On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL

Frédéric Jouault and Olivier Beaudoux

Groupe ESEO, Angers, FRANCE,
{*firstname.lastname*}@eseo.fr

Abstract. Many relations between model elements are expressed in OCL. However, tool support to enable synchronization of elements based on OCL-expressed relations is lacking. In this paper, we propose to use active operations in order to achieve incremental execution of some OCL expressions. Moreover, bidirectionality can also be achieved in non-trivial cases.

1 Introduction

Relations between model elements are often expressed as OCL [9] expressions. These may be intra-model relations, which may for instance specify the values of derived features with respect to the values of other features. They may also be inter-model relations, which may for instance specify transformations. Although it is generally easy to compute the value of OCL expressions, doing so in such cases is often not enough.

Consider two variables a and b. A relation between them can be expressed in several ways: 1) $a = f(b)$, 2) $b = g(a)$, 3) $h(a) = i(b)$, or 4) $j(a, b) = true$, where f, g, h, i, and j are functions denoting potentially complex OCL expressions involving their arguments. It is easy to compute the value of a given b from 1), or of b given a from 2). However, computing the value of b given a from 1), of a given b from 2), or of a or b given the other from 3) or 4) can be much more complex.

Moreover, models do change, thus potentially invalidating relations. *Synchronization* corresponds to performing appropriate changes to make relations hold again. It is generally not trivial. If changes always happen on one part (e.g., non-derived features, or source models), then relations may be expressed such as the other part (e.g., derived features, or target models) can be computed easily. If changes can happen on any part (e.g., with changeable derived features, or bidirectional transformations), then there is no way to express relations in OCL such that computation is always easy.

Figure 1 illustrates synchronization of two models M_A and M_B related by relation R (decomposable into model-element-level relations). At some point, M_A evolves into M'_A after some changes (denoted by an arrow labeled a), and relation R may not hold between M'_A and M_B. Synchronizing M_B with M'_A consists in evolving it into M'_B by performing some changes (denoted by an

arrow labeled b) such as R holds between M'_A and M'_B. Alternatively, changes denoted by arrow b may happen before those denoted by arrow a, and M_B may be the first model to evolve into M'_B, requiring M_A to be evolved into M'_A.

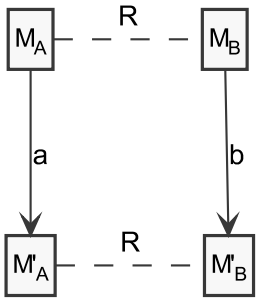

Fig. 1. Synchronization of models M_A and M_B related by relation R

Active operations [2,1] enable evaluation of operations on collections in a way that is both: 1) incremental (i.e., propagating changes instead of recomputing whole expressions), and 2) bidirectional (i.e., enabling changes to the value of an expression to be propagated back to its source collection). Following a proposal made during the OCL 2014 Workshop panel discussion (see Section 5 of [5]), this paper presents how active operations can be applied to OCL in order to partially address this synchronization problem. A Java implementation of an active operation framework supporting EMF[1] has been developed. As of writing this paper, it is available in a development branch[2] of Papyrus[3]. Manual rewriting of OCL expressions into active operations has been experimented. The resulting active operations have been used in a bidirectional transformation between a profiled UML model, and a model conforming to a metamodel corresponding to the profile. Change propagation in both directions have been extensively tested to behave as expected.

Incrementality is defined in Section 2. Section 3 exposes what active operations consider as immutable and mutable values. Active operations are introduced in Section 4, and their application to OCL is presented in Section 5. In Section 6, our implementation is briefly described. Section 7 discusses some limitations of the approach, along with some ways to mitigate them. Finally, some related works are listed in Section 8, and Section 9 concludes.

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

[2] `http://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/tree/extraplugins/aof?h=committers/fnoyrit/aofacade&id=8bec1ad60253cc854cbd3734efa424bfed0e0bbe`

[3] `https://eclipse.org/papyrus/`

2 Incrementality

When a model is represented by an immutable data structure (e.g., in purely functional approaches), then the only way to perform synchronization is to compute a new model. For derived features, this requires computing a new version of the model in which the corresponding relations hold again. For transformations, the model that was not changed need to be recomputed from the one that did change such that the corresponding relations hold again. However, models are often represented by mutable data structures (e.g., with EMF in Java). In such a case, it is still possible to recompute whole models like with immutable data structures. However, another possibility is to update models in-place by performing small changes that make relations hold again. This is called *incremental* synchronization.

Incremental synchronization has the potential to be more efficient because only relatively small changes are typically required when compared to whole model recomputation. It also updates traceability links between the synchronized models instead of creating new ones with the recomputed model. Moreover, it also avoids creating new elements but rather updates existing ones. This results, for instance, in an interesting advantage when models are being edited in graphical views. Visual shapes are typically bound to the model elements they represent, and are updated when they change. Updating models in-place means that visual editors can directly reflect changes to users. Conversely, recomputation results in whole new models with elements not bound to any visual shape, which means no change is being made visible to users.

Incremental evaluation of OCL expressions can be used to achieve incremental model synchronization. It is based on the same idea of in-place updates, and similarly relies on mutable values[4]. Values resulting from expression evaluation as well as intermediate values corresponding to sub-expressions are updated in-place. For instance, given an ordered set `s` with initial value `OrderedSet {1, 2, 3}`, and expression `s->collect(e | e + 2)->select(e | e > 4)`. The initial evaluation of the expression yields value `OrderedSet {5}` with intermediate value (after the `collect`, but before the `select`): `OrderedSet {3, 4, 5}`. If `s` changes into `OrderedSet {1, 4, 3}` (i.e., `2` is replaced by `4`), then incremental evaluation will start by updating the intermediate value to `OrderedSet {3, 6, 5}` (i.e., replacing `4` by `6`). Then, the value of the whole expression will be updated into `OrderedSet {6, 5}` by adding `6` to it.

3 Mutability of Values

This section starts with values that cannot change before going into mutable values that active operations consider. Finally, it is applied to models.

[4] This does not prevent OCL expressions from always having the same values as if they operated on immutable values, following the OCL specification [9].

3.1 Immutable Values

Primitive values are defined as immutable. These notably include: booleans, numbers, and strings. This is consistent with OCL, and is a choice made by many programming languages (e.g., Java), even if they support mutability.

An immediate consequence is that operations on immutable values do not need to propagate changes. Therefore, these operations are defined in the usual way.

3.2 Mutable Values

Mutable values wrap other (mutable or immutable) values. There are two kinds of mutable values supported by active operations: boxes and objects. A box can be a singleton, or a collection. Each box is *observable*. It notifies *listeners* of its changes: addition, removal, replacement, or move of a wrapped value.

Remark: the value wrapped by a singleton box may be replaced by another one. Since a mutable primitive type variable cannot have its immutable primitive value mutated, it is actually defined as a mutable singleton wrapping a primitive value. In this way, changing the value of the variable actually corresponds to replacing the value wrapped by the singleton box with another one (i.e., changing the content of the box without changing the box itself).

The two kinds of singleton boxes, and four kinds of collection boxes are:

- **Mandatory singletons** (called **one**) are boxes that must contain exactly one value. The contained value can be a different one at different times.
- **Optional singletons** (called **opt**) are boxes that may be empty or contain exactly one value. They may notably become empty, or full.
- **Collections** are boxes that may contain any number of values. They are further classified into four kinds, according to ordering and uniqueness:
  - **Sets** (called **set**) forbid duplicate values and are unordered.
  - **Ordered sets** (called **oset**) forbid duplicate values and are ordered.
  - **Bags** (called **bag**) may contain duplicate values and are unordered.
  - **Sequences** (called **seq**) may contain duplicate values and are ordered.

The type of a box is immutable. This means that, for instance, a **set** will always remain a **set** and never become an **oset**, a **bag**, or a **seq**. The type of elements contained in a box (its *element type*) may be written in brackets (e.g., **one**(String) for a mandatory singleton wrapping a string value). Objects are a special kind of mutable value that contain named slots holding boxes as values.

3.3 Application to Models

Each model element is represented by an object having a fixed type, which is a meta-element coming from a metamodel. The type of an object constrains the types of its slots. Each slot of an object corresponds to a property (attribute or reference) belonging to its meta-element. The type of box used to hold the value of a slot is given by the multiplicity of its corresponding property: an **opt**

for 0..1, a **one** for 1..1, and a collection box for n..m with $m > 1$. In the last case, bounds narrower than 0..* are not enforced by the box itself. The specific kind of collection box depends on ordering and uniqueness (as defined above). The element type of a slot value is either a primitive type (for attributes) or a meta-element (for references).

This scheme can be applied to any kind of modeling framework supporting observation, even one not explicitly based on boxes. We have notably applied it to EMF. We leverage the fact that EMF can notify listeners (called `Adapter`s) of changes to model element properties to provide a box-based view on models. A box is created to represent each slot, but it delegates storage of its contents to actual `EObject`s: there is no need for content duplication.

4 Active Operations

4.1 Operations on Boxes

Active operations enable bidirectional incremental evaluation of expressions involving boxes. The result of each active operation is also a box. Available operations correspond to well-known OCL operations (e.g., conversion between box types, concatenation, `isEmpty`, `notEmpty`, `size`) and iterators (e.g., `collect`, `select`) on collections or operations available in other languages.

An example of the latter category is the `zipWith` operation (e.g., available in Haskell). `zipWith` operates on two collections, and is given a function (called a `zipper`) taking two arguments. It traverses both collections in parallel, and expects them to have the same size. It returns a collection in which every element is obtained by applying the `zipper` function to one element from each collection. For instance, applying `zipWith` on `OrderedSet {1, 2, 3}` and `OrderedSet {1, 1, 2}` with integer addition as `zipper` function results in `OrderedSet {2, 3, 5}`.

The additional `bind` operation can be used to propagate changes between two result boxes. Each active operation propagates changes from its source(s) to its result, and *vice versa*. It does so using operation-specific algorithms (see [2,1]).

Active operations distinguish between: a) forward change propagation from source to result, and b) reverse change propagation from result to source. Forward direction is always supported, but reverse direction often requires more information. Consequently, reverse direction is only supported if enough information is available. In practice, on a concrete bidirectional transformation, reverse direction can be made to work in most cases. Note that only a) is necessary for incrementality, but both a) and b) are necessary for bidirectional incrementality.

Here is how an operation like `size` works. It observes its source box, and its result is a **one**(Integer) that contains as value the size of the box on which it is applied (its source box). It is updated upon removal or addition of values in its source box. The `size` operation is currently only implemented in a unidirectional way: it only supports forward change propagation. However, limited support for reverse change propagation could make sense, and therefore be implemented. For

instance, replacing the value in its result box with a lower value could actually resize its source box by dropping tail elements. Moreover, the size operation could be augmented to a fully bidirectional operation `size(p)` where `p` would be a *provider* function. Provider `p` would return the elements to append to the source box whenever its size increases.

Although this paper is not the place to look at the algorithms between each operation, an overview of how the `collect` operation works is useful. This operation is applied on a source box containing source elements, and is given a *collector* function. It returns a box containing the result of applying the collector to each source element. There are actually ten variants of the `collect` operation[5] exposed to users. Each variant handles a specific combination of change propagation direction (i.e., supporting reverse or not), and mutability of its collector. There is also a forward variant that keeps traceability information, and uses it in order to retrieve already computed elements. Finally, there is a corresponding reverse variant that can read this traceability information.

Reverse change propagation is typically only supported if a reverse collector function is also provided. Property navigation is handled in a special way, and is able to support limited reverse change propagation without requiring a reverse collector. Collecting different values depending on a mutable predicate (e.g., `aCollection->collect(e | if p(e) then f(e) else g(e) endif)`) also requires special handling.

The collector function given to `collect` is either immutable or mutable. An *immutable collector* is a function taking a value as argument, and returning an immutable value. A *mutable collector* is a function taking a value as argument, and returning a box. Collecting with an immutable collector only requires listening for changes on the container boxes (on the source box for forward change propagation, and on the target box for reverse change propagation). Collecting with a mutable collector further requires listening for changes on the result of applying the collector to every source element (called inner boxes).

4.2 Lifting Immutable Operations

Immutable operations (functions or operators) defined on immutable values may be lifted[6] to work on boxes wrapping such immutable values. This enables change propagation for many existing operations. It is trivial for bijective operations such as number or boolean negation, by leveraging `collect`. Non-bijective operations (e.g., absolute value on numbers) can be easily lifted to support forward change propagation on boxes. However, reverse change propagation can generally be performed in several way. For instance, setting the result of an absolute value operation to a positive number (e.g., 5) may be reversed by setting its

[5] Remark: there are also five variants of the `select` operation.

[6] In this context, lifting consists in taking a function operating on simple values, and transforming it into a function operating on boxed values. It works by taking the values out of the boxes before operating on them, and putting them back in boxes afterward.

source to that number (e.g., 5), or to its opposite (e.g., −5). In general, this requires *ad hoc* handling, but a default behavior may be provided (e.g., always returning the positive value). In the general case where no default behavior can solve every problem, users may have to implement the reverse behavior, or at least choose among several possible behaviors (e.g., using annotations).

While `collect` can be used to lift unary operations, binary operations (e.g., the conjunction of boolean values) can be lifted by leveraging `zipWith`. As explained above, there are variants of `collect` without reverse collector that only support forward change propagation, as well as variants with a reverse collector that also support reverse change propagation. When the reverse direction is required to be supported, a reverse collector can be specified to implement default or specific reverse behavior. Similarly, `zipWith` also exists in two variants: one with only a forward zipper function supporting only forward change propagation, and one with an additional reverse zipper function also supporting reverse change propagation. When the reverse direction is required to be supported, a reverse zipper can be specified to implement default or specific reverse behavior.

5 Application to OCL

There are two main aspects to consider in order to apply active operations to OCL: mapping OCL types to active operation types, and rewriting OCL expressions to use active operations such as presented in Section 4.

5.1 Types Mapping

Collection box types and OCL collection types are very closely related, with simple correspondences: **set** for `Set`, **oset** for `OrderedSet`, **bag** for `Bag`, and **seq** for `Sequence`. Model elements are mapped to objects. Although we have not experimented with OCL tuples yet, it seems that they could also map relatively easily to objects.

As for singleton boxes, OCL does not explicitly distinguish nullable values from non-nullable values. However, modeling languages like UML, MOF, and Ecore do. The value of a slot typed by a property with multiplicity [0..1] is mapped to an **opt**. The value of a slot typed by a property with multiplicity [1..1] is mapped to a **one**. In order for every singleton expression to have a definite **one** or **opt** box, static analysis of OCL expressions need to be extended with nullability analysis, which determines whether each sub-expression can actually be null.

5.2 Expression Rewriting

Firstly, operations on primitive values can be made to be the same with active operations and OCL. Secondly, active operations on boxes are quite close to operations on OCL collections. However, several active operations actually correspond to some OCL iterators such as `collect` or `select`. Therefore, in order to know

which one to use, static analysis of OCL expressions need also be extended with mutability analysis. That is, whether each sub-expression can actually mutate must be determined. Moreover, complex collector functions need to be rewritten into several simpler functions in order to make sure that each navigation step is actually observed by an active operation. For instance, `persons->collect(p | p.bestFriend.name)` needs to be rewritten into `persons->collect(p | p.bestFriend)->collect(p | p.name)`.

Manually writing relatively complex active operations expressions is cumbersome. But it showed that static mutability analysis can work. Indeed, in our Java-based implementation of active operations, mutable values are distinguished as being instances of interface `IBox`. Any expression that types as an `IBox` is therefore mutable. Of course, this still needs to be implemented for OCL.

Finally, the `bind` active operation generally corresponds to the OCL equality operator applied on collections.

6 Implementation, Debugging and Testing

So far, our implementation of active operations does not handle translation from OCL. However, it does support EMF models, and a significant-enough subset of active operations that enables writing bidirectional incremental transformations.

Debugging is supported by three tools:

- **Inspection.** The first one is the `inspect` pseudo-operation. It has no effect on the data flow, and returns its source box. It is similar to the `trace` function in Haskell, or to the `debug` operation in ATL, except that it stays active and listens to its source box. It logs every change in the console.
- **Data Flow Serialization.** Once an active operations expression has been evaluated, a data flow graph exists in memory to handle change propagation. It is possible to display this data flow graph textually, and to show it in the variable inspection view of the Java debugger.
- **Data Flow Visualization.** Finally, the whole data flow corresponding to the evaluation of all expressions can also be serialized to a textual file. This file can then be further processed by graph layouting tools[7].

Testing active operation expressions (e.g., used for derived features or transformations) has two aspects. First, the passive functionality of the expression (i.e., the computation it initially performs in the absence of change) needs to be tested (e.g., using unit testing). This can be performed using traditional techniques. Second, the active behavior needs to be tested to make sure: 1) that the active operations implementation behaves properly (but this is not the responsibility of a user of active operations), and 2) that the reverse change propagation behaves as intended. The second point is especially important considering that reverse change propagation can often be performed in several ways, but only one may make sense in a given context. We built some tools to do this by comparing the result of change propagation with full passive reexecution.

[7] We use PlantUML: `http://plantuml.com/`.

7 Limitations

Although the approach presented in this paper enables bidirectional incremental evaluation of many useful OCL expressions, it has the following limitations in addition to general bidirectionality issues:

1. Reverse change propagation typically requires more information than available in OCL expressions (e.g., reverse collector or zipper functions).
2. It is not clear yet that all OCL expressions can be rewritten using a fixed set of active operations, and if so, what these operations are.
3. The current active operations algorithms do not support arbitrary-level traversals such as can occur in arbitrary-level collection flattening or closures.

While all these points are open issues, it is possible to mitigate them:

- To mitigate 1:
  - Lifted immutable operations may be annotated with information about which reverse behavior to choose from. For instance, a lifted number addition may be annotated to reverse propagate changes by modifying only one of its operand, or both. If both are modified, it may distribute the change evenly, or not.
  - Specific active operations may be defined with semantics appropriate for a given context. However, this is not trivial since this requires designing a custom bidirectional propagation algorithm.
  - A search-based approach that explores possible solutions to find a "good" one (according to a fitness function) could be used. However, it is unclear at this time how this could interact with active operations algorithms.
  - Finally, if reverse propagation cannot be defined in a meaningful way, it is always possible to make corresponding properties read only in metamodels.
- To overcome 2 and 3, other techniques can be used for forward change propagation (e.g., reevaluating whole sub-expressions). However, reverse change propagation may not be possible in such a case.
- To ease the developer's job despite 1 and 2, a development environment could help. It could warn expression writers of problems (e.g., missing reverse annotation, rewriting impossibility), thus making it easier to write in the appropriate subset of OCL. At least it would make it easy to know when we loose some property (e.g., support for reverse change propagation).

8 Related Work

Firstly, *ad hoc* solutions for bidirectional synchronization can be implemented in general purpose languages such as Java. However, such approaches mix together domain-specific concern with technical aspects. Some frameworks like AngularJS do support data-binding that can perform change propagation. However, the kind of supported relations is quite limited.

Secondly, functional approaches based on lenses can be used to express relations similarly to what can be done with OCL. Some approaches such as [6] enable bidirectional transformations based on lenses. However, functional approaches are not easy to marry efficiently [7] with side-effect based modeling frameworks like EMF.

Finally, there are some graph-pattern based approaches such as IncQuery [10]. IncQuery uses a custom query language based on graph patterns. There is tool support to translate some OCL expressions into these patterns [3]. Although active operations also require some rewriting from OCL, expressions built using active operations are structurally similar to corresponding OCL expressions, whereas graph patterns have different structure. Furthermore, IncQuery may be used as part of VIATRA [4] transformations. VIATRA enables fine-grained custom reactions to specific change events. It should be possible to achieve similar results with active operations, by coupling them with as powerful a transformation language (e.g., possibly by making VIATRA use active operations in addition to IncQuery).

9 Conclusion

This paper proposes an approach that enables incremental (i.e., supporting forward change propagation) evaluation of OCL expressions. Furthermore, this approach can also be made to support reverse change propagation directly in some cases, and with additional annotations in others. Bidirectional incremental evaluation of OCL expressions can thus be achieved in some cases. Applying this approach to the implementation of a model transformation has shown that bidirectionality can be achieved in non-trivial cases.

The approach is based on active operations, of which an overview has been given. Not all OCL expressions can be translated into active operations, and of those which can, not all can propagate changes in the reverse direction. However, it is not clear yet which subset of OCL can be translated, and which (smaller) subset can support reverse propagation. We expect that further works on the rewriting of OCL expressions into active operations will help define these subsets more precisely. The presented approach is still missing automatic rewriting of OCL expressions into active operations, and still has to be evaluated for performance and scalability.

While working on this approach, we noted that it could be useful to have the notion of optional values in OCL. This would enable tools to support null-checks and safe navigation. By combining the approach presented in this paper with the OCLT approach presented in [8], it may become possible to express bidirectional incremental transformations directly in OCL.

References

1. Olivier Beaudoux. *Vers une programmation des systèmes interactifs centrée sur la spécification de modèles exécutables.* Habilitation à diriger des recherches, Université d'Angers, August 2014.

2. Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Active Operations on Collections. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 91–105. Springer Berlin Heidelberg, 2010.
3. Gábor Bergmann. Translating OCL to Graph Patterns. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 670–686. Springer International Publishing, 2014.
4. Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, volume 9152 of *Lecture Notes in Computer Science*, pages 101–110. Springer International Publishing, 2015.
5. Achim D. Brucker, Tony Clark, Carolina Dania, Geri Georg, Martin Gogolla, Frédéric Jouault, Ernest Teniente, and Burkhart Wolff. Panel Discussion: Proposals for Improving OCL. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling*, pages 83–99, 2014.
6. J Nathan Foster, Alexandre Pilkiewicz, and Benjamin C Pierce. Quotient lenses. In *ACM Sigplan Notices*, volume 43, pages 383–396. ACM, 2008.
7. Sochiro Hidaka and Massimo Tisi. ATLGT: bidirectional ATL on top of GRoundTram. 2015. To appear.
8. Frédéric Jouault, Olivier Beaudoux, Matthias Brun, Mickael Clavreul, and Guillaume Savaton. Towards Functional Model Transformations with OCL. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, volume 9152 of *Lecture Notes in Computer Science*, pages 111–120. Springer International Publishing, 2015.
9. Object Management Group (OMG). Object Constraint Language (OCL), Version 2.4. `http://www.omg.org/spec/OCL/2.4/`, February 2014.
10. Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98:80–99, 2015.