Movie Database Case: An EMF-INCQUERY Solution*

Gábor Szárnyas Oszkár Semeráth Benedek Izsó Csaba Debreceni
Ábel Hegedüs Zoltán Ujhelyi Gábor Bergmann
Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
H-1117 Magyar tudósok krt. 2., Budapest, Hungary
{szarnyas, semerath, izso, debrecenics, abel.hegedus, ujhelyiz, bergmann}@mit.bme.hu

This paper presents a solution for the Movie Database Case of the Transformation Tool Contest 2014, using EMF-INCQUERY and Xtend for implementing the model transformation.

1 Introduction

The use of automated model transformations is a key factor in modern model-driven system engineering. Model transformations allow to query, derive and manipulate large industrial models, including models based on existing systems, e.g. source code models created with reverse engineering techniques. Since such transformations are frequently integrated to modeling environments, they need to feature both high performance and a concise programming interface to support software engineers.

The objective of the EMF-INCQUERY [3] framework is to provide a declarative way to define queries over EMF models without needing to manually define imperative model traversals. EMF-INCQUERY extended the pattern language of VIATRA with new features (including transitive closure, role navigation, match count) and tailored it to EMF models [2]. The semantics of the pattern language is similar to VTCL [7], but the adaptation of the rule language is an ongoing work.

EMF-INCQUERY is developed with a focus on *incremental query evaluation* and uses the same incremental algorithm as VIATRA. The latest developments extend this concept by providing a preliminary rule execution engine to perform transformations. As the engine is under heavy development, the design of a dedicated rule language (instead of using the API of the engine) is currently subject to future work.

Conceptually, the current execution environment provides a method for specifying graph transformations (GT) as rules, where the LHS (left hand side) is defined with declarative EMF-INCQUERY graph patterns, defined in EMF-INCQUERY Pattern Language [2] and the RHS (right hand side) as imperative model manipulations formulated in the Xtend programming language [5]. The rule execution engine is also configured from Xtend code.

One case study of the 2014 Transformation Tool Contest describes a movie database transformation [6]. The main characteristics of the transformation related to the application of EMF-INCQUERY are that i) it only adds new elements to the input model (i.e. couples and groups are created without modifying the input model), and ii) it is non-incremental (i.e. adding a new group with a rule will not affect the applicability of rules).

*This work was partially supported by the MONDO (EU ICT-611125) and TÁMOP (4.2.2.B-10/1–2010-0009) projects. This research was realized in the frames of TÁMOP 4.2.4. A/1-11-1-2012-0001 „National Excellence Program – Elaborating and operating an inland student and researcher personal support system". The project was subsidized by the European Union and co-financed by the European Social Fund.

Submitted to:
TTC 2014

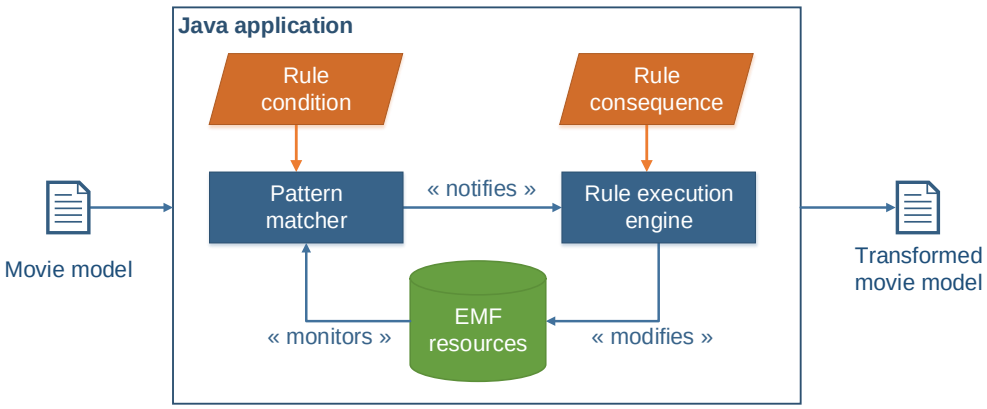

Figure 1: Overview of the specification and runtime.

The rest of the paper is structured as follows: Section 2 gives an overview of the implementation, Section 3 describes the solution including design decisions, benchmark results, and Section 4 concludes our paper.

2 Architecture Overview

The overview of the rule-based solution is illustrated in Figure 1. The input of the transformation is a *movie model*. The result is a *transformed movie model* containing additional model elements, including various groups (couples and *n*-cliques) and their average rating [6]. The transformation runs in a Java application, that uses *pattern matchers* provided by EMF-INCQUERY and model modification code specified in Xtend. The model *modifications* are performed over EMF resources, while the pattern matcher *monitors* the resources to incrementally update match sets. The application initially reads the input movie database resource, creates the output resources (organizing them into a resource set), then executes the transformation, and finally serializes the results into files.

The whole solution is implemented in two languages. Rule conditions are formulated as EMF-INCQUERY graph patterns, while the rule consequences (model manipulations) in Xtend. Since the advanced transformation constructs of EMF-INCQUERY are tailored for event-driven (incremental) execution, the current solution only uses pattern matchers to access query results, without additional constructs.

3 Solution

3.1 Specification

In the provided Ecore model, no containment hierarchy is used and all objects are held in the contents list of the EMF resource. However, this also means that the performance of the transformation can be affected by the resource implementation used (since it will determine the implementation of the list operations).

For performance considerations, we used an extended version of the metamodel, which has a Root

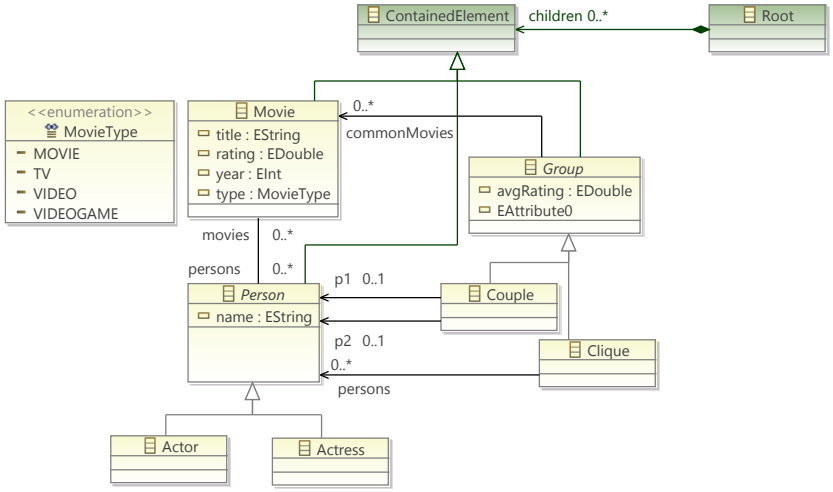

Figure 2: Our extended metamodel.

object (see Figure 2). This object serves as a container for all Group, Movie and Person objects. According to our experiments, this increases the speed of the pattern matching by a factor of two.

We ensured that our solution works with the models provided and also persists outputs in a format that does not include the root element. This way, the transformation part of our solution is independent of resource implementation (binary, UUID-based, regular XMI, etc.).

3.2 Patterns and Transformations

3.2.1 Task 1: Generating Test Data

The synthetic test data is generated in Xtend (see Listing **??**). The code tightly follows the specification defined in the case description [6]. Since the task is simple model construction without any querying, EMF-INCQUERY is not used in this task.

3.2.2 Task 2: Finding Couples

Couples are listed with the following pattern:

```
pattern personsToCouple(p1name, p2name) {
  find cast(p1name, M1); find cast(p2name, M1);
  find cast(p1name, M2); find cast(p2name, M2);
  find cast(p1name, M3); find cast(p2name, M3);

  M1 != M2; M2 != M3; M1 != M3;

  check(p1name < p2name);
}

pattern cast(name, M) {
  Movie.persons.name(M, name);
}

pattern personName(p, pName) {
  Person.name(p, pName);
}
```

Note that the cast pattern returns the names of persons that play in a given movie. This is important since the names of the persons can be used to avoid symmetric matches in the personsToCouple pattern by sorting. The Couple objects are created and configured in Xtend (see createCouples in line **??** of Listing **??**). This includes setting the p1 and p2 references using a personName pattern and computing the commonMovies by simple set intersection operators (retainAll).

3.2.3 Task 3: Computing Average Rankings

The average rankings are computed in Xtend by calculating the mean of the rating attributes of a couple's common movies (see calculateAvgRatings in line **??** of Listing **??**). The movies are enumerated with the following pattern:

```
1 pattern commonMoviesOfCouple(c, m) {
2   Couple.commonMovies(c, m);
3 }
```

3.2.4 Extension Task 1: Compute Top-15 Couples

This task is mostly implemented in Xtend (see topGroupByRating in line **??** and topGroupByCommonMovies in line **??** of Listing **??**), however, it uses the groupSize pattern in order to filter the groups with the particular number of members.

```
1 pattern groupSize(group, S) {
2   Group(group);
3   S == count find memberOfGroup(_, group);
4 }
```

This pattern uses the count find construct which computes the number of matches for a given pattern. Additionally, specific comparators are used to sort and determine the top-15 lists by rating or number of common movies (see Listing **??**).

3.2.5 Extension Task 2: Finding Cliques

The pattern for finding cliques is implemented similarly to the personsToCouple pattern 3.2.2. The pattern for 3-cliques is defined as follows:

```
1  pattern personsTo3Clique(P1, P2, P3) {
2    find cast(P1, M1); find cast(P2, M1); find cast(P3, M1);
3    find cast(P1, M2); find cast(P2, M2); find cast(P3, M2);
4    find cast(P1, M3); find cast(P2, M3); find cast(P3, M3);
5
6    M1 != M2; M2 != M3; M1 != M3;
7
8    check(P1 < P2); check(P2 < P3);
9    check(P1 < P3);
10 }
```

The creation of cliques is done similarly to couples (see createCliques in line **??** of Listing **??**). However, this pattern has a redundant check constraint, as $P_1 < P_2$ and $P_2 < P_3$ already imply $P_1 < P_3$. This works as a hint for the query engine and allows it to filter the permutation of the results (e.g. $(a_2, a_1, a_3), (a_1, a_3, a_2), \ldots$) earlier.

To achieve high query performance, patterns for 4- and 5-cliques are defined manually. For larger cliques ($n > 5$), patterns could be automatically generated using code generation techniques.

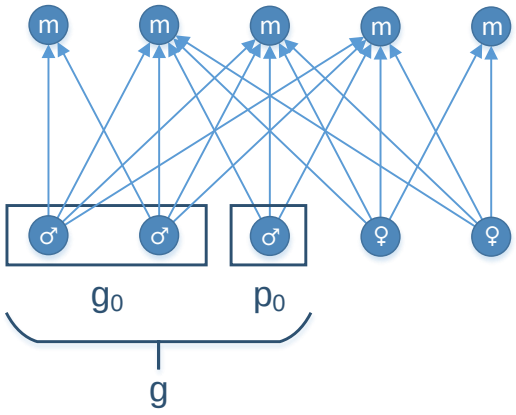

Figure 3: Matching 3-clique groups in the positive test pattern. g_0 is a couple.

General solution for n-cliques. We also provide the outline for a more general solution (for arbitrary n values). For the sake of clarity, we will refer to couples as 2-cliques. In this approach, the cliques are built iteratively. Suppose we already have all k-cliques in the graph (e.g. we already added the 2-, 3-, 4- and 5-cliques with the previous patterns). To get the $(k+1)$-cliques, we look for a group g_0 and a person p_0 that (i) have at least 3 movies in common, (ii) $g = g_0 \cup \{p_0\}$ is a group that is not a subset of any other groups (see Figure 3).

Formally, (ii) can be expressed as $(\nexists g') : g \subseteq g'$. Using $g = g_0 \cup \{p_0\}$, we derive the following expression $(\nexists g') : (g_0 \subseteq g') \wedge (p \in g')$. The $g_0 \subseteq g'$ expression can be formulated as follows: $(\forall p_0 \in g_0) : p_0 \in g'$. As the EMF-IncQuery Pattern Language does not have a universal quantifier, we rewrite this using the existential quantifier: $(\nexists p_0 \in g_0) : p_0 \notin g'$.

The resulting expression for condition (ii) is the following: $(\nexists g') : ((\nexists p_0 \in g_0) : p_0 \notin g') \wedge (p \in g')$. This is equivalent to the following EMF-IncQuery pattern:

```
/**
 * This pattern returns with g0 and gx pairs, where Group g0 is a subset of Group gx.
 */
pattern subsetOfGroup(g0 : Group, gx : Group) {
  neg find notSubsetOfGroup(p0, g0, gx);
}

/**
 * This pattern returns is a helper for the subsetOfGroup pattern.
 */
pattern notSubsetOfGroup(p0 : Person, g0 : Group, gx : Group) {
  find memberOfGroup(p0, g0);
  neg find memberOfGroup(p0, gx);
}

/**
 * This pattern returns p and g pairs, where Person p is a member of Group g.
 * A Group is either a Couple or a Clique.
 */
pattern memberOfGroup(p, g) {
  Couple.p1(g, p);
} or {
  Couple.p2(g, p);
} or {
```

```
  Clique.persons(g, p);
}
```

Based on the subsetOfGroup pattern, we may implement the nextClique pattern like follows:

```
pattern nextCliques(g : Group, p : Person)  {
  neg find alphabeticallyLaterMemberOfGroup(g, p);
  n == count find commonMovieOfGroupAndPerson(g, p, m);
  check(n >= 3);
  neg find union(g, p);
}

pattern alphabeticallyLaterMemberOfGroup(g : Group, p : Person)  {
  find memberOfGroup(m, g);
  Person.name(p, pName);
  Person.name(m, mName);
  check(mName >= pName);
}

pattern commonMovieOfGroupAndPerson(g, p, m) {
  find commonMoviesOfGroup(g, m);
  Person.movies(p, m);
}

pattern commonMoviesOfGroup(g, m) {
  Group.commonMovies(g, m);
}

pattern union(g0, p) {
  find memberOfGroup(p, gx);
  find subsetOfGroup(g0, gx);
}
```

Given a model containing all k-cliques, the nextClique pattern is capable of determining the $(k+1)$-cliques. While this solution is functionally correct, it only works for very small input models and hence is omitted from our implementation.

3.2.6 Extension Task 3: Compute Average Rankings for Cliques

The average rankings are computed the same way as in *task 3* (section 3.2.3).

3.2.7 Extension Task 4: Compute Top-15 Cliques

The top 15 average rankings are computed the same way as in *extension task 2* (section 3.2.5).

3.3 Optimizations

To increase the performance of the transformations, we used some optimizations.

- After the matcher engine produced the initial result set, the engine is turned off. This way, we spare the cost of incrementally maintaining the result set. As the transformation is non-incremental, this does not affect the result of the performance.
- The common movies of the two Person objects are computed from Xtend instead of EMF-INCQUERY.
- The patterns for 3-, 4- and 5-cliques are implemented manually.

We looked for *common subpatterns* and extracted them into separate patterns. This results in better performance as the engine can reuse the pattern for each occurrence, and makes the query definition file easier to maintain. For an example, see the cast pattern in **??**.

3.4 Build Automation

Our solution was developed in the Eclipse IDE. While it is fully functional in Eclipse, it can also be compiled with the Apache Maven [1] build automation tool. This offers a number of benefits, including easy portability and the possibility of continuous integration. The build process uses the Tycho Maven plug-in [4] to build the Eclipse plug-ins defined in the project.

3.5 Benchmark Results

The implementation was benchmarked in the SHARE cloud, on an Ubuntu 12.04 64-bit operating system running in a VirtualBox environment. The virtual machine used one core of an Intel Xeon E5-2650 CPU and had 6 GB of RAM.

The benchmark used Maven to build the binary files. The transformations were ran in a timeout window of 10 minutes.

3.6 Synthetic model

Results are displayed in Figure 4. The diagram shows the transformation times for creating couples and cliques for synthetic models. The results show that the transformations run in near linear time.

The dominating factor of the running time is the initialization of the query engine. However, after initialization, creating groups can be carried out efficiently.

Furthermore, our experiments showed that the limiting factor for our solution is the memory consumption of the incremental query engine. Given more memory, the solution is capable of transforming larger models as well.

3.7 IMDb model

In the given time range and memory constraints, the transformation of the IMDb model could only generate the couples and 3-cliques for the smallest instance model. Finding the couples took 3 minutes, while finding 3-cliques took 6. However, in case of a live and evolving model, our solution is capable of incrementally running the transformation which in practice results in near instantaneous response time.

3.8 Transformation Correctness and Reproducibility

The transformation runs correctly for the provided test cases on SHARE[1], and the source code is also available in a Git repository[2]. The results of the transformations were spot-checked for both synthetic and IMDb models.

3.9 Tool Support for Debugging and Refactoring

As the transformation is written in two languages, debugging and refactoring is dependent on the tooling for these languages and the capabilities of the query engine.

[1] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_TTC14_64bit_TTC14-EIQ-imdb.vdi`

[2] Homepage: `https://git.inf.mit.bme.hu/w?p=projects/viatra/ttc14-eiq.git` (use the `anonymous` user with no password), clone URI: `https://anonymous@git.inf.mit.bme.hu/r/projects/viatra/ttc14-eiq.git`

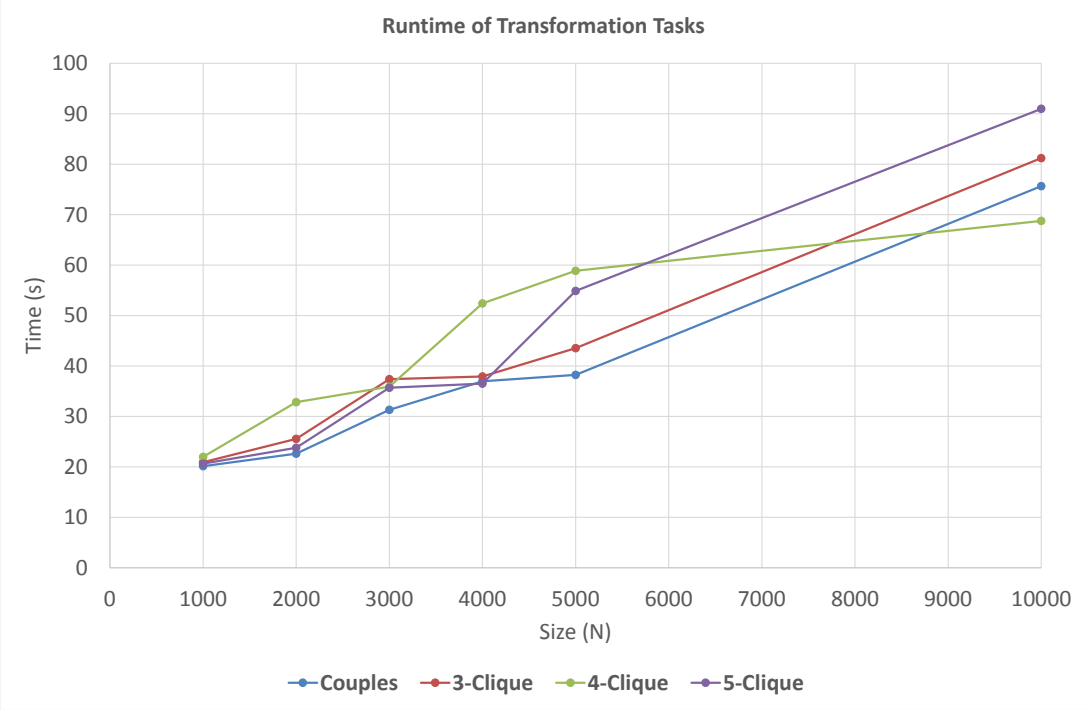

Figure 4: Benchmark results.

Xtend and the EMF-INCQUERY pattern editors are based on Xtext, and while provide some refactoring operations. Declarative EMF-INCQUERY graph patterns cannot be efficiently debugged at runtime. However, smaller instance models can be loaded into the *Query Explorer* view, which is very handy to the debug matches of the patterns. The engine controller code can be debugged and refactored as well, as it is implemented in Java. Debug messages of the execution engine can be turned on, which prints useful messages about rule firings and activations. Firings of the transformation operations can be debugged by placing breakpoints in the Xtend code.

4 Conclusion

In this paper we have presented our implementation of the Movie Database Case. The solution is based on EMF-INCQUERY which is used as a model query engine. Because its dedicated rule language is yet to be implemented, the transformation rules are defined in Xtend.

The transformation is specified using declarative graph pattern queries over EMF models for rule preconditions, and Xtend code which can be executed to obtain the desired effect of the rule.

References

[1] Apache.org (2014): *Maven*. http://maven.apache.org.

[2] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF models*. In: *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Lecture Notes in Computer Science* 6707, Springer, pp. 167–182, doi:10.1007/978-3-642-21732-6_12.

[3] Eclipse.org (2014): *EMF-IncQuery*. http://eclipse.org/incquery/.

[4] Eclipse.org (2014): *Tycho home*. https://www.eclipse.org/tycho/.

[5] Eclipse.org (2014): *Xtend – Modernized Java*. https://www.eclipse.org/xtend/.

[6] Matthias Tichy Tassilo Horn, Christian Krause (2014): *The TTC 2014 Movie Database Case*. In: *7th Transformation Tool Contest (TTC 2014)*, EPTCS.

[7] Dániel Varró & András Balogh (2007): *The model transformation language of the {VIATRA2} framework*. *Science of Computer Programming* 68(3), pp. 214 – 234, doi:http://dx.doi.org/10.1016/j.scico.2007.05.004. Available at http://www.sciencedirect.com/science/article/pii/S016764230700127X. Special Issue on Model Transformation.