Efficient OCL-based Incremental Transformations

Frédéric Jouault and Olivier Beaudoux

Groupe ESEO, Angers, France
`firstname.lastname@eseo.fr`

Abstract. Active operations have recently been shown to be a possible back-end to implement OCL-based incremental model transformations. However, the scalability of this approach had not been evaluated yet. This paper presents work done to address this issue by leveraging the VIATRA CPS Benchmark. We show that our implementation of the benchmark transformation using the Active Operation Framework scales similarly to VIATRA. Furthermore, it presents the following advantages: it is able to preserve collection order, and is applicable to OCL-based approaches such as QVT and ATL.

Keywords: model transformation, incrementality, scalability, active operations

1 Introduction

Incremental model transformations are becoming more and more widespread. Several reasons make them necessary in different contexts. For instance, re-executing whole transformations to update target models on each source model change may not be possible for performance reasons. Another requirement may be the preservation of target model elements identity: creating new elements is not always the same as updating existing ones (e.g., if the target model is observed by a third party such as a graphical editor).

However, incremental model transformations are significantly more difficult to implement than regular ones. In general, they only become cost-efficient to develop when they are based on a good theory, and a reusable and robust implementation, which we will call *incremental framework* in this paper. VIATRA [17] is an example of such an incremental framework, providing incremental execution, while scaling to large models. No similarly mature tool exists for the scalable and incremental execution of OCL-based model transformation approaches such as QVT[1] or ATL [11].

In a previous work [10], we have already shown how active operations [1] can be used to incrementally evaluate OCL expressions. We also showed that this approach can be applied to model transformation [2], and have provided a robust implementation called *Active Operation Framework* (AOF). However, the

[1] `http://www.omg.org/spec/QVT/`

scalability of this OCL-compatible incremental framework had not been asserted before the present work.

In order to fill this gap, we leveraged the VIATRA CPS Benchmark [8]. The first step was to implement the specified CPS to Deployment model-to-model transformation using AOF. Then, running the performance measurements helped discover which parts needed optimization.

After implementing all the necessary optimizations to the framework, and to the transformation itself, our AOF-based transformation scales similarly to the VIATRA-based solutions. However, relying on active operations provides additional benefits over VIATRA: it enables 1) preservation of collection order, and 2) the use of OCL expressions.

Section 2 briefly presents the benchmark, as well as AOF. Implementation of the benchmark in AOF is described in Section 3 along with some performance results. Then, Section 4 explains how we achieved scalability, and Section 5 discusses the advantages of using AOF over VIATRA. Some related works are presented in Section 6, and finally, Section 7 gives some concluding remarks.

2 Context

2.1 CPS to Deployment Benchmark

The VIATRA CPS demonstrator[2] is a model-driven engineering (MDE) scenario going from a high-level Cyber-physical System (CPS) metamodel to Java code, via an intermediate Deployment metamodel. It consists of several steps, notably a model-to-model transformation from CPS to Deployment, and a model-to-text transformation from Deployment to Java code. In this paper, we only consider the CPS to Deployment model-to-model transformation. Multiple implementations of this transformation are provided: from simple batch-only to complex incremental ones.

Its source and target models[3], as well as its specification[4] are fully described in the VIATRA documentation. All models are handled using the Eclipse Modeling Framework (EMF).

The VIATRA CPS benchmark [8] is an extension of this demonstrator for the purpose of MDE tool performance evaluation. It provides:

1. **A CPS model generator**, which can generate models with different statistical distribution of elements and connections between them. Each of these possible distributions is called a *case*. Furthermore, it can generate these models at various scales (i.e., with varying number of elements and connections).

[2] Used as motivating example in [17], and fully described in the VIATRA documentation: `https://github.com/viatra/viatra-docs/blob/master/cps/Home.adoc`

[3] `https://github.com/viatra/viatra-docs/blob/master/cps/Domains.adoc`

[4] `https://github.com/viatra/viatra-docs/blob/master/cps/CPS-to-Deployment-Transformation.adoc`

2. **An execution machinery** to launch the different transformation implementations.
3. **Performance reporting** tools to generate visualizations.

This benchmark is based on the MONDO-SAM framework [9].

2.2 Active Operations Framework

Active operations [1] are an approach for the incremental evaluation of OCL-like operations on collections. Over the years, we worked on several prototype implementations. The latest implementation is called Active Operations Framework (or AOF). It is designed as an incremental framework for the development of model transformations, and has been developed while aiming for robustness.

AOF represents mutable values by *boxes* implementing the IBox<E> interface. A box is either a collection (Bag, OrderedSet, Sequence, or Set) or a singleton value (One, or Option). The types of collections are equivalent to those available in OCL, which does not have equivalent for singleton boxes. One (respectively Option) is necessary to represent mandatory non-nullable (resp. optional nullable) mutable values. For instance, a Java int value is not mutable, and its standard object-based representation Integer is also immutable. A IOne<Integer> wraps an Integer object, and may be mutated by replacing the wrapped object by another one. All boxes, collections and singletons, notify observers upon change.

Beside boxes, the second central concept in AOF is *operation.* An operation produces a target box from a source box. Each operation is furthermore able to propagate changes occurring on its source box into corresponding changes on its target box. AOF operations are also generally (often with additional parameters such as a reverse lambda for collect) capable of propagating changes occurring on their target box into corresponding changes on their source box. This enables bidirectional incremental evaluation. AOF provides incremental algorithms for elementary OCL operations such as select, collect, union, size, isEmpty, notEmpty.

Every OCL expression can be represented by a graph with boxes as nodes, and operations as oriented edges. Each box node represents an intermediate value of the expression. Each operation edge represents how its target is generated and can be updated from its source.

AOF is available in the Papyrus git repository[5], where it is notably used for experimental diagram synchronization. It provides a Java 6 API. Many of the methods it provides (e.g., select, collect) take functions as arguments. Although AOF is compatible with Java 6 and Java 7, more conciseness can be achieved using the lambda expressions of either Java 8 or Xtend[6].

[5] `https://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/tree/extraplugins/aof?h=committers/fnoyrit/aofacade`
[6] `http://www.eclipse.org/xtend/`

The main potential scalability issue is the cost of maintaining enough information for incremental algorithms in terms of computation time, and required memory.

3 Overview of Benchmark Implementation with AOF

There are multiple possible implementations of a given transformation with AOF. In this section, we detail the model traversal strategy we selected in Section 3.1, and the concrete syntax used to write expressions in Section 3.2. Finally, we present some performance and scalability results in Section 3.3. It should be noted that although AOF supports bidirectional incremental evaluation of OCL expressions, this is neither required nor evaluated by the VIATRA CPS benchmark. Therefore we have not tried to achieve bidirectionality.

3.1 Source Model Traversal Strategy

Transformation rules may be called explicitly[7], or implicitly[8], by relying on automatic trace link resolution. Explicit rule call is more complicated when rule selection can happen at multiple call sites. Conversely, implicit rule call can be more costly because of rule selection mechanisms. However, the CPS to Deployment transformation does not require rule selection because there is a one-to-one correspondence between metamodel classes and rules. Therefore, we decided to use explicit rule call, in order to ensure better performance.

Thus, the AOF-based transformation is similar to how it would be written with explicit unique lazy rule in ATL.

3.2 Expressions Concrete Syntax

As mentioned earlier, using Java 8 or Xtend enables much more concise lambda syntax than earlier version of Java. Furthermore, we preferred Xtend over Java because it is already used in the benchmark code, and it enables even more concise and OCL-like syntax. Consider, for instance, Listing 1.1, which shows how Xtend enables writing the bindings of a rule.

Listing 1.1. Bindings of rule *hostInstance2DeploymentHost* in Xtend

```
target._ip <=> source._nodeIp

target._applications <=>
  source._applications.collectTo(
    applicationInstance2DeploymentApplication
  )
```

[7] An example of explicit rule call is ATL lazy rules.

[8] Implicit rule call is notably the default for ATL rules.

The first binding (line 1) binds the value of the nodeIp feature of the source element, to the ip feature of the target element (both being strings). The underscore prefixing the _ip and _nodeIp is there to avoid ambiguity with EMF accessor methods. target.ip would correspond to target.getIp(), which would return an immutable String, whereas target._ip calls a helper method which returns a mutable IBox<String>. The spaceship operator (<=>) has been overloaded to specify a call to the `bind` operation. Its visual symmetry indicates that binding is a bidirectional operation in AOF, although this is not required here. The second binding (lines 3-6) binds source applications transformed by rule *applicationInstance2DeploymentApplication* to target applications. The collectTo operation is equivalent to collect in OCL, but additionally maintains source-to-target traceability links so that it can return the same target element for a given pair (source element, rule). In order to achieve such a concise syntax, we used metamodel-specific helpers, which can be automatically generated from the source and target metamodels.

In Java, the same code would look like Listing 1.2. Here, properties of source and target elements are computed by static methods provided by the DEP and CPS classes. These static methods are metamodel helpers that leverage Xtend extension method syntax [9], and property access syntax for accessor method invocation [10]. This is why the Xtend syntax from Listing 1.1 is more concise.

Listing 1.2. Bindings of rule *hostInstance2DeploymentHost* in Java using metamodel helpers

```
DEP._ip(target).bind(CPS._nodeIp(source));

DEP._applications(target).bind(
  CPS._applications(source).collectTo(
    applicationInstance2DeploymentApplication
  )
);
```

Without these metamodel helpers, the first binding would look like the code of Listing 1.3. An additional issue with this syntax, beyond its verbosity, is that the `createPropertyBox` method has no way to know the type of the property it is given since it is not provided by EMF EStructuralFeatures at the type system level with generics. Conversely, the metamodel helpers used in Listing 1.1 enable static type checking by the Xtend compiler.

Listing 1.3. First binding of rule *hostInstance2DeploymentHost* in Java without metamodel helpers

```
factory.createPropertyBox(
  target,
```

[9] https://eclipse.org/xtend/documentation/202_xtend_classes_members.html#extension-methods

[10] https://eclipse.org/xtend/documentation/203_xtend_expressions.html#property-access

```
    DeploymentPackage.eINSTANCE.getDeploymentHost_Ip()
).bind(
    factory.createPropertyBox(
        source,
        CyberPhysicalSystemPackage.eINSTANCE
                .getHostInstance_NodeIp()
    )
);
```

3.3 Scalability Results Overview

We used the benchmark machinery to measure execution time, and memory usage of all implementations of the transformation on each case, and at all scales[11]. The benchmark was executed on a HP Z620 Workstation with 96 gigabytes of RAM. There is not enough place here to present all results, therefore we present a representative sample: the publish-subscribe case.

Figure 1 shows the execution time results obtained for AOF compared to the different VIATRA-based incremental[12] approaches[13] on a log-log scale. As can be seen, AOF scales as well as VIATRA: the slopes of the different curves are roughly equal at high scales. AOF even appears to be slightly faster, but additional measurements on a separate machine show that it depends on the actual hardware. The HP Z620 Workstation has faster memory, and more cores[14] than the other test machine. Therefore, we suppose the difference in performance is due to the fact that AOF is slightly more demanding on memory allocation and collection.

Figure 2 shows the memory usage results obtained for AOF compared to the different VIATRA-based incremental approaches, as reported by the benchmark (for the Transformation phase at iteration 6). Again, AOF appears to scale similarly to Viatra. However, these memory results are not as reliable as the execution time ones (for instance, we have no explanation for the negative slope of two VIATRA solutions). Measuring memory usage is particularly difficult. Notably, taking a single measurement at a specific time does not show

[11] A mapping from benchmark-defined scale to number of elements and links is given at `https://github.com/viatra/viatra-cps-benchmark/wiki/Performance-evaluation#publish-subscribe-scenario-1`.

[12] Results for batch implementations are not shown but are already compared with incremental VIATRA at `https://github.com/viatra/viatra-cps-benchmark/wiki/Performance-evaluation`.

[13] INCR_VIATRA_AGGRE corresponds to *Partial batch transformation*, INCR_VIATRA_EXPLICI to *Explicit traceability*, INCR_VIATRA_QUERY to *Query result traceability*, and INCR_VIATRA to *VIATRA EMF-based tranformation API*, which are described at `https://github.com/viatra/viatra-docs/blob/master/cps/Alternative-transformation-methods.adoc#incremental`.

[14] Although the transformations are all single-threaded, the number of cores is relevant for the parallel garbage collector of the Java virtual machine.

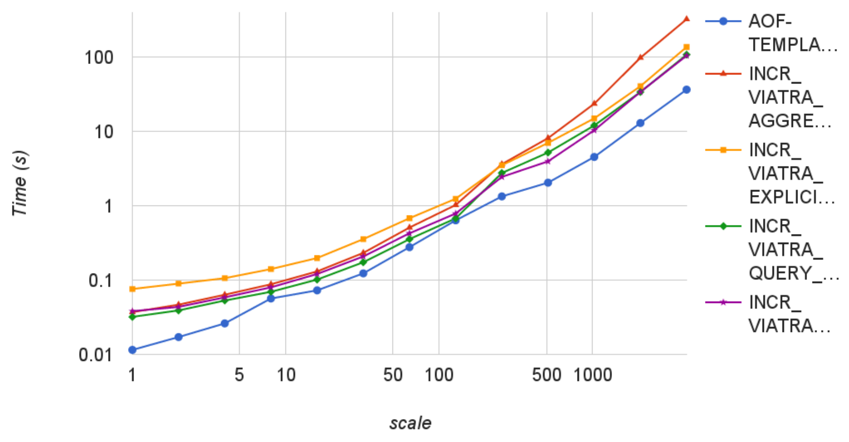

Fig. 1. Compared transformation execution time

peak memory usage. That being said, we observed that the maximum scale at which transformations were executable within available memory was roughly the same for all transformations. This shows that peak required memory is roughly equivalent.

4 Optimization

Three kinds of optimizations have been necessary in order to achieve the scalability results presented above: AOF optimizations (Section 4.1), transformation writing methodology (Section 4.2), and specific operations (Section 4.3). Finally, more optimization perspectives are listed in Section 4.4.

4.1 AOF Optimizations

Initial development of AOF focused on correctness (with a significant amount of unit testing). We consider this effort as fruitful since only a couple of bugs have been discovered (and fixed) in AOF since then. However, we did not take performance into account at that time. Implementing the VIATRA CPS benchmark significantly helped trigger performance bottlenecks, which we located with the CPU profiler of the visualvm tool provided by the Java Development Kit.

The main scalability issue in AOF was due to sanity checks. Instead of using development time assertions (using Java `asserts`), we wanted to throw meaningful exceptions. For instance, each addition of an element to a collection without

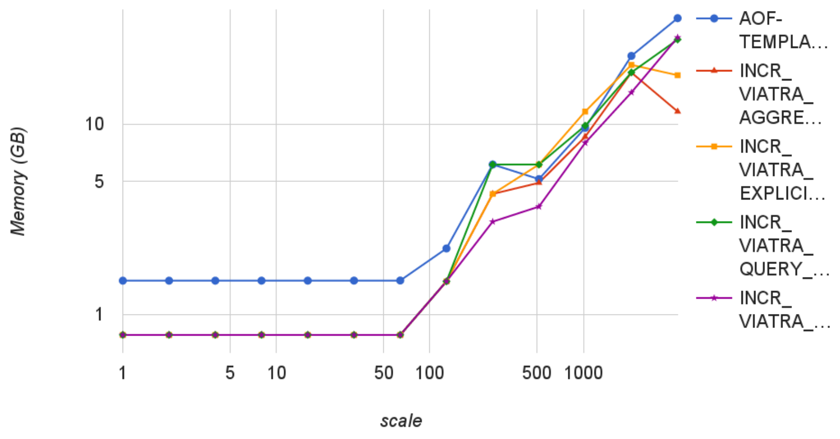

Fig. 2. Compared transformation memory usage

duplicates (i.e., OrderedSet or Set) induced a check that that element was not already present in the collection. Although this is helpful during AOF development, this is actually not necessary for its use. As a matter of fact, code written using AOF never needs to directly use these methods because boxes should only be modified by the operation creating them (except root boxes, which are bound to EMF lists on which the changes are performed). Moreover, all operations guarantee that they do not violate these checks. Therefore, only bugs in AOF can result in sanity check violations, which makes these checks only useful while debugging AOF. Turning them into asserts does not sacrifice correctness. This change alone significantly reduced time complexity.

We also performed a few local optimizations in propagation algorithms, notably in the operation that converts boxes with duplicates (i.e., Bag or Sequence) into boxes without duplicates (i.e., OrderedSet or Set).

4.2 Transformation Writing Methodology

Scalability can only be achieved if duplicate computations, and duplicate mutable values (i.e., boxes) are avoided. Special care has to be taken to make sure of it when writing AOF-based transformations with Java or Xtend as front-ends.

AOF internally caches boxes produced by its `CollectBox` operation, which is a variant of collect used when the given lambda returns a mutable value. This cache is necessary for correctness. However, we realized while working on the benchmark that most operations should use caches to avoid duplication of work and memory. Concretely, we must avoid having two or more boxes computed from the same initial box through the same sequence of operations with the exact same parameters. We have not integrated caching in all AOF operations yet, but have simply implemented it in the benchmark transformation. One difficulty in implementing caching is that lambda expressions given as parameter to many operations do not have intrinsic identity in Java or Xtend. This means that we cannot use lambdas inline if we want caching, and must create them separately. To help discover where caching is required, we created a simple tool that analyzes the box-operation graph to detect duplicate boxes, and report where they were created. This tool uses introspection to build a full lambda identity by taking into account captured values. An OCL front-end could automate (and hide to the user) most of these considerations.

4.3 Specific Operations

Operations provided by AOF are sufficient to implement the benchmark transformation. However, in two specific locations they cannot provide adequate scalability. This section presents the two specific operations that we developed to overcome this issue. These operations could be later integrated in AOF. Being able to create such specific operations shows AOF extensibility. In practice, we expect an OCL front-end to automatically and transparently rewrite OCL expressions in terms of these operations in order to provide adequate performance.

GroupBy is well-known from database query languages such as SQL. Listing 1.4 gives an example implementation of this operation in OCL. Given a collection (myCollection) and a function to get a key from each of its elements (getKey), we return one tuple per unique key. This tuple contains the key, and a collection of all elements from myCollection for which getKey returns that key. The main issue is that there is a select nested inside of a collect. This results in quadratic performance. We discovered this issue by using the visualvm CPU profiler.

Listing 1.4. Possible implementation of `GroupBy` in OCL

```
let keys : Set(OclAny) =
  myCollection.collect(e | getKey(e))->asSet() in
keys->collect(key |
  Tuple {key = key, elements =
      myCollection->select(e | getKey(e) = key)}
)
```

However, such an operation is required for the generation of the trace model of the benchmark transformation. Indeed, the specification of this trace model mandates that trace links for 2-to-1 transformation rule (e.g., from the pair (ApplicationInstance, StateMachine) to DeploymentBehavior) be represented as 1-to-many trace links (e.g., from each StateMachine to all DeploymentBehaviors created from a pair with that StateMachine), while forgetting the second source element (e.g., ignoring the ApplicationInstance). VIATRA can work with such a trace[15], but computing it from AOF internal trace requires additional effort.

Linear performance can be achieved by first computing a HashMap while traversing the source collection. We implemented an active `GroupBy` operation using this technique.

SelectBy Another scalability issue was due to the creation of a relatively large number of mutable booleans (i.e., singleton boxes containing a boolean) in the lambda of a select operation. This is mainly a memory issue, but has nonetheless an impact on execution time due to the additionally required allocation and garbage collection work. Consider an expression such as:

```
myCollection.select(e | selector(e) = someMutableValue)
```

where `selector` can be any arbitrarily complex mutable expression. Because both operands of the equality check are mutable, a mutable boolean has to be created for every compared pair. This produces the correct result, but can require the creation of a relatively large number of boxes depending on where in the transformation it appears, and which shape the model has. In practice, the StatisticsBased case of the benchmark is the only one where the problem has a visible impact due to its specific statistical distribution of model element types and links.

In order to avoid this issue, we created a `SelectBy` operation, which can be used in the following way:

```
myCollection.selectBy(selector, someMutableValue)
```

Internally, it works with a HashMap, which it actively maintains up-to-date upon changes in myCollection, or in properties navigated in the selector. To detect this kind of issue, which can be resolved with `SelectBy`, we created a simple box profiler that shows hot spots in transformation code that are responsible for the creation of many boxes. This tool is easier to use than the memory profiler of visualvm because it focuses on boxes only, and does not need to be run separately from the transformation.

4.4 More Optimization Perspectives

Although the optimizations presented above were enough to achieve scalability comparable to VIATRA's, other optimizations are possible.

First, we consider some optimizations that could be implemented in the current version of AOF. Our optimization work so far focused on the parts of AOF actually used in the benchmark. Other parts likely need optimization as well. Identifying them will require a new benchmark, or operation-specific performance tests. Memory usage could also be lowered by using specific storage for

[15] There was however a bug in the plain xtend batch implementation of the benchmark transformation that was linked to this trace representation: `https://github.com/viatra/incquery-examples-cps/issues/72`.

singleton values, which currently use ArrayLists like collections. Finally, by giving control of caches to transformation developers, we could avoid the need for costly weak references.

Second, other optimizations are possible with a significant rewriting work. Mainly, relaxing order preservation on Set and Bag would improve performance (e.g., by enabling constant time `includes` as can be provided by Java HashSet). Indeed, AOF currently preserves order even on these collections for which it is not necessary. Moreover, it is mostly because of order preservation that AOF was under-optimized as detailed in Section 4.1: uniqueness sanity checks are slower than they could be because all collections are backed by ArrayLists instead of hash-based structures like HashSet. Laziness [15] could also help lower memory usage. Finally, parallel execution may be possible, for instance by using Java 8 streams.

5 Advantages of Active Operations

Because VIATRA and AOF take significantly different approaches to the model transformation problem, they exhibit different properties. In this section, we focus on two advantages of AOF-based transformations over VIATRA-based ones: order preservation (Section 5.1), and extensive OCL compatibility (Section 5.2).

5.1 Order Preservation

By design, all AOF operations preserve collection order. They even do it for unordered collections (i.e., Set and Bag), as mentioned in Section 4.4.

Order preservation is not explicitly required by the benchmark transformation specification[16], but all multivalued structural features of both the source and target metamodels are specified as being ordered. Furthermore, the plain xtend implementations of the transformation do preserve order, but none of the VIATRA-based transformations do, as VIATRA has no support for this, and is thus unable do it efficiently.

Actually, even when order preservation would bring significant benefits, VIATRA approaches do not do it. Source model generation is performed using the VIATRA query engine, and it is currently not fully deterministic for reasons that seem to be linked to order preservation.

Beyond transformation-specific requirements on order preservation (e.g., displaying target elements in the same order as source elements), it helps debugging by resulting in deterministic target models and execution paths. It also lets developers make more assumptions when stepping through transformations.

It should be noted that preserving collection order is not always enough to achieve order preservation of all model element properties. The reason is that bidirectional associations can be initialized from an unordered or singleton end.

[16] `https://github.com/viatra/viatra-docs/blob/master/cps/CPS-to-Deployment-Transformation.adoc`

If order is to be maintained, the ordered end of these associations should always be used, or order should be restored by other means[17].

5.2 Extensive OCL Compatibility

As described in Section 2, active operations enable incremental evaluation of expressions by using mutable intermediate results produced by operations with change propagation capabilities. This approach is applicable to OCL, as explained in [10]. The only limiting factor is availability of incremental operation implementations.

Although not all OCL operations are currently provided by AOF, some unsupported ones can be expressed in terms of the supported ones, with the drawback of being generally less scalable or at least slower than a specific implementation. For instance, many non-active functions operating on singleton values can be lifted to operate on mutable values: by using `collect` when it only depends on a single mutable value (e.g., x+1, −x), or by using `zipWith` when it depends on two mutable values (e.g., x+y). For functions depending on more than two mutable values, these values can be `zip`ped before applying `collect`.

Besides, there is no theoretical issue preventing any operation from being implemented in an incremental way. The only main difficulty in integrating a new operation is to find efficient propagation algorithms, and implement them correctly. For instance, `iterate` is not easy to make incremental in the general case. However, if it is used with an associative operator, it can be turned into a `reduce`, and computations can be arranged into a tree, which provides logarithmic update cost.

6 Related Work

The authors are not aware of any mature OCL-based scalable and incremental transformation tool. A scalable and incremental QVT implementation is in progress [18] but is only at the proof-of-concept level. The closest solution is to use VIATRA [17] along with the work presented in [4] to translate a subset of OCL to VIATRA. However, although our approach can be extended to support all of OCL, this is not the case of that one. Another difference is that we can preserve collection order, while it cannot (see Section 5). Moreover, our approach has some bidirectional capability [2] that VIATRA does not have. It should be possible to jointly use VIATRA and AOF by agreeing on a change event API, and thus benefit from the best of both approaches.

Proof-of-concepts like incremental ATL [12] and Reactive ATL[18] are able to incrementally update target models, but they have not been shown to scale.

[17] See the following bugzilla entry for a discussion on order preservation at transformation level: `https://bugs.eclipse.org/bugs/show_bug.cgi?id=490172`.

[18] `https://github.com/atlanmod/org.eclipse.atl.reactive`

Furthermore, although they correctly detect when to reevaluate OCL expressions, they have to completely reevaluate them. Conversely, AOF only performs a minimal amount of re-computation.

The work presented in [6] focuses on rewriting integrity check OCL constraints in order to enable more fine-grained incremental model validation. However, when a constraint needs to be reevaluated, this approach does not provide means to do so incrementally. Therefore, we consider it complementary to active operations.

A possible alternative way to support incrementality on large models is to translate the problem to a different technical space. For instance, [13] translates OCL expressions into SQL queries. Using this approach for model transformation would require extending it, and would not be easy to integrate with modeling frameworks such as EMF, and tools based on such frameworks.

Other kinds of scalability than with respect to source model size are possible. For instance, [14] focuses on graph transformation size scalability. We have not considered this issue yet, but rule selection would likely be a performance bottleneck.

Other approaches focus on non-incremental model transformation scalability. Parallel ATL [16] is thus able to leverage the many cores of modern machines to execute different parts of a transformation in parallel. As mentioned earlier, it should be possible to parallelize the execution of active operations, and parallel ATL could provide some valuable insight to that end. Mogwaï [7] translates OCL expressions into NoSQL queries, thus supporting very large models. Approaches such as LinTra [5] or ATL on MapReduce [3] enable distributed execution of model transformations. LinTra can even do it in-place. We have not considered distributed execution of active operations yet.

7 Conclusion

The work presented in this paper leverages the VIATRA CPS benchmark [8] to show that active operations can be implemented in a scalable, time and memory efficient way. Although the Active Operation Framework (AOF) is not as mature as an incremental framework such as VIATRA, it offers unique advantages: it can preserve collection order, and can be used as an OCL back-end. The main missing element is an actual OCL front-end built on top of AOF.

Scalability of AOF has only been evaluated for model transformation, but it should also apply to incremental integrity constraint checking. Top performance could likely be achieved by combining the integrity constraint rewriting work from [6] with active operations.

Finally, the VIATRA CPS benchmark mostly evaluates scalability with respect to source model size. Change propagation performance should also be evaluated.

Acknowledgement

The authors would like to thank Ábel Hegedüs for his invaluable assistance in implementing the VIATRA CPS Benchmark.

References

1. Beaudoux, O., Blouin, A., Barais, O., Jézéquel, J.M.: Active Operations on Collections. In: ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10). pp. 91–105. Oslo, Norway, Norway (2010), https://hal.inria.fr/inria-00542763
2. Beaudoux, O., Jouault, F.: Bidirectional incremental transformations with Active Operation Framework – Application to Facades. In: 1st Papyrus Workshop - DSML Technologies (CEA). Toulouse, France (Jun 2015)
3. Benelallam, A., Gómez, A., Tisi, M., Cabot, J.: Distributed model-to-model transformation with atl on mapreduce. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering. pp. 37–48. SLE 2015, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2814251.2814258
4. Bergmann, G.: Translating OCL to Graph Patterns, pp. 670–686. Springer International Publishing, Cham (2014), http://dx.doi.org/10.1007/978-3-319-11653-2_41
5. Burgueño, L., Troya, J., Wimmer, M., Vallecillo, A.: Parallel In-place Model Transformations with LinTra. In: Proceedings of the 3rd Workshop on Scalable Model Driven Engineering (BigMDE 2015) part of the Software Technologies: Applications and Foundations (STAF 2015) federation of conferences. pp. 52–62 (2015)
6. Cabot, J., Teniente, E.: Incremental evaluation of ocl constraints. In: In: Proc. 18th Int. Conf. on Advanced Information Systems Engineering, LNCS. p. 4001. Springer (2006)
7. Daniel, G., Sunyé, G., Cabot, J.: Mogwaï: a Framework to Handle Complex Queries on Large Models. In: International Conference on Research Challenges in Information Science (RCIS 2016). Grenoble, France (Jun 2016), https://hal.archives-ouvertes.fr/hal-01344019
8. IncQuery Labs Ltd.: Performance benchmark using the viatra cps demonstrator, https://github.com/viatra/viatra-cps-benchmark
9. Izsó, B., Szárnyas, G., Ráth, I., Varró, D.: MONDO-SAM: A Framework to Systematically Assess MDE Scalability. In: BigMDE 2014 2nd Workshop on Scalable Model Driven Engineering. p. 40. ACM, ACM (2014)
10. 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 co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015. pp. 35–45 (2015), http://ceur-ws.org/Vol-1512/paper03.pdf
11. Jouault, F., Kurtev, I.: On the interoperability of model-to-model transformation languages. Science of Computer Programming 68(3), 114–137 (Oct 2007)
12. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations, pp. 123–137. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-13688-7_9

13. Oriol, X., Teniente, E.: Incremental checking of OCL constraints through SQL queries. In: Proceedings of the 14th International Workshop on OCL and Textual Modelling co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014), Valencia, Spain, September 30, 2014. pp. 23–32 (2014), http://ceur-ws.org/Vol-1285/paper03.pdf
14. Strüber, D., Kehrer, T., Arendt, T., Pietsch, C., Reuling, D.: Scalability of model transformations: Position paper and benchmark set. In: Proceedings of BigMDE 2016: Workshop on Scalability in Model Driven Engineering. (Jul 2016)
15. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 28, 2015. pp. 46–61 (2015), http://ceur-ws.org/Vol-1512/paper04.pdf
16. Tisi, M., Martínez, S., Choura, H.: Parallel Execution of ATL Transformation Rules, pp. 656–672. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-41533-3_40
17. 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), http://dx.doi.org/10.1007/s10270-016-0530-4
18. Willink, E.: Optimized declarative transformation - first eclipse qvtc results. In: Proceedings of BigMDE 2016: Workshop on Scalability in Model Driven Engineering. (Jul 2016)