Solving the Movie Database Case with QVTo

Christopher Gerking and Christian Heinzemann

Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn,
Zukunftsmeile 1, 33102 Paderborn, Germany
{christopher.gerking|c.heinzemann}@uni-paderborn.de

Abstract. This paper proposes a solution to the IMDb movie database case of the Transformation Tool Contest 2014. Our solution is based on the Eclipse implementation of the OMG standard QVTo. We implemented all of the tasks including all of the extension tasks. Our benchmark results show that QVTo is able to handle models with a few thousand objects.

Keywords: Transformation Tool Contest, Model Transformation, QVT Operational

1 Introduction

This paper proposes a solution to the movie database case [4] of the Transformation Tool Contest 2014. The objective of the movie database case is deriving a set of performance results that indicate the ability of model transformation languages of processing large models with millions of objects. The case study is based on the IMDb movie database [1] that stores information about movies, actors, actresses, and movie ratings.

We use QVT Operational Mappings (QVTo, [5]) for implementing the different parts of the movie database case. QVTo is a textual, imperative model transformation language that is standardized by the OMG. It is based on MOF [6] and OCL [7]. Therefore, it natively supports metamodels specified in EMF [8] such as the provided IMDB metamodel. In particular, we rely on the Eclipse implementation of QVTo that is part of the Eclipse Modeling tools[1].

The Eclipse implementation of the QVTo standard is open source and already widely used in open-source and academical projects. It is used, for example, within the Graphical Modeling Framework (GMF[2]) and in the Papyrus project[3]. Recently, it has been used for translating software design models to verification models [2] and for generating operational behavior specification out of declarative ones [3].

In our implementation, we created seven transformations for solving the movie database cases. We implemented the three main tasks including all of

[1] http://projects.eclipse.org/projects/modeling.mmt.qvt-oml
[2] http://eclipse.org/gmf-tooling/
[3] https://www.eclipse.org/papyrus/

the extensions tasks. Our implementation demonstrates that QVTo enables a concise specification of the solutions. Five out of seven tasks require less than 30 lines of code. Our benchmark results show that the Eclipse implementation of QVTo is currently able to handle input models with a few thousand objects in reasonable time.

The paper is structured as follows. We first briefly review the movie database case in Section 2 and QVTo in Section 3. Thereafter, Section 4 describes our solution that we implemented in QVTo. We provide benchmark results concerning runtime of our transformation in Section 5 before concluding the paper in Section 6.

2 The Movie Database Case

The movie database case is based on the metamodel shown in Figure 1. It provides classes for Movies, Actors, and Actresses while the latter two inherit from Person. In addition, the metamodel specifies Groups which may be Couples or Cliques. A clique consists of at least n persons that played together in at least 3 movies while $n \geq 3$. A couple is a clique with $n = 2$, i.e., two persons who played together in at least 3 movies.

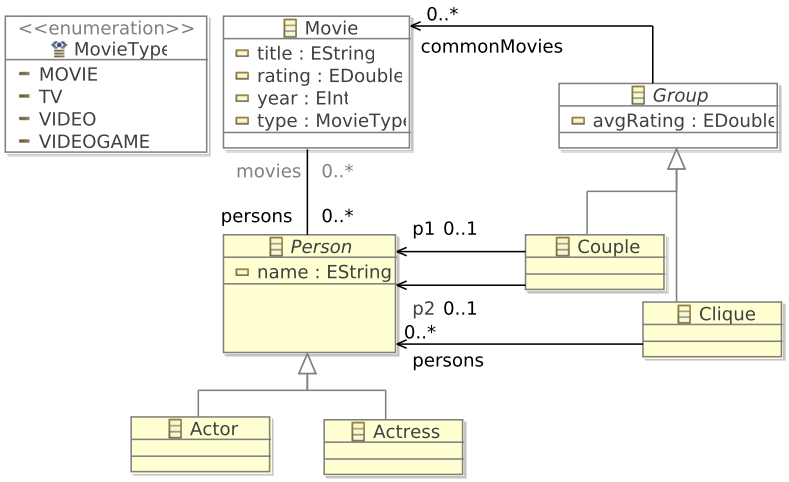

Fig. 1. Metamodel for the Movie Database Case [4]

The case study requires to generate a synthetic set of test data based on the metamodel shown in Figure 1. In addition, it requires to provide a number of queries for computing couples and cliques including average movie ratings for them. Furthermore, the computed couples and cliques need to be sorted for

obtaining the top 15 with respect to the average movie ratings of their common movies and the number of common movies [4].

3 QVT Operational

QVT Operational (QVTo, [5]) is a textual, imperative language for defining unidirectional model-to-model transformations. The current Eclipse implementation of QVTo natively supports the specification of model transformations based on EMF metamodels such as the metamodel shown in Figure 1. Since QVTo is an imperative extension of the OCL [7], the Eclipse implementation also provides access to numerous OCL operations that enable to build collections (e.g., sets) of objects.

A QVTo transformation defines one or more input metamodels and one or more output metamodels. Then, the transformation transforms instances of the input metamodels to instances of the output metamodels. By defining metamodels as `inout`, QVTo enables inplace transformations where the input model is modified. Each transformation has a name and a unique entry point denoted by `main()`. Using so called *configuration properties*, QVTo supports the parametrization of transformations by means of primitive data types.

The transformation itself consists of a set of mappings, queries, helpers, and constructors. A *mapping* translates an object of an input model to an object of an output model. *Queries* derive information from one of the models without modifying it. They may be defined for the transformation itself or for one of the classes of the input or output metamodels. A *helper* may be used to perform auxiliary computations but also for creating additional objects in the output model. Finally, *constructors* enable to provide explicitly constructors with parameters for classes in the output metamodel.

4 Solution

In the following, we present our solutions to the tasks that were set as part of the IMDb database case. For each of the tasks, we provide QVTo model transformation operating on instances of the IMDb metamodel. All design decisions made were influenced by the intention to reuse built-in QVTo functionality as much as possible. Thus, by reusing native language features instead of manual implementation, we focus on keeping the solutions concise with respect to the size of the transformation scripts.

4.1 Task 1: Generating Test Data

For the generation of test data, we developed a QVTo transformation with a single model parameter that corresponds to the IMDb model to be created. The model parameter named `imdb` is declared as `out` in order to reflect the circumstance that the transformation generates a new model instead of reading or manipulating an existing one:

```
transformation Task1(out imdb : IMDb);
```

We declare the transformation parameter N using a QVTo `configuration property`. Thus, a desired value for N may be specified along with the invocation of the transformation:

```
configuration property N : Integer;
```

The implementation of our transformation reflects the structure of the given Henshin specification in terms of imperative operation calls. Thus, the Henshin units *createExample*, *createTest*, *createPositive*, and *createNegative* correspond to dedicated `helper` operations parametrized by means of integer values. The operations `createPositive` and `createNegative` generate the actual test data. For this purpose, they invoke dedicated `constructor` operations as another imperative operation type introduced by QVTo. Our solution specifies constructors for the types `Movie`, `Actor`, and `Actress` which enable the instantiation of the required model elements. After the instantiation, we assign the appropriate set of movies to the `movies` feature of each created person. In the listing below, we provide the entire implementation of the `createPositive` operation as an example:

```
helper createPositive(n : Integer) {

    var movie1 = new Movie(10*n);
    var movie2 = new Movie(10*n + 1);
    var movie3 = new Movie(10*n + 2);
    var movie4 = new Movie(10*n + 3);
    var movie5 = new Movie(10*n + 4);

    var a : Person = new Actor("a" + (10*n).toString());
    var b : Person = new Actor("a" + (10*n + 1).toString());
    var c : Person = new Actor("a" + (10*n + 2).toString());
    var d : Person = new Actress("a" + (10*n + 3).toString());
    var e : Person = new Actress("a" + (10*n + 4).toString());

    a.movies += Set{ movie1, movie2, movie3, movie4 };
    b.movies += Set{ movie1, movie2, movie3, movie4 };
    c.movies += Set{ movie2, movie3, movie4 };
    d.movies += Set{ movie2, movie3, movie4, movie5 };
    e.movies += Set{ movie2, movie3, movie4, movie5 };

    return null;
}
```

4.2 Task 2: Finding Couples

Unlike our solution described in Section 4.1, the `imdb` model parameter to the transformation for this task is declared as `inout`:

```
transformation Task2(inout imdb : IMDb);
```

This declaration implies that the transformation accepts an existing input model, and will output the same model after manipulating in in a certain sense. The required manipulation for this task is the addition of `Couple` elements referring to all pairs of persons who played together in at least three movies. In order to detect the set of couples, our approach is to traverse all potential pairs by means of a for-loop with two iterator variables, iterating over all pairs of persons. QVTo allows to obtain the set of all persons by means of its built-in `objectsOfType` operation:

```
var persons = imdb.objectsOfType(Person);
```

For a given model, this operation retrieves all existing element instances of a certain type (`Person` in our case). While iterating over all potential pairs of persons, we create a couple element for every unique pair that is a valid couple, i.e., has at least three common movies. Thus, we need to ensure uniqueness and validity of the couples created.

Uniqueness implies that no couple should be created when another couple referring to the same two persons already exists (regardless of the order in which the two persons are referred). To achieve uniqueness, we exploit one of the basic language features of QVTo in terms of operational mappings. An operational mapping is an imperative operation that behaves according to a partial mathematical function, i.e., maps each input to at most one output. Thus, the first invocation of a mapping with a certain input will potentially create the appropriate result. However, invoking a mapping again with an equal input will not produce another result, but return the cached result of the prior invocation.

Our solution comprises an operational mapping named `createCouple` that generates a couple for every distinct input pair with at least three common movies. To exploit the mapping behavior for the creation of unique couples, we need to ensure that every two pairs referring to the same persons are regarded as equal. Hence, before invoking the operational mapping `createCouple`, we convert the two persons into an OCL `Set` that serves as the actual mapping input:

```
mapping createCouple(persons : Set(Person)) : Couple
```

The equality behavior of the OCL `Set` type ensures that two sets compare equal if they refer to the same elements, regardless of the internal element ordering. Thus, when invoking the `createCouple` mapping again for the same two persons ordered differently, the conversion to a set ensures equality to the input mapped before. Therefore, instead of creating a new couple, the mapping will just return the existing couple, ensuring the required uniqueness of all couples created.

The remaining challenge is to detect the set of couples which are valid in the sense of this task, i.e., comprise at least three common movies. Checking this precondition for a given pair of persons requires to obtain the set of common movies first. We achieve this by intersecting the two sets of movies that each of

the persons played in. Again, we exploit built-in QVTo functionality in terms of the `intersection` operation defined for OCL's `OrderedSet` type:

```
commonMovies := p1.movies->intersection(p2.movies);
```

The actual evaluation of the number of common movies is carried out inside a dedicated query operation named `isValidCouple`, which obtains the movie intersection as described and returns true if and only if the number of common movies is at least three. In order to make this number mandatory for any valid couple, we invoke the `isValidCouple` operation from inside the `when` clause of the `createCouple` mapping. This causes the mapping to execute (and generate a new result) only if the query return true, i.e., the input pair of persons has at least three common movies.

```
mapping createCouple(persons : Set(Person)) : Couple
    when {persons->isValidCouple()}
```

4.3 Task 3: Computing Average Rankings

Similar to the transformation described in Section 4.2, our solution to this task is based on a transformation with a single `inout` model parameter. The task is to compute and store the average rating of all common movies for each existing couple. Since our transformation is independent from the solution to Task 2, we need to obtain the set of existing couples first. Again, we carry out this task using QVTo's built-in `objectOfType` operation to obtain the set of `Couple` elements inside the given model:

```
var couples = imdb.objectsOfType(Couple);
```

Subsequently, we traverse the obtained set of couples by means of an imperative `forEach` loop. During each iteration, we compute and store the average rating for one of the detected couples. To obtain the sum of ratings for the common movies, we use the `sum` operation defined for an arbitrary OCL collection. Given the sum of ratings, we compute the arithmetic mean by simply dividing the sum by the number of movies, assigning the result to the `avgRating` feature of the respective couple.

```
couples->forEach(couple) {
    couple.avgRating :=
        couple.commonMovies.rating->sum() /
        couple.commonMovies->size();
};
```

4.4 Extension Task 1: Compute Top-15 Couples

Unlike the prior tasks, this task is not about model transformation as such, but rather requires to query information from a given model. In particular, the challenge is to query the top-15 couple elements according to their average

ratings and their number of common movies. Hence, our QVTo-based solution relies on an `imdb` model parameter declared as `in` only:

```
transformation ExtensionTask1(in imdb : IMDb);
```

After retrieving the existing `Couple` elements using QVTo's `objectsOfType` operation, we sort the couples as required. We carry out the sorting by means of the predefined `sortedBy` operation available on OCL collections, using the `avgRating` feature and respectively the size of the `commonMovies` feature as the sorting criteria. The listing below illustrates the sorting by average rating:

```
var sorted = couples->sortedBy(-avgRating);
```

Based on the sorted sequence of couples, we iterate over first 15 elements and print out the desired information about each couple using QVTo's `log` operation.

4.5 Extension Task 2: Finding Cliques

Our solution to this extension task comprises a QVTo `configuration property` named `n` that represents the desired size of the cliques to be obtained:

```
configuration property n : Integer;
```

The major challenge in comparison to Task 2 is to retrieve the sets of persons with the desired size n in order to check each of these sets for being a valid clique with at least three common movies. Since n is not fixed to a certain value (such as two for Task 2), it is not possible to solve this problem using a fixed number of iterator variables.

Instead, we construct the required sets of n persons using an incremental approach. Starting with the empty set, we iterate over every person in the input `imdb` model and create new sets by adding the current person to each of the sets already created before. We avoid the construction of duplicates by storing the obtained sets of persons inside another OCL `Set`, which does not allow duplicates per default. We illustrate the declaration of this set below, initializing it by means of an empty set of persons:

```
var sets : Set(Set(Person)) = Set{Set{}};
```

In order to save runtime, we evaluate any constructed set for being a valid clique on the fly. This means that we discard every constructed set if the number of common movies goes below three, because no valid extension to a clique with three or more common movies exists for such a set. In addition, our solution does not construct sets with more than n persons, which would be an evitable overhead. After constructing all clique sets consisting of n persons with at least three common movies, we create an appropriate `Clique` instance for each of these sets.

4.6 Extension Task 3: Computing Average Rankings for Cliques

Our solution to this extension task is almost identical to the approach described in Section 4.3. The only difference is that we obtain the set of `Clique` elements instead of `Couple` elements, before computing and storing the average rating for each of these as described before.

4.7 Extension Task 4: Compute Top-15 Cliques

The contributed solution for this task is only an adjustment of the approach described in Section 4.4. The difference is that we obtain the set of `Clique` rather than `Couple` elements, and slightly extended the printing routine for cliques in comparison to the one for used couples.

5 Evaluation and Benchmarks

In this section, we evaluate and discuss the applicability of QVTo to complex transformation tasks such as the IMDb movie database case. In particular, we are interested in how far QVTo's conciseness relates to its runtime performance. In Table 1, we present the size of our solutions in terms of the underlying source lines of code, as well as the measured transformation runtime from invocation to termination. The performance testing was carried out on a quad-core 2,2 GHz machine with 8 GB of main memory. Our measurements are based on the parameter values *N=100* for the size of the input data, and *n=2* for the size of cliques to be detected.

Table 1. Evaluation of Conciseness and Performance

Task	**LOC**	**Runtime**
Task 1	60	357ms
Task 2	23	2m 17s
Task 3	8	162ms
Extension Task 1	27	142ms
Extension Task 2	45	2m 57s
Extension Task 3	8	181ms
Extension Task 4	36	294ms

Obviously, QVTo's conciseness (reflected by the small number of source code lines) is out of proportion to the measured runtime. In case of complex challenges such as Task 2 oder Extension Task 2, QVTo did not provide an acceptable transformation runtime. Focusing on these critical tasks in particular, the detected performance limitations are traceable to QVTo's missing native support for the construction of subsets of model elements (which is required to cover all potential cliques). Compared to QVTo as an imperative language, declarative approaches

might achieve considerable runtime improvements by obtaining all possible sets using nondeterministic matching techniques.

As another limitation, QVTo as a dedicated model transformation language does not provide a broader scope of actions when it comes to performance tweaks. In contrast, the usage of general-purpose languages such as Java gives rise to specific implementational variations that could drastically improve the performance.

6 Conclusions

This paper presents a solution to the movie database case of the transformation tool contest 2014 based on QVTo. Our implementation relies on the open-source Eclipse implementation of QVTo. Our results show that QVTo enables for a concise specification of the required queries. However, QVTo is only able to handle test models based on IMBD with a few thousand objects in reasonable time, but not test models with a million objects.

Based on the aforementioned experiences made with QVTo, we propose future work in two different directions. On the one hand, it might be a promising effort to improve the QVTo interpreter towards more efficient implementations of the built-in language features. On the other hand, QVTo's lack of native support for the construction of subsets of model element might be subject to further investigation as well. Extending the QVT specification [5] by new language features is a reasonable option, provided that the added features enable more efficient implementations or do even lead to further improvements in terms of code conciseness and readability.

References

1. Internet movie database (IMDB). Alternative interfaces: http://www.imdb.com/interfaces
2. Gerking, C.: Transparent Uppaal-based Verifcation of MechatronicUML Models. Master's thesis, University of Paderborn (May 2013)
3. Heinzemann, C., Becker, S.: Executing reconfigurations in hierarchical component architectures. In: Proceedings of the 16th international ACM Sigsoft symposium on Component based software engineering. pp. 3–12. CBSE '13, ACM, New York, NY, USA (Jun 2013)
4. Horn, T., Krause, C., Tichy, M.: The TTC 2014 movie database case. In: Transformation Tool Contest. TTC'14 (2014)
5. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/-Transformation, v1.1 (Jan 2011), http://www.omg.org/spec/QVT/1.1/, document formal/2011-01-01
6. Object Management Group: OMG Meta Object Facility (MOF) Core Specification (Jun 2011), http://www.omg.org/spec/MOF/2.4.1/, document formal/2013-06-01
7. Object Management Group: Object Constraint Language (OCL) 2.3.1 (Jan 2012), http://www.omg.org/spec/OCL/2.3.1/, document formal/2012-01-01
8. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. The Eclipse Series, Addison-Wesley, 2nd edn. (Dec 2008)