Dynamic typing of expressions enables the type of the result to be calculated in real time during the manipulation process. This dynamic approach means the OCL programmer no longer has to “force” the type using the OCL primitive `oclAsType`, thus simplifying the way rules are written without weakening the control of the expressions. The simultaneous use of several meta-models is perfectly integrated into the dynamic process as all meta-classes belong to the first lattice. The result is two classes originating from two separate meta-models that have at least one common OCL meta-class `oclAny`, and one OCL meta-class that specialises them `oclVoid`. These two classes are essential for the type synthesiser as they define type `top` and type `bottom` of the lattice respectively, bottom being the reference type when calculating the type of the elements in a collection.

The type system defined in this way is homogeneous and is not disturbed by the simultaneous manipulation of several meta-models. The result is therefore a simple integration of the multi-meta-model mode into the type system.

### 3.3 Primitives allowing transformations

OCL allows structural constraints to be expressed and to define semantics that UML graphic notation is incapable of expressing. We can mention, for instance WFR (well formed rules) that are a part of all meta-models. So, the extension presented includes notions of constraint, request and transformation. The evaluation of a constraint, introduced by the basic OCL keyword `inv`, leads to a logical result. A pOCL query introduced using the keyword `query` gives a result that is more general i.e. not restricted to a boolean. In contrast the pOCL expression introduced by the keyword `transform` enables modifications to be programmed. The three following texts illustrate constraints navigation and modification:

```plaintext
context UML[m]::Attribute
inv : self.type.oclIsKindOf (String)

context UML[m]::Attribute
query : self

context UML[m]::Class
transform : Sequence
  {table <- new DB[DBmodel]::Table,
   table.name <- self.name}
```

The second pOCL rule is an example of navigation in the meta-class `Attribute` of the UML meta-model of in one of its descendents. The last rule shows an example of a transformation sequence such as can be found in imperative language. This transformation sequence makes pOCL a hybrid language that combines standard OCL functional aspects and the transformation of examples of meta-class. The inference type system ensures that expressions are correctly typed. Compared with other tools, pOCL checks the type correctness of an expression.

Finally two other syntactic constructions have been added to pOCL to create an example of a meta-class (new operator) and for updating an example (-> transformation operator). Similar to the `OclModel` type, we introduce a type called `Rule`. This type allows handling transformation rules. When the pOCL interpreter detects a transformation rule (->), a variable of “Rule” type is instantiated. This type is a structure containing two attributes of type `OclVoid` corresponding to the left and right parts of the rule.

### 4 pOCL in a multi-model context

We now have available a unified pOCL environment which combines navigation expressions recognised to be standard OCL and model transformation primitives [8], [9]. Without questioning the standard language defined by the OMG, we can now simultaneously link several models [10] and propose designers the possibility to specify transformation contracts. In the context of an MDE development process, it would be necessary to establish to what extent this new interpreter would be able to guarantee that contracts are respected during a series of transformations.

Figure 2 shows a first example of a series of treatments that include check before and after transformations. In this scheme, the verification activity is performed by the OCL interpreter, initially designed for the European project NEPTUNE¹. The transformation activity suggests the generation of a target model from a source model.

---

¹ [http://neptune.irit.fr](http://neptune.irit.fr)
Fig. 2 : Use of OCL and pOCL in a multi-model context

The first set of rules checks that the source model satisfies a certain number of structural properties, in particular, that the model is well formed and that, for instance, it respects the requirements for the application of a project defined by its analysts-designers. This first set of rules can be considered as a prerequisite for transformation. The second set of pOCL rules contains the description of the transformation. These rules include operators allowing limit conditions to be tested on the models. The third set of OCL rules check the properties on the model that has been generated. Finally, the 4th pOCL set of rules establish the inter-model correspondences that the model must guarantee such as the preservation of uniqueness of a meta-element for instance. Sets of rules 3 and 4 can be seen as post-conditions of transformation. The sequence of events thus established enables the conformity of the phases and the activity of a process to be checked considering, for instance, that an activity is considered solely as the controlled and assisted transformation of source models into target models.

5 pOCL examples

In this section, we illustrate the pOCL concepts in two punctual examples applied to UML diagrams transformations into relational scheme and Petri Nets.

5.1 From a class diagram to a relational scheme

This first example detailed below again considers the classic transformation of a data structure – the class diagram – into a relational scheme. Sequence (2) of the following pOCL script initially transforms all metaclasses Class in the meta-model Simple UML into the meta-class Table for the meta-model DB; the attribution name of the meta-class Table is identical to that of the meta-class Class. Then, a meta-class Column is associated to the table previously created. Column represents the primary key. This transformation only becomes possible if the attributes of a class are either strings of characters, or integers (\(c\)). Respecting the types of the attributes of the source model, those of the target model must be the data types of relational model (\(e\)). Rule (4) guarantees that a Varchar2 type is always associated to a String type. This inter-model verification implies navigation within the UML model starting from an attribute of the DB model. The same would be true for the type Number with respect to the type Integer.

5.2 From an activity diagram to a Petri network

This example illustrated a transformation based on a recursive algorithm. The transformation considered at the input of a model respecting a simplified version of the meta-model of the activity diagrams described in Fig. 3 and provides, at the output, a model respecting a meta-model of the Petri networks described in Fig.4.
The fragment of code presented below initiates the transformation from the activity of the START type (①). A place (②) and a transition (③) created and linked by an edge of the ArcPT type. Then, the recursive call createNextPlace (④) goes through the source graph source of the activity diagram and creates edges ArcPT/Transition, as well as the connections Transition/ArcTP/Place of the following nodes.

```oc
model new petri[PetriModel]
context MMactivity[ActivityModel]::Activity
transform:
  if self.kind = ActivityKind::START then ①
  Sequence{
    place <- new petri[PetriModel]::Place,
    place.name <- 'Start',
    place.token <- '',
    aPT <- new petri[PetriModel]::ArcPT, ②
aPT.arcPTinscr <- '',
    transition <- new petri[PetriModel]::Transition, ③
transition.name <- self.name,
    transition.arcPTsource <- aPT,
aPT.arcPTtarget <- transition },
  self.begin->asSequence->
  first.end.createNextPlace( ④
transition,
  Bag{
    Tuple{
      a:Activity=self,
      p:Place=place}})->flatten
  else Sequence {false}
endif
```

One of the contracts related to this transformation stipulates that to any activity must correspond one and just one place in the Petri network.

### 6 pOCL experiments in MDE processes

The industrial feedbacks show that without a global view of the different development process activities, the code generation many time requires manual readjusting directly on the generated code. These readjusting are mainly due to the fact that designers do not completely realize the last activities of the process. These readjusting make incoherent models [11] and components. We present in this section two industrial applications using OCL/pOCL. In the first one, languages are introduced in the processes as UML models including syntactic and semantics properties. The second one consists in indentifying design patterns during the software design.

#### 6.1 Modelling of languages in UML/OCL

Modelling of languages in UML/OCL [12] is part of the DOMINO² project financially supported by the French Government. In order to provide a better visibility of the models evolution and of the realized software during the use of a software process, the idea is to build the UML meta-model of the target language used to generate the software components. In this context, pOCL is used for checking and transforming models during the activities constituting the process.

The use of meta-models of the languages inserted during terminal phases of a process has been applied for two case studies: the first one deals with designing mission control systems for space applications. The second one deals with embedded software components in aeronautic systems where certification requirements are very relevant.

Such a meta-model, directly produced from the abstract syntax of the language and completed both with OCL properties for defining the static language semantics and with operations written using an action language, can be an intermediate step between models and components code [13].

The analysts and designers can extend the design of the software until to component abstract models, and anticipate the technical solutions from models elaboration, taking into account the criticality level of the software requirements.

#### 6.2 Automatic detection of design patterns

To give a consistent and more valuable feature on models, we propose that model-driven processes should be able to reuse the expert knowledge generally expressed in terms of patterns [14]. We also used pOCL in this context in order to automatic detect model fragments substitutable by structural design patterns. Among the pOCL proposed extensions, model query is the most pertinent for this study [15] because it allows extracting information from a model.

Our query is a set of OCL rules automatically generated which finds occurrences of an “alternative solution” with respect to the one using the design pattern. In this context, an alternative solution is a valid solution which

---

² http://www.domino-rtf.org/
solves the same problem as the pattern, with a more complex or different structure than the pattern. Each time one applies the OCL detection query of one alternative solution associated to a given pattern, one retrieves all substitutable model fragments by the pattern itself. Therefore, this method is determinist and the result is complete with the set of alternative solutions dedicated to the pattern. These alternative solutions have been extracted from an experimentation which consists in designing standard problems of the GoF catalog [14] solvable with the seven structural patterns, in UML notation.

When detection is successful, we propose to the designer to substitute its alternative solution with the corresponding pattern. This integration can be done thanks to an automatic model refactoring written in pOCL.

7 Conclusion and perspectives

The NEPTUNE is operational and is being used in projects associating industries and universities such as the project DOMINO² which focuses on model coherence, traceability of modelling artefacts, and the verification of transformation components. It is also used in research projects for the detection of model fragments that can be substituted by patterns.

Other transformation languages working with the QVT standard, such as ATL³, KerMeta⁴ and SmartQVT⁵, integrate the OCL language in order to execute requests on the meta-models. However, in contrast to these languages, pOCL proposes an extension of the OCL language to operate these transformations and does not define, in the real sense of the word, a new language. pOCL is rather like an action language such as that of the USE⁶ platform, which is also based on OCL and allows transformation sequences to be carried out.

Owing to its modular structure, NEPTUNE is regularly updated with new functions all contributing to increase the quality of the models produced. We intend to expand NEPTUNE by the use of metrology tools for models based on pOCL rules and simulation tools allowing the static and dynamic coherence of a system to be checked.

References:

³ http://www.sciences.univ-nantes.fr/lina/atl/
⁴ http://www.kermeta.org/
⁵ http://www.casetool.eu/smartqvt_en.html
⁶ http://www.db.informatik.uni-bremen.de/projects/USE/