Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Efficient ATL Incremental Transformations

Théo Le Calvar[a] Frédéric Jouault[b] Fabien Chhel[b]
Mickael Clavreul[b]

a. LERIA, University of Angers, France

b. ERIS Team, Groupe ESEO, France

Abstract Incrementally executing model transformations offers several benefits such as updating target models in-place (instead of creating a new copy), as well as generally propagating changes faster (compared with complete re-execution). Active operations have been shown to offer performant OCL-based model transformation incrementality with useful properties like fine-grained change propagation, and the preservation of collection ordering. However, active operations have so far only been available as a Java library. This compels users to program at a relatively low level of abstraction, where most technical details are still present. Writing transformations at this level of abstraction is a tedious and error prone work. Using languages like Xtend alleviates some but not all issues.

In order to provide active operation users with a more user-friendly front-end, we have worked on compiling ATL code to Java code using the active operations library. Our compiler can handle a significant subset of ATL, and we show that the code it generates provides similar performance to hand-written Java or Xtend code. Furthermore, this compiler also enables new possibilities like defining derived properties by leveraging the ATL refining mode.

Keywords Incremental model transformation Active operations ATL.

1 Introduction

Incremental model transformation execution offers many advantages over batch (i.e., non-incremental) execution, such as faster change propagation. However, there are only few incremental model transformation approaches that really scale to complex transformations on large models, Viatra [VBH^{+}16] probably being the most well-known. The approach used in Viatra is based on graph patterns and is quite different from approaches like ATL or QVT, which rely on OCL [Obj14] as navigation language. Each approach has its benefits, and it would be useful to have efficient incremental engines for all of them. Previous work [JB16] showed that active operations, which are an OCL-based approach to achieve fine-grained incremental execution, can be used to achieve performance similar to what is achievable with Viatra. However, active operations were used as a Java API from Xtend code. Consequently, low-level details were visible, and hindered the development and readability. In this paper, we present an approach that translates declarative ATL transformations into Java code that uses active operations in order to achieve efficient incremental execution of ATL transformations. We evaluate this approach on two public benchmarks, and show how incremental ATL can be used to support the definition of derived properties, which were not previously supported by any ATL implementation.

Section 2 presents the motivation for the incremental execution of ATL. The principles of how active operations achieve incrementality are presented in Section 3. Section 4 discusses the main issues related to the translation of ATL code into active operations, while Section 5 describes our compiler. Our approach is evaluated in Section 6. Some related works are discussed in Section 7. Finally, Section 8 gives some concluding remarks.

2 Motivation

This section briefly presents three applications of incremental execution: incremental queries, incremental transformations, and derived properties.

Incremental Queries. A model query (or simply *query* in the remainder of this paper) is typically used to compute a single value from a model. Such a query cannot make changes to a model, such as creating new model elements. With ATL, queries are expressed in OCL, and can compute values as simple as a single integer, but also any kind of OCL value such as a collection of elements. Computing complex queries can be expensive (in time, and memory usage), and can benefit from being performed incrementally. Indeed, the live contest of TTC 2018 (Transformation Tool Contest) consisted in implementing two such queries on specific social media models [Hin18]. Change propagation performance was then evaluated by executing these queries on models that were then changed multiple times to simulate user activity (e.g., adding comments to posts). In order to evaluate scalability, this was repeated on models of increasing size.

Incremental Transformations. A model transformation is typically used to create a set of new target models from a set of existing source models. There is no constraint on what these models may represent, only that they are defined using a modeling framework compatible with the transformation language used. In some situations, source models may change, triggering the need to compute updated target models. Sometimes it is enough to execute the transformation again, thus producing a new set of target models consistent with the changed source models. But there are cases in which it is necessary to update the target models produced by the transformation's first execution. This is for instance the case when a target model is actually a view displayed in a modeling tool. Computing a new model would not update what users see, whereas updating the existing view in place would. The incremental execution of the transformation can achieve this. Moreover, like queries, complex transformations can be expensive to execute. Executing them incrementally often provides a huge speed up over executing them fully after each source model change. The Viatra CPS (Cyber Physical System) to Deployment benchmark [Inc] specifies a small but non-trivial transformation along with tools to generate source models, source model changes, as well as reports, based on the MONDO-SAM framework [ISRV14] These tools can be used to compare the performance of transformation languages given implementations of the CPS to Deployment transformation in these languages.

Derived Properties. Some metaclass properties can be computed from other parts of the model like other properties of their owning metaclass, or properties of other metaclasses reachable from their owning metaclass. Such properties are called *derived properties*. OCL can be used to specify the value of a derived property based on other parts of its context model [Obj14, Section 7.3.7, page 9], for instance:

```
context Node::name : String derive: 'node_'.concat(self.id)
```

This line of OCL code means that the property named `name` belonging to the metaclass named `Node` has type `String`, and must at all times be equal to the concatenation of the string `'node_'` with the value of property `id`. However, the OCL specification does not state how and when derived properties should be computed. The notion of incremental query as presented above may be applied to the derivation expression, so that its value can then be automatically updated whenever the parts of the model on which it depends are changed. However, the resulting value must also be assigned to the derived property itself in such a way that the latter is updated along with the former. We evaluated derived properties in a transformation not presented in this paper but available on GitHub[1].

3 Incrementality with Active Operations

In this section, we present how active operations make incremental transformations, as discussed in the previous section, possible.

3.1 Overview

Active operations were introduced in [BBBJ10] as a way to specify incremental queries. They operate on a concept of mutable value called *box*. A *box* can be observed, and every change is notified to each of its observers. There are several kinds of boxes mostly mirrored after the different kinds of OCL collections: `Bag` (also known as multiset), `OrderedSet`, `Sequence`, and `Set`. In addition to these collection boxes, there are also two kinds of singleton boxes: `One`, which always has a value, and `Option`, which may be empty, thus representing a kind of nullable value.

In order to perform computations on such boxes, one must use active operations. Available active operations are inspired by OCL collection operations, for instance: `collect`, `select`, `first`. Non OCL operations are also supported for convenience, such as: `zip`, `zipWith`, `groupBy`. Each of these operations is equipped with change propagation algorithms. Not only can they compute their target values given their source values, but they can also update their target values when their source values change. By composing active operations, more complex expressions can be formed that are still able to update their target values upon source value changes. These expressions, especially the lambda expressions given to active operations, must be side effect free because propagation may basically trigger computations in arbitrary order.

[1] `https://github.com/TheoLeCalvar/scheduling-example`

$$\begin{tikzcd}
A \arrow[r, "op_1"] \arrow[d, no head] & B \arrow[r, "op_2"] \arrow[d, dashed, no head] & C \arrow[d, no head] \\
\delta_A \arrow[r, "prop_1"] \arrow[d] & \delta_B \arrow[r, "prop_2"] \arrow[d, dashed] & \delta_C \arrow[d] \\
A' \arrow[r, dashed, "op_1"] & B' \arrow[r, dashed, "op_2"] & C'
\end{tikzcd}$$

Figure 1 – Active operation composition example

3.2 Example

Figure 1 gives an example of active operation composition. Operation op_1 produces value B from value A, and operation op_2 further produces value C from value B. Later, a change δ_A occurs on value A, turning it into changed value A'. The propagation algorithm $prop_1$ of operation op_1 produces update δ_B from change δ_A. Then, propagation algorithm $prop_2$ of operation op_2 produces update δ_C from update δ_B. Finally, update δ_C is applied to value C in order to produce updated value C'. The dashed lines represent redundant computations that do not need to be performed. For instance, update δ_B could be applied to value B in order to obtain updated value B'. Then, the value of B' would be the same as if operation op_1 was applied to value A', and the value of C' would be the same as if operation op_2 was applied to value B'. But, in general, intermediate values do not need to be stored and updated in this way. There are therefore two ways to compute updated value C' given value A and update δ_A. The first way (re-execution) consists in applying update δ_A to value A, then successively applying op_1 and op_2. If we represent each operation and each update as a function, this may be written in the following way:

$C' = op_2(op_1(\delta_A(A)))$

The second way (incremental propagation) consists in first applying op_1 followed by op_2 to value A, then applying update δ_C, which may be written as:

$C' = \delta_C(op_2(op_1(A)))$

which becomes, if we represent the propagation algorithms as functions, and replace δ_C by its expression in terms of δ_A, $prop_1$, and $prop_2$:

$C' = prop_2(prop_1(\delta_A))(op_2(op_1(A)))$

As these expressions show, propagation algorithms compose in the same way as regular operations, but operate on updates instead of operating on values. Note that some propagation algorithms require access to the source value in addition to the update, which could be represented as $\delta_B = prop_1(A, \delta_A)$ instead of $\delta_B = prop_1(\delta_A)$. This makes the situation slightly more complex, but does not change it significantly from what is explained above.

We call *propagation graph* the graph formed by values as nodes, and operations as edges. Because changes on source values can happen at any time, this propagation graph must be kept as long as one needs to be able to propagate changes.

3.3 Lifting Operations

Singleton mutable values are wrapped in boxes so that they can notify observers of changes to their internal values, even if these values are immutable (e.g., like instances of `java.lang.Integer` in Java). Standard operations on such values (e.g., addition of integers) must therefore be lifted into active operations: using `collect` for a single mutable value, or using `zipWith` for multiple mutable values. For instance, `x + 1` where `x` is mutable becomes:

```
x−>collect(e | e + 1).
```

With two mutable values `x`, and `y`, `x + y` becomes:

```
x−>zipWith(y)(a, b | a + b),
```

where `zipWith` takes both `x` and `y` as source boxes, as well as a lambda expression to apply to their contents.

3.4 Current Implementation: AOF

The current active operation implementation is called AOF for Active Operations Framework. It is an online, or in memory, implementation of active operations that keeps the complete propagation graph in memory. This has the advantage of making it possible to propagate changes with reduced latency, but requires the propagation graph to fit in available memory.

AOF can operate on its own type of boxes, but it can also wrap mutable values provided by other libraries. For instance, it provides means to view the properties of EMF model elements as boxes on which active operations can be composed [JB16]. JavaFX[2] properties can also be viewed as boxes, which makes it possible to use AOF-based incremental transformations to visualize models graphically.

4 Translation from ATL to Active Operations

The previous section has presented how active operations can be used to specify incremental expressions. This section gives an overview of how ATL transformations can be expressed in terms of active operations.

4.1 OCL Expressions

The two main issues with the translation of OCL expressions into code using active operations are:

- **Accessing model element properties.** Each access to a property must be translated into the creation of a box from which active operations can be composed to form an expression. In this way, when the property value changes, the value of the whole expression will be updated.
- **Dealing with mutable values.** Some OCL expressions are immutable, like primitive type literals (e.g., `Integers`, `Strings`), and variables referring to contextual elements. But other values are mutable, like property accesses, and variables initialized using mutable expressions. Expressions that build on other expressions (e.g., operation or operator calls), must therefore be able to deal with both mutable and immutable values.

[2]JavaFX (`https://openjfx.io/`) is a Java framework that is used to create graphical views.

Dealing with both immutable and mutable values requires being able to rewrite all expressions building on other expressions depending on their mutability. This includes operation lifting discussed in Section 3.3, but also specific kinds of expressions like `if−then−else−endif`. In order to appropriately rewrite these expressions, knowing whether they are mutable or not is crucial. This can be achieved by analyzing the mutability status of every expression. Leaf expressions are known to be either mutable or immutable, and compound expressions are mutable as soon as one of their nested expressions is mutable.

4.2 Refining Mode

Refining mode ATL transformations are used to perform relatively small changes on models. However, this objective had to be reconciled with the basic semantics of ATL, which only considers read-only source models, and write-only target models. The original refining mode execution strategy consisted in automatically copying all parts of a model that were not explicitly matched by transformation rules. This way, the write-only target model is a slightly modified copy of the read-only source model. This *copy* strategy is trivially compatible with incremental execution because it is basically equivalent to explicitly writing a copy rule for every element that does not need to be transformed. For the same reason, its usefulness is limited to reducing size of the transformation code, which can actually be achieved via another technique: module superimposition [WVDSD10]. Moreover, copying whole models can be quite costly when performed on large ones.

This is why a new *in place* execution strategy was designed [TMJC11], and implemented in the current version of ATL. It consists of two phases: 1) first computing all changes that must be performed on the model without actually touching it, and 2) applying these changes. Every OCL expression is thus evaluated on the original unchanged source model, which remains compatible with the ATL semantics. We therefore decided to look into the problem of marrying incrementality with an *in place* refining strategy. The core issue is that incremental in place updates can result in propagation cycles, for instance if two properties are computed from each other. Repeated in place updates are naturally supported by other approaches like graph transformation [Hec06] but not by active operations. We could define some kind of fixpoint semantics, but this would basically turn ATL into a search tool, which is beyond the scope of the present paper. Instead, we elected to only consider refining transformations that do not involve cycles.

5 Proof of Concept Compiler: ATOL

The previous section has presented how ATL transformations can be translated into code that uses active operations. In order to evaluate this approach, we created ATOL: a proof of concept compiler from ATL to Java code that uses AOF.

5.1 Objectives and Trade-offs

The main reason that motivated the development of ATOL was to make it possible to evaluate the performance and expressiveness of our approach. We chose to compile ATL code into Java code instead of writing an ATL interpreter in Java in order to get better performance. However, developing a full-fledged ATL compiler is not trivial, and would require more resources than were available to us. We therefore had to accept some trade-offs between full ATL support and development cost. We decided to develop a proof of concept implementation focused on the subset of ATL necessary to achieve good performance. For instance, in order to achieve acceptable performance, standard rule selection requires more advanced static analysis than we have implemented so far. Therefore, ATOL only supports unique lazy rules. Moreover, in order to further simplify static analysis of ATL code, metamodel properties with colliding names must be used with a numeric disambiguation suffix when performing an implicit collect. These and other trade-offs prevent many existing ATL transformation from compiling unmodified with ATOL, but do not impact the validity of the evaluation performed in this paper.

5.2 Metamodel Representation

A model transformation must have means to access its source and target models and metamodels. There are three main aspects to consider:

- **Type introspection** is required to test whether a given model element has a given metaclass for type. This is not only used in OCL to support the `oclIsKindOf`, and `oclIsTypeOf` operations, but also in ATL to check which rule matches a given element.
- **Element creation** is necessary so that transformation rules can create their target elements.
- **Property access** is used both to navigate source models, but also to initialize target element properties. In order to use active operations, a mechanism to obtain boxes associated to properties of elements is required.

Moreover, in order to support different kinds of models, the compiler must not have dependencies to a specific modeling framework like EMF. The metamodel representation used by ATOL takes all these requirements into account. Listing 1 gives a Java code excerpt of the class representing the simple metamodel from Figure 2. Each metaclass has a corresponding field with the same name (e.g., see line 2), and type `IMetaClass`, which is an interface offering type introspection using an `isInstance` method, and element creation with a `newInstance` method. This interface takes a type parameter for the same reason that `java.lang.Class` does: so that the Java compiler can perform some type checking, for instance on the return value of the `newInstance` method. Furthermore, each metaclass property is captured by two methods: a field access method, and a navigation method. The field access method of property `id` is shown at line 3. It takes an instance of the metaclass as parameter, and returns an AOF boxed value. The name of this field access method is formed by prefixing an underscore to the name of the field. This is of no consequence for ATOL, but is useful when writing Xtend helpers: these methods can be used as extension methods[3], but their names would collide with the actual EMF accessor methods without the prefix. The navigation method (e.g., see line 6) takes a box of instances of the metaclass or one of its children, and returns a box. These navigation methods may be used to support the implicit `collect` navigation offered by OCL, but may

[3]An extension method is a syntactic sugar allowing methods to be added to an object outside after it has been defined. Which can later be used in the same way as any other method defined by the object.

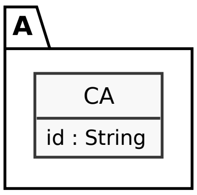


Figure 2 – Class diagram representing metamodel A

also simply be used to access properties of mutable singleton values. Such metamodel representation classes can be written manually, which makes it possible to use any kind of model. For EMF support, we have implemented an Xtend active annotation[4]: `AOFAccessors`. It takes as argument the class generated by EMF from the metamodel package, and automatically generates the Java code of the metamodel representation class. Thus, a user only has to write the code of Listing 2 to automatically obtain the code of Listing 1.

```
class A {
  static IMetaClass<CA> CA = // [...]
  IBox<String> _id(CA a) {
    // [...]
  }
  IBox<String> id(IBox<? extends CA> a) {
    // [...]
  }
}
```

Listing 1 – Java class representing metamodel A generated from the metamodel of Figure 2 and the Xtend Class of Listing 2

```
@AOFAccessors(APackage)
class A {
}
```

Listing 2 – Source Xtend class representing metamodel A of Figure 2 for ATOL

5.3 Compilation Process

There are multiple possible ways to generate Java code from ATL code. Developing a standalone compiler would reduce the number of required dependencies. However, it would be necessary to write plugins for such a compiler in order to integrate it into development environments, and build systems. Therefore, we elected to follow the same approach used to generate the metamodel representation class presented above: an Xtend active annotation, which we called `@ATOLGen`. Figure 3 shows the overall ATOL compilation process in three steps denoted by circled numbers:

- First, given an Xtend file `A2B.xtend` like the one of Listing 3 with the `@ATOLGen` annotation, the Xtend compiler calls the corresponding annotation processor called `ATOLGenProcessor`.
- Second, this annotation processor reads the annotation arguments, which in-

[4]`https://www.eclipse.org/xtend/documentation/204_activeannotations.html`

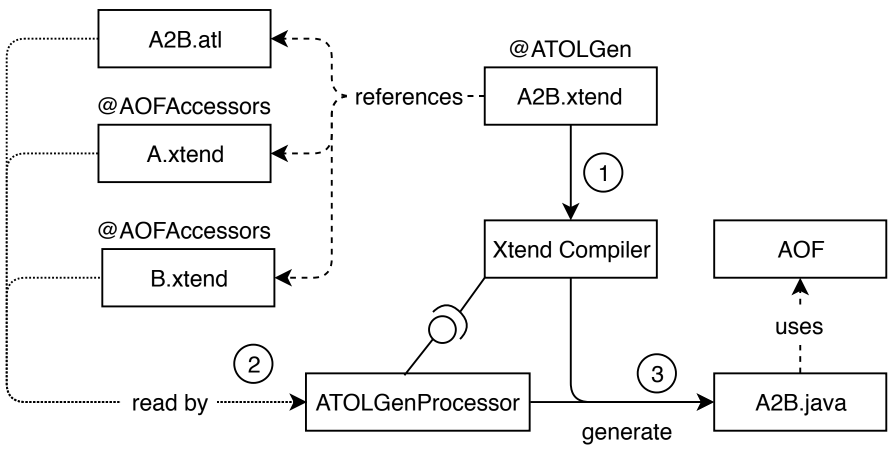


Figure 3 – Overview of the ATOL compilation process

clude the path to an ATL transformation file `A2B.atl` like the one of Listing 4, and a list of the metamodels used in the transformation.

- Third, it then proceeds to parse this ATL file, and generate the Java code corresponding to the translation of the ATL rules, and helpers, resulting in a file `A2B.java` like the one of Listing 5.

The second argument to the `@ATOLGen` annotation is the list of metamodels (see lines 2–3 of Listing 3). Each metamodel argument has a name, which must be the same as in the ATL file, and an implementation (`impl`) of a metamodel representation class, which must follow the scheme presented in Section 5.2. The ATOL compiler generates immutable tuple classes for the source and target patterns of each rule (lines 2–9 of Listing 5). These tuple classes are used to package source and target elements so that they can be stored in a hash map implementing the transformation trace. Each unique lazy rule is also translated into a method (see lines 10–11 of Listing 5) that can be called from external Java code. ATL helpers like the one at lines 9–10 of Listing 4 are transformed into a method computing the helper (see lines 13–15 of Listing 5), as well as a navigation method (see lines 16-18 of Listing 5). This is similar to how properties are represented in the metamodel representation class. Finally, users can also write helpers in Xtend (e.g., see lines 6–8 of Listing 3), which the Xtend compiler translates into Java code (e.g., see lines 19–21 of Listing 5). This makes it possible to call any Java code from the ATL transformation.

```
1 @ATOLGen(transformation="A2B.atl",metamodels=#[
2   @Metamodel(name="A", impl=A),
3   @Metamodel(name="B", impl=B)
4 ])
5 class A2B {
6   String _h2(CA self) {
7     // [...]
8   }
9 }
```

Listing 3 – Example ATOL declaration in Xtend for the transformation from Listing 4

```
module A2B;
create OUT : B from IN : A;

unique lazy rule CA2CB {
    from s : A!CA
    to t : B!CB
}

helper context A!CA def: h1 : Real =
  -- [...]
```

Listing 4 – Example ATOL transformation

```
class A2B {
  static class SourceTupleCA2CB {
    final CA s;
    // [...]
  }
  static class TargetTupleCA2CB {
    final CB t;
    // [...]
  }
  TargetTupleCA2CB CA2CB(CA s) {
    // [...]
  }
  // [...]
  Double _h1(CA self) {
    // [...] with caching
  }
  IBox<Double> h1(IBox<CA> self) {
    // [...] with caching
  }
  String _h2(CA self) {
    // [...]
  }
}
```

Listing 5 – Example Java code generated from the transformation in Listing 4, and corresponding declaration in Listing 3

6 Evaluation on Case Studies

The ATOL compiler presented in the previous section makes it possible to evaluate our approach on the case studies mentioned in Section 2. Section 6.1 and Section 6.2 respectively present the results of the TTC 2018 live contest benchmark, and the Viatra CPS to Deployment benchmark. Section 6.3 gives an example of using ATL to define derived properties.

6.1 TTC2018 Social Network Queries

The case study, presented in [Hin18], defines a social network metamodel along with two queries Q1 and Q2. The corresponding benchmark code is available on GitHub[5]. We focus here on the first query: Q1. Listing 6 gives the code of our ATL implementation of this query. Note that during the TTC 2018 live contest we submitted an Xtend implementation using AOF [BJLCT18], with O(n) propagation time complexity, which we later optimized to O(log(n)). The incremental ATL solution presented here was created later from that optimized solution, once the ATOL compiler sup-

[5]https://github.com/TransformationToolContest/ttc2018liveContest

ported the necessary features. The performance results were presented in [BJLCT18], and they show that the incremental ATL solution is as efficient as the Xtend-based one. This is not surprising considering that both use the same AOF back-end, but it still shows that the compilation from ATL code does not impact performance significantly. More recent results can be obtained from a specific GitHub repository[6].

```
-- @path SN=/socialmedia/social_network.ecore
library Q1;

helper context SN!SocialNetworkRoot def: topPosts : Collection(String) =
  self.posts−>sortedBy(e |
    e.timestamp
  )−>sortedBy(e |
    e.score
  )−>subSequence(1, 3)−>collect(e | e.id);

helper context SN!Post def : score : Integer =
  let allComments:Collection(SN!Comment)=self.allContents(SN!Comment) in
  10∗allComments−>size() +
  allComments−>collect(e|e.likedBy−>size())−>sum();
```

Listing 6 – ATL implementation of query Q1 from the TTC2018 live contest [Hin18]

There are two main differences between the code of Listing 6, and the original ATL implementation we submitted during the live contest (see [BJLCT18, Listing 1]). First, ATOL does not support the rarely used ATL query syntax yet, only modules and libraries. We therefore used a library, and simply call the compiled `topPosts` helper from Java code. Second, the set of all comments was computed with a recursive helper in our original ATL solution, which was inefficient. We changed it into a call to `allContents`, an operation that retrieves all elements of a given type contained in a given element. This is basically equivalent to calling EMF's `eAllContents` operation followed by a `select` filtering by type, but incremental, and optimized for performance.

6.2 Viatra CPS to Deployment Transformation

This case study has already been used to evaluate AOF [JB16], showing that it basically scaled as well as Viatra. However, the transformation was written by hand in Xtend. We have now implemented this case study in ATL, and used the ATOL compiler to execute it with AOF as backend. The source code of this implementation is available on GitHub[7] with full benchmark execution results also available on GitHub[8]. Note that apart from the reference batch Viatra implementation, we only executed the incremental solutions integrated in the benchmark. Figure 4a shows execution time of the initial transformation of the publish-subscribe scenario[9]. None of the evaluated solution is parallelized, but all can benefit from the parallel garbage collector provided by the Java virtual machine. The Java heap size was forced for all executions to be 130 GB. Manually written AOF transformation used from Xtend (`INCR_AOF`), and AOF transformation generated from ATL code (`INCR_ATL`) have a slight advantage over incremental Viatra implementations (i.e., the other implementations starting with `INCR_`), but basically scale similarly. The incremental Viatra

[6]`https://github.com/TheoLeCalvar/ttc2018liveContest/`

[7]`https://github.com/TheoLeCalvar/viatra-cps-benchmark`

[8]`https://theolecalvar.github.io/viatra-benchmark-results/`

[9]Executed on a machine with two Intel® Xeon® X5670 - 2.93GHz 6-core 12MB CPUs, and twelve 16 GB PC3-8500 (DDR3-1066Mhz) Registered CAS-7 of RAM using OpenJDK 64-bit version 1.8.0_181.

implementations timed out before producing solutions at scale 16384, which may be due to their slightly higher memory demand triggering too many garbage collections. Figure 4b shows propagation time for the same scenario. Performance of the AOF-based implementations fall between the incremental Viatra implementations, and do not achieve the same performance as the best ones. However, the trends seem similar. The ATL implementation is slower than the AOF/Xtend one, which is probably due to an overhead introduced by compilation that we have not been able to track yet. As discussed in [JB16], specific non-OCL operations were required for optimization purpose like an incremental implementation of `groupBy`.

6.3 Derived Properties

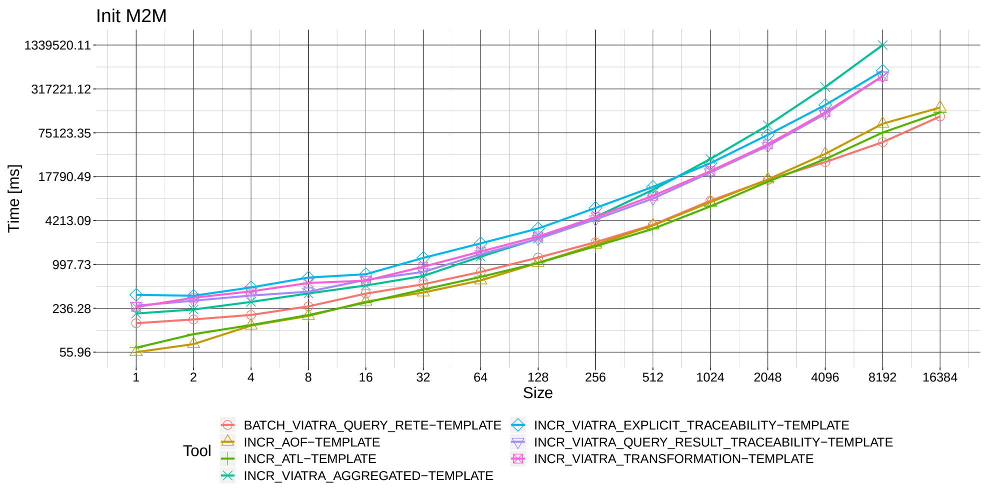
In ATL, values are assigned to properties via *bindings*, which are written using a right to left arrow with a property name on the left, and an OCL expression on the right. In the standard execution mode of ATL, the property must belong to a target model element, whereas the expression must be computed from source model elements only. However, with ATL's refining execution mode, the target and source models may conceptually be the same model, and actually are with the *in place* execution strategy, as discussed in Section 4.2. It is therefore possible to use the refining mode in order to compute a property from other parts of the model, as long as there are no cyclic dependencies. Listing 7 gives an example transformation that refines a model conforming to metamodel `MM` (line 2) by specifying how property `derivedProperty` of metaclass `Metaclass` is computed using `someExpression`. This is more verbose than using OCL for this purpose, but has the advantage of having the precise semantics given by active operations, whereas OCL does not state how and when derived properti should be computed. If using EMF, the derived property must not be declared as *derived* so that the refining transformation can bind an incremental value to it. In addition to being automatically updated, the derived property will also generate notifications when it is modified, which is especially useful if it is used in a downstream transformation, or in a user interface.

```
 1 module DerivedProperties;
 2 create OUT : MM refining IN : MM;
 3
 4 rule Task {
 5   from
 6     s : MM!Metaclass
 7   to
 8     t : MM!Metaclass (
 9         derivedProperty <− someExpression
10     )
11 }
```

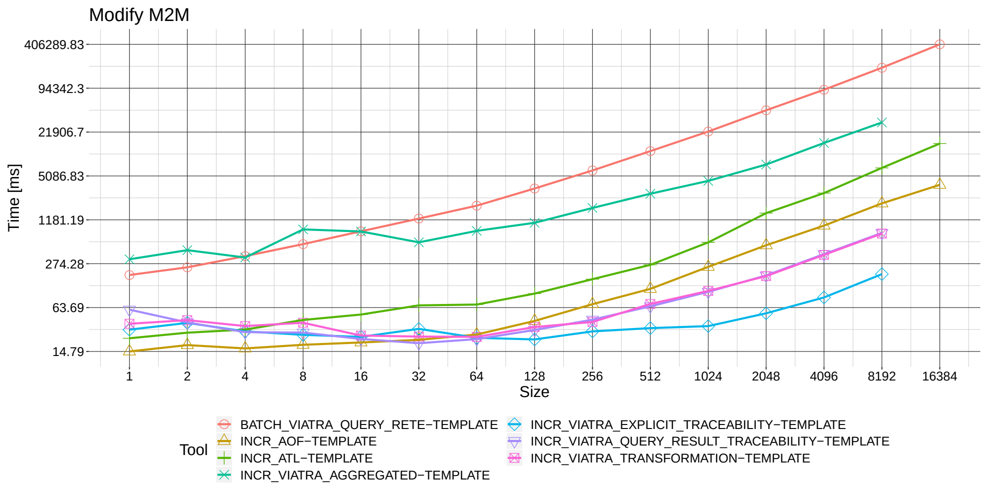
Listing 7 – ATL implementation of a derived property

7 Related Work

The work presented in this paper builds upon the results presented in [JB16], which already compared the active operations approach to other approaches. Viatra [VBH$^+$16] achieves performance similar to our approach, while having a more robust implementation. However, although a subset of OCL can be translated into graph patterns [Ber14], Viatra does not support the kind of OCL-based transformations that can be written in ATL. Moreover, active operations can preserve the order of elements



(a) Transformation execution time



(b) Change propagation time

Figure 4 – Viatra CPS to Deployment benchmark results

in collections, whereas Viatra cannot. The micromapping approach to QVT execution has made some progress [Wil17], but has only been shown to work on simpler benchmarks.

YAMTL [Bor18a] is a recently developed transformation language defined as an Xtend internal DSL. This is similar to the approach we used in our previous work [JB16] to write incremental transformations using AOF, while benefiting from a user friendly syntax that Xtend makes possible (notably via extension methods, and operator overloading). A batch YAMTL implementation of the CPS to Deployment transformation was integrated into the Viatra benchmark, but the incremental version has not been integrated yet as of writing this paper, which is why it does not appear in Figures 4a and 4b. The results obtained by YAMTL on the TTC 2018 live contest [Bor18b] are similar to the one we obtained [BJLCT18].

Our approach to make ATL incremental is not the first, and there are several other proposed ATL runtimes that support incremental transformations. [JT10, MTD17] present incremental ATL engines. Both keep track of which properties of source models are used in bindings in order to be able to recompute them whenever an update occurs on the source. Both support only a limited subset of ATL, namely the declarative part of ATL. The exact subset of ATL that is supported by each approach is not exactly the same. For instance, [MTD17] supports lazy evaluation of non-lazy rules, whereas both [JT10] and our approach do not support it. Unlike [MTD17, JT10], our approach supports incremental refining in-place transformation, which can be used to compute derived properties.

Interested readers can refer to [KBC$^+$18, KEK$^+$13] for a more comprehensive survey of existing incremental transformation techniques.

8 Conclusion

This paper has presented our approach to achieve efficient incremental ATL transformation execution. We show that active operations, when used as a back-end for ATL, make it possible to achieve performance similar to state of the art Viatra, but with an OCL-based language. The ATOL compiler that we developed to evaluate our approach needs more work in order to support a larger subset of ATL, but it can already be useful. We deliberatly limited the support of ATL to what was necessary to write the transformations showcased in the article and prove that active operations could be used to execute ATL transformations.

Section 4.2 mentioned propagation cycles in incremental transformations. Active operations cannot be used in such cyclic transformations. We partly overcame this limitation in [LCCJS19], in which we presented an approach that leverages the extensibility of the ATOL compiler so as to support mixing classic ATL with constraints declarations. These constraints can be used instead of bindings when cyclic relations are required. We plan to work on furthering the integration of active operations and constraints, and to use ATOL for this purpose.

References

[BBBJ10] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Active Operations on Collections. In *Model Driven Engineering Languages and Systems - 13th International Conference,* *MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2010.

[Ber14] Gábor Bergmann. *Translating OCL to Graph Patterns*, pages 670–686. Springer International Publishing, Cham, 2014.

[BJLCT18] Valentin Besnard, Frédéric Jouault, Théo Le Calvar, and Massimo Tisi. The TTC 2018 Social Media Case, by ATL and AOF. In *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences*, volume 2310 of *CEUR Workshop Proceedings*, pages 79–83, Toulouse, France, June 2018. CEUR-WS.org.

[Bor18a] Artur Boronat. Expressive and efficient model transformation with an internal dsl of xtend. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 78–88, New York, NY, USA, 2018. ACM.

[Bor18b] Artur Boronat. YAMTL Solution to the TTC 2018 Social Media Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences*, volume 2310 of *CEUR Workshop Proceedings*, pages 65–78. CEUR-WS.org, June 2018.

[Hec06] Reiko Heckel. Graph transformation in a nutshell. *Electronic notes in theoretical computer science*, 148(1):187–198, 2006.

[Hin18] Georg Hinkel. The TTC 2018 Social Media Case. In Antonio Garcia-Dominguez, Georg Hinkel, and Filip Krikava, editors, *Proceedings of the 11th Transformation Tool Contest, a part of the Software Technologies: Applications and Foundations (STAF 2018) federation of conferences*, volume 2310 of *CEUR Workshop Proceedings*, pages 39–43. CEUR-WS.org, June 2018.

[Inc] IncQuery Labs Ltd. Performance benchmark using the viatra cps demonstrator. URL: `https://github.com/viatra/viatra-cps-benchmark`.

[ISRV14] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. MONDO-SAM: A Framework to Systematically Assess MDE Scalability. In *BigMDE 2014 2nd Workshop on Scalable Model Driven Engineering*, page 40. ACM, ACM, 2014.

[JB16] Frédéric Jouault and Olivier Beaudoux. Efficient OCL-based Incremental Transformations. In *Proceedings of the 16th International Workshop in OCL and Textual Modeling*, volume 1756 of *CEUR Workshop Proceedings*, pages 121–136, Saint-Malo, France, October 2016.

[JT10] Frédéric Jouault and Massimo Tisi. Towards incremental execution of atl transformations. In Laurence Tratt and Martin Gogolla, editors, *Theory and Practice of Model Transformations*, pages 123–137, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[KBC+18] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Daniel Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, Mar 2018. URL: `https://doi.org/10.1007/s10270-018-0665-6`, `doi:10.1007/s10270-018-0665-6`.

[KEK+13] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. A survey on incremental model transformation approaches. In *ME 2013–Models and Evolution Workshop Proceedings*, page 4, 2013.

[LCCJS19] Théo Le Calvar, Fabien Chhel, Frédéric Jouault, and Frédéric Saubion. Toward a Declarative Language to Generate Explorable Sets of Models. In *34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*, Limassol, Cyprus, April 2019.

[MTD17] Salvador Martinez, Massimo Tisi, and Rémi Douence. Reactive Model Transformation with ATL. *Science of Computer Programming*, 136:1 – 16, March 2017. URL: `https://hal.inria.fr/hal-01627991`, `doi:10.1016/j.scico.2016.08.006`.

[Obj14] Object Management Group (OMG). Object Constraint Language (OCL), v2.4. `http://www.omg.org/spec/OCL/2.4/`, February 2014.

[TMJC11] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Refining Models with Rule-based Model Transformations. Research Report RR-7582, INRIA, March 2011.

[VBH+16] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth, and Zoltán Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the viatra framework. *Software & Systems Modeling*, 15(3):609–629, 2016.

[Wil17] Edward D. Willink. The micromapping model of computation; the foundation for optimized execution of eclipse qvtc/qvtr/umlx. In Esther Guerra and Mark van den Brand, editors, *Theory and Practice of Model Transformation*, pages 51–65, Cham, 2017. Springer International Publishing.

[WVDSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software & Systems Modeling*, 9(3):285–309, Jun 2010.

About the authors

Théo Le Calvar is a PhD student at ESEO and the University of Angers. His research focuses on the interactions between Constraint Programming (CP) and Model Driven Engineering (MDE). He received his Master's degree in 2016 from the University of Angers. His studies focused on CP and optimization. To conclude his Master's Degree he spent 6 months as an intern at the National Institute of Informatics in Tokyo. During this internship he studied multiobjective constraint optimization in distributed environments. Contact him at `theo.lecalvar@univ-angers.fr`.

Frédéric Jouault is a research associate at ESEO, Angers, France. He received his PhD from the University of Nantes before doing a postdoc at the University of Alabama at Birmingham. His research interests involve model engineering, transformation, synchronization, and execution, as well as their application to Domain-Specific Languages (DSLs) and model-based reverse engineering. Frédéric created ATL, a DSL for model-to-model transformation. He is now co-leading the development of ATL (language and toolkit) on Eclipse.org. Contact him at `frederic.jouault@eseo.fr`.

Fabien Chhel is an associate professor at ESEO. He received his PhD from the University of Angers in 2014. His previous fields of research focused on combinatorial optimisation and metaheuristic for analysing biological data. Currently, he works to improve model transformation engines for the model-driven engineering community with the help of the constraint satisfaction problem (CSP) framework. Contact him at `fabien.chhel@eseo.fr`.

Mickael Clavreul is an associate professor at ESEO, Angers, France. He received his PhD from the University of Rennes 1 in 2011. His research interests involve model engineering, model transformation, model composition, model synchronization and language engineering applied to Domain-Specific Languages (DSLs) and legacy systems. Mickael invented the ModMap language and the associated tool support to specify correspondences relations and alignment rules between both homogeneous and heterogeneous languages. Contact him at `mickael.clavreul@eseo.fr`.