Introducing the Active Map operation to unify and improve efficiency of active operations

Frédéric Jouault and Fabien Chhel

ERIS, ESEO-TECH, Angers, France
{firtname.lastname}@eseo.fr

Abstract. The active operations approach enables incremental evaluation of OCL-like expressions, and can also be used to implement incremental model transformation. Each active operation corresponds to a basic building block such as select, or collect, and encapsulates both its initial computation algorithm as well as its change propagation algorithms. Complex operations such as groupBy can generally be expressed using simpler operations such as select and collect, but have better performance with an efficient map data structure (e.g., a HashMap) as internal state. However, implementing new algorithms for each new kind of complex active operation is costly and error prone. In this paper, we introduce the Active Map active operation that can be used as a basis to express several complex operations, such as groupBy, without specific propagation algorithms. Several complex operations are shown to be expressible in terms of the Active Map operation without sacrificing scalability when compared to ad-hoc implementations.

1 Introduction

Incremental evaluation of OCL-like expressions, including in the context of model transformation, has many potential applications. It can for instance be used to improve the performance of querying or transforming rapidly changing models. Another usage scenario is to update existing target models in-place to make sure other connected elements (e.g., diagrammatic views) are updated automatically. In order to address this need for incremental evaluation, active operations [1] provide a conceptual approach for incremental evaluation of OCL-like expressions [7]. All mutable values are wrapped in boxes, which may be either singletons: options that may be empty (then equivalent to `null`), or ones that may never be empty; or collections. Each active operation basically corresponds to an OCL operation like `size`, `collect`, or `select`. Active operations are not only able to compute result boxes from source boxes (like most OCL implementations), but they are also able to propagate changes occurring on either side to the other. When a change occurs, the source and target boxes are temporarily inconsistent, and it is the role of the active operation to restore consistency by propagating the change.

The Active Operations Framework (AOF) [8] provides an implementation of this approach. One of the main difficulty in implementing a framework like AOF is to handle the many existing operations from the OCL standard library [10], for all the types of boxes (two singletons, plus the four OCL collection types) with algorithms for all possible kinds of changes (element addition, removal, replacement, and move). In order to alleviate this difficulty, several choices have been made. All kinds of boxes share as much code as possible, and each operation supports all kinds of boxes. Moreover, not all OCL standard library operations are implemented as specific operations. Many operations are expressed in terms of a limited set of basic operations.

However, when scalability becomes an issue, some specific operations require *ad-hoc* implementations. This was notably observed in [8] for two operations: `groupBy` and `selectBy`. These are not standard OCL operations, but they can be expressed using standard OCL operations. However, such implementations are not scalable, which made it necessary to implement optimized `groupBy` and `selectBy` operations, each using a `HashMap` as internal state.

All such specific operations cannot be integrated in a framework like AOF, otherwise its complexity would increase, at the risk of becoming unmaintainable. It would therefore be useful to define a single active operation using which operations like `groupBy`, and `selectBy` can be expressed without sacrificing scalability.

This paper presents such an operation called *Active Map*. This operation also wraps a `HashMap`, but provides general-purpose access to multiple changeable views, which makes it possible to express `groupBy`, and `selectBy` with it.

The Active Map operation makes it possible to express several other operations, while preserving scalability properties. It thus enables the unification of multiple operations relying on `HashMap`s to ensure scalability, and makes it easier for developers to create new scalable operations.

The remainder of the paper is organized as follows. Section 2 details the tackled problem, and gives a motivating example. The Active Map operation is defined in Section 3, and its API is presented in Section 4. Section 5 presents several applications of the Active Map operation. Finally, some related work are discussed in Section 6, and Section 7 gives some concluding remarks.

2 Problem statement

This section states the problem in more details by starting with some context on operation reuse (Section 2.1) before explaining why preserving efficiency and scalability is not trivial (Section 2.2). Finally, Section 2.3 presents a motivating example.

2.1 Reusing operations

When writing OCL code, developers use the set of operations (including iterator expressions such as `collect`, and `select`) provided in the OCL standard library to write their own expressions, or even create their own operations (e.g., using the `def` keyword). Having such a rich standard library makes the developers' job easier: they do not need to implement the same basic algorithms constantly. However, for OCL implementors, the richer the standard library, the more costly its implementation is. This is even more the case for incremental implementations: in addition to designing and implementing algorithms for specific computations, algorithms for change propagation are also required for each possible kind of change: adding, removing, replacing, and moving elements. Moreover, these specific algorithms also need to be tested. Non-incremental implementations must be tested for each corner case of their computation, but incremental implementations must also be tested for each propagation corner case for each possible kind of change. Then there is also maintenance cost.

Fortunately, many OCL operations can actually be expressed in terms of other operations, which is notably done in the OCL specification [10, Section 11.9, pp. 177–183] to define iterator expressions. For instance, `reject` is defined in terms of `select`, while `exists` and most others are defined in terms of `iterate`. But, although not detailed in the specification, most non-iterator operations can also be expressed in terms of other operations, and some iterator operations can also be expressed without `iterate`. We can, for instance, reuse the syntax used in the OCL specification to define iterator expressions in order to express `notEmpty` in terms of `size`, `includes`, and `exists` in terms of `select`, and `notEmpty`, as well as express `count` in terms of `select` and `size`, as shown in Listing 1.1.

Listing 1.1. `notEmpty`, `includes`, `exists`, and `count` expressed using other operations

```
1 source->notEmpty() = source->size() <> 0
2 source->includes(o) = source->select(e | e = o)->notEmpty()
3 source->exists(iterator | body) =
4   source->select(iterator | body(iterator))->notEmpty()
5 source->count(o) = source->select(e | e = o)->size()
```

Note that this syntax, although used in the OCL specification, cannot itself be OCL compliant because OCL has no mechanism to define general lambda expressions. Adding lambdas to OCL has already been discussed [3,13], but not done yet. OCL only supports the limited mechanism provided by iterator expressions, and does not provide any mechanism to call lambda expressions. Therefore, the way `iterator` and `body` are expressed at line 3, and the way `body` is applied to `iterator` at line 4 are not OCL compliant. Moreover, it would actually be possible to express `notEmpty`, `includes`, and `count`, which are not iterator expressions, in pure OCL syntax, for instance by defining `def` constraints. However, we choose here to use the same syntax for all such expressions of one operation in terms of others for simplification reasons.

There are generally several ways to express one operation in terms of others. For instance, `exists` is defined using `iterate` in the OCL specification, whereas we defined it above using `select`, and `notEmpty`. Which expression of an operation in terms of others is the most useful depends on the context.

2.2 Preserving scalability

For incrementality purposes, expressing operations in terms of `iterate` is not especially useful. Indeed, `iterate` is hard to make efficiently incremental because it computes its result by successively applying its body to both each element of its source collection as well as the intermediate result it computed for the previous element. Therefore, a change to any element in its source collection requires recomputing the iteration for that element as well as all iterations for the following elements. This typically results in linear change propagation time: a change on any source element can entail retraversing the whole source collection.

There are consequently two main ways to preserve the scalability of all operations: 1. lots of code, and 2. smart rewriting. Writing specific code for each operation (i.e., solution 1) results in more efficient code, but high development costs as mentioned at the beginning of Section 2.1. The second approach consists in implementing a relatively small number of basic operations, and finding a way to express most others in terms of the basic ones while preserving computational complexity. This second approach has a certain overhead, but as long as computational complexity is kept as low as possible, scalability is not threatened.

Let us, for instance, consider the operations defined in Listing 1.1: `notEmpty`, `includes`, `exists`, and `count`. They are all directly or indirectly expressed in terms of `size`, which has trivial constant time propagation algorithms if it maintains the current size in a variable. This is true even if its source collection is a linked list, for which length computation requires linear time[1]. The current size variable can be incremented when an element is added to the source collection, or decremented when an element is removed from the source collection. Replacing, or moving elements has no impact on the size. `notEmpty` also has constant change propagation time when expressed in terms of `size` because the only other operation involved is comparing a scalar value (i.e., the result of `size`) to 0.

`includes`, `exists`, and `count` all require linear initial computation time because they need to traverse their source collections at least once. Expressing them in terms of `select` does not increase complexity since `select` can also be implemented with a linear initial computation time. Because `size` and `notEmpty` both have constant time propagation algorithms, `includes`, `exists`, and `count` can also have constant time propagation algorithms if `select` has constant time propagation algorithms. As a matter of fact this is the case if `select` does not need to preserve order, which is true here (only size of result is used):

- **Adding** an element to its source results in adding it to its target if its body evaluates to `true` for that element.
- **Removing** an element from its source results in removing it from its target if its body evaluates to `true` for that element.
- **Replacing** an *old* element by a *new* one in its source can be treated as removing the *old* one, then adding the *new* one.
- **Moving** an element in its source has no influence on its target, because order does not need to be preserved.

Other expressions of these operations would not necessarily have constant change propagation time but may rather have linear change propagation time. This is notably the case when expressing `exists` in terms of `iterate`.

[1] This is an example application of the classical time-memory trade-off: it is often possible to get a lower change propagation complexity by storing specific data.

At this point, several interesting questions can be asked: (1) What is the best change propagation complexity achievable for each OCL standard library operation? (2) Is there a minimal set of operations from which all other OCL standard library operations can be expressed while preserving scalability? (3) If so, what is this minimal set? (4) What is the time-memory complexity trade-off for each operation? However, they are all beyond the scope of the present paper. The problem tackled in this paper is to define an operation that can be reused to express operations that so far only have specific implementations relying on `HashMap`s, such as: `groupBy`, and `selectBy`. These operations are not in the OCL standard but were introduced in [8] in order to ensure scalability of the AOF implementation of the VIATRA CPS benchmark [6]. As mentioned in [8], these operations do not need to be made available to users. Instead, they can be used internally by an execution engine, which could detect patterns corresponding to their semantics and rewrite user-specified expressions in terms of these more efficient implementations. However, we place ourselves in the position of an implementor who must define all necessary optimized operations.

2.3 Motivating example: `groupBy`

The previous sections presented the problem of reusing operations while preserving scalability, and concluded by focusing on the problem of finding a reusable operation for cases where a `HashMap` is necessary to achieve scalability. One such case presented itself (see [8]) while implementing the VIATRA CPS benchmark [6]: computing the trace model requires performing a `groupBy` operation. This operation is well-known in database query languages such as SQL, but is not provided by the OCL standard library.

Listing 1.2 gives a definition of this `groupBy` operation in terms of `collect`, `asSet`, and `collect`. `groupBy` is defined as a custom iterator expression taking a body that returns the keys by which the elements of its source collection must by grouped. The `Set` of keys is first computed by collecting the results of calling `groupBy`'s body over its source collection, and then converting it into a `Set` using the `asSet` operation. In a second time, a tuple is constructed for each unique key by collecting over this `Set` of keys. The `left` part of the tuple is set to the key, while its `right` part is set to the collection of elements having that key. This `right` part is computed using a `select` nested inside the `collect`. Because of this nesting of "loops", a naive implementation therefore has quadratic computation time. Moreover, even if a version of `select` with constant propagation time is used, there is one call to `select` per key. This results in linear propagation time when an element is added to or removed from the source collection, as well as when the key of an element changes. Depending upon its implementation, the `asSet` operation may also have linear propagation time.

Listing 1.2. Possible implementation of `GroupBy` in OCL

```
source->groupBy(iterator | body) =
  let keys : Set(OclAny) =
      source->collect(iterator | body(iterator))->asSet() in
  keys->collect(key |
      Tuple {left = key, right =
          source->select(iterator | body(iterator) = key)}
  )
```

The implemented solution proposed in [8] consists in developing a specific *ad-hoc* `groupBy` operation, which uses a `HashMap` internal state in order to achieve linear computation time, and constant change propagation time. Having to develop and maintain such a specific operation would not be such an issue if it was one of its kind. However, a second operation called `selectBy` (described in [8] as well) also had to be developed around a `HashMap` in order to preserve scalability of the same model transformation. This situation triggered the search for a more general operation in terms of which both `groupBy`, and `selectBy` could be expressed without sacrificing scalability.

3 Active Map definition

The previous section has motivated the need for a reusable operation able to wrap a `HashMap` in such a way that other operations requiring `HashMap`s to ensure their scalability can be expressed using it. This section defines such an operation, which we call *Active Map*. Section 3.1 defines the Active Map operation in terms of the views it keeps synchronized, and Section 3.2 extends this definition to support multiple values per key.

3.1 Synchronized views

The Active Map operation is inspired by the `Map` (or associative array) data type found in many programming languages, notably Java in which AOF is implemented. In Java, a `Map` (as defined in interface `java.util.Map`) provides three *collection views*[2]: a set of keys accessible via the `keySet` method, a collection of values accessible via the `values` method, as well as a set of key-value mappings accessible via the `entries` method. When a Java `Map` is changed, its collection views are also updated. However, they are neither observable nor changeable from without the `Map` itself. As can be seen on Figure 1, the Active Map operation also provides these views, called after the corresponding Java methods. The central ellipse represents the Active Map operation, while the rectangles denote boxes. The two-ended arrows link the Active Map operation to the boxes it keeps synchronized.

Figure 1 also shows that there are two other boxes in addition to the three collection views. They both also correspond to the Java methods with the same names, but where the Java methods only return plain objects, they are also boxes for the Active Map operation. The reason is that they may be changed externally, or be made to change by the Active Map operation when propagating a change that occurred on one of its other boxes. These additional boxes are: `size` that contains a single integer whose value is the size of the Active Map,

[2] As specified in the Javadoc documentation: `https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`.

and `isEmpty` that contains a single boolean whose value is true if and only if the Active Map is empty.

Finally, each of the two “3D” boxes shown at the bottom of Figure 1 corresponds to a set of boxes. They correspond to Java methods that take a key as argument: `get(key)` that contains the single value associated to the key, and `containsKey(key)` that contains a single boolean indicating whether the Active Map has a mapping for that key. Each set contains one box per possible key.

An Active Map is consistent if all its boxes have the values a non-active `Map` would have. The purpose of the Active Map operation is to restore consistency upon changes: whenever a change occurs on any of its boxes[3], it must update the other boxes so that it is again consistent.

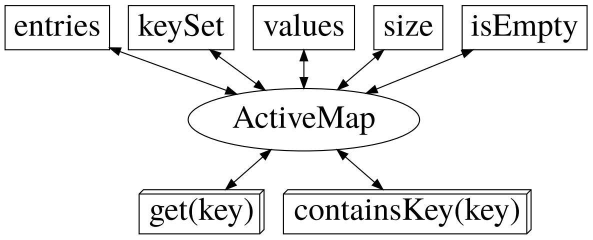

Fig. 1. Active Map views

Remark: calling the Active Map an *operation* may seem counter-intuitive to the reader. A `Map` is generally considered to be a data type, and a `HashMap` to be a data structure implementing such a data type (among other possible implementations). We could arguably have considered it a *map* box, or defined it as a new kind of active data structure. The choice of calling it an operation may be controversial, but is based on the following consideration. Each active operation keeps a set of boxes synchronized. This set often has a fixed size of 2, and simply consists of a source and target boxes. This is notably the case for `collect`, and `select`, which each have a source collection, and a result collection. An Active Map also keeps a set of boxes synchronized, even if this set does not have a fixed size (there may be any number of *get* and *containsKey* boxes). This shared characteristic of keeping a set of boxes synchronized is why we call the Active Map an operation.

3.2 Extension to `Multimap`

A `Multimap` is the generalization of a `Map` to the case where multiple values may be associated to each key. There are several `Multimap` implementations for

[3] Conceptually, the Active Map operation maintains synchronization upon changes occurring on any of its boxes, and is thus multidirectional. This makes it quite versatile. However, depending on how a given Active Map operation is created, some of its boxes may be read-only, but this is implementation-dependent.

Java[4], but none is provided in the standard library. It is similar to a `Map` with collections as values, but is designed to hide its implementation. For instance, it is possible to add new entries without having to write the code that creates collections for new keys. One of the most notable differences when compared to a `Map` is that the `get(key)` method of a `Multimap` returns a collection view instead of a single plain object. As a matter of fact the Active Map operation as defined in the previous section already has `get(key)` boxes. Extending the Active Map operation to handle multiple values per keys is therefore mostly a matter of allowing multiple objects in the `get(key)` boxes. We therefore chose to extend the Active Map operation instead of defining a separate Active Multimap.

Figure 2 shows the boxes kept synchronized by the Active Map operation extended to `Multimap`. The extended Active Map provides two additional boxes: `groupedEntries` that contains a set of key-collection of values mappings, and `keys` that contains the list of all keys, including duplicates when a key is associated to multiple values. Depending on whether the boxes used as `get(key)` boxes are singletons, or collections, the extended Active Map behaves respectively either as a `Map`, or as a `Multimap`. When it behaves as a `Map`, `keys` and `keySet` contain the same set of keys, and the value part of the `groupedEntries` mappings are all collections with a single element.

An Active Map extended to `Multimap`, which we will simply call an Active Map from now on, is consistent if all its boxes have the values a non-active `Multimap` would have. The purpose of the extended Active Map operation can be summarized in the the same way as its non-extended version was: whenever a change occurs on any of its boxes, it must update the other boxes so that it is again consistent.

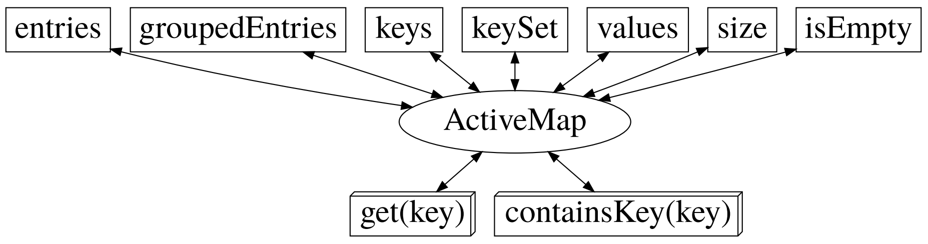

Fig. 2. Views of the Active Map operation extended to Multimap

[4] Guava notably provides a `Multimap` class: `https://google.github.io/guava/releases/23.0/api/docs/com/google/common/collect/Multimap.html`, as does Eclipse Collections: `https://www.eclipse.org/collections/javadoc/9.2.0/org/eclipse/collections/api/multimap/Multimap.html`.

4 Active Map API

The two previous sections defined the Active Map operation in terms of the boxes it keeps synchronized, and in terms of how it should behave. This section presents its API: how its boxes can be retrieved (Section 4.1), and how Active Maps can be created (Section 4.2).

4.1 Accessing boxes

Figure 3 gives an overview of the Active Map API in the form of a class diagram. The `Box` interface represents all mutable values by wrapping them, and is generic over the type of these values. It notably provides all pre-existing operations such as `collect` and `select`, but they are not all detailed here. Note that type $\lambda < E, R >$ denotes the type of anonymous functions (also known as lambda expressions or closures) taking an E as argument, and returning an R. Non-OCL operations such as `zip`, or `zipWith` (not shown on Figure 3), borrowed from functional languages like Haskell, and already discussed in [7,8,9], are also defined. The `zip` operation requires the definition of a `Pair`, which has a left, and right parts, over which types it is generic. This is similar to an OCL tuple, but is represented explicitly here to improve readability of the diagram. `zip` returns a `BoxOfPairs`, which is a specialization of `Box` to the case where its elements are `Pair`s, as denoted by the $E \rightarrow Pair < K, V >$ template binding. Finally, `Box` is extended with specific operations to create Active Maps.

The `ActiveMap` interface represents Active Maps, and is generic over the type of their keys (K), and values (V). It provides one zero-argument operation for each Active Map box: `entries`, `groupedEntries`, `keys`, `keySet`, `values`, `size`, and `isEmpty`, as well as one single-argument operation for each set of boxes: `get(key)`, and `containsKey(key)`. Note that they are no operation to get the `get(key)`, and `containsKey(key)` boxes from mutable keys. They could be integrated to the API, but they are not strictly necessary because they can be expressed using `collect`. For instance, a `getMutable` method taking a box of keys as argument can be defined in the following way by leveraging the fact that `collect` can not only propagate changes occurring in its source collection but also in the boxes its body returns:

```
sourceMap->getMutable(key:Box<K>) = key->collect(e|sourceMap->get(key))
```

Note also that getting the box corresponding to a non-existing mapping (i.e., when `containsKey(key)` is false) is allowed. Such a box will be empty, but may be populated later when changes occur. These empty boxes do not show up in `groupedEntries(key)`, and their existence does not impact any other box until they are populated. This mechanism is useful to be able to use the Active Map operation in more situations. Guava's `Multimap` behaves similarly.

4.2 Creating Active Maps

There are multiple ways to create an Active Map depending on which one of its boxes is initially available. The corresponding operations are provided by interface `Box`, and its specializations `BoxOfPairs`, and `BoxOfMutablePairs`. The latter specializes `Box` to the case where its elements are `Pair`s with a mutable (and therefore wrapped in a box) right part, as denoted by the $E \rightarrow Pair < K, Box < V >>$ template binding.

An Active Map can be created from a box of keys. If one wants a simple `Map`, not a `Multimap`, one either uses the `keysToMap`, or `keysToMapM` operations that both take a lambda expression specifying how to compute a value from a key. `keysToMapM` allows the key corresponding to a value to change (e.g., because it is a changeable property of a model element), whereas with `keysToMap` a given value always has the same key. If one wants a `Multimap`, one uses `keysToMultimapM` which is similar, but with a lambda that always returns a collection box. Because it always returns a box, their is no "immutable" `keysToMultimap` operation.

An Active Map may also be created from a box of values, given a lambda to compute the corresponding immutable keys (one per value) or mutable keys (possibly several per value). Operations `valuesToMultimap`, and `valuesToMultimapM` perform these roles.

An Active Map can also be created from a box of pairs representing its `entries`. The `entriesToMap`, and `entriesToMultimap` operations perform these roles depending on whether one respectively wants a `Map`, or a `Multimap`. In the first case, there should not be multiple pairs with the same key as left part, otherwise a runtime error will occur.

Finally, an Active Map can also be created from a box of "mutable" pairs, each with an immutable key, and a mutable value. There are two overloaded operations named `groupedEntriesToMultimap` for this purpose. The second one takes a lambda as argument that computes keys from values. This makes it possible to have a changeable `values` box.

Remarks: other ways to create Active Maps exist, but we focused on the ones we found useful in practice. Upon creation, an Active Map starts observing the box from which it is created in order to propagate changes to all its views. Which boxes are changeable or not depending on how the Active Map operation is created cannot be discussed here systematically for space reasons. A rule of thumb is: if there is enough information to properly propagate a change occurring on a box to the other boxes, then it is changeable, otherwise changing it results in a runtime error. Active Maps observe all their changeable views.

5 Using Active Maps

The two previous sections defined the Active Map operation and its API. This section shows how it can be used to express various other operations, while preserving scalability. Like in Section 2.1, the syntax used in this section is similar to the one used in the OCL specification [10, Section 11.9, pp. 177–183] with some extensions.

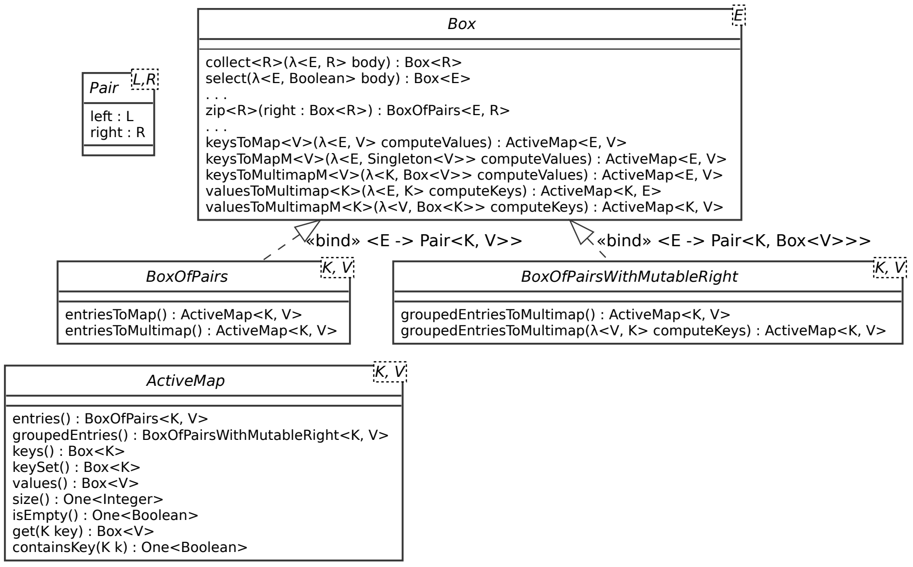
Box
E
collect<R>(λ<E, R> body) : Box<R>
select(λ<E, Boolean> body) : Box<E>
. . .
zip<R>(right : Box<R>) : BoxOfPairs<E, R>
. . .
keysToMap<V>(λ<E, V> computeValues) : ActiveMap<E, V>
keysToMapM<V>(λ<E, Singleton<V>> computeValues) : ActiveMap<E, V>
keysToMultimapM<V>(λ<K, Box<V>> computeValues) : ActiveMap<E, V>
valuesToMultimap<K>(λ<E, K> computeKeys) : ActiveMap<K, E>
valuesToMultimapM<K>(λ<V, Box<K>> computeKeys) : ActiveMap<K, V>
Pair
L,R
left : L
right : R
«bind» <E -> Pair<K, V>>
«bind» <E -> Pair<K, Box<V>>>
BoxOfPairs
K, V
entriesToMap() : ActiveMap<K, V>
entriesToMultimap() : ActiveMap<K, V>
BoxOfPairsWithMutableRight
K, V
groupedEntriesToMultimap() : ActiveMap<K, V>
groupedEntriesToMultimap(λ<V, K> computeKeys) : ActiveMap<K, V>
ActiveMap
K, V
entries() : BoxOfPairs<K, V>
groupedEntries() : BoxOfPairsWithMutableRight<K, V>
keys() : Box<K>
keySet() : Box<K>
values() : Box<V>
size() : One<Integer>
isEmpty() : One<Boolean>
get(K key) : Box<V>
containsKey(K k) : One<Boolean>

Fig. 3. Active Map API

5.1 `groupBy`

The first operation we consider is the one given as motivating example in Section 2.3. The initial expression of `groupBy` given in Listing 1.2 used `collect`, `asSet`, and `select`. It had quadratic computation time, and linear propagation time. The new expression of `groupBy` in terms of the Active Map operation is given below for an immutable body (i.e., always the same grouping key for a given value):

```
source->groupBy(iterator | body) =
  source->valuesToMultimap(iterator | body(iterator)).groupedEntries()
```

Supporting a mutable body (i.e., a body expression returning a mutable value) is simply a matter of switching from `valuesToMultimap` to `valuesToMultimapM`:

```
source->groupByM(iterator | body) =
  source->valuesToMultimapM(iterator | body(iterator)).groupedEntries()
```

Both versions define a `Multimap` with the source collection as values, and the `groupBy` body as key computation lambda argument. The grouped entries of the `Multimap` are then returned, resulting in a collection of pairs with each one's left part equal to a key, and its right part equal to a collection box of all associated values. There is a slight typing difference between these expressions of `groupBy` and the one given earlier in Listing 1.2: we had tuples, and we now have `Pair`s. However, this is mostly a cosmetic issue due to the fact that we decided to add an explicit `Pair` interface to the API class diagram in Figure 3. There is no fundamental difference, and a concrete implementation can be made to return the same kind of tuples in all cases. Remark: the mutable version also supports grouping a given value with multiple keys if its body returns a collection box instead of a singleton box.

5.2 `selectBy`

The second operation we consider is `selectBy`, defined in [8] to be:

```
source->selectBy(searchedKey, iterator | body) =
  source->select(iterator | body(e) = searchedKey)
```

Note that `selectBy` requires two arguments: the key to search (`searchedKey`), and a lambda expression to compute keys from values. No standard OCL operation has such a signature, but this should not prevent us from defining one: it may not even be made available to users if the execution engine automatically optimizes a standard `select` used according to the above pattern into a `selectBy`. If `searchedKey` changes, propagation requires retraversing the whole source collection. Moreover, as noted in [8], this version requires a significant amount of memory due to the fact that each comparison results in a mutable boolean. A more scalable version expressed using the Active Map operation is the following:

```
source->selectBy(searchedKey, iterator | body) =
  source->valuesToMultimap(iterator | body(iterator)).get(searchedKey)
```

Multiple variants exist. Firstly, if the body is mutable, then `valuesToMultimapM` can be used instead of `valuesToMultimap`. Secondly, if `searchedKey` is a mutable value, then `getMutable` can be used instead of `get`. All versions based on the Active Map operation have linear computation time, constant propagation time, and use less memory than the original expression.

5.3 Previous and next elements from `OrderedSet`

Another application example consists in finding the previous or next element in an `OrderedSet`. These can both be expressed using `iterate`, or `zip` plus `select` (expressions not detailed here), but is easier to express, and more scalable using an ActiveMap. We can compute an Active Map for the previous elements using the following expression:

```
source->prevMap() =
  source->zip(source->prepend(null))->entriesToMap()
```

Each element is paired with its previous one by `zip` thanks to the `prepend` of `null` perform on the right box. Given an element `a` of the source `OrderedSet`, getting its previous element consists in calling `get(a)` on the result of `prevMap`.

Similarly, we can compute an Active Map for the next elements using the following expression:

```
source->nextMap() =
  source->prepend(null)->zip(source)->entriesToMap()
```

Remark: both `prevMap`, and `nextMap` rely on a `zip`. For them to work correctly, the change alignment problem discussed in [9] must have been solved.

6 Related work

There are two main categories of related work: those related to integrating `Maps` into OCL, and those about incremental computations. Regarding the first category, QVT [11] defines a `Dict` mutable data structure that behaves like a mutable `Map`. Immutable `Maps` are supported in ATL, and Eclipse OCL[5]. Integrating such `Maps` into the OCL standard has notably been proposed a few years ago [13, slide 23]. However, none of these works consider incremental evaluation of `Maps`, which are always considered as some kind of data structure, and none consider `Multimaps`. Moreover, immutable `Maps` may not actually be much more efficient than other structures like collections of pairs. Finally, factory operations like those described in Section 4.2 are generally not available.

Regarding the second category of related work about incremental computations, IncQuery and VIATRA [12] are based on an entirely different incremental approach built around the Rete [4] algorithm. A subset of OCL can be translated to the kind of graph patterns used by IncQuery and VIATRA [2], but active operations are able to support OCL more extensively. Although it does not necessarily make sense to compare the Active Map operation to Rete, one can observe that in [8], both VIATRA and AOF (with *ad-hoc* `groupBy` and `selectBy` that should scale like the Active Map operation) were shown to scale similarly. Therefore, there is probably already in VIATRA a mechanism playing some of the roles that an Active Map operation can play. Finally, works in databases on the view update problem (e.g., [5]) do support incremental `groupBy` computation. However it does not seem that these works have defined a more general operation similar to the Active Map operation.

7 Conclusion

This paper has presented a new versatile active operation called Active Map. It keeps multiple boxes synchronized in a scalable way thanks to the `HashMap` it wraps. Specific operations such as `groupBy` can therefore be expressed in terms of it without sacrificing scalability, thus reducing the need for *ad-hoc* implementations.

Several aspects of the Active Map operation have not been discussed in the present paper for space reasons. Possible extensions of this work therefore include developing these aspects. For instance, this paper is limited to discussing time complexity without proofs. It may be possible to write such proofs, or at least to benchmark various implementations of operations such as `groupBy` with and without relying on the Active Map operation. Other notable aspects that would benefit from being examined include memory complexity, different kinds of `Multimaps` (depending on the type of `get(key)` boxes such as `Sequence`, or `Set`), order preservation, and further unification by expressing more operations

[5] `https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FMap.html`

in terms of the Active Map operation. Finally, multiple open questions have been asked in Section 2.2. Looking for answers should prove useful.

References

1. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.: Active Operations on Collections. In: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6394, pp. 91–105. Springer (2010)
2. Bergmann, G.: Translating OCL to Graph Patterns. In: Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E. (eds.) Model-Driven Engineering Languages and Systems: 17th International Conference, MODELS 2014, Valencia, Spain, September 28 – October 3, 2014. Proceedings. pp. 670–686. Springer International Publishing, Cham (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. CEUR Workshop Proceedings, vol. 1285, pp. 83–99 (2014)
4. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence **19**(1), 17 – 37 (1982)
5. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. ACM SIGMOD Record **22**(2), 157–166 (1993)
6. IncQuery Labs Ltd.: VIATRA CPS Benchmark: Performance benchmark using the VIATRA CPS demonstrator, `https://github.com/viatra/viatra-cps-benchmark`
7. Jouault, F., Beaudoux, O.: On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling. CEUR Workshop Proceedings, vol. 1512, pp. 35–45. Ottawa, Canada (Sep 2015)
8. Jouault, F., Beaudoux, O.: Efficient OCL-based Incremental Transformations. In: Proceedings of the 16th International Workshop in OCL and Textual Modeling. CEUR Workshop Proceedings, vol. 1756, pp. 121–136. Saint-Malo, France (Oct 2016)
9. Jouault, F., Beaudoux, O., Brun, M., Chhel, F., Clavreul, M.: Improving Incremental and Bidirectional Evaluation with an Explicit Propagation Graph. In: Seidl, M., Zschaler, S. (eds.) Software Technologies: Applications and Foundations. pp. 302–316. Springer International Publishing, Cham (2018)
10. Object Management Group (OMG): Object Constraint Language (OCL), v2.4. `http://www.omg.org/spec/OCL/2.4/` (Feb 2014)
11. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.3. `http://www.omg.org/spec/QVT/1.3/` (Jun 2016)
12. Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z.: Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. Software & Systems Modeling **15**(3), 609–629 (2016)
13. Willink, E.: OCL 2.5 Plans. Presentation given at the 14th International Workshop on OCL and Textual Modelling (2014), `http://software.imdea.org/OCL2014/slides/OCL25Plans`