Lazy Evaluation for OCL

Massimo Tisi[1], Rémi Douence[2], Dennis Wagelaar[3]

[1] AtlanMod team (Inria, Mines Nantes, LINA), Nantes, France
massimo.tisi@mines-nantes.fr
[2] Ascola team (Inria, Mines Nantes, LINA), Nantes, France
remi.douence@mines-nantes.fr
[3] HealthConnect NV, Vilvoorde, Belgium
dennis.wagelaar@healthconnect.be

Abstract. The Object Constraint Language (OCL) is a central component in modeling and transformation languages such as the Unified Modeling Language (UML), the Meta Object Facility (MOF), and Query View Transformation (QVT). OCL is standardized as a strict functional language. In this article, we propose a lazy evaluation strategy for OCL. We argue that a lazy evaluation semantics is beneficial in some model-driven engineering scenarios for: i) lowering evaluation times on very large models; ii) simplifying expressions on models by using infinite data structures (*e.g.*, infinite models); iii) increasing the reusability of OCL libraries. We implement the approach on the ATL virtual machine EMFTVM.

1 Introduction

The Object Constraint Language (OCL) [1] is widely used in model-driven engineering (MDE) for a number of different purposes. For instance, in the Unified Modeling Language (UML), OCL expressions are used to specify: queries, invariants on classes and types in the class model, type invariants for stereotypes, pre- and post-conditions on operations and methods, target (sets) for messages and actions, constraints on operations, derivation rules for attributes. Besides its role in UML, OCL is embedded as expression language within several MDE languages, including metamodeling languages (e.g., the Meta Object Facility, MOF) and transformation languages (e.g., the Query View Transformation language, QVT, and the AtlanMod Transformation Language, ATL [2]).

In the standard specification of the OCL semantics [1], the language is defined as a side-effect-free functional language. While several implementations of the specification exist as a standalone language (e.g., [3]), or as an embedded expression language (e.g., in [2]), they all compute OCL expressions by a strict evaluation strategy, *i.e.*, an expression is evaluated as soon as it is bound to a variable. Conversely, a lazy evaluation strategy, or call-by-need [4] would delay the evaluation of an expression until its value is needed, if ever. In this paper we want to: 1) clarify the motivation for lazy OCL evaluation and capture the main opportunities of application by means of examples; 2) propose a lazy evaluation strategy for OCL by focusing on the specificities of the OCL language

w.r.t. other functional languages; 3) present an implementation of the approach in the ATL virtual machine EMFTVM[4] [5].

The first effect we want to achieve is a performance increase in some scenarios by avoiding needless calculations. Companies that use MDE in their software engineering processes need to handle large amounts of data. In MDE, these data structures would translate into very large models (VLMs), *e.g.*, models made by millions of model elements. Examples of such model sizes appear in a range of domains as shown by industrial cases from literature: AUTOSAR models [6], civil-engineering models [7], product families [8], reverse-engineered software models [9]. A lazy evaluation strategy for a model navigation language like OCL would allow to 1) delay the access to source model elements to the moment in which this access is needed by the application logic and, by consequence, 2) reduce the number of processed model elements, by skipping the unnecessary ones (if any). When the OCL evaluator is embedded in an MDE tool, lazy OCL evaluation may have a significant impact on the global tool performance.

Our second purpose is enabling the use of infinite data structures in the definition of algorithms with OCL. Indeed, infinite data structures make some algorithms simpler to program. For instance, they allow to decouple code in a producer-consumer pattern: a producer function defines data production without caring for the actual quantity of data produced; a consumer function explores the data structure, implicitly driving the production of the necessary amount of data. For instance, it is simpler to lazily generate infinite game trees and then explore them (e.g., by a min-max algorithm), rather than estimating at each move the part of the game tree to generate. In this paper we argue that infinite data structures simplify also the development of common queries in MDE.

Finally our third objective is to use laziness to improve the reusability of OCL libraries, by reducing their dependencies. Indeed, laziness promotes definitions reuse. For instance, the minimum of a collection can be defined as the composition of sorting with selection of the first element. Such a definition reuses code but it can be vey inefficient in a strict evaluation strategy, requiring the full collection sorting. Laziness makes it practical, at least for some sorting algorithms, since only the computation for sorting the first element will be executed. Similarly, composing libraries in a producer-consumer pattern, enables the definition of general (hence reusable) generators that compute many (possibly infinite) results. Consumers specialize generators to the context of use by demanding only part of the generated elements.

The remainder of this paper is organized as follows: Section 2 motivating the need for lazy evaluation in OCL by introducing two running scenarios; Section 3 describes our approach; Section 4 discusses the implementation strategy; Section 5 lists the main related works; Section 6 concludes the paper with a future research plan.

[4] available from `http://wiki.eclipse.org/ATL/EMFTVM`

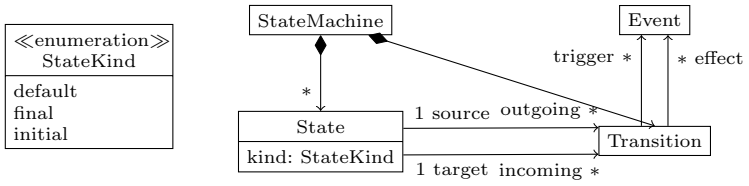

Fig. 1. State Machine metamodel (excerpt)

2 Motivating Examples

This section introduces two examples of OCL queries with the purpose of highlighting the benefits of lazy evaluation in the specific case of model queries.

The state machine in Fig. 2 conforms to the State Machine metamodel displayed in Fig. 1. This metamodel defines a `StateMachine` as composed of several `State` elements. A *kind* property is used to distinguish special states, such as the unique `initial` state and possibly several `final` states. `Transitions` connect couples of `States`. Each transition is triggered by a set of events (`trigger`) and when it fires it produces new events (`effect`).

We provide this state machine with a simple execution semantics. The machine maintains a queue of events to process, that is initially not empty. The execution starts from the initial state and checks the top of the queue for events that match the trigger of some outgoing transition. If such events are found, the transition is fired: the machine moves to the target state of the transition, the triggering events are removed from the top of the queue and the effect events are added to the bottom of the queue. In our simple model the machine proceeds autonomously (no external events are considered) and deterministically (triggers outgoing from the same state are disjoint).

2.1 Queries on Large Models

As a first example scenario we check if there exists a non-final state that contains a self-transition[5]:

[5] The query structure is identical to the one introduced in [9] and used in several works to compare the execution performance of query languages, but here we apply it to state machines instead of class diagrams.

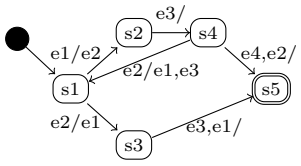

Fig. 2. State machine example (transitions are labeled as *trigger/effect*)

```
State.allInstances()->select(s | not s.kind = 'final')
  ->exists(s | s.outgoing->exists(t | t.target = s))
```

If we assume the number of states of the input state machine to be very large, the time and memory cost to evaluate such query may be high. Here are the steps that a strict evaluation of OCL typically performs:

1. **Computation of the extent of class `State`.** In this first step, the OCL evaluator typically traverses the whole model on which the query is evaluated in order to compute the collection of all elements that have `State` as type (directly, or indirectly via inheritance).
2. **Filtering out final states.** Then, the whole collection computed in previous step is traversed in order to keep only states that are not final.
3. **Finding a state with a self-transition.** Finally, the list of non-final states is traversed in order to discover if one of them satisfies the condition.

Several optimizations may be supported by an OCL evaluator. For instance, with **extent caching**, the result of calling `allInstances()` on a model for a given type (`State` in our example) may be cached. Thus, a second extent computation will not require traversal of the whole model. In our case, this will not reduce the cost of the first evaluation, but will reduce the cost of subsequent evaluations (provided the source model is not modified, which may invalidate our cache, and require a new extent computation). However, with these optimizations alone, even if a non-final state satisfying the condition appears near the beginning of the model, the whole model still needs to be traversed for the first computation of the extent of `State`, and the whole list of states needs to be traversed for each evaluation in order to filter out final states.

Especially when the query is performed as part of an interactive tool, there may be a significant need to reduce the query response time. Moreover, if the queried model is too large to fit in RAM (e.g., it may be stored in a database and traversed lazily using frameworks such as CDO[6]), evaluation of the query will simply fail. In such a case, the computation of the extent of `Class` will force all elements typed by `Class` to be loaded into RAM (at least a proxy per element if not the values of all their properties). However, we do not actually need all such elements to be in memory at the same time.

2.2 Infinite Collections in Model Queries

In the queries of this section we consider also the state machine semantics and in particular event consumption. The following OCL query computes if a final state is reachable from the current state in a given number of steps while consuming all the given events (in this case we say that the state is *valid*):

```
1 context State::isValid (events:Sequence(Event), steps:Integer) : Boolean
2 body :
3   if (steps<0) then false else
4     if (events->isEmpty()) then self.kind = 'final'
```

[6] `http://www.eclipse.org/cdo/`

```
    else self.outgoing->exists(t | events->startsWith(t.trigger)[7]
      and t.target.isValid(events->difference(t.trigger)
        ->union(t.effect)), steps-1);
    endif
  endif;
```

The following query searches for a repeating state in the state machine execution (e.g., to possibly optimize the state machine execution):

```
context State :: repeatingState (events: Sequence(Event)) : State body :
  self.repeatingStateRec (events, Set{});
context State :: repeatingStateRec (events: Sequence(Event),
  visited: Set(State)) : State body :
  if (visited->includes(self)) then self
  else self.outgoing->select(t | events->startsWith(t.trigger))
    ->any().target.repeatingStateRec(events->difference(t.trigger)
      ->union(t.effect), visited->including(self)))
  endif;
```

The logic of the two recursive queries have clear similarities, being both based on a simulation of the state-machine execution. However the simulation logic is embedded in the query definitions, and interleaved with query-specific logic, i.e. validity or repetition checks. Factorizing the logic for state-machine simulation would simplify the definition of the queries, avoid code duplication, and increase code-reusability. We may try to achieve this factorization by writing a `simulate` OCL query that given a set of events returns an execution trace:

```
context State :: simulate (events: Sequence(Event)) :
  Sequence(Tuple(state: State, events: Sequence(Event))) body :
  let tr : Transition = self.outgoing
    ->select(t | events->startsWith(t.trigger))->any() in
  Sequence{Tuple{state=self, events=events}}
    ->union(tr.target.simulate(events->difference(tr.trigger)
        ->union(tr.effect));
```

Reusing the `simulate` function considerably simplifies the definition of the previous queries, that can be re-written as:

```
context State :: isValid(events:Sequence(Event), steps:Integer): Boolean
body : self.simulate(events)->subSequence(1,steps)
    ->exists(tu | tu.state.kind = 'final' and tu.events->isEmpty());
```

```
context State :: repeatingState (events: Sequence(Event)) : Boolean
body : self.simulate(events)->collect(tu | tu.state)->firstRepeating()[8];
```

However the result of `simulate` is in general an infinite sequence of states and the use we describe would be possible only by providing OCL with a lazy semantics.

3 Lazy Evaluation of OCL

3.1 Approach Overview

In general, lazy evaluation consists in delaying computations, detecting when the result of such a delayed computation is needed, and forcing the delayed compu-

[7] `startsWith` is a shortcut for as `self.subSequence(1,argument->size())=argument`

[8] `firstRepeating` is defined as an operation on ordered collections (independent from state machines) finding the first repeating occurrence

tation. In functional languages, there is a single way to define computation: functions. When function (application) is lazy, the language is lazy. Object-oriented languages (with late binding) do not fit well with laziness. Indeed, evaluation of a method call requires to evaluate its receiver in order to lookup the method definition. Overloading also requires to evaluate arguments. Hence, method call in object orientation is essentially strict. For this reason, in OCL we choose to restrict laziness to collections. Our approach relies on iterators which allow us to produce and consume incrementally (lazily) the elements of a collection.

3.2 Laziness and the OCL Specification

One of the main design goals of our approach for lazy OCL is maximizing compatibility with standard (strict) OCL.

We choose not to extend or change the OCL syntax. In particular we avoid introducing language constructs to control if an expression (or data value, function call...) will be eagerly/lazily computed, like `strict/lazy` keywords or explicit lazy data types (e.g., `LazySet`). This enables programmers to directly reuse existing programs and libraries. We also argue that this choice preserves the advantage of declarative languages like OCL, i.e. programmers do not need to worry about how statements are evaluated. As we will see in the next section, keeping laziness completely implicit is indeed a challenge for the lazy evaluation of high-level declarative languages like OCL.

We also do not change the semantics of existing terminating OCL programs: if a query terminates in strict OCL and returns a value, it also terminates in lazy OCL and returns the same value (although it may require less computation to do so). The only exception to this property are queries that during their computation produce an `invalid` value, as we will soon see.

We are not only backward compatible, but some non-terminating OCL queries terminate in lazy OCL. In particular, we allow the definition of infinite collections and the application of OCL collection operations to them, with some restrictions that we discuss in the next section. Queries that make use of infinite collections terminate, as long as only a finite part of the collection is required by the computation. This is a deviation (extension) of the OCL standard, which defines that all collections are finite: potential infinite sets such as `Integer.allInstances()` are `invalid` in the standard.

As we mentioned, the error management mechanism of OCL has a significant impact on the backward compatibility of the lazy semantics. In OCL, errors are represented as `invalid` values that propagate: for instance when `invalid` is added to a collection, the resulting whole collection is `invalid`. In lazy OCL, the value of an element is unknown until it is accessed. So, if an `invalid` element is never accessed, it does not propagate and the prefix of the collection is well defined. This means that strict queries that return `invalid`, may return a different value in lazy semantics.

Moreover OCL provides the programmer with the `oclIsInvalid` function to handle `invalid` values (somehow analogously to catching exceptions in Java). The function returns `true` if its argument is `invalid` and at the same time stops

the propagation of `invalid`, allowing the program to recover from the error and possibly terminate correctly. Hence terminating queries in strict semantics that use `oclIsInvalid` may produce a different valid value than the same queries in lazy semantics.

Summarizing: 1) expressions that return a valid result in strict semantics return the same result in lazy semantics, 2) expressions that return an invalid result or do not terminate in strict semantics may return a valid result in lazy semantics, 3) expressions that use the `oclIsInvalid` function are an exception to (1) and (2), as they are in general not compatible with the lazy semantics. Note that the other special OCL value, `null`, is a valid value that can be owned by collections, hence it does not pose any compatibility problem to the lazy semantics.

3.3 OCL Operations

OCL functions benefit from laziness in a different degree. In Table 1 we list all the OCL operations on collections and Table 2 all the iterators (according to [1]). For each operation, and each kind of collection it can be applied to, we provide two properties that characterize its lazy behavior:

- We add a constraint to the *Restrictions* column to indicate that the operation/iterator may not terminate, or it is simply not well-defined, if its source (context) or argument is an infinite collection. Examples of such cases are: appending an element at the end of an infinite collection, reversing it, calculating its maximum.
- We specify in the *Strictness* column if the operation/iterator always evaluates the totality of the source or argument collection. Simple examples are: sorting the collection, summing it, or generically iterating over it (`iterate`).

The properties in Tables 1 and 2 implicitly categorize OCL operations and iterators w.r.t. laziness: operations that can be lazily applied without restrictions (e.g., `product`), operations that can lazily navigate only some of the arguments (e.g., `src - c` lazily navigates the source/context collection `src` but strictly evaluates the argument `c`) and operations that do not support lazy evaluation (e.g., `iterate`).

For brevity, in the following we illustrate in detail only a subset of the OCL functions. The reader may extend the principles we introduce to analogous functions.

AllInstances. While not being an operation in the context of a collection type, `allInstances` returns a collection, made by the instances of the type in argument, and this collection can be lazily computed. OCL implementations usually perform a depth-first traversal on the model containment tree to find the model instances and populate the result collection, but this traversal order is not defined in the OCL specification. We propose a lazy evaluation semantics for `allInstances` that supports the navigation of infinite models. However, even

Table 1. Laziness for OCL collection operations

Context	Operation	Restrictions	Strictness
Collection	`=/<> (c : Collection)`	`src` and `c` finite	-
Collection	`size ()`	`src` finite	strict on `src`
Collection	`includes/excludes (o : OclAny)`	`src` finite	-
Collection	`includesAll/excludesAll (c : Collection)`	`src` and `c` finite	-
Collection	`isEmpty/notEmpty ()`	-	-
Collection	`max/min/sum ()`	`src` finite	strict on `src`
Set/Bag OrdSet/Sequence	`including (o : OclAny)`	- `src` finite	- -
Collection	`excluding (o : OclAny)`	-	-
Set/Bag OrdSet/Sequence	`union (c : Collection)`	- `src` finite	- -
Collection	`product (c : Collection)`	-	-
Collection	`selectByKind/selectByType (t: OclType)`	-	-
Collection	`asSet/asOrdSet/asSequence/asBag ()`	-	-
Set/Bag OrdSet/Sequence	`flatten (c : Collection)`	- `src` finite	- -
Set/Bag	`intersection (c : Collection)`	-	-
Set	`- (c : Set)`	`c` finite	strict on `c`
Set	`symmetricDifference (c : Set)`	`src` and `c` finite, strict on `src` and `c`	
OrdSet/Sequence	`append (o : OclAny)`	`src` finite	-
OrdSet/Sequence	`prepend (o : OclAny)`	-	-
OrdSet/Sequence	`insertAt (n : Integer, o : OclAny)`	-	-
OrdSet/Sequence	`subOrdSet/subSeq (f : Integer, l : Integer)`	-	-
OrdSet/Sequence	`at (n : Integer)`	-	-
OrdSet/Sequence	`indexOf (o : OclAny)`	-	-
OrdSet/Sequence	`first ()`	-	-
OrdSet/Sequence	`last ()`	`src` finite	-
OrdSet/Sequence	`reverse ()`	`src` finite	-

Table 2. Laziness for OCL collection iterators

Iterator	Restrictions	Strictness
`iterate`	`src` finite	strict on `src`
`any`	-	-
`closure`	`src` finite	strict on `src`
`collect`	-	-
`collectNested`	-	-
`count`	`src` finite	strict on `src`
`exists`	`src` finite	-
`forAll`	`src` finite	-
`isUnique`	`src` finite	-
`one`	`src` finite	-
`reject`	-	-
`select`	-	-
`sortedBy`	`src` finite	strict on `src`

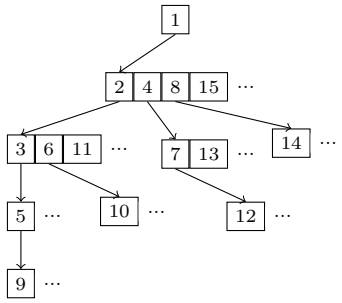

Fig. 3. Fair-first traversal (numbers indicate traversal order)

when models are finite but large, lazy `allInstances` can lead to easier programming and better performance.

Applying `allInstances` to infinite models is not trivial. Depth-first traversal of the containment tree does not work in the case of models with infinite depth: a query like `Type.allInstances()->includes(e)` will not terminate if `e` appears in a rightmost branch w.r.t. an infinite-depth branch. Dually a breadth-first traversal will not work if the model contains a node with infinite children: the traversal will never move to the next tree level. According to our principle of *implicit* laziness, we avoid introducing user-defined model traversals (e.g., `State.allInstancesBreadthFirst()`), that would leave to the user the burden of selecting the correct traversal strategy for `allInstances` depending on the model structure.

Instead, we propose a specific model-traversal order for lazy evaluation of `allInstances`. We still traverse the containment tree with a traversal strategy that alternates at each step a movement in depth and one in width (in an ideally *diagonal* way). Listing 1.1 formalizes the semantics of the traversal in Haskell (function `fairFS`) and Figure 3 graphically illustrates the traversal order.

For instance, applying the example query of Section 2.1 in strict semantics to a state machine with infinite states, the first `allInstances` would never terminate and the following select would never start computing. In our lazy semantics instead, `allInstances` would traverse the infinite model by need, and the full query would actually terminate if a non-final state containing a self-transition was found.

Listing 1.1. Fair traversal for lazy semantics of `allInstances`

```
1  class Tree t where
2      subs :: t a -> [t a]
3
4  data RoseTree a = Node a [RoseTree a] deriving Show
5
6  label :: RoseTree a -> a
7  label (Node l _) = l
8
9  instance Tree RoseTree where
10     subs (Node _ ts) = ts
11
```

```
type Visitor a = a -> [a]

type Brothers t = [t]

nextBrother :: Visitor (Brothers t)
nextBrother [_]    = []
nextBrother (_:ts) = [ts]

nextSon :: Tree t => Visitor (Brothers (t a))
nextSon (t:_) | null (subs t) = []
              | otherwise     = [subs t]

fairFS :: Tree t => Visitor (t a)
fairFS t = ffsIter [[t]]
   where ffsIter (ts:tss) = head ts:ffsIter (tss++nextSon ts++nextBrother ts)
         ffsIter [] = []
```

Union. The `union` operator computes the union of two collections. In lazy OCL each collection is represented as an iterator of elements, hence their `union` is also represented as an iterator of elements.

Four versions of the union, in function of the type of their arguments, are detailed in Listing 1.2. When the collection arguments are `Sequences` the union appends (recursively) the elements of the first collection to the head of the second one. When the arguments are `Bags` the union is a fair interleaving of the two collections. When the arguments are `OrderedSets` the union concatenates the first collection to the second, but elements of the first collection are deleted from the second, to preserve the unicity property. Finally, when the arguments are `Sets`, the union interleaves the two collections while deleting duplicated elements.

The different lazy behavior of the four `union` semantics stands out when they are used with infinite collections. When collections are not ordered no restriction is required, since the interleaving allows to fairly navigate and merge both of the infinite collections. When the collections are ordered if the first argument is infinite the elements of the second arguments will not occur in the infinite result, because in the declarative semantics of OCL the elements of the first collection must occur before the elements of the second collection. In other words, if `c1` is infinite and ordered, than `c1.union(c2)` is equivalent to `c1` for all uses in OCL.

Listing 1.2. Lazy `union`

```
unionSequence (x:xs) ys = x:unionSequence xs ys
unionSequence []     ys = ys

unionOrderedSet (x:xs) ys = x:unionOrderedSet xs (delete x ys)
unionOrderedSet []     ys = ys

unionBag (x:xs) ys = x:unionBag ys xs
unionBag []     ys = ys

unionSet (x:xs) ys = x:unionSet (delete x ys) xs
unionSet []     ys = ys
```

Intersection. The *intersection* operator computes the intersection of two Sets or Bags. Such a computation requires an occurrence check (an element belongs

to the result only if it belongs to both collections), that is in general an operation that is not applicable to infinite sets.

In the lazy execution algorithm we propose (Listing 1.3) both collections are inspected in parallel (note how the arguments are swapped in the recursive call) and the check of occurrence in the other collection is performed with respect to the already considered elements.

Listing 1.3. Lazy `intersection`

```
1 intersect xs ys = intersect' xs [] ys []
2     where intersect' (x:xs) seenInXs ys seenInYs
3               | x `elem` seenInYs = x:intersect' ys (delete x seenInYs) xs
                    seenInXs
4               | otherwise         = intersect' ys seenInYs xs seenInXs
5           intersect' [] _ _ _ = []
```

Table 3 shows an example of execution trace of this algorithm where two infinite integer collections are intersected. In the columns of Table 3 we show, for each step: the part of the collections that is still to evaluate (columns `set1` and `set2`), the elements of the two collections that have already been considered (columns `buffer1` and `buffer2`), the test applied at the current step (column `test`), the result being built (column `set1`∩`set2`).

Table 3. Example of lazy intersection: {powers of 2} ∩ {squares}

set1 (powers of 2)	buffer1	test	buffer2	set2 (squares)	set1∩set2
{1,2,4,8,16,32,64,128,256...}	{}		{}	{1,4,9,16,25,36,49,64...}	{}
{2,4,8,16,32,64,128,256...}	{}	$1 \notin$	{}	{1,4,9,16,25,36,49,64...}	{}
{2,4,8,16,32,64,128,256...}	{1}	$\ni 1$	{}	{4,9,16,25,36,49,64...}	{}
{4,8,16,32,64,128,256...}	{}	$2 \notin$	{}	{4,9,16,25,36,49,64...}	{1}
{4,8,16,32,64,128,256...}	{2}	$\not\ni 4$	{}	{9,16,25,36,49,64...}	{1}
{8,16,32,64,128,256...}	{2}	$4 \in$	{4}	{9,16,25,36,49,64...}	{1}
{8,16,32,64,128,256...}	{2}	$\not\ni 9$	{}	{16,25,36,49,64...}	{1,4}
{16,32,64,128,256...}	{2}	$8 \notin$	{9}	{16,25,36,49,64...}	{1,4}
{16,32,64,128,256...}	{2,8}	$\not\ni 16$	{9}	{25,36,49,64...}	{1,4}
{32,64,128,256...}	{2,8}	$16 \in$	{9,16}	{25,36,49,64...}	{1,4}
{32,64,128,256...}	{2,8}	$\not\ni 25$	{9}	{36,49,64...}	{1,4,16}
{64,128,256...}	{2,8}	$32 \notin$	{9,25}	{36,49,64...}	{1,4,16}
{64,128,256...}	{2,8,32}	$\not\ni 36$	{9,25}	{49,64...}	{1,4,16}
{128,256...}	{2,8,32}	$64 \notin$	{9,25,36}	{49,64...}	{1,4,16}
{128,256...}	{2,8,32,64}	$\not\ni 49$	{9,25,36}	{64...}	{1,4,16}
{256...}	{2,8,32,64}	$128 \notin$	{9,25,36,49}	{64...}	{1,4,16}
{256...}	{2,8,32,64,128}	$\ni 64$	{9,25,36,49}	{...}	{1,4,16}
{...}	{2,8,32,64,128}	$256 \notin$	{9,25,36,49}	{...}	{1,4,16,64}

4 Lazy OCL in ATL/EMFTVM

We have implemented lazy OCL evaluation upon the ATL virtual machine EMFTVM. We compile the underlying OCL expression into imperative byte codes, like INVOKE, ALLINST and ITERATE as explained in [5]. In order to lazily evaluate collections, we implemented the `LazyCollection` type and its subtypes, `LazyList`, `LazySet`, `LazyBag`, and `LazyOrderedSet` corresponding to

four collection types of OCL (`Sequence`, `Set`, `Bag` and `OrderedSet`). An iterator such as `select` or `collect` does not immediately iterate over its source collection, but rather returns a lazy collection to its parent expression that keeps a reference to the source collection, and to the body of the iterator. This is possible because EMFTVM supports closures (also known as lambda-expressions). Then, when a collection returned by an iterator is traversed, it only executes the body of the iterator on the source elements as required by the parent expression.

Listing 1.4 for instance shows the relevant code excerpts for implementing the `collect` operation for `Bags`. A `LazyBag` class extends `LazyCollection` and defines methods for each operation on `Bags`, e.g. `collect()`. In the strict version the `collect()` method would contain the code for computing the resulting collection (i.e., applying the argument function to each element of the source collection). In our lazy implementation the method just returns another `LazyBag`. A `LazyBag` is constructed by passing an `Iterable` as the data source of the collection. In the case of `collect` the `Iterable` is built around a `CollectIterator` (from `LazyCollection`), and the `collect` logic is embedded in the two methods `next()` and `hasNext()` of the iterator. In the `CollectIterator` the `next()` method executes a `function CodeBlock`, representing the lamba-expression associated with it.

Listing 1.4. LazyCollection

```
1  public class LazyBag<E> extends LazyCollection<E> {
2      // ...
3      /**
4       * Collects the return values of <code>function</code> for
5       * each of the elements of this collection.
6       * @param function the return value function
7       * @return a new lazy bag with the <code>function</code> return values.
8       * @param <T> the element type
9       */
10     public <T> LazyBag<T> collect(final CodeBlock function) {
11         // ...
12         return new LazyBag<T>(new Iterable<T>() {
13             public Iterator<T> iterator() {
14                 return new CollectIterator<T>(inner, function, parentFrame);
15             }
16         });
17     }
18     // ...
19 }
20 public abstract class LazyCollection<E> implements Collection<E> {
21     //...
22     public static class CollectIterator<T> extends ReadOnlyIterator<T> {
23 
24         protected final Iterator<?> inner;
25         protected final CodeBlock function;
26         protected final StackFrame parentFrame;
27 
28         /**
29          * Creates a {@link CollectIterator} with <code>condition</code> on <
                code>inner</code>.
30          * @param inner the underlying collection
31          * @param function the value function
32          * @param parentFrame the parent stack frame context
33          */
34         public CollectIterator(final Iterable<?> inner, final CodeBlock
                function, final StackFrame parentFrame) {
35             super();
```

```
36              this.inner = inner.iterator();
37              this.function = function;
38              this.parentFrame = parentFrame;
39          }
40
41          public boolean hasNext() {
42              return inner.hasNext();
43          }
44
45          public T next() {
46              return (T) function.execute(parentFrame.getSubFrame(function,
                    inner.next()));
47          }
48      }
49      // ...
50  }
```

In EMFTVM, `allInstances()` returns a lazy list that traverses the source model lazily, as illustrated in Listing 1.5. The method `allInstancesOf()` in the class `ModelImpl` is executed at each call to OCL `allInstances`. The method returns a `LazyList` whose data source is a `ResourceIterable`. `ResourceIterable` contains a `DiagonalResourceIterator` that implements in its `next()` method the fair tree traversal strategy specified in Listing 1.1[9].

Listing 1.5. allInstances

```
1  public class ModelImpl extends EObjectImpl implements Model {
2      // ...
3      public LazyList<EObject> allInstancesOf(final EClass type) {
4          return new LazyList(new ResourceIterable(getResource()), type));
5      }
6      // ...
7  }
8  public class ResourceIterable implements Iterable<EObject> {
9      // ...
10     public Iterator<EObject> iterator() {
11         // the DiagonalResourceIterator implements the fair tree traversal
12         return new DiagonalResourceIterator<EObject>(this, false)
13     }
14     // ...
15 }
```

Our implementation allows to define and use lazy queries on very large or infinite models, including the examples of Section 2. We have not performed a systematic performance experimentation and time execution performance of the lazy implementation clearly depends on the ratio of the large collections that is actually visited by the query. When performance is the main concern, lazy semantics has to be preferred if a small part of collections is used; strict semantics is still faster in other cases because of the lower overhead.

As an example, we perform the OCL query from Section 2.1 in a strict way with the classic (strict) ATL virtual machine and in a lazy way with the lazy

[9] Note that the current implementation of `allInstances()` in standard ATL returns a `Sequence` of elements in depth-first order, instead of a `Set`. This deviation from the OCL standard may improve the engine performance (by avoiding occurrence checks). The drawback is that the traversal order is exposed to the user, that can consider it in its transformation. In such cases our change in traversal order may break backward-compatibility.

EMFTVM on an Intel core i7, 2.70GHz x 8, x86_64 CPU with 8GiB of RAM. We provide a large state machine made of 38414 elements, where the first state satisfies the query condition. Then, we compare results returned from the two OCL evaluation methods and summarize them in Table 4. The column `Calls` presents the number of operation calls on elements of the underlying collection, i.e., iterations over the `->select()` and the `->exists()`. As shown in Table 4, the lazy evaluator stops the iteration on both `->select()` and `->exists()` as soon as the condition is satisfied (i.e., for the first state), resulting in a much faster execution.

5 Related Work

Lazy evaluation of functional languages is a subject with a long tradition [4], yet it is still studied [10]. We refer the reader to [11] for an example based on Lisp, and to [12] for its formal treatment. Lazy evaluation can be mixed with strict one [13][14]. Hughes has argued that laziness makes programs more reusable [15]. Our approach based on lazy iterators is a simplified version of *iteratees* [16]. Indeed, iteratees are composable abstractions for incrementally processing of sequences. However, our iterators do not isolate effects with a monad, nor distinguish producers, consumers and transducers. Moreover, in our iterators either there is a next value or the iteration is over, but we do not consider raising errors.

The idea of defining and using infinite models has been already addressed in previous work. In [17] transformation rules are lazily executed, producing a target model that can be in principle infinite. In [18] the authors extend MOF to support infinite multiplicity and study co-recursion over infinite model structures. Both works do not provide the query language with an explicit support of infinity. Streaming models can be considered a special kind of infinite models, and their transformation has been recently studied in [19] with languages like IncQuery, but the focus is more on incrementality than laziness.

As alternatives to laziness, other improvements to OCL evaluation have been explored in several works. In [20] the OCL execution engine has been optimized "locally" (i.e., by changing code generated for a given construct). With laziness, we perform only the necessary iterations in many more cases. However,

Table 4. Lazy vs. Strict OCL evaluation in ATL.

Query	**Model Size**	**Lazy Eval.**		**Strict Eval.**	
		Calls	Time	Calls	Time
Example 1	38414	2	0.002 s	38412	0.200 s
`State.allInstances()->select(s | not s.kind = 'final')`		1		25608	
`->exists(s | s.outgoing->exists(t | t.target = s))))`		1		12804	

from a performance point of view, laziness overhead should also be considered. The paper in [21] proposes a mathematical formalism that describes how the implementation of standard operations on collections can be made active. In that way they could evaluate the worst case complexities of active loop rules on collections with a case study. The work in [22] reports on the experience developing an evaluator in Java for efficient OCL evaluation. They aim to cope the novel usages of the language and to improve the efficiency of the evaluator on medium-large scenarios. [23] proposes to extend the OCL evaluator to support immutable collections. Finally, an issue tightly coupled to lazy navigation, is on-demand physical access to the source model elements, i.e. lazy loading. For lazy loading of models for transformation we refer the reader to [24].

6 Conclusions

In this paper we argue that a lazy evaluation semantics for OCL expressions would increase the performance of OCL evaluators in some scenarios, simplify the definition of some queries and foster the development of more reusable OCL libraries in a producer-consumer pattern. We illustrates by example the main challenges of lazy OCL, we provide novel lazy algorithms for some OCL operations (i.e., `allInstances` and `intersection`) and perform an implementation of the approach in the ATL virtual machine EMFTVM.

In future work we plan to perform an extensive performance evaluation on a corpus of real-world OCL queries used in ATL transformation projects. From this study we plan to derive a systematic approach for identifying queries that benefit from lazy evaluation.

Acknowledgements. This work was partially funded by the AutoMobile EU 7th FP SME Research project.

References

1. OMG: Object Constraint Language Specification, version 2.4. Object Management Group. (February 2014)
2. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Sci. Comput. Program. **72**(1-2) (2008) 31–39
3. Eclipse Model Development Tools Project: Eclipse OCL website `http://www.eclipse.org/modeling/mdt/?project=ocl`.
4. Wadsworth, C.P.: Semantics And Pragmatics Of The Lambda-Calculus. PhD thesis, University of Oxford (1971)
5. Wagelaar, D., Tisi, M., Cabot, J., Jouault, F.: Towards a general composition semantics for rule-based model transformation. In: MoDELS. (2011) 623–637
6. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental evaluation of model queries over emf models. In: Model Driven Engineering Languages and Systems. Springer (2010) 76–90
7. Steel, J., Drogemuller, R., Toth, B.: Model interoperability in building information modelling. Software & Systems Modeling **11**(1) (2012) 99–109

8. Pohjonen, R., Tolvanen, J.P., Consulting, M.: Automated production of family members: Lessons learned. In: Proceedings of the Second International Workshop on Product Line Engineering-The Early Steps: Planning, Modeling, and Managing (PLEES'02), Citeseer (2002) 49–57
9. Sottet, J.S., Jouault, F.: Program Comprehension. GraBaTs 2009 Case Study, `http://www.emn.fr/z-info/atlanmod/index.php/GraBaTs_2009_Case_Study`
10. Chang, S., Felleisen, M.: Profiling for laziness. SIGPLAN Not. **49**(1) (January 2014) 349–360
11. Henderson, P., Morris, Jr., J.H.: A lazy evaluator. In: Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages. POPL '76, ACM (1976) 95–103
12. Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A call-by-need lambda calculus. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '95, New York, NY, USA, ACM (1995) 233–246
13. Wadler, P., Taha, W., MacQueen, D.: How to add laziness to a strict language, without even being odd. In: Workshop on Standard ML. (1998)
14. Mauny, M.: Integrating lazy evaluation in strict ML. Research Report RT-0137 (1992)
15. Hughes, J.: Why Functional Programming Matters. Computer Journal **32**(2) (1989) 98–107
16. Kiselyov, O., Peyton-Jones, S., Sabry, A.: Lazy v. yield: Incremental, linear pretty-printing. In Jhala, R., Igarashi, A., eds.: Programming Languages and Systems. Volume 7705 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 190–206
17. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy execution of model-to-model transformations. In: MoDELS. (2011) 32–46
18. Combemale, B., Thirioux, X., Baudry, B.: Formally defining and iterating infinite models. In France, R., Kazmeier, J., Breu, R., Atkinson, C., eds.: Model Driven Engineering Languages and Systems. Volume 7590 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 119–133
19. Dávid, I., Ráth, I., Varró, D.: Streaming model transformations by complex event processing. In Dingel, J., Schulte, W., Ramos, I., Abrahão, S., Insfran, E., eds.: Model-Driven Engineering Languages and Systems. Volume 8767 of Lecture Notes in Computer Science. Springer International Publishing (2014) 68–83
20. Cuadrado, J.S., Jouault, F., Molina, J.G., Bézivin, J.: Optimization patterns for ocl-based model transformations. In: Models in Software Engineering. Springer (2009) 273–284
21. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active operations on collections. In: MoDELS. Volume 6394 of LNCS., Springer (2010) 91–105
22. Clavel, M., Egea, M., de Dios, M.A.G.: Building an efficient component for OCL evaluation. ECEASST **15** (2008)
23. Cuadrado, J.S.: A proposal to improve performance of atl collections. MtATL2010 (2010)
24. Jouault, F., Sottet, J.: An AmmA/ATL Solution for the GraBaTs 2009 Reverse Engineering Case Study. In: 5th International Workshop on Graph-Based Tools, Grabats. (2009)