The TTC 2018 Social Media Case, by ATL and AOF

Valentin Besnard
ERIS, ESEO-TECH
Angers, France
valentin.besnard@eseo.fr

Frédéric Jouault
ERIS, ESEO-TECH
Angers, France
frederic.jouault@eseo.fr

Théo Le Calvar
LERIA, Université d'Angers
Angers, France
theo.lecalvar@univ-angers.fr

Massimo Tisi
IMT Atlantique, LS2N (UMR CNRS 6004)
Nantes, France
massimo.tisi@imt-atlantique.fr

Abstract

Incremental model queries are a key solution to apply model-driven engineering to rapidly evolving models. This paper describes alternative solutions to the live competition of the Transformation Tool Contest 2018. The case study requires to query large models of social networks to derive the most influential and controversial contributions. We compare two batch solutions implemented in a general-purpose language (Xtend) and a model transformation language (ATL), to one incremental solution implemented by the Active Operations Framework.

1 Introduction

This paper describes the solutions to the Social Media case [Hin18] that were developed by the authors during the live competition of the Transformation Tool Contest 2018 (TTC2018)[1].

The goal of the contest was to make queries on models conforming to a social network metamodel (Figure 1). On each model, the chosen solution had to compute the result of two queries: a first time on the initial model and then after each change made on the source model. The goal of the first query (Q1) is to find the three most controversial posts (i.e., most commented and liked posts), while the goal of the second query (Q2) is to search the three most influential comments (i.e., comments commented by biggest groups of friends). The interested reader should refer to the case description paper in [Hin18] for a detailed presentation of the case study.

Our team submitted three solutions to this live contest. The first one has been implemented in ATL [JK05], a declarative rule-based model transformation language. The second one is a batch solution implemented in Xtend[2] mainly used for comparison with and validation of the two other solutions. The last one takes avantage of the Active Operations Framework [JB15] (AOF) based on active operations [BBBJ10] for incremental computation.

During this live contest, our solutions were competing against various solutions based on other tools. An evaluation process has been used to classify all solutions according to four criteria: correctness, conciseness,

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

[1] `https://www.transformation-tool-contest.eu/solutions_liveContest.html`

[2] `https://www.eclipse.org/xtend/`

understandability, and performance. As outcome of this evaluation, our team won two awards: the "Most Concise Solution Award" for the ATL solution, and the "Audience Award" for the AOF solution.

The remainder of this paper is structured as follows. The thee submitted solutions are presented in Section 2 for ATL, Section 3 for Xtend, and Section 4 for AOF. Then, all these solutions are evaluated and compared in Section 5, where we draw some conclusions.

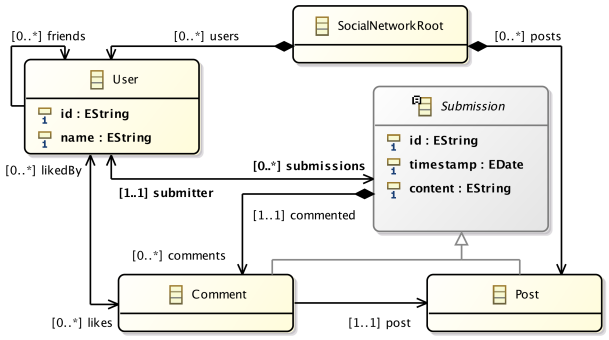

Figure 1: Social Network metamodel.

2 A solution in standard ATL

Our first solution is written as a pure ATL query and executed on the most recent ATL virtual machine (EMFTVM). Queries in ATL are OCL expressions, that can call helper OCL functions and libraries. Thus, this solution includes a complete encoding of the case study as declarative and functional OCL code.

The main objective of the solution is maximizing conciseness and readability. The whole program comprises only 13 lines (Q1) + 31 lines (Q2) = 44 lines. The execution in standard ATL is not incremental, but it has good batch performance since both queries are implemented by algorithms with linear complexity.

The full code for Q1 is presented in Listing 1. The recursive `allComments` helper gathers the set of comments for a given post, and a score for the post is computed by the given formula (line 11) considering the number of comments and likes to the post. The main query `topPosts` sorts the set of posts by score (and timestamp) and picks the top three posts.

Listing 1: Q1 in ATL

```
1  query topPosts = SN!Post.allInstances()->sortedBy(e | e.timestamp)->sortedBy(e | e.score)->reverse()
2    ->subSequence(1, 3);
3
4  helper context SN!Submission def : allComments : Sequence(SN!Comment) =
5    self.comments->union(self.comments->collect(e | e.allComments)->flatten());
6
7  helper context SN!Post def : countLikes : Integer =
8    self.allComments->collect(e| e.likedBy.size())->sum();
9
10 helper context SN!Post def : score : Integer =
11   10*self.allComments->size() + self.countLikes;
```

The code for Q2 is shown in Listing 2. In particular, the `allComponents` helper implements a one-pass algorithm for the detection of all the connected components. The algorithm iterates on the likers: if the liker has not been visited then compute a new component by the `allFriends` function. The `allFriends` helper (whose implementation is not shown in the listing) is just a standard depth-first traversal, limited to the subgraph `s`. Finally a score is computed for each comment (line 5), and the top three comments are identified similarly to Q1 (lines 1-2).

Listing 2: Q2 in ATL

```
1  query topComments = SN!Comment.allInstances()->sortedBy(e | e.timestamp)->sortedBy(e | e.score)
2    ->reverse()->subSequence(1, 3);
3
4  helper context SN!Comment def : score : Integer =
5    self.allComponents->collect(c | c.size()*c.size())->sum();
6
7  helper def : allFriends(u: SN!User, s:Sequence(SN!User)) : TupleType(component : Sequence(SN!User),
8    remaining : Sequence(SN!User)) = ...
```

```
9
10 helper context SN!Comment def : allComponents : Sequence(Sequence(SN!User)) =
11   self.likedBy->iterate(u;
12     acc : TupleType(components : Sequence(Sequence(SN!User)), visited : Sequence(SN!User)) =
13       Tuple{components=Sequence{}, visited=Sequence{}} |
14     if (acc.visited->includes(u))
15       then acc
16       else let component : TupleType(component : Sequence(SN!User), remaining : Sequence(SN!User)) =
17         thisModule.allFriends(u, self.likedBy->excluding(acc.visited)).component in
18           Tuple{components = acc.components.append(component), visited =
19       acc.visited->union(component)}
20     endif).components;
```

The solution being completely declarative, some degree of implicit incrementality can be added by switching to an incremental execution engine, without requiring to modify the user code. Incremental engines for ATL exist [JT10, MTD17] and perform an on-demand activation of transformation rules. However they do not incrementally update the computation of OCL expressions. Hence, they would not have an impact on pure queries like the one we present. In Section 5 we show the performance of this ATL code on an experimental execution mode for ATL that leverages Active Operations for incrementality (see Section 4).

3 A batch solution in Xtend

In parallel with the ATL solution, an implementation of both queries has been made in Xtend[3], a modern Java dialect suited for rapid prototyping thanks to its flexibility and expressiveness. Results obtained with this solution have been used to validate results of both ATL and AOF solutions.

We have written a first batch implementation of Q1 and Q2 (i.e., without incrementality) in pure Xtend, using the Eclipse Modeling Framework (EMF) plugin to perform loading and navigation into models. In a second step, we optimize this solution using Java 8 Streams to parallelize some operations on collections. The Xtend code used for the implementation of Q1 (Listing 3) shows that this mechanism is used two times: (1) to process all posts in parallel, and (2) to compute the sum of all likes received by comments of a post in the `computeScore` method. For better performance, we have also implemented a specific Stream operation, called `Greatest3`, to avoid sorting the whole list of posts while only the top 3 posts can be considered.

The code of Q2 is similar to the implementation of Q1 except for the `computeScore` method. Indeed, the second query requires to find connected groups of users through the friend relationship. For this purpose, the `computeScore` method uses a connected components algorithm based on Tarjan algorithm [Tar72].

Listing 3: Q1 in Xtend with Java Streams

```
1 def private queryQ1() {
2   return socialNetwork.posts.parallelStream.collect(Collector.of([
3     new Greatest3(
4       Comparator.comparingInt[
5         if(it === null) { Integer.MIN_VALUE } else { computeScore}
6       ].thenComparing(Comparator.comparing[timestamp])
7     )
8   ], [$0.add($1)], [$0.merge($1)], [asList])).map[id].join("|")
9 }
10 def private computeScore(Post p) {
11   val comments = p.eAllContents.filter(Comment).toList
12   return comments.size*COMMENT_SCORE + comments.parallelStream.mapToInt[likedBy.size].sum*LIKE_SCORE
13 }
```

4 An incremental solution in AOF

Active operations [BBBJ10] are OCL-like operations such as `collect`, `select`, etc. equipped with incremental propagation algorithms. Each operation is able to perform an initial computation, and then to update its result when its source changes (and *vice versa* when possible). Furthermore, it is possible to build complex incremental expressions by composing active operations. They may thus be used to incrementally evaluate OCL expressions [BCD+14, Section 5] such as found in ATL-like model transformations. It is therefore possible to use active operations to write incremental queries and transformations.

The AOF implementation [JB15] of active operations is based on observation, and notably supports EMF models. It is implemented in Java, and can be used from Java or Xtend code. Each mutable value is wrapped in an observable box, which is either a collection, or a singleton value. Each active operation observes its source box, and updates its target box upon changes by applying its propagation algorithm.

[3] `https://www.eclipse.org/xtend/`

AOF provides enough basic active operations to implement the case study. However, building complex queries out of basic operations does not always guarantee scalability. We observed that creating specific operations sometimes helps [JB16]. Listing 4 shows how AOF can be used in Xtend to implement the first query. The code corresponding to the second query in given in Listing 5. For this case study, we developed four new operations:

1. **sortedBy** (see line 2 in Listing 4, and line 2 in Listing 5) returns a sorted copy of its source collection using one or more criteria. This is a standard OCL operation for which AOF does not have a specific implementation yet. We implemented this operation around a balanced binary tree, which makes it possible to have a logarithmic change propagation time.

2. **take** (see line 3 in Listing 4, and line 3 in Listing 5) returns the n first elements of a collection.

3. **allContents** (see line 8 in Listing 4, and line 2 in Listing 5) retrieves all model elements contained in a given source element, filtering them by type. This is not a standard OCL operation but is rather a kind of mix between `closure` applied on the contents of an element, and `select`. This operation can be implemented relatively efficiently on observable EMF models, which already provide access to all transitively contained elements of a given element.

4. **layering** (see line 8 in Listing 5) implements an incremental connected component algorithm.

The first two operations (i.e., `sortedBy`, and `take`) are relatively generic, and may ultimately be integrated into AOF. `allContents` is not a basic operation, but should prove useful in other transformations. Finally, `layering` is more specific to some graph-related transformations.

Listing 4: Q1 in Xtend using AOF

```
1  def private queryQ1() {
2    return socialNetwork._posts.sortedBy([computeScore], [_timestamp.asOne(null)])
3      .take(3).collect[id]
4  }
5  val scroreByPost = new HashMap<Post, IOne<Integer>>
6  def private computeScore(Post p) {
7    return scoreByPost.get(p) ?: {
8      val comments = p._allContents(Comment)
9      val score = comments.size * COMMENT_SCORE + comments.likedBy.size.sum * LIKE_SCORE
10     val r = score.asOne(0)
11     scoreByPost.put(p, r)
12     r
13   }
14 }
```

Listing 5: Q2 in Xtend using AOF

```
1  def private queryQ2() {
2    return socialNetwork._allContents(Comment).sortedBy([computeScore], [_timestamp.asOne(null)])
3      .take(3).collect[id]
4  }
5  val scroreByComment = new HashMap<Comment, IOne<Integer>>
6  def computeScore(Comment c) {
7    return scoreByComment.get(c) ?: {
8      val s = c._likedBy.layering[u |
9        u._friends.selectMutable[f | f._likes.select[it == c].notEmpty]
10     ].collectMutable[it?.size?.square ?: emptyOne].sum
11     scoreByComment.put(c, s)
12     s
13   }
14 }
```

5 Evaluation and Conclusion

Figure 2 shows performance of the AOF solution versus NMF[4]. Our two best performing solutions are represented in Figure 2: the AOF solution (Section 4), and the ATL solution (Section 2) on an AOF-powered execution mode. The latter is a still-experimental recent development, which enables incremental evaluation of ATL by leveraging AOF. It basically behaves similarly to the AOF solution. However, it requires simpler ATL code than what is presented in Section 2. For instance, this incremental ATL solution reuses the AOF-compatible connected component algorithm mentioned in Section 4. The Xtend solution with Stream optimization shows suitable

[4]Some scalability issues of our AOF solution have been solved since the submission to the live competition.

results for the initial computation but, as a batch solution, it lacks performance on performing changes. The ATL solution on the standard engine is relatively slow, and not incremental. AOF seems to be slightly more efficient than NMF on Figure 2 but the slight differences may be in part due to the fact that measures are performed differently for .Net-based NMF and Java-based AOF. The very last high update time measure for NMF is likely due to a too small heap size requiring too much garbage collection.

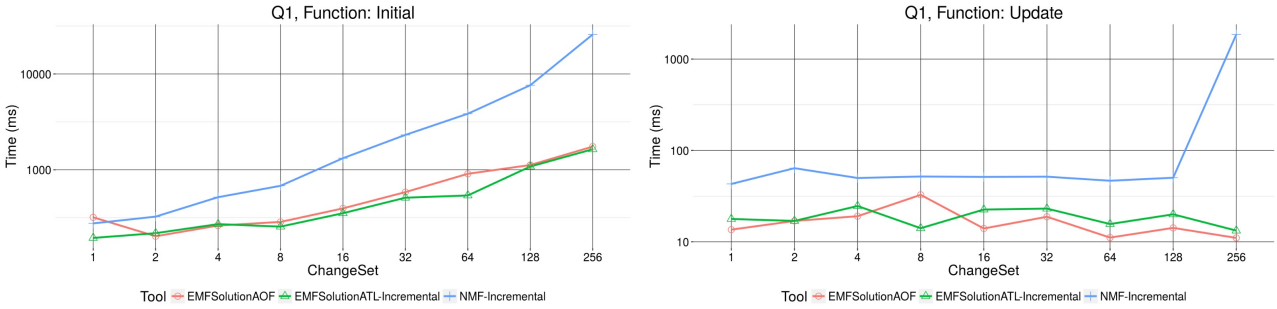

Figure 2: Initial computation of Q1 query: NMF vs. AOF vs. AOF-powered Incremental-ATL

In conclusion, the pure ATL solution maximizes conciseness and readability while the pure AOF solution optimizes incremental performance. The AOF-powered backend for ATL aims to jointly address the two dimensions, and the experimental results in figures are promising. We plan in future work to extend this initial prototype to a full fledged solution for efficient incremental execution of declarative query and transformation code.

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.

[BCD+14] Achim D. Brucker, Tony Clark, Carolina Dania, Geri Georg, Martin Gogolla, Frédéric Jouault, Ernest Teniente, and Burkhart Wolff. Panel discussion: Proposals for improving OCL. In *Proceedings of the 14th International Workshop on OCL and Textual Modelling*, volume 1285 of *CEUR Workshop Proceedings*, pages 83–99, 2014.

[Hin18] Georg Hinkel. The TTC 2018 Social Media Case. *Transformation Tools Contest 2018*, 2018.

[JB15] Frédéric Jouault and Olivier Beaudoux. On the Use of Active Operations for Incremental Bidirectional Evaluation of OCL. In *Proceedings of the 15th International Workshop on OCL and Textual Modeling*, volume 1512 of *CEUR Workshop Proceedings*, pages 35–45, Ottawa, Canada, September 2015.

[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.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proc. of the Model Transformations in Practice Workshop at MoDELS 2005*, volume Satellite, pages 128–138. Springer, 2005.

[JT10] Frédéric Jouault and Massimo Tisi. Towards incremental execution of ATL transformations. In *Theory and Practice of Model Transformations*, pages 123–137. Springer, 2010.

[MTD17] Salvador Martínez, Massimo Tisi, and Rémi Douence. Reactive model transformation with ATL. *Science of Computer Programming*, 136:1–16, 2017.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.