Solving the Movie Database Case with QVTo

Christopher Gerking
Software Engineering Group,
Heinz Nixdorf Institute,
University of Paderborn, Germany
christopher.gerking@uni-paderborn.de

Christian Heinzemann
Fraunhofer IPT,
Project Group Mechatronic Systems Design,
Software Engineering, Paderborn, Germany
christian.heinzemann@ipt.fraunhofer.de

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.

1 Introduction

This paper proposes a solution to the movie database case [3] of the Transformation Tool Contest 2014. The objective of the movie database case is to derive a set of performance results that indicate the ability of model transformation languages to process large models with millions of objects. The case study is based on the IMDb movie database that stores information about movies, actors, actresses, and ratings.

We use QVT Operational Mappings (QVTo, [4]) for solving the different tasks of the movie database case. QVTo is a textual, imperative model transformation language based on OCL [5] that is standardized by the OMG. It natively supports metamodels specified in EMF [6] such as the provided IMDb metamodel. In this paper, we use the QVTo implementation of the Eclipse Model to Model Transformation (MMT) project[1].

The Eclipse implementation of the QVTo standard is open source and already widely used in other 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 [1] and for generating operational behavior specifications out of declarative ones [2].

In our implementation, we created seven transformations for solving the different tasks of the movie database case. We implemented the three main tasks and all of the extension tasks. Our implementation demonstrates that QVTo enables a concise specification of the solutions. Four 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 a reasonable amount of 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 transformations in Section 5 before concluding the paper in Section 6.

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

Submitted to:
TTC 2014

2 The Movie Database Case

The movie database case is based on a simple metamodel for storing movies and the actors who played in these movies [3]. Therefore, it provides the classes Movie, Actor, and Actress. In addition, the metamodel comprises classes to represent a Couple or Clique. A clique consists of n persons that played together in at least 3 movies while $n \geq 2$. A couple is a clique with $n = 2$, i.e., two persons who played together in at least 3 movies.

3 QVT Operational Mappings

QVT Operational Mappings (QVTo, [4]) 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. Since QVTo is an imperative extension of OCL [5], the Eclipse implementation also provides access to numerous OCL operations that enable to build collections (e.g., sets) of objects.

A QVTo transformation refers to one or more input metamodels and one or more output metamodels. Then, a transformation run transforms instances of the input metamodels to instances of the output metamodels. QVTo also enables inplace transformations where one and the same model instance acts as both input and output, enabling model modifications as required for the movie database case. 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.

In our implementation, we use mappings, helpers, and constructors. A *mapping* translates an object of an input model to an object of an output model. A *helper* may be used to perform auxiliary computations but also for creating additional objects in the output model. Finally, *constructors* enable the parametrized instantiation of classes which are part of the output metamodel.

4 Solution

In the following, we present our solutions to the tasks that were given as part of the IMDb movie database case. For each of the tasks, we provide a QVTo model transformation operating on concrete instances of the IMDb metamodel. Our overall design goal is to keep the solutions concise with respect to the transformation size. Thus, we prefer using high-level native language features provided by QVTo, avoiding more complex manual implementations by means of low-level constructs whenever possible.

4.1 Generating Test Data

For the generation of test data, we developed a QVTo transformation with a single IMDb output model. In addition, we declare a transformation parameter *N* using a QVTo *configuration property* such that the resulting amount of test data may be configured along with the invocation of the transformation.

The implementation of our transformation reflects the structure of the given Henshin specification [3] in terms of imperative operation calls. Thus, the given Henshin units correspond to dedicated *helper* operations parametrized by means of integer values. The actual instantiation takes places inside dedicated *constructor* operations for the types Movie, Actor, and Actress.

4.2 Finding Couples

The solution for this task is based on an inplace transformation that accepts an existing IMDb input model and will output the same model after manipulating it. The required manipulation for this task is the addition of Couple elements as defined in Section 2. In order to detect the set of couples, our approach is to traverse all pairs of persons. We achieve this by iterating over the set of persons using an imperative `forEach` loop with two iterator variables. During the iteration, we create a Couple element for every unique pair that is a valid couple:

```
persons->forEach(p1,p2) {
    Set{p1,p2}->map createCouple();
};
```

In order to achieve uniqueness, we must not create a new Couple if an existing one already refers to the same two persons (regardless of their order). Therefore, we use an operational *mapping* operation to generate a Couple for every valid and unique input pair. The *mapping* is declared as follows:

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

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, reinvoking the mapping with an equal input will not produce another result, but return the cached result of the prior invocation. To exploit this mapping behavior for the creation of unique couples, we represent input pairs using the OCL `Set` type. The equality behavior of this built-in collection type ensures that two pairs compare equal if they refer to the same persons regardless of their ordering.

In order to not generate any invalid couples, we check the validity inside a `when` clause of the `createCouple` mapping. This causes a mapping invocation to be skipped whenever the input `Set` is not a pair or has less than three common movies.

4.3 Computing Average Rankings

Task 3 and Extension Task 3 of the IMDb movie database case require to compute the average rankings for couples or cliques. Our solution to this challenge is based on a QVTo inplace transformation. We traverse the set of groups inside the given IMDb model by means of an imperative `forEach` loop. During each iteration, we compute the average rating for one of the detected groups. To obtain the sum of ratings for the common movies, we use the `sum` operation defined for OCL collections. We compute the arithmetic mean by simply dividing the sum of ratings by the number of movies:

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

4.4 Computing the Top-15 Groups

Extension Task 1 and 4 require to query information from the given IMDb model. In particular, the challenge is to query the top-15 couples/cliques according to their average ratings and their number of common movies. Hence, our QVTo-based solutions declare only input and no output models.

In order to obtain the top-15, we sort all existing groups as required. Whereas this approach constitutes a computational overhead since only the top-15 is of interest, it allows for a concise solution. By using the predefined `sortedBy` operation for OCL collections, we avoid more complex manual implementations. The listing below illustrates the sorting of couples by average rating. Based on the sorted

sequence of couples or cliques, we iterate over the first 15 elements and print out the desired information about each group using QVTo's `log` operation.

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

4.5 Finding Cliques

Our solution to Extension Task 2 comprises a QVTo *configuration property* that represents the desired size *n* of the cliques to be obtained. The major challenge in comparison to Task 2 is to retrieve all candidate sets consisting of *n* persons in order to check each of these sets for being a valid clique. Since *n* is not fixed to a certain value (such as *n=2* for Task 2), it is not possible to solve this problem using a fixed number of iterator variables as described in Section 4.2. Instead, we construct the candidate sets explicitly inside a *helper* operation. The signature below illustrates that the operation accepts a set of persons and returns all candidate subsets for cliques:

```
helper Set(Person) :: candidates() : Set(Set(Person))
```

The implementation of the `candidates` operation is based on an incremental approach. Starting with an empty set of persons, we iterate over every given person and create new sets by adding the current person to each of the sets already created before.

In order to save runtime, we evaluate the validity of any constructed set on the fly. This means that we discard a 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. In addition, our solution does not construct sets with more than *n* persons, which would be an evitable overhead. After constructing all clique candidates, we map each valid candidate set to an appropriate Clique instance similar to our solution for Task 2.

5 Evaluation and Benchmarks

In our evaluation, we particularly focus on the relationship between code conciseness and runtime performance for QVTo. Table 1 summarizes the measured runtime for the transformations from invocation to termination, as well as the transformation size in terms of the underlying source lines of code (SLOC). The performance testing was carried out on a quad-core 2,2 GHz machine with 8 GB of main memory running the 3.4.0 release version of Eclipse QVTo. Our measurements are based on the parameter values $N \in \{50, 100, 150, 200, 250, 300, 350, 400\}$ for the size of the synthetic test data. For each N, Table 1 also shows the resulting number of model elements generated in Task 1. Our measurements are based on the size $n = 3$ for the cliques to be detected in Extension Task 2.

Table 1: Evaluation of Conciseness and Performance

	Runtime								**SLOC**
N	50	100	150	200	250	300	350	400	
# Elements	1000	2000	3000	4000	5000	6000	7000	8000	
Task 1	2.2s	0.1s	0.2s	0.2s	0.2s	0.3s	0.3s	0.4s	59
Task 2	4.3s	15.2s	36.6s	63.3s	93.9s	134.1s	179.2s	234.7s	24
Task 3	0.3s	0.1s	0.2s	0.3s	0.4s	0.6s	0.7s	0.9s	8
Ext. Task 1	0.3s	0.1s	0.2s	0.3s	0.4s	0.6s	0.7s	1.0s	28
Ext. Task 2	14.1s	53.6s	122.6s	225.6s	345.5s	487.3s	654.3s	1371.9s	47
Ext. Task 3	0.2s	0.1s	0.3s	0.5s	0.8s	0.9s	1.2s	3.5s	8
Ext. Task 4	0.2s	0.2s	0.3s	0.5s	0.8s	1.0s	1.2s	3.4s	37

Table 1 indicates a superlinear increase for the runtime of complex challenges such as Task 2 or Extension Task 2. Consequently, QVTo is not able to provide an acceptable transformation runtime for realistic models based on the IMDb movie database. Thus, the conciseness enabled by QVTo (reflected by the small number of source code lines) is obviously out of proportion to the measured runtime.

The detected performance limitations are traceable to QVTo's missing native support for the construction of powersets (which is required to generate all candidate sets for couples or cliques). In contrast to QVTo as an imperative language, declarative approaches might achieve considerable runtime improvements by obtaining all possible subsets using nondeterministic matching techniques. Furthermore, QVTo as a dedicated model transformation language does not provide a broader scope of actions when it comes to performance tweaks. In contrast, using general-purpose languages (such as Java) gives rise to specific implementational variations that could drastically improve the performance.

Nevertheless, focusing on the small number of source code lines illustrated in Table 1, our evaluation shows that QVTo enables a concise specification of the transformations for all tasks. Thus, despite the detected performance drawbacks, we regard our major design goal as reached.

6 Conclusions

This paper presents a solution to the movie database case of the Transformation Tool Contest 2014 based on the Eclipse implementation of QVTo. Our results show that QVTo enables for a concise specification of transformations. However, QVTo is only able to handle synthetic test models with a few thousand objects in a reasonable amount of time but not realistic models based on the IMDb movie database.

Our benchmark results indicate two promising directions for future works. First, realizing parts of a QVTo transformation inside a Java *blackbox* [4] is an option to integrate more efficient implementations. We excluded blackboxes in order to keep the focus on plain model-to-model transformations. Second, the QVT/OCL specifications [4, 5] could be extended by missing operations such as computing a power set for Extension Task 2. A promising approach in that direction is the implementation of an extensible standard library for OCL [7]. However, such library extensions are only useful if the additional operations are equipped with an efficient implementation or further improve the code conciseness.

References

[1] Christopher Gerking (2013): *Transparent UPPAAL-based Verifcation of MechatronicUML Models*. Master's thesis, University of Paderborn.

[2] Christian Heinzemann & Steffen Becker (2013): *Executing reconfigurations in hierarchical component architectures*. In Philippe Kruchten, Dimitra Giannakopoulou & Massimo Tivoli, editors: *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering*, ACM, pp. 3–12.

[3] Tassilo Horn, Christian Krause & Matthias Tichy (2014): *The TTC 2014 Movie Database Case*.

[4] Object Management Group (2011): *Meta Object Facility (MOF) 2.0 Query/View/Transformation*. Available at `http://www.omg.org/spec/QVT/1.1/`. Document formal/2011-01-01.

[5] Object Management Group (2012): *Object Constraint Language (OCL) 2.3.1*. Available at `http://www.omg.org/spec/OCL/2.3.1/`. Document formal/2012-01-01.

[6] David Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2008): *EMF: Eclipse Modeling Framework*, 2nd edition. The Eclipse Series, Addison-Wesley.

[7] Edward D. Willink (2011): *Modeling the OCL Standard Library*. Electronic Communications of the EASST 44. Available at `http://journal.ub.tu-berlin.de/eceasst/article/view/663`.