TTC'16 Live Contest Case Study

Execution of dataflow-based model transformations

Antonio Garcia-Dominguez

Department of Computer Science, Aston University, Birmingham, UK
a.garcia-dominguez@aston.ac.uk

4 July 2016

Incremental execution of model transformation has received much attention in the past few years, due to the potential speedups in keeping an output model up to date with regards to an input model. Most of the proposed approaches are alternative execution engines for existing languages, or frameworks for programming event-driven transformations.

This live competition proposes a dataflow-driven notation that may be more directly mappable to an incremental strategy. Contestants will implement an execution engine that takes in one of the four example transformations and runs it in batch or incremental mode, using their favorite approach and technology. Solutions will be compared in terms of batch correctness and performance, incremental correctness and performance, and maintainability and extensibility. A reference implementation is made available.

1 Introduction

Model-to-model transformation frameworks and tools have evolved in the recent years to process larger models more efficiently. Traditional *batch* transformation engines took the entire source model(s) and produced the entire target model(s) exactly once. A small change would require going again through the entire model. Alternatively, an *incremental* transformation engine processes only the change itself, updating the target model(s) as needed.

Various approaches for incremental transformations are present in the literature: for instance, ReactiveATL [5] is an alternative execution engine of ATL which only computes and updates results on-demand as the model changes. VIATRA3 is a general purpose framework for reactive transformations [1], in which users write rules that trigger as the model changes. Streaming transformations have been studied by Cuadrado and Lara [2] with the Eclectic tool and its ATL-like language, and by David, Rath, and Varró [3] through complex event processing.

Most of these systems can be seen as complex transformations of rule-based systems into event-driven systems, or frameworks that allow for writing these event-driven systems through Java code. This case suggests studying a type of notation that may be more directly amenable to incremental execution: a graph of model-oriented primitives which generate and process streams of tuples. The notation is inspired on popular Extract-Transform-Load (ETL) tools such as Pentaho Data Integration[1] or Talend Data Integration[2]. As data integration can be considered a form of model transformation, perhaps there might be lessons to learn from these languages. In addition to their potentially simpler incrementality, breaking rules into smaller primitives might increase the level of detail of the execution traces of the transformation.

This case study presents an initial and simplified version of such a notation, called *FlowM2M*, with primitives subdivided into "minimal" and "extended" sets. Participants are tasked with writing an execution engine using their favorite approach (e.g. code generation or model interpretation) and tool, which may support batch and/or incremental transformations.

All the resources are included in the official Github repository[3]. These resources are mentioned in Section 2. The evaluation criteria for the provided solutions are described in Section 3. Contestants are invited to raise questions through GitHub issues should there be any unclear points in the description.

2 Case description

The present case study requires contestants to use their favorite technology to write a batch or incremental execution engine for a dataflow-oriented model-to-model transformation language. The transformation language has been designed with a simplistic syntax and a limited set of primitives, in order to be feasible for the time available during TTC.

The first part of the description is dedicated to introducing the language. Section 2.1 presents the abstract syntax of the language and describes the intended semantics for its various primitives. Section 2.2 summarizes the concrete syntaxes for the language through an example: these include a HUTN-like simple textual notation and a boxes-and-arrows graphical notation.

After this, two tasks are defined: one is to support the most basic primitives so the simple example shown in Section 2.2 can run. The other task is to support the rest of the primitives and enable a more complex transformation to be run.

2.1 Abstract syntax

The abstract syntax of the language has been implemented as an Ecore metamodel, which is available on the Github repository. Broadly speaking, it is based on a graph of primitives (shown in Figure 1) which may contain expression trees (Figure 2).

[1] `http://community.pentaho.com/projects/data-integration/`
[2] `https://www.talend.com/products/data-integration`
[3] `https://github.com/TransformationToolContest/ttc16-live-contest`

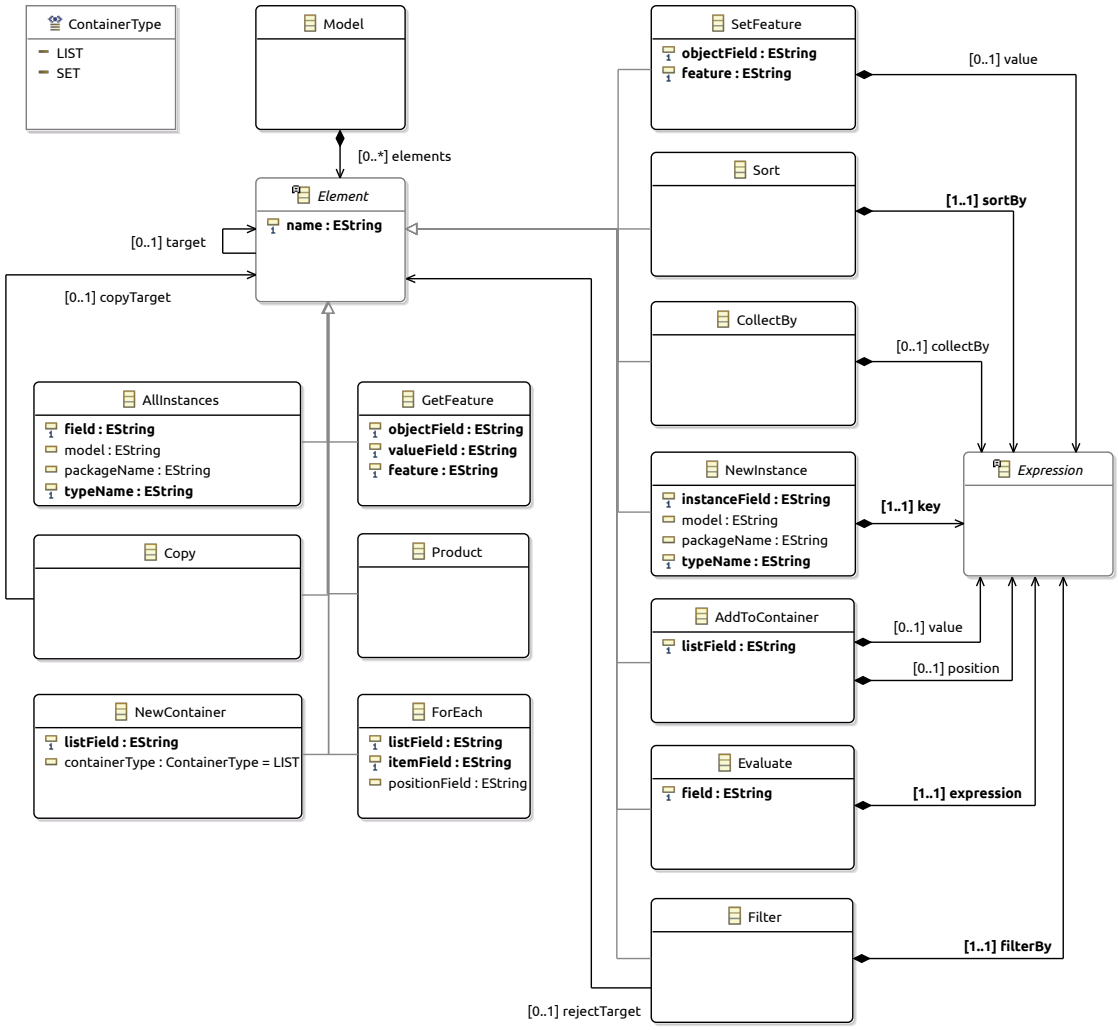

Figure 1: Abstract syntax: primitives

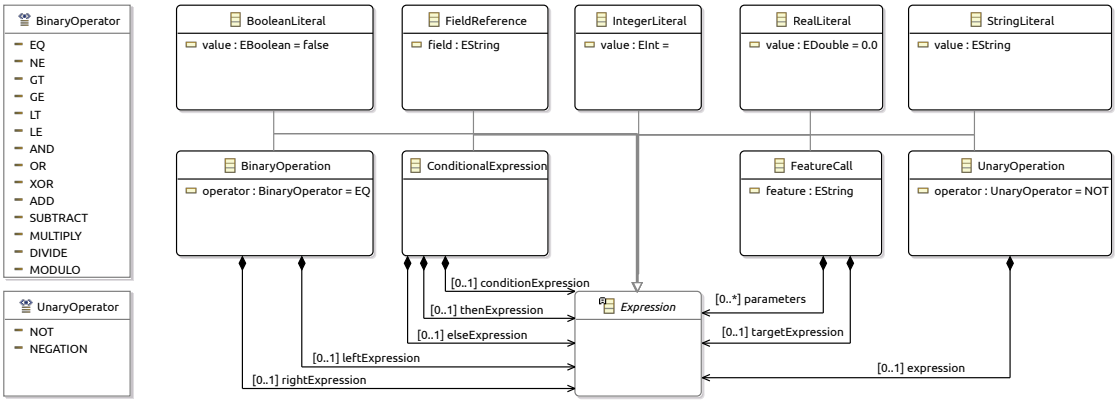

Figure 2: Abstract syntax: expressions

2.1.1 Primitives

As shown in Figure 1, a transformation is defined as a MODEL that contains a collection of connected ELEMENTs. These generally take a sequence of "rows" (a collection of key/value pairings) and process them in some way, forwarding the resulting tuples to the *target* ELEMENT. Some elements have more than one possible target in order to allow for more advanced behaviour (e.g. FILTER can split tuples into two streams according to a condition).

The primitives can be divided into several groups. First, there are the object handling primitives:

- ALLINSTANCES (needed for tasks 1 and 2): for each input row r, it produces one new row r' for each instance of the specified type within the specified package and model, where the specified field will be set to that instance. Package and model are optional.

  If this step has no incoming edges, it will act as it received a single empty row, producing one row per instance of the specified type from scratch. It will be very commonly the first step in most transformations.

- NEWINSTANCE (needed for tasks 1 and 2): evaluates the *key* expression against each input tuple and tests if an instance of the same type was previously created against the same key (potentially from another NEWINSTANCE step). If it was not, it will set the field mentioned in *instanceField* to a new instance of the specified type and package within the specified model. Otherwise, it will set the same field to the previously created element.

  The usage of a key is meant to provide a similar facility to the *equivalent()* operation in popular languages such as ATL or ETL.

- GETFEATURE (not needed for TTC): reads a feature (e.g. an attribute or reference) from an object in a field of the input row and places it as a field of the output row. Useful in combination with ADDTOCONTAINER later.

- SetFeature (needed for tasks 1 and 2): sets a feature of an object to the result of a certain Expression on each input row.

There are primitives for computing derived values and operating on collections:

- Evaluate (needed for tasks 1 and 2): computes an arbitrary expression on the current input row and places it as a field on the output row.
- NewContainer (not needed for TTC): creates a new, empty container as a field on each input row. The container may be a set, or a list.
- AddToContainer (needed for task 2): adds an element (potentially at a certain position) to the container present in a certain field of the input row. This operation has been treated as a primitive, as it may need specific considerations for incremental processing.

More primitives are available for managing rows:

- Filter (needed for tasks 1 and 2): for each input row, evaluates the *filterBy* expression. Rows that produce a *true* value are sent through the *target* element, and those that produce a *false* value are sent through the *rejectTarget* element.
- Copy (not needed for TTC): duplicates the same input rows in the same order to the *target* and *copyTarget* elements.
- Product (not needed for TTC): this step is intended to receive rows from two Elements and generate the cartesian product of the two sets of rows (e.g. produce all pairs of rows from both elements).
- Sort (not needed for TTC): reads in all the rows and then sorts them according to the values produced by the *sortBy* expression.
- ForEach (needed for task 2): for each input row with a collection on a certain field, it produces as many rows as elements in that collection, setting a certain field to each of its elements.
- CollectBy (not needed for TTC): for each sequence of contiguous rows with the same value for *collectBy*, it will produce one row replacing all fields with collections of their respective values in the sequence.

While these are quite a few primitives, only some of them need to be implemented to run the transformations proposed in this case study. This has been mentioned above, and will be listed again within each task.

2.1.2 Expressions

As mentioned above, some of these ELEMENTs can embed EXPRESSIONs in order to compute derived values, sorting/grouping keys or filtering conditions. Figure 2 shows the available elements for the small expression language. The objective of the language is to be side-effect free as much as possible. An expression is a tree of elements of various types:

- Literals of a certain type, e.g. BOOLEANLITERAL for boolean values and so forth.
- FIELDREFERENCEs to a certain field within the row, by name.
- UNARYOPERATIONs, which take the result of a subexpression and apply to it logical negation (*NOT* in UNARYOPERATOR) or arithmetic negation (*NEGATION*).
- BINARYOPERATIONs, which combine the result of two subexpressions in various ways. The language supports equality comparison (*EQ* in BINARYOPERATOR), inequality (*NE*), greater than (*GT*), greater or equal (*GE*), less than (*LT*), less or equal (*LE*), logical AND with shortcircuit, logical OR with shortcircuit, logical XOR, numeric addition / string concatenation (*ADD*), subtraction, multiplication, division or the modulo operation.

  Note: to simplify some cases, logical operators operate with "truish" values as well (as in JavaScript). A "truish" value is a "true" Boolean literal, a non-empty string, a non-null EObject, a non-empty sequence or a non-zero number.
- FEATURECALLs, which access a certain property or method within an object contained in the present row. If it is a method call, it will include a list of parameter subexpressions. The transformations to be run assume that "a.x" will be translated to "a.eGet(feature x)" in EMF terms, and that "a.eClass" and "a.eContainer" will also be available.
- CONDITIONALEXPRESSIONs, inspired by the ternary operator in C/C++ and by the *if..then..else* expression in Python. If the conditional expression produces a *true* value it will compute and return the result of the *thenExpression*, otherwise it will use the *elseExpression*.

2.2 Example: Families to Persons

Since the current case study does not involve writing new transformations but rather executing existing ones, the concrete syntax will be simply illustrated through the basic example to be executed: the classic "Families to Persons" transformation, as listed in the ATL Zoo[4].

This transformation takes a collection of FAMILY instances containing MEMBERS, whose genders are derived from their role in the FAMILY, and transforms them into

[4] `https://www.eclipse.org/atl/atlTransformations/#Families2Persons`

either MALE or FEMALE instances depending on their gender, while preserving their first and last names.

Two concrete syntaxes are provided in the Github repository for the abstract syntax shown in Section 2.1. The main syntax is textual (Xtext-based) and similar to OMG HUTN [4], allowing users to comfortably write the embedded expressions included in primitives such as EVALUATE or FILTER. A graphical syntax has also been developed using Sirius, for simpler visualization of the connections between the primitives. The same model can be edited through both notations at the same time, thanks to the viewpoint approach taken by Sirius.

The textual representation of the "Families to Persons" example is shown in Listing 1. The transformation starts by retrieving all the instances of MEMBER in "AllMembers", and then computing the full name in advance through the "ComputeFullName" element. "SplitByGender" receives the resulting rows and distributes them between the "NewMale" and "NewFemale" elements. This creates the appropriate MALE or FEMALE instance, which is sent to "SetPersonName" to have its full name set.

The textual syntax provides a slight simplification over the metamodel in Figure 1: if a particular primitive only takes one field name, the textual syntax simply uses the "field" keyword to refer to it. This is the case for ALLINSTANCES (which has "field"), NEWINSTANCE (has "instanceField"), SETFEATURE (has "objectField"), NEWCONTAINER (has "listField"), ADDTOCONTAINER (has "listField"), and EVALUATE (has "field"). For the primitives that use multiple field names, these are preserved as-is: this is the case for GETFEATURE and FOREACH.

Another note regarding the textual syntax: in some cases, field or feature names may conflict with keywords in the language (e.g. "type"). To solve this issue, identifiers can be escaped with "^": "^type" will be understood by the notation as a reference to the "type" field or feature, instead of the "type" keyword.

The graphical representation of the "Families to Persons" example is shown in Figure 3. As it can be seen, it is only meant to be a viewpoint on the full model that emphasizes the flows between the different activities: it does not represent the embedded expressions themselves. The representation is mostly a simple "boxes and arrows" affair, mentioning the primitive type and the name of the ELEMENT. This partial representation is editable, nevertheless, as it can be useful for detecting disconnected or wrongly connected elements in the flow. For clarity, the outgoing edges of the FILTER steps are color-coded and labelled depending on whether they take the rows that passed the condition or not.

2.3 Task 1: base primitives

The first task of this case study is implementing an approach that enables the batch and/or incremental execution of two basic transformations: the "Families to Persons" example shown in Section 2.2 and a "Tree to Graph" transformation that was also inspired from a case in the ATL Transformation Zoo. Both transformations are available as subfolders of the `examples` folder in the GitHub repository.

The basic folder layout is the same for all examples:

Listing 1: Families to Persons, textual *FlowM2M* notation

```
1  AllInstances AllMembers {
2    field member
3    type Families!Member
4    target ComputeFullName
5  }
6  Evaluate ComputeFullName {
7    field fullName
8    expression member.firstName + ' ' + member.eContainer.lastName
9    target SplitByGender
10 }
11 Filter SplitByGender {
12   filterBy member.familyMother or member.familyDaughter
13   target NewFemale
14   rejectTarget NewMale
15 }
16 NewInstance NewMale {
17   field person
18   key member
19   type Persons!Male
20   target SetPersonName
21 }
22 NewInstance NewFemale {
23   field person
24   key member
25   type Persons!Female
26   target SetPersonName
27 }
28 SetFeature SetPersonName {
29   field person
30   value fullName
31   feature fullName
32 }
```

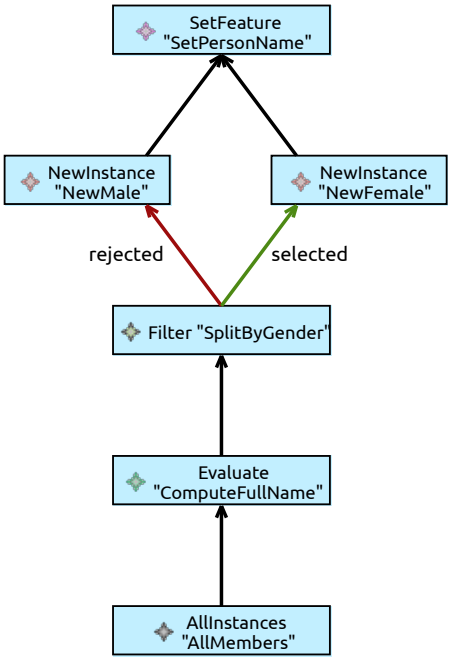

Figure 3: Families to Persons, graphical *FlowM2M* notation

- `input`: manually constructed model, plus a set of small, medium and large synthetic models (with the `GS`, `GM` and `GL` suffixes, respectively).
- `output`: expected output for the manually constructed example.
- `metamodels`: Ecore metamodels for the input and output models.
- `generator`: the Epsilon Model Generator[5] script that produced the synthetic models.
- `transformation`: a `.dataflow` file specifying the transformation according to the abstract syntax in Section 2.1 and the textual concrete syntax used in Listing 1.
- The root folder of the example contains a `.launchconfig` with the Ecore-based launch configuration used for the sample Epsilon-based interpreter, a set of Eclipse launch configurations to use the model generator, and an `.aird` file with a graphical representation of the transformation.

A reference Epsilon[6]-based interpreter of `.dataflow` files is available in `solutions`. It is not designed for performance nor scalability, however: contestants are free to use their favorite approach to enable execution (e.g. Java code generation, transformation to another tool or better interpretation). Contestants are also invited to slightly tweak

[5] `https://github.com/sop501/ModelCodes`
[6] `http://eclipse.org/epsilon`

semantics if it improves incrementality, maintainability or performance, as long as the transformation produces the intended result.

The base set of primitives for this task are AllInstances, Evaluate, Filter, NewInstance and SetFeature. Contestants are free to limit the implementation of the expression language to the particular operators and methods that are required by these transformations.

2.4 Task 2: extended primitives

The second task is a direct extension of the first one: this time, the two transformations to be enabled are more complex. One is an instance of the (in)famous "Class to RDB" example, while the other is a "Flowchart to HTML" transformation. These transformations follow the same folder layout mentioned in Section 2.3.

This task requires implementing the primitives in the previous task, plus the ForEach and AddToContainer tasks. The fact that these primitives handle collections will naturally complicate their execution, especially in an incremental scenario. As for the expression language, more operators will have to be implemented: in particular, "Flowchart to HTML" includes an alternative version that takes advantage of the ternary "if...then...else..." operator.

3 Evaluation

Contestants should submit their solutions at the end of the day before the contest in a form that allows for easy execution by attendees. Solutions will be submitted as pull requests to the main GitHub repository, adding new subfolders to the `solutions` folder. Solutions should also report their execution times in milliseconds, to aid performance evaluation.

The solutions shall be evaluated according to the following criteria:

- Batch correctness – how many of the manually constructed models are transformed correctly?
- Incremental correctness – how many of the manually constructed models are recalculated/updated correctly after a change in the model?

  The actual changes to the model will be:

  - For "Families to Persons": change the role of a member of a family, so that their gender changes.
  - For "Tree to Graph": remove a subtree, which should result in removing a subgraph.
  - For "Class to RDB": add a new multivalued scalar attribute, producing a new table.
  - For "Flowchart to HTML": add a new transition, which should result in a new link.

- Batch performance – is the solution faster than the others at transforming the large synthetic model? *Requires batch correctness.*
- Incremental performance – is the solution faster than the others at producing the updated model after changing the large synthetic model as mentioned above? *Requires incremental correctness.*
- Maintainability and extensibility – can the provided solution be extended naturally to cover the other primitives which were not used in this case study?

References

[1] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3: A Reactive Model Transformation Platform. In Dimitris Kolovos and Manuel Wimmer, editors, *Theory and Practice of Model Transformations*, number 9152 in Lecture Notes in Computer Science, pages 101–110. Springer International Publishing, July 2015.

[2] Jesús Sánchez Cuadrado and Juan de Lara. Streaming Model Transformations: Scenarios, Challenges and Initial Solutions. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations*, number 7909 in Lecture Notes in Computer Science, pages 1–16. Springer Berlin Heidelberg, June 2013. DOI: 10.1007/978-3-642-38883-5_1.

[3] István Dávid, István Ráth, and Dániel Varró. Streaming Model Transformations By Complex Event Processing. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, number 8767 in Lecture Notes in Computer Science, pages 68–83. Springer International Publishing, September 2014.

[4] Object Management Group. Human-Usable Textual Notation (HUTN) 1.0, August 2004.

[5] Massimo Tisi, Salvador Martínez, Frédéric Jouault, and Jordi Cabot. Lazy execution of model-to-model transformations. In *Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2011.