Towards Functional Model Transformations with OCL

Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickaël Clavreul, Guillaume Savaton

To cite this version:

Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickaël Clavreul, Guillaume Savaton. Towards Functional Model Transformations with OCL. ICMT 2015 - 8th International Conference on Model Transformation, Jul 2015, L'Aquila, Italy. pp.111-120, 10.1007/978-3-319-21155-8_9 . hal-01179294

HAL Id: hal-01179294

https://hal.archives-ouvertes.fr/hal-01179294

Submitted on 27 Jan 2018

Towards Functional Model Transformations with OCL

Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickael Clavreul, and Guillaume Savaton

ESEO, Angers, France

Abstract. Several model transformation approaches such as QVT and ATL use OCL as expression language for its model-querying capabilities. However, they need to add specific and incompatible syntactic constructs for pattern matching as well as model element creation and mutation.

In this paper, we present an exploratory approach to enable the expression of whole model transformations in OCL. This approach lever-ages some OCL extensions proposed for inclusion in the upcoming OCL 2.5: pattern matching and shadow objects. It also relies on a specific execution layer to enable traceability and side effects on models.

With model transformations as OCL functions, it becomes possi-ble to use a single, standard, well-known, functional, and formalized model querying language to perform tasks traditionally assigned to model transformation languages. Thus, functional techniques such as func-tion composition and higher-order become directly applicable to model transformations.

Keywords: Model transformation · OCL · Functional transformation

1 Introduction

The Object Constraint Language [6] (OCL) progressively evolved from a language focused on the expression of constraints (invariants, pre- and post- conditions) on UML models to a more general metamodel-independent language for model query and navigation. Some model transformation approaches (such as QVT and ATL) started making use of these capabilities by integrating (or hosting) OCL as an expression languages. These *host* languages typically leverage OCL to express *guards* (i.e., predicates selecting elements that match transformation rules) and for navigation (i.e., path expressions over models).

Because OCL is a purely functional language, it cannot be directly used to perform changes on models or their elements. Therefore, host languages must define specific syntax and semantics around OCL for these purposes. However, recent OCL extension proposals [3, 5] considered for inclusion in the next version of OCL [12] give even more capabilities to OCL. For instance, structural *pattern* *matching* enables declarative data analysis, and *shadow objects* enable creation and processing of immutable versions of model elements. Making use of shadow objects does not require performing any side effect such as creating elements in models. This constraint is mandatory to keep OCL purely functional.

In this paper, we explore the possibility of directly using OCL as a transformation language. For this purpose, we define our own variant of OCL called OCLT (where the *T* stands for transformation). OCLT is based on OCL 2.4 [6] and integrates pattern matching and shadow objects extensions in a way that is similar to the work presented in [5], but with syntax closer to the one used in [3]. These custom extensions are likely to become unnecessary when they actually become standard by being integrated in OCL 2.5. In the mean time, OCLT lets us start investigating their capabilities. In addition to these extensions, OCLT also needs some means to actually create elements in models. To this end, we additionally integrate to OCLT a specific layer that can translate shadow objects to actual model elements. This layer is also responsible for *trace link* resolution, which consists in linking elements created separately by using traceability links between source and target elements.

Model transformations expressed in OCLT are pure functions taking as arguments a collection of source model elements, and returning a collection of target elements. Transformation composition thus becomes function composition. Other functional techniques such as partial application and higher-order functions also become applicable to model transformation. We illustrate our approach on the well-known *ClassDiagram2Relational* model transformation case-study.

The paper is organized as follows. Section 2 gives an overview of the shadow objects and pattern matching OCL extensions. Section 3 presents the specific execution layer of OCLT, and shows how our approach can be applied to the well-known *ClassDiagram2Relational* transformation. Section 4 discusses the merits of the OCLT approach. Relation to some related works is given in Sect. 5. And finally Sect. 6 concludes.

2 Overview of Proposed OCL Constructs

Over the years, many different OCL extensions have been proposed and discussed (notably in the OCL Workshop series since the year 2000). We focus here on two extensions that facilitate functional model transformation: *shadow objects*, and *pattern matching.* They are both considered for inclusion in the next version of OCL, as explained in [12]. They were first introducted in [5], and are also discussed in [3] along with other extensions such as lambda expressions and active operations [1]. Although these other extensions could be useful, they are not strictly necessary for the approach presented in this paper. This section presents shadow objects and pattern matching with emphasis on their application to model transformation.

2.1 Shadow Objects

OCL already offers immutable tuples with labeled components. These tuples notably help with complex computations by enabling the construction of temporary data structures. The following example shows a possible tuple-based representation of class named C owning an attribute named a:

```
1 Tuple {name = 'C', attr = OrderedSet {Tuple {name = 'a'}}}
```

The outermost tuple is a class, and the innermost tuple an attribute. One can note that these facts are not captured in the tuple representation. Although it would be possible to add an explicit *type* component to both tuples, shadow objects extend tuples with an attached model element type, as illustrated below:

```
1 Class {name = 'C', attr = OrderedSet {Attribute {name = 'a'}}}
```

Like tuples, shadow objects are immutable and can be processed by OCL expressions. The semantics of OCL is only modified so that they are mostly indistinguishable from actual model elements. Shadow objects can be useful in side effect-free OCL expressions (e.g., as metamodel-typed tuples). But they are especially convenient when explicitly supported by a host language. For instance, a model transformation language may create an actual element in a model when a shadow object is assigned to a property of an existing model element. Model element creation can thus use the same standard OCL syntax in all host languages.

2.2 Pattern Matching with OCL

Pattern matching is a construct found in several successful functional languages (e.g., Haskell, ML, Scala), but not in OCL. It is typically used to analyze the structure of data. Existing OCL-based model transformation languages typically heavily rely on OCL guards for rule matching. For instance, to match all `Attribute`s named `'id'` and not multivalued, one may write (in ATL-like syntax):

```
1 a : Attribute (
2     a.name = 'id' and not a.multiValued
3 )
```

To each `Attribute` in turn a variable named a is bound (line 1), and then a guard (line 2) is evaluated to test if `Attribute` a matches or not. The guard becomes more verbose when the values of more properties need to be examined. With pattern matching, one may write:

```
1 a@Attribute {
2     name = 'id',
3     multiValued = false
4 }
```

The `@` character (line 1) denotes an as-pattern (like in Haskell and Scala), which binds the matched value to the variable. The pattern we have here is an object pattern that matches model elements (or shadow objects). It consists of a type: `Attribute` (line 1), and a set of slots (lines 1-2) between curly braces. Each slot details the value (right of equal symbol) that its associated property (named on the left of the equal symbol) must have for a match. More complex pattern matching can be performed: all values can be matched (e.g., `Tuples`, `Collections`), and multiple variables may be bound in a single pattern.

Moreover, in the context of this paper, we decide to support non-linear patterns (i.e., patterns in which a given variable may be bound several times). Nonetheless, guards are still useful, and can be combined with pattern matching.

3 Application to Model Transformation

3.1 Traceability and Side Effects

As mentioned earlier, OCL is purely functional and does not permit side effects on mutable data structures. However, models and their elements are often represented as such. This is notably the case in EMF[1]-based tools. Whether they should rather be represented as immutable data structures or not is beyond the scope of this paper. We want to find a solution that plays well with mutable models as well. The resolution of trace links is another issue: it typically works by linking (and therefore updating) elements created at different places.

In order to address these problems, we add a specific layer to OCLT. After the execution of an OCLT transformation, this layer translates shadow objects into actual model elements, and performs trace links resolution. These actions are only performed at the end of each transformation before their results can be reused (e.g., by another transformation). We also impose that whole models are created by OCLT transformations (i.e., no update to existing models). Therefore, model creation can happen atomically, models as seen from OCLT can be considered as immutable, and the purely functional property of OCLT can be preserved. We add a new type of OCL expression called *transfo* in order to identify which OCLT functions require this specific layer to kick in. This is its only syntactically visible aspect. The workings of trace link resolution are best explained on a case study. They are therefore explained in the next section.

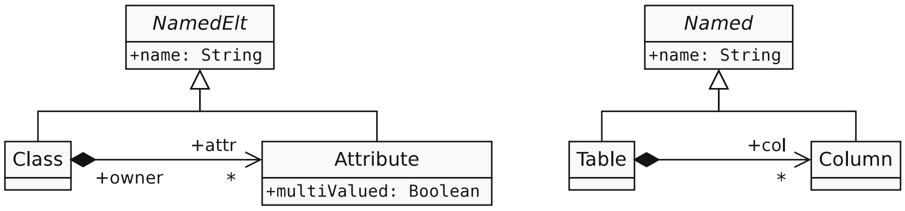
3.2 ClassDiagram2Relational in OCLT

This section shows how the *ClassDiagram2Relational* transformation can be encoded in OCLT, as given in Listing 1. The source and target metamodels are given in Fig. 1. They were adapted from [9].

The transformation is written as OCLT function *classDiagram2Relational* with type *transfo* (line 1). It is composed of three parts similar to model transformation rules: *Class2Table* (line 4), *SingleValuedAttribute2Column* (line 14), and *MultiValuedAttribute2ColumnsAndTable* (line 17). Each *rule* is encoded as a *case* in a single *collect* over the whole source model contents (line 2). Although the syntax of *case*s is different from the one presented in [5], it is equivalent. *collect* ignores elements that do not match any pattern, like an implicit *select*.

Rule *Class2Table* selects instances of *Class* from the source and binds them to variable *a* since they trivially match the empty object pattern (line 4). A shadow object instance of *Table* is then created before being collected to the target (lines 5 to 12). The mapping between the class and the relational table is defined

[1] Eclipse Modeling Framework: https://www.eclipse.org/modeling/emf/.



(a) ClassDiagram (b) Relational

Fig. 1. Metamodels for the *ClassDiagram2Relational* transformation.

within the shadow object directly by setting its properties. The name of the table matches exactly the name of the class (line 6), and its columns consist of a column defining the primary key (line 8) union the set of columns representing the single-valued attributes of the class (line 10).

According to the Relational metamodel, the *cols* property of `Table` only accepts `Column`s as values. Therefore, trying to put `Attribute`s in this property is an issue. OCLT relaxes the type system for shadow object so as to temporarily allow it to happen, until trace link resolution kicks in. Once the whole transformation has been executed, all source elements stored in the properties of target elements (such as `Attribute`s being stored in property *cols* of `Table` here) are resolved into their corresponding target elements. The trace links between source elements and target elements required for resolution are automatically created during the execution of every *collect* iterator that has a collection of source elements as input, and a collection of target elements as output. Therefore, our single-valued `Attribute`s stored in property *cols* are ultimately replaced by the `Column`s created in the *case* labeled *SingleValuedAttribute2Column*. This mechanism is similar to the implicit trace link resolution of ATL.

The next two rules follow a similar construct based on the use of pattern matching and shadow objects. They however differs from the first rule by introducing variables *n* and *on* (lines 14 and 18) directly within the pattern expression for capturing the values of object properties, rather than using a single variable representing the matched object *c* (line 4). This example illustrates the two styles that can be used for writing pattern expressions (navigation or deconstruction), but using a single style for a whole transformation may be preferable.

Listing 1. *ClassDiagram2Relational* in OCLT.

```
 1 transfo: classDiagram2Relational(sourceModelContents:
 2 OrderedSet(NamedElt)): OrderedSet(Named)=sourceModelContents->collect(
 3         -- Class2Table
 4         case c@Class {} |
 5             Table {
 6                 name = c.name,
 7                 cols = OrderedSet {
 8                         Column {name = 'id'}
 9                     }->union(
10                         c.attrs->select(a | not a.multiValued)  -- resolving!
11                     )
12             }
13         -- SingleValuedAttribute2Column
```

```
		case Attr {name = n, multiValued = false} |
			Column {name = n}
		-- MultiValuedAttribute2ColumnsAndTable
		case Attr {
				owner = Class {name = on},
				name = n, multiValued = true
			} |
			Table {
				name = on + '_' + n,
				cols = OrderedSet {
						Column {name = 'idref'},
						Column {name = n}
					}
			}
	)
```

4 Discussion

The previous sections presented the OCLT approach, and its application to a well-known case-study. In this section, we briefly discuss five points: model transformations seen as functions in Sect. 4.1; interoperability with model transformation languages in Sect. 4.2; performance benefits of pattern matching in Sect. 4.3; an alternative *rule* structuring in Sect. 4.4; and some limitations of the OCLT approach in Sect. 4.5.

4.1 Model Transformations as Functions

When model transformations are functions, functional programming techniques become usable. External model transformation composition [11] is simply achievable via function composition.

Considering model transformations as functions is not a new idea. For instance, the type system introduced in [10] gives a function type to every model transformation. It thus enables type checking of model transformation compositions. However, this type system only considers black-box functions. With OCLT, even the internals of transformations are expressed in a functional language.

The case of higher-order transformations [8] (HOTs) is similar: existing techniques are closer to transformation generation. It is the black-box view of these transformations as functions, which has a higher-order functional type. Adding lambda expressions and partial application to OCLT would enable HOTs as high-order functions.

4.2 Interoperability with Model Transformation Languages

We consider two different motivations for interoperability between model transformation languages. (1) *Reusing* transformations written in other languages. (2) *Leveraging capabilities* of several languages.

Motivations 1 and 2 can be achieved by existing transformation composition approaches. Moreover, OCLT could be extended to support functional composition of transformations written in several languages. In this case, these transformations are considered as black-box functions.

However, sometimes only part of a transformation may need to be written in a different language. Because OCL is used in several existing model transformation languages, internal composition [11] with OCLT becomes possible by integrating the OCL extensions of OCLT into these transformation languages. Concretely, partial OCLT transformations could be integrated anywhere the host language allows OCL expressions. The host language could then benefit from OCLT capabilities (motivation 2).

Finally, OCLT could also be compiled into existing model transformation languages, which would achieve motivations 1 and 2. This would also be one way to implement OCLT. Pattern matching can be relatively easily transformed into regular OCL guards for languages that do not support complex patterns such as ATL. Thus, flat OCLT transformations such as the one presented in Sect. 3.2 would be relatively trivial to compile to QVT or ATL. Nonetheless, it may be more difficult to compile complex rule dependencies such as could potentially be achieved in more complex OCLT transformations. There may also be some issues if the target language only offers declarative rules with specific scheduling incompatible with OCLT.

4.3 Performance Benefits of Pattern Matching

Pattern matching can make OCL expressions more readable and less verbose [5]. But it can also have a positive impact on performance. For instance, to match a `Class` with an `Attribute` it owns, one may write (in ATL-like syntax):

```
1 c : Class,
2     a : Attribute (
3     c.attr->includes(a)
4 )
```

Naive execution is very expensive because the cartesian product of the sets of all `Class`es and of all `Attribute`s must be filtered with the guard (line 3). Deep guard analysis can result in a significant optimization: given a `Class`, only the `Attribute`s it owns need to be considered. But it relies on extracting the intent behind the guard, which is not a trivial task in the general case. With pattern matching, the intent is directly expressed at the right level of abstraction:

```
c@Class {
    attr = Set {a : Attribute, ...}
}
```

The dots at the end of the set denote that the matched set may contain other elements than the matched attribute. With such a pattern, it is relatively simple for each `Class` c to iterate only on `Attribute`s it owns.

Of course, pattern matching cannot express all relationships between model elements. Therefore, guards must still be permitted. In OCLT as presented here, guards may be encoded using pre-filtering (using the *select* iterator) or with the *if-then-else-endif* expression. A possibly better solution would be to integrate the *selectCollect* iterator proposed in [12] into OCLT.

4.4 ClassDiagram2Relational Without Cases

Listing 2 gives a different version of the *ClassDiagram2Relational* transformation that does not make use of cases. It relies on the implicit selection performed by *collect* when patterns do not match. If a guard is required, then *selectCollect* could be used. A drawback of this new version is that a naive implementation would traverse the whole source model three times instead of once. However, it has the advantage that each *collect* may traverse different collections. This may prove useful to apply different *rules* to different models. Another potential use is to *collect* on a cartesian product of model element collections (with multiple iterators). This is one possibility to express model transformation rules that take multiple source elements.

Another way to express rules without relying on cases is to follow an approach similar to the definition of functions with equations, which is used in functional programming languages like Haskell. However, such an approach would not easily support rules with different numbers of source elements.

Listing 2. *ClassDiagram2Relational* in OCL without cases.

```
transfo: classDiagram2Relational_WithoutCases(sourceModelContents:
OrderedSet(NamedElt)): OrderedSet(Named) = sourceModelContents->collect(
sourceModelContents->collect(
              [...]    -- Class2Table
    )->union(
        sourceModelContents->collect(
              [...]    -- SingleValuedAttribute2Column
        )
    )->union(
        sourceModelContents->collect(
              [...] -- MultiValuedAttribute2ColumnsAndTable
)
```

4.5 Limitations of the Approach

The OCL extensions presented in this paper enable writing whole transformations in OCLT. We have nonetheless identified the three following limitations:

- **Explicit trace link resolution** is not currently possible. All trace link resolution is performed entirely automatically by the specific layer of OCLT. However, our experience with ATL has shown that explicit trace link resolution (with *resolveTemp*) is sometimes useful.
- **Model refining** transformations leave most of a model unchanged, and only perform few changes. This is notably what the refining mode is for in ATL. OCLT does not currently offer such a capability. This mostly becomes an issue when in-place changes must be performed. Otherwise, it is always possible to copy all unchanged elements.
- **MxN rules** transform *M* source elements into *N* target elements. OCLT can currently handle multiple source elements by *collect*ing over cartesian products as discussed in Sect. 4.4. However, multiple target elements is not currently supported. It would be possible to return a collection of elements for matched source element. This may work because *collect* automatically

flattens collections. However, such rules may need to be specified separately (e.g., using *union* as in Listing 2). A more critical issue would be to enable trace link resolution to one target element among several. This would be difficult to support without explicit trace link resolution.

5 Related Work

In [5], Clark proposes to add pattern matching and object expressions (similar to shadow objects) to OCL and already addresses the similarities with functional programming languages and graph-based transformation languages. While Clark tackles the issue of navigation expressions and their verbosity for expressing constraints, our proposal focuses on model transformation. Of course, all advantages noted by Clark also apply to OCLT.

In [7], Pollet et al. propose new constructs for implementing model manipulation in OCL using the concept of actions where navigation through the elements of the models is available. Our approach extends OCL to enable similar declaration of model manipulation actions. Pollet et al. and Cariou et al. also propose to express contracts [4,7] on OCL actions. This is currently not a concern for OCLT.

In [2], Bergmann proposes to tranform OCL constraints into EMFQuery to improve the performance of querying models. In [13], Winkelmamm et al. propose to transform a subset of OCL constraints into graph constraints. The intent of this approach is to generate valid instances of model for a given metamodel for testing purposes. While the generation of instances might be considered as a specific kind of model transformation, our approach focuses on the definition of model transformation rules. The use of these rules for model synthesis could be investigated in further research. These two works show that translation of OCL guards into patterns is possible in some cases.

6 Conclusion

This paper has presented OCLT, an OCL-based approach to express model transformations. OCLT relies on two OCL extensions (pattern matching and shadow objects) that are considered for inclusion in OCL 2.5 [12]. Therefore, the only lasting difference with OCL may be the new *transfo* type of expressions along with its semantics. *transfo* expressions are post-processed by instantiating shadow objects in actual models, and by resolving trace links.

The *ClassDiagram2Relational* transformation written in OCLT looks similar to, and is as readable as with more traditional rule-based model transformation languages. Because OCLT transformations are purely functional, they can directly use techniques such as functional composition. Partial application and higher-order functions have not been deeply investigated yet but look promising.

As an exploratory work, OCLT still need further work to become actually usable. Notably, its specific *transfo* type and associated layer should be given clear and precise semantics. Then, a full implementation should be created.

Finally, the addition of other proposed OCL extensions should be evaluated. For instance, adding an active operations semantics [1,3] to OLCT has the potential of enabling incremental synchronization, with at least partial bidirectional updates. However, such an addition may be difficult to reconcile with the purely functional aspect of OCLT.

References

1. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.-M.: Active operations on collections. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 91–105. Springer, Heidelberg (2010)
2. Bergmann, G.: Translating OCL to graph patterns. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) MODELS 2014. LNCS, vol. 8767, pp. 670–686. Springer, Heidelberg (2014)
3. Brucker, A.D., Clark, T., Dania, C., Georg, G., Gogolla, M., Jouault, F., Teniente, E., Wolff, B.: Panel Discussion: proposals for Improving OCL. In: Proceedings of the 14th International Workshop on OCL and Textual Modelling, pp. 83–99 (2014)
4. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: OCL for the specification of model transformation contracts. In: OCL and Model Driven Engineering on UML 2004 Conference Workshop, vol.12, pp.69–83 (2004)
5. Clark, T.: OCL pattern matching. In: Proceedings of the MODELS 2013 OCL Workshop, pp. 33–42 (2013)
6. Object Management Group (OMG). Object Constraint Language (OCL), Version 2.4. February 2014. http://www.omg.org/spec/OCL/2.4/
7. Pollet, D., Vojtisek, D., Jézéquel, J.-M.: OCL as a core uml transformation language. In: Workshop on Integration and Transformation of UML models WITUML (held at ECOOP 2002), Malaga(2002)
8. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009)
9. Tisi, M., Jouault, F., Delatour, J., Saidi, Z., Choura, H.: FUML as an assembly language for model transformation. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 171–190. Springer, Heidelberg (2014)
10. Vignaga, A., Jouault, F., Bastarrica, M.C., Brunelière, H.: Typing artifacts in megamodeling. Softw. Sys. Model. **12**(1), 105–119 (2013)
11. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 152–167. Springer, Heidelberg (2008)
12. Willink, E.: OCL 2.5 Plans. Presentation given at the 14th International Workshop on OCL and Textual Modelling, September 2014. http://www.software.imdea.org/OCL2014/slides/OCL25Plans
13. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. Electron. Notes Theor. Comput. Sci. **211**, 159–170 (2008)