



Active Operations on Collections

Olivier Beaudoux, Arnaud Blouin, Olivier Barais, Jean-Marc Jézéquel

► **To cite this version:**

Olivier Beaudoux, Arnaud Blouin, Olivier Barais, Jean-Marc Jézéquel. Active Operations on Collections. ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10), 2010, Oslo, Norway, Norway. pp.91–105, 2010. <inria-00542763>

HAL Id: inria-00542763

<https://hal.inria.fr/inria-00542763>

Submitted on 3 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Active Operations on Collections

Olivier Beaudoux¹, Arnaud Blouin², Olivier Barais³, and Jean-Marc Jézéquel³

¹ ESEO Group, Angers - GRI Team `olivier.beaudoux@eseo.fr`

² INRIA Rennes - Triskell Team `arnaud.blouin@inria.fr`

³ University of Rennes 1 - Triskell Team `{barais, jezequel}@irisa.fr`

Abstract. Collections are omnipresent within models: collections of references can represent relations between objects, and collections of values can represent object attributes. Consequently, manipulating models often consists of performing operations on collections. For example, transformations create target collections from given source collections. Similarly, constraint evaluations perform computation on collections. Recent research works focus on making such transformations or constraint evaluations active (*i.e.* incremental, or live). However, they propose their own solutions to the issue by the introduction of specific languages and/or systems. This paper proposes a mathematical formalism, centered on collections and independent of languages and systems, that describes how the implementation of standard operations on collections can be made active. The formalism also introduces a reversed active assignment dedicated to bidirectional operations. A case study illustrates how to use the formalism and its Active Kermeta implementation for creating an active transformation.

1 Introduction

The promise of model-driven engineering (MDE) is that the development and maintenance effort can be reduced by working at the model instead of the code level. Models define what is valuable in a system, and code generators produce the functionality that is common in the application domain. One of the main current issue for the MDE community is to support evolution in the stage of the software development process (*e.g.* to support incremental code generation). To address this issue, this paper works on formalizing operation on collections to support incremental models manipulation. Indeed, collections are omnipresent in the Model Driven Engineering field: collections of references can represent relations between objects, and collections of values can represent object attributes. Consequently, manipulating models often consists of performing operations on collections. Two essential model manipulations are constraint checking and model transformation: the former often uses (OCL) iterators to traverse collections, and the later generates target from source collections. This illustrates the importance of collections in Model Driven Engineering. However, whenever a model is *modified*, these operations are not efficient since they require a full re-execution as if the model was newly created. Many different solutions have

been proposed to solve such an issue; they are all based on the concept of incremental [1], live [2] or active [3] model manipulation. Despite their omnipresence, collections are not the central piece of these approaches that are often tied to a specific language and/or system [4,5], or use usual manipulations combined with merge strategies [6].

This paper proposes a formalism that makes standard operations on collections active, independently from languages and systems. It shows how active transformations can be reduced to active operations on collections, thus underlining the interest of collections as first class objects for model manipulation. It evaluates the approach by studying complexities of active operations, and by explaining how active operations can be written with Active Kermeta, an implementation of our formalism on top of Kermeta [7].

The remainder of this paper is structured as follows. Section 2 explains how standard operations on collections can be made active through active loops, thus showing the foundation of our proposal. Section 3 illustrates the use of the formalism and its Active Kermeta implementation for writing active transformations. Section 4 evaluates complexities of active operations, discusses the possible optimizations, and compares the approach with related works. Finally, section 5 concludes on our contribution and its perspective.

2 From Standard to Active Operations

This section explains the semantics of active operations by describing their active loops: usual binary operations, application, selection, sort, and reversed assignment. Such a set of operations has been mainly inspired by OCL [8].

2.1 Preliminary: Definitions

The set of all collections is noted \mathbb{C} . Its subsets \mathbb{U} and \mathbb{O} define collections that respectively manage *uniqueness* and *order*. The four combinations define usual collection types: order set ($oset = \mathbb{U} \cap \mathbb{O}$), set ($set = \mathbb{U} - \mathbb{O}$), sequence ($seq = \mathbb{O} - \mathbb{U}$), and bag ($bag = \mathbb{C} - \mathbb{U} - \mathbb{O}$).

The following table introduces the minimal set of operations that is sufficient to express any other operation:

Operation	$C \notin \mathbb{O}$	$C \in \mathbb{O}$
$ C $	cardinality of C	
$e \in C$	presence of e in C	
$e \in_i C$	n/a	presence of e in C at position $i \in [0.. C $
$C[i]$	n/a	element in C at position $i \in [0.. C $
$C[i..j]$	n/a	sub-collection of C from pos. i to pos. j
$C + e$	adds e into C appends e at the end of C	
$C +_i e$	n/a	inserts e into C at position $i \in [0.. C $
$C - e$	removes the first occurrence of e from C	
$C -_i$	n/a	removes from C the element at pos. $i \in [0.. C $

If $C \in \mathbb{U}$, operations $C + e$ and $C +_i e$ check the uniqueness of e within C : if $e \in C$ before the insertion, operations have no effect. The following table gives the notation used to iterate on collections:

Iteration	$C \notin \mathbb{O}$	$C \in \mathbb{O}$
$\forall e \in C, p(e)$		calls $p(e)$ for each element e of C
$\forall e \in_i C, p(e, i)$	n/a	calls $p(e, i)$ for each element e at position i in C
$\forall' e \in C, p_a(e)$		calls $p_a(e)$ each time an element e is added into C
$\forall' e \notin C, p_r(e)$		calls $p_r(e)$ each time an element e is removed from C
$\forall' e \in_i C, p_a(e, i)$	n/a	calls $p_a(e, i)$ each time e is inserted at position i
$\forall' e \notin_i C, p_r(e, i)$	n/a	calls $p_r(e, i)$ each time e is removed from position i

The first two rows represent usual iterations throughout loops (symbol \forall): the iteration is performed once for all elements of the collection. The next two rows define an iteration throughout an *active loop* (symbol \forall') that is composed of two *rules*: the *addition rule* ($\forall' \dots \in \dots$) *immediately* invokes procedure p_a for each element e of C (similar to usual loops), and *subsequently* invokes p_a each time a new element e is added into C ; the *removal rule* ($\forall' \dots \notin \dots$) *subsequently* invokes procedure p_r each time element e is removed from C . The last two rows define the *indexed active loop* dedicated to ordered collections; such a loop can also be used in some situations with unordered collections (*e.g.* selection and sort, see sections 2.4 and 2.5). Active loops thus observe additions and removals performed on collections; they also observe replacements since a replacement is considered as a removal+addition pair (see section 4.2). Usual and active loops can use a predicate; for example, $\forall e \in C \mid e \neq 1, p(e)$ calls p for each e different from 1.

Usually operation $B = op(A_1, \dots, A_n)$ computes the resulting collection B from the source collections A_i . The computation is based on a usual loop: changing A_i collections afterward does not change B . Conversely, an *active operation* is based on an active loop, and thus reevaluates B each time an addition or a removal occurs on A_i . One may find that using operator $=$ is ambiguous since it suggests bidirectionality; however, a change on B does *not* affect A_i . For this reason, operator $:=$ is used so that expression $B := op(A_1, \dots, A_n)$ becomes unambiguous.

2.2 Union, Intersection and Difference

Usual binary operations, such as union, intersection and difference, have simple active loops. For example with operation $C := A \cup B$, each time an element e is added into A or B , it is also added into C ; conversely, each time e is removed for A or B , it is also removed from C :

Operation	Order	Active loop	
$C := A \cup B$	$(A, B) \notin \mathbb{O}^2$	$\forall' e \in A, C + e$	$\forall' e \notin A, C - e$
		$\forall' e \in B, C + e$	$\forall' e \notin B, C - e$
	$(A, B) \in \mathbb{O}^2$	$\forall' e \in_i A, C +_i e$	$\forall' e \notin_i A, C -_i$
		$\forall' e \in_i B, C +_{ A +i} e$	$\forall' e \notin_i B, C -_{ A +i}$

Union preserves uniqueness, *i.e.* $(A, B) \in \mathbb{U}^2 \Rightarrow C \in \mathbb{U}$, and order, *i.e.* $(A, B) \in \mathbb{O}^2 \Rightarrow C \in \mathbb{O}$. Active loops for intersection and difference have been defined similarly.

2.3 Application

Application $B := A(f)$ consists of applying f on each element of A . It preserves the order and guarantees $|A| = |B|$; it cannot preserve uniqueness since f may have introduced pairs.

Application allows the definition of navigation *paths*. For example, if elements e of A define property⁴ p , path $B := A.p$ is equivalent to $B := A(e \rightarrow e.p)$. However, the result must be flattened since $A(e \rightarrow e.p)$ returns a collection of properties, *i.e.* a collection of collections. The following table gives the active loops for applications and paths:

Operation	Order	Active loop
$B := A(f)$	$A \notin \mathbb{O}$	$\forall e \in A, B + f(e) \quad \forall e \notin A, B - f(e)$
	$A \in \mathbb{O}$	$\forall e \in_i A, B +_i f(e) \quad \forall e \notin_i A, B -_i$
$B := A.p$	$A \notin \mathbb{O}$	$\forall e \in A,$ $\forall e' \in e.p, B + e' \quad \forall e' \notin A,$ $\forall e' \notin e.p, B - e' \quad \text{stop observation of } e.p$ $\forall e' \in e.p, B - e'$
	$A \in \mathbb{O}$	$\forall e \in_i A,$ $\forall e' \in_j e.p, B +_k e' \quad \forall e' \notin_i A,$ $\forall e' \notin_j e.p, B -_k \quad \text{stop observation of } e.p$ $\forall e' \in_j e.p, B -_k$ where $k = j + \sum_{n=0}^{i-1} e[n].p $

2.4 Selection

Selection $B := A[f]$ consists of selecting elements of A that match predicate f . It preserves uniqueness and order since it filters A . Other operations can be derived from selection, such as operations *reject*, *detect*, *exists* and *forAll* defined by OCL [8], or operation $B := toUnique(A)$ that converts $A \in \mathbb{C}$ into $B \in \mathbb{U}$.

Operation	Order	Active loop
$B := A[f]$	$A \notin \mathbb{O}$	$\forall e \in A f(e), B + e \quad \forall e \notin A f(e), B - e$
	$A \in \mathbb{O}$	$\forall e \in_i A f(e, i), B +_j e \forall e \notin_i A f(e, i), B -_j$ where $j = A[0..i][f] $

As one can note, the previous active loops do not take into account any reevaluation of f required in some situations. For example, selection $persons[p \rightarrow p.age < 18]$ returns a collection of persons under 18. The active loop works fine whenever a person is added or removed from the collection; however, it fails whenever the age of a person goes above 18.

⁴ A property is either a relation or an attribute. As explained in section 3.1, all properties are considered as collections.

Thus, we propose to *reify* (symbol $'$) predicate f into a *predicate collection* represented as a sequence of booleans. Let us consider that collection *persons* contains three people with ages 16, 42 and 12. Expression *persons.age* [$a \rightarrow a < 18$]' returns predicate collection (*true, false, true*) indicating that *persons*[0] and *persons*[2] are below 18. Here we assume that all collections, including unordered ones, store their elements in an array (see section 4.1), thus allowing the use of the indexed accessor $C[i]$ and indexed loops $\forall' e \in_i C$. By *overriding* operation $B := A[f]$ with $B := A[P]$ where $P = A[f]'$, the desired selection can be performed: *persons[persons.age[a \rightarrow a < 18]']*.

Operation	Order	Active loop
$P := A[f]'$		$\forall' e \in_i A, P +_i f(e) \quad \forall' e \notin_i A, P -_i$
$B := A[P]$	$A \notin \emptyset$	$\forall' p \in_i P \mid p, B + A[i] \quad \forall' e \notin_i A \mid e \in B, B - e$ $\forall' p \in_i P \mid \neg p, B - A[i]$
	$A \in \emptyset$	$\forall' p \in_i P \mid p, B +_j A[i] \quad \forall' e \notin_i A \mid e \in_j B, B -_j$ $\forall' p \in_i P \mid \neg p, B -_j$ where $j = \mid P[0..i][p \rightarrow p] \mid$

The addition rule of $B := A[P]$ is based on observing P but not A , which implies that $P := A[f]'$ must be computed *before* $B := A[P]$. The removal rule is based on observing A but not P , which allows retrieving element e of A that must be removed from B . Predicate collections can be combined through usual boolean operators:

Operation	Active loop
$P' := \neg P$	$\forall' p \in_i P, P' +_i \neg p \quad \forall' p \in_i P, P' -_i$
$P := P_1 \wedge P_2$	$\forall' p_2 \in_i P_2, P +_i (P_1[i] \wedge p_2) \quad \forall' p_2 \notin_i P_2, P -_i$
$P := P_1 \vee P_2$	$\forall' p_2 \in_i P_2, P +_i (P_1[i] \vee p_2) \quad \forall' p_2 \notin_i P_2, P -_i$

These active loops are simple. However, it is necessary to decide which collection P_1 or P_2 must be observed for operators \wedge and \vee . By convention, we fix that P_1 is defined *before* P_2 so that a change on P_1 is followed by a change on P_2 ; rules are thus based on observing P_2 , which guaranties that $\mid P_1 \mid = \mid P_2 \mid$ when the rule is called ($P_1[i]$ can thus be used).

2.5 Sort

Sort $B := A\{f\}$ consists of sorting A accordingly to the value returned by f assuming that its type defines operator $<$. The sort operation preserves uniqueness but, naturally, not the order.

Operation	Order	Active loop
$B := A\{f\}$	$A \notin \emptyset$	$\forall' e \in A, B +_j e \quad \forall' e \notin A, B -_j$
	$A \in \emptyset$	$\forall' e \in_i A, B +_j e \mid \forall' e \notin_i A, B -_j$ where $j = \mid A[e' \rightarrow f(e') < f(e)] \mid$

As for selection, the previous active loops do not take into account any reevaluation required if f uses paths on e . For example, sort $persons\{p \rightarrow p.name\}$ returns a collection of persons sorted by their name. This works fine whenever a person is added or removed from the collection but fails whenever the name of a person changes.

Thus, we propose again to *reify* (symbol $'$) function f into an *order collection* represented as a sequence of integers that gives positions after the sort. Let us consider that collection $persons$ contains three persons named “Emma”, “Oliver” and “Alice”. Expression $persons.name\{n \rightarrow n\}'$, abbreviated on $\{persons.name\}'$, returns order collection $(1, 2, 0)$, which means that $persons[0]$ representing “Emma” will occupy position $\{persons.name\}'[0] = 1$ after sorting. The order can then be used for sorting the collection: $persons\{\{persons.name\}'\}$. The following table *overrides* the previous one where $O := \{A\}'$ returns an order collection and $B := A\{O\}$ sorts A according to order O :

Operation	Active loop	
$O := \{A\}'$	$\forall' e \in_i A, O +_i j$ where $j = A[e' \rightarrow e' < e] $	$\forall' e \notin_i A, O -_i$
$B := A\{O\}$	$\forall' j \in_i O, B +_j A[i]$	$\forall' j \notin O, B -_j$

The active loop of $O := \{A\}'$ consists of adding or removing order j at/from position i . However, operation $+$ and $-$ must be refined for order collections to manage resulting positions correctly. For example, $O +_i j$ requires to increment (silently) all orders greater than j .

Moreover, the previous active loops do not allow sorting on multiple criteria. The full version is based on *partial order collections* that specify all *possible* positions; a partial order collection is represented by a sequence of sequences of integers. The previous example can be extended so that the persons are sorted by their last names and then by their first names: $persons\{\{persons.lastName\}' \wedge \{persons.firstName\}'\}$. Let us now consider that collection $persons$ contains three persons named “Emma G.”, “Oliver B.” and “Alice B.”. We now have $\{person.lastName\}'$ that returns $((2), (0, 1), (0, 1))$ and $\{person.firstName\}'$ that returns $((1), (2), (0))$: “Oliver B.” and “Alice B.” have the same possible positions (0 or 1) represented by the two sequences $(0, 1)$, and final positions are given by combining the two partial order collections: $\{persons.lastName\}' \wedge \{persons.firstName\}' = ((2), (1), (0))$.

2.6 Reversed Assignment

Previous operations are unidirectional: in operation $B := op A$, modifying A induces a change on B , but modifying B does not induce any change on A , thus motivating the use of operator $:=$ instead of $=$. Bidirectionality implies that op is reversible, *i.e.* $A := op^{-1} B$, so that a change on B impacts collection A . Union, intersection, difference, selection and sort are not reversible; the only operation that can be reverted is the application: $B := A(f)$ can be reverted as long as f^{-1} exists.

However, application $B := A(f)$ and its reversed version $A := B(f^{-1})$ cannot be defined together since they both create a *new* resulting collection (respectively B and A). We thus introduce the *reversed assignment* operator (symbol $=$) that can only be used on applications: application $B := A(f)$ creates B from A , while its reversed version $B =: A(f)$ (also written $B(f^{-1}) =: A$) allows the reverse *update*. Since A is always initialized before the reversed assignment, its addition rule *must only* be called subsequently to additions. Having $B =: A(f)$ implies that $B := A(f)$: we use operator $=$ so that $B = A(f)$ defines a *bidirectional application*. Active loops for reversed applications are defined as follows:

Operation	Order	Active loop
$B =: A(f)$	$(A, B) \notin \mathbb{O}^2$	$\forall' e \in B, A + f^{-1}(e) \mid \forall' e \notin B, A - f^{-1}(e)$
	$(A, B) \in \mathbb{O}^2$	$\forall' e \in_i B, A +_i f^{-1}(e) \mid \forall' e \notin_i B, A -_i$

Navigation throughout collections is based on the flattening version of the application. In order to preserve the semantics of the active loop of path $B := A.p$ (see section 2.3), the active loop of the reverse path assignment $B =: A.p$ should define the following addition rule: $\forall' e' \in B, A + e_n, e_n.p + e'$ where e_n is the owner element of property p that contains e' . Figure 1 helps in understanding this rule.



Fig. 1. Path principle

Such a definition has no general meaning: if e_n is already contained in A , which one is it (*e.g.* e_1 or e_2 of figure 1)? if not, to what corresponds e_n ? This demonstrates that the path operation is not reversible since the transformation loses the required information due to the flattening. However, if $|A| = 1$ at any time, $e_n = A[0]$ necessarily: in this specific case, operations $B := A.p$ and $B =: A.p$, written $B = A.p$, mean that property p of singleton A equals B at any time. Operation $B = A.p$ is very useful to “bind” a property to another property. Reversed path assignment is defined as follows:

Operation	Order	Active loop
$B =: A.p$ with $ A = 1$	$(A, B) \notin \mathbb{O}^2$	$\forall' e \in B, A[0].p + e \mid \forall' e \notin B, A[0].p - e$
	$(A, B) \in \mathbb{O}^2$	$\forall' e \in_i B, A[0].p +_i e \mid \forall' e \notin_i B, A[0].p -_i$

The following case study includes such a reversed path assignment.

3 Case Study

This section illustrates how previous active operations can be used for implementing active transformations. We first motivate the use of collections for representing any object property. We then give the active operations required for implementing an active transformation in the context of a user interface. We finally explain how the active transformation has been successfully implemented within Kermeta using our *Active Kermeta* framework.

3.1 Requirement: all Properties are Collections

An object property can be either a relation or an attribute, and is *always* represented by a collection. This means that, if the property has a cardinality 0..1 or 1..1, its representing collection is a *singleton*. In such a case, an empty collection represents a null property value.

This requirement is implied by the use of paths that extends the dotted notation of OOP. For example, in expression $B := o.p_1.p_2$ where $|p_1| \leq 1$, $o.p_1$ represents the dotted notation of OOP while $p_1.p_2$ represents a path (see section 2.3). If p_1 is not considered as a singleton but as a value, the property value $o.p_1$ can be *null*, thus resulting in a null reference error within expression $o.p_1.p_2$. This requirement has no real impact on performance since observing a singleton is equivalent to observing changes on a value: in this last case, the value needs to be encapsulated within a dedicated class (e.g. a class *ObservableValue* $\langle T \rangle$).

3.2 Active Transformation

Figure 2 gives the outline of the sample transformation: the left part represents source domain data, a directory of contacts; the right part represents the associated user interface (UI) that displays the contacts within a list widget, and allows editing contact properties throughout three text fields⁵.

Linking domain data to UI is usually performed using “UI bindings” that are platform dependent and offer limited features. Using active operations avoids such drawbacks, and addresses a more general problem than UI binding [9]. A comparison between UI bindings and active operations is however beyond the scope of this paper. A complex example is provided within the Active Kermeta framework.

In the example of figure 2, *D2L* transforms directory d into list l that displays the contacts sorted by their last name and first name:

$$l.items := d.contacts\{\{d.contacts.lastName\}' \wedge \{d.contacts.firstName\}'\}(C2I)$$

C2I transforms each contact c into an item i that displays his/her first and last name, and also saves the link between c and i in reversed relation *contact*:

⁵ UI objects are rendered through a graphical server not represented in the figure. Adding a contact is achieved through the button “Add” of the user interface: clicking on the button creates a new contact in the source data directly..

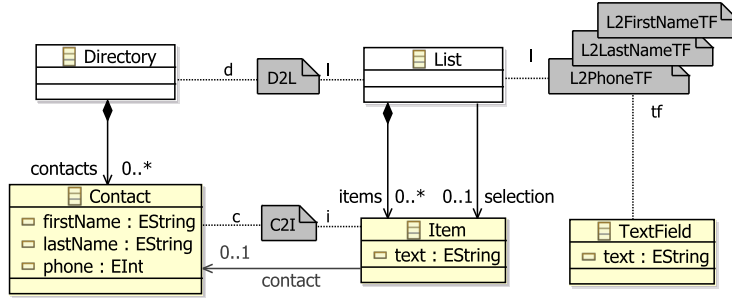


Fig. 2. Transformation outline

$$i.text := c.firstName + " " + c.lastName$$

$$i.contact := c$$

Operation $+$, not presented in this paper, is also active for a *String* singleton (see section 3.3). Relation *contact* allows a *reversible navigation* from the transformation target of the transformation source: $i = C2I(c)$ and $c = C2I^{-1}(i) = i.contact$. Such a relation is called a *trace*.

$L2FirstNameTF$ (respectively $L2LastNameTF$) transforms the first name (respectively the last name) of the selected contact (relation $l.selection.contact$) into text-field tf in a bidirectional way:

$$tf.text = selection.contact.firstName$$

Finally, $L2PhoneTF$ performs the same transformation but adds a bidirectional conversion between the *phone* number (an *Integer*) and the text field content (a *String*):

$$tf.text := selection.contact.phone(IntegerToString)$$

$$tf.text(StringToInteger) =: selection.contact.phone$$

3.3 Kermeta Implementation

Active operations have been implemented on top of Kermeta by the *Active Kermeta* framework, freely available at <http://gri.eseo.fr/software/activekermeta>. The framework proposes two packages dedicated to the four collection classes *Set*, *OrderedSet*, *Bag* and *Sequence*. Package *kermeta::observable* defines the minimal set of operations and active loops for these classes; the following table gives the syntax of Kermeta active loops:

$\forall e \in C, \dots$	$C.eachAdded\{e \dots\}$	$\forall e \in_i C, \dots$	$C.eachAddedAt\{e,i \dots\}$
$\forall e \notin C, \dots$	$C.eachRemoved\{e \dots\}$	$\forall e \notin_i C, \dots$	$C.eachRemovedAt\{e,i \dots\}$

Package *kermeta::active* implements active operations based on the active loops presented in section 2; the following table gives the syntax of Kermeta active operations:

$C := A \cup B$	$C := A.union(B)$	$B := A(f)$	$B := A.collect\{e f(e)\}$
$C := A \cap B$	$C := A.intersection(B)$	$B := A.p$	$B := A.path\{e e.p\}$
$C := A - B$	$C := A.difference(B)$	$B := A.p$	$A.assignPath\{e e.p\}.from(B)$
$P := A[f]'$	$P := A.predicate\{e f(e)\}$	$O := \{A\}'$	$O := A.sortOrder()$
$B := A[P]$	$B := A.select(P)$	$B := A\{O\}$	$B := A.sortedBy(O)$
$P' := \neg P$	$Pbis := P.not()$	$O := O_1 \wedge O_2$	$O := O1.and(O2)$
$P := P_1 \wedge P_2$	$P := P1.and(P2)$	$P := P_1 \vee P_2$	$P := P1.or(P2)$

Writing an active Kermeta transformation respects the same principle as writing a usual Kermeta transformation: aspects are used to add new transformation operations on Ecore models [7]. The transformation of figure 2 has been implemented with active operations as follows:

```

1  aspect class Directory {
2      operation D2L(): List is do
3          result := List.new
4          result.items := contacts.sortedBy(
5              contacts.path{c|c.lastName}.sortOrder().and(
6                  contacts.path{c|c.firstName}.sortOrder())
7              ).collect {c|c.C2L()}
8      end
9  }
10
11 aspect class Contact {
12     operation C2L(): Item is do
13         result := Item.new
14         result.text := firstName.plusValue(" ").plus(lastName)
15         result.contact.add(self)
16     end
17 }
18
19 aspect class List {
20     operation L2PhoneTF(): TextField is do
21         result := TextField.new
22         var contact: Set<Contact> init selection.path{i|i.contact}
23         result.text := contact.path{c|c.phone}.collect(a|a.toString())
24         contact.assignPath{c|c.phone}.from(result.text.collect {t|t.toInteger()})
25     end
26
27     // ...
28 }

```

Operation *plus* (line 15), not described in this paper, is an active operation that concatenates two string singletons; its companion operation *plusValue* concatenates the literal string with the string singleton. Relation *contact* is added through an aspect of class *Item*.

As one may note, such a Kermeta code can be easily generated from the formal specification given in the previous section.

4 Evaluation

This section evaluates the worst case complexities of active loop rules, discusses the resulting performance in the contexts of both constraint evaluation and model transformation, and then compares the approach with related works.

4.1 Worst Case Complexities

The following table gives the *worst* case complexities of elementary operations on collections implemented as *array lists*:

Operation	$C \notin \mathbb{U} \mid C \in \mathbb{U}$
$ C , e \in_i C, C[i]$	$\mathcal{O}(1)$
$C + e$	$\mathcal{O}(1) \mid \mathcal{O}(n)$
$e \in C, C +_i e, C - e, C -_i$	$\mathcal{O}(n)$

Due to the choice of the elementary operations, these complexities cannot be better for linked lists, nor for sorted sets ($C + e$ is even worse). Moreover, using hash sets should only improve the *average*-case complexities (*e.g.* $e \in C$ is $\mathcal{O}(1)$ in average). We can thus infer *worst*-case complexities of active loops from the previous table.

Since active loops of ordered collections differ from those of unordered collections, we must study complexities for each of the four collection types (*bag*, *seq*, *set* and *oset*). Moreover, we distinguish three cases of the active construction of collections: the initialization (“i.”) that invokes the addition rule n times; the addition (“a.”) that invokes the addition rule on each addition performed in the source collection; and the removal (“r.”) that invokes the removal rule on each removal performed in the source collection. The following table synthesizes complexities for each rule of action loops:

Operation	bag		seq		set			oset	
	i.	a./r.	i.	a./r.	i.	a.	r.	i.	a./r.
$A \cup B, A(f), A.p$	n	1	n	n	n	1	n	n	n
$A\{O\}, A[f'], A[P]$	n	1	n	n	n^2 (n)	n (1)	n	n^2 (n)	n
$B =: A.p$	-	1	n	-	-	1	n	-	n
$\neg P, P_1 \wedge P_2, P_1 \vee P_2, O_1 \wedge O_2$	-	-	-	n	n	-	-	-	-
$A \cap B, A - B, \{A\}'$	n^2	n	n	n^2	n	n^2	n	n	n^2

Complexity for an initialization is not necessarily equal to $n \times c$ where c is the complexity of its addition rule. For example, the addition rule of $A \cup B$ where $(A, B) \in seq^2$ is based on operation $C +_i e$ that costs $\mathcal{O}(n)$: the initialization would thus cost $\mathcal{O}(n^2)$; however, operation $C +_i e$ here appends e at the *end* of C , which costs only $\mathcal{O}(1)$: the initialization phase thus costs $\mathcal{O}(n)$.

Operations in the first four rows are mainly $\mathcal{O}(n)$: additions cost $\mathcal{O}(1)$ for unordered collections, but cost $\mathcal{O}(n)$ for ordered ones because of the required shifting; removals always cost $\mathcal{O}(n)$ since they require a search in unordered

collections, or a shifting of ordered collections. Note that the complexity of operation $A.p$ is given considering $|A.p| = n$, but not $|A| = n$. Collections with uniqueness have $\mathcal{O}(n^2)$ operations because $C + e$ must ensure the uniqueness. However, operations on row 2 do not require uniqueness from operation $+$: for example, selection $B := A[f]$ guaranties that $B \in \mathbb{U}$ if $A \in \mathbb{U}$ since it filters A ; these operations can thus reduce their complexities (values surrounded by parenthesis). The last row has $\mathcal{O}(n^2)$ complexities: $A \cap B$ and $A - B$ require presence tests that cost $\mathcal{O}(n)$, and $\{A\}$ naturally requires two loops.

4.2 Discussion

The previous table illustrates that complexities for removals are always $\mathcal{O}(n)$. Consequently, replacing element e_a by element e_b in collection C costs $\mathcal{O}(n)$ since it performs a removal of e_a and an addition of e_b . This might be optimized for ordered collections since replacing one of its element only costs $\mathcal{O}(1)$. This optimization requires the introduction of a *replace rule* within active loops. Moreover, replace rule can be mandatory in some circumstances; for example, replacing the value of a property with cardinality 1..1 violates the minimal cardinality constraint if a removal is performed. The replace rules can be easily inferred from addition and removal rules; they have not been considered in this paper for clarity.

An active transformation that counts m operations constructs its initial result in between $\mathcal{O}(m \times n)$ and $\mathcal{O}(m \times n^2)$, and subsequently updates each *individual* resulting collection in between $\mathcal{O}(1)$ and $\mathcal{O}(n)$. However, operations are not independent: modifying one source collection can result in multiple chained updates. A simplified view of such dependencies consists of representing the operations in a two dimensional space where height h counts the independent chains of operations, and w counts the operations involved in a chain: $\mathcal{O}(m) = \mathcal{O}(h \times w)$. The number of operations Δw required to reify functions involved in selections and sorts (see sections 2.4 and 2.5) is included in number w . For example, single selection $persons[p \rightarrow p.age < 18]$ must be rewritten in active selection $persons[persons.age[a \rightarrow a < 18]']$ that counts $\Delta w = 2$ (path + predicate collection), *i.e.* $w = 3$.

Complexity of the initial construction varies from $\mathcal{O}(h \times w \times n)$ to $\mathcal{O}(h \times w \times n^2)$, and complexity of the subsequent chained updates varies from $\mathcal{O}(w)$ to $\mathcal{O}(w \times n)$: their ratio λ thus varies from $\mathcal{O}(h)$ to $\mathcal{O}(h \times n^2)$. This result illustrates the interest of active operations, especially for large models ($n \gg 1$) and/or complex transformations ($h \gg 1$).

The active loops proposed in section 2 should be implemented in a way that depends on the *context of use*, thus allowing possible optimizations that would increase λ by reducing width w of the dependency chains. In the context of incremental constraint evaluation, Cabot and Teniente [4] propose to take into account *only* changes that can induce constraint violation, thus defining a new specific context of use. Such a specific context requires that our model for observing collections should be refined so that addition and removal rules does not systematically invoke their associated procedures. Section 3 has presented the

specific context of active transformations for user interfaces: the transformation binds the domain objects to their presentation. In such a context, the user works on a presentation that represents a (small) part of the full application model: this typically means that the transformation starts by a *selection* that filters the full model. Such a selection thus naturally “optimizes” the transformation by pre-filtering changes that do not impact the presentation. Moreover, the user can perform many changes on a single object in a short time-slot, thus resulting in many reevaluations of active operations. Once again, the observability model can be refined by using an asynchronous invocation of addition and removal rules, as done within Viatra [10]. This would improve performances by filtering any redundant modifications, such as multiple intermediate changes of a single property (only the last change should be considered).

Since it is centered on operations on collections, our approach is more suited to imperative transformations (*e.g.* Kermeta) than to declarative ones (*e.g.* ATL [11]). However, we think that our formalism can *help* in making declarative transformations active. Moreover, mappings expressed with higher level languages, such as Malan [12], should be automatically converted into active operations.

4.3 Related Work

Many research have been done on incremental evaluation of constraints and incremental transformations. We herein only cite some of the most recent ones.

Blanc *et al.* propose an original approach for detecting model inconsistency (constraint violation): the detection is performed on the model considered as a sequence of elementary construction operations, rather than a model considered as a set of elements [13]. The approach is thus naturally incremental. It shares some similarities with active operations: their elementary construction operations match the elementary rules (addition and removal) of active loops, and our addition rules are also used to initially build the content of collections. However, their approach is dedicated to constraint evaluation only, and the implementation is based on Prolog which is not widely used and not well adapted to MDE.

Cabot et Teniente optimize OCL constraint evaluation by considering only constraint violations: model changes that cannot violate constraints are filtered [4]. They also translate OCL contexts to better contexts. The proposed optimization is interesting and forms a specific context of use, as previously explained. However, users often temporally violate constraints when editing models (*e.g.* a user omits the type of a class attribute within an Ecore diagram): transition from state *violated* to state *respected* should not be ignored. The optimization, specific to constraint checking, cannot be used in the context of incremental transformation.

XSLT is probably the best known transformation language. Villard and Layaïda have developed incXLST, an incremental XSLT processor, thus showing the broader interest of incremental transformation [5]. The processor is based on re-instantiating transformation rules and merging the resulting fragments within the target document, and has limited featured. Framework eXAcT allows

the transformation of DOM documents into DOM presentations (*e.g.* SVG presentations) [14]. However, eXAcT transformations are complex Java programs with limited features. Moreover, both incXSLT and eXAcT are not MDE tools.

QVT has established that incremental transformation is an important issue of MDE [15], but no incremental QVT-based transformation engine has been implemented yet. Xiong *et al.* proposed SyncATL, an incremental ATL processor [6], on the same principle as incXSLT: elicited ATL rules are re-executed and their results are merged with the target. As for incXSLT with XSLT, the processor is dedicated to ATL only. Hearnden *et al.* propose an original approach based on the use of SLD resolution, where SLD trees store the transformation context and dependency tables record dependencies between the transformation and the source model [2]. The drawback of the approach is the maintenance cost of the SLD trees and dependency tables.

The previous works make *declarative* transformations incremental by implementing new processors and/or algorithms tied to specific languages and/or systems. Using active operations on collections allows their direct execution on model instances, without requiring any specific processor or complex algorithm: *definition* of operations are directly *executable* in an active manner. We have shown that active operation can be easily used to implement Kermeta *imperative* transformations. We think that our formalism can *help* in making declarative transformation languages active, such as ATL [11], by generating the active operations for a given “passive” transformation. Some authors considered that declarative transformations should be expressed as mappings [3,1,12]. Here again, we think that active implementations of mappings, as defined by Akehurst [3], can be achieved by active operations.

5 Conclusion and perspective

This paper proposes a formalism, based on active loops, for implementing active operations on collections. The standard set of operations, mainly inspired by OCL [8], is supplemented by a reversed assignment that allows the definition of bidirectional operations. A case study, fully implemented in Kermeta, illustrates that making a transformation active by using such a formalism does not require to change much the usual (*i.e.* passive) transformation; it also gives a specific context that requires active transformations with bidirectionality features: user interfaces. The complexity study shows that running active operations results in an interesting gain when compared to running all the “passive” operations. Moreover, such a gain can be increased by reducing operation dependencies with optimization strategies that can be implemented depending of the contexts of use (*e.g.* transformation within UI or evaluation of constraint violation).

We first plan to create an active implementation of the Malan language [12] based on *Active Kermeta*, thus showing the ability of active operations to implement *declarative* mappings and transformations. We will secondly focus on the use of active operations for incremental constraint validation by extending the proposed set of active operations, and by defining active class invariants through

Kermeta aspects. We thirdly plan to optimize the collection observability model of *Active Kermeta* with filtering and asynchronous treatment capabilities. Finally, we will use active operations in the context of user interfaces to link each of their components [16,17].

References

1. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling* **8**(1) (2008) 21–43
2. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: *MoDELS 2006, LNCS 4199*. (2006) 321–335
3. Akehurst, D.H.: Model Translation: A UML-based specification technique and active implementation approach. PhD thesis, University of Kent (2000)
4. Cabot, J., Teniente, E.: Incremental evaluation of OCL constraints. In: *CAiSE 2006, LNCS 4001*. (2006) 81–95
5. Villard, L., Layaïda, N.: An incremental XSLT transformation processor for XML document manipulation. In: *Proc. of WWW'02, ACM* (2002) 474–485
6. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: *Proc. of ASE'07, ACM* (2007) 164–173
7. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: *Proc. of MoDELS'05, LNCS 3713, Springer* (2005) 264–278
8. Warmer, J.B., Kleppe, A.G.: The object constraint language: getting your models ready for MDA. Addison-Wesley
9. Beaudoux, O., Blouin, A.: Linking data and presentations: from mapping to active transformations. In: *Proc. of DocEng'10 (in press), ACM* (2010)
10. Varró, D., Balogh, A.: The model transformation language of the viatra2 framework. *Sci. Comput. Program.* **68**(3) (2007) 187–207
11. Jouault, F., Kurtev, I.: Transforming models with ATL. In: *Satellite Events at MoDELS 2005, LNCS 3844, Springer* (2006) 128–138
12. Blouin, A., Beaudoux, O., Loiseau, S.: Malan: A mapping language for the data manipulation. In: *Proc. of DocEng'08, ACM* (2008) 66–75
13. Blanc, X., Mounier, I., Mougnot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: *Proc. of ICSE'08, ACM* (2008) 511–520
14. Beaudoux, O.: XML active transformation (eXAcT): transforming documents within interactive systems. In: *Proc. of DocEng'05, ACM* (2005) 146–148
15. OMG: MOF QVT final adopted specification. OMG document, OMG (2005)
16. Blouin, A., Beaudoux, O.: Improving modularity and usability of interactive systems with Malai. In: *Proc. of EICS'10, ACM* (2010) 115–124
17. Beaudoux, O., Beaudouin-Lafon, M.: OpenDPI: A toolkit for developing document-centered environments. In: *Enterprise Information Systems VII, Springer* (2006) 231–239