Incremental MTL vs. GPLs: Class into Relational Database Schema

Sandra Greiner[1], Stefan Höppner[2], Frédéric Jouault[3,4], Théo Le Calvar[5] and Mickael Clavreul[3,4]

[1]Software Engineering Group - University of Bern, 3012 Bern, Switzerland
[2]University of Ulm, 89069 Ulm, Germany
[3]University of Angers, LERIA, 49000 Angers, France
[4]ESEO-TECH / ERIS, 49100 Angers, France
[5]IMT Atlantique, LS2N (UMR CNRS 6004), France

Abstract

Model transformation languages (MTLs) are domain-specific languages tailored to express model-to-model transformation programs. In contrast to general-purpose languages (GPLs), MTLs typically offer specific higher-level syntactic constructs, such as rules, and specific features, such as automatic traceability support. Moreover, some MTLs allow for multiple execution modes, such as incremental or bidirectional, based on a single specification. Many MTLs have been proposed over the past decades, but GPLs are still widely used to write model transformations in practice. Previous work has identified some reasons for this, in the context of the batch execution mode, such as the fact that modern GPLs are not much more verbose than MTLs. Our working hypothesis is that the situation is different for other execution modes. Therefore, this transformation tool contest case calls for incremental solutions implemented using various MTLs and GPLs, with the purpose of building a data set consisting of labeled solutions specified in diverse languages. The overall objective is to leverage this data set to gain a better understanding whether MTLs are better suited than GPLs to perform incremental tasks.

Keywords

Incremental Transformations, Model-Driven Software Engineering, Model Transformation Languages

1. Introduction

Within the Model Driven Engineering methodology, *model transformation languages* (MTLs) are typically seen as the best means of expressing model transformations. However, many transformations are defined in *general purpose languages* (GPLs), particularly, in real-world situations, which poses several challenges [1]. One main issue is a lack of understanding of the benefits and functionality of MTLs compared to GPLs [2]. This can lead to "hidden" model transformations that may not be explicitly denoted as such, and the unawareness that a transformation is performed.

To address this problem, the aim of the Incremental Class2Relational transformation tool contest case is to compare GPL solutions specified in heterogeneous languages, such as Python, Java, C#, and Xtend, with MTL solutions, focusing on their syntactic complexity [3]. For comparing the solutions, we require submissions to label their transformation code with the syntactic complexity of each statement (see Section 3.3) and the purpose the statement serves for in the transformation process. By comparing the complexity of these languages, we can guide software developers in deciding which type of language to use and provide suggestions for developing transformation-specific language support in GPLs.

Through the tool case, our goal is to evaluate and compare the case solutions and work towards a journal paper that examines the differences between GPLs and MTLs for writing incremental transformations answering the following research questions:

RQ 1: *How does the distribution of incremental transformation parts, such as model loading and saving or transformation rules, of GPLs compare with MTLs?*

RQ 2: *How do GPLs differ from MTLs in terms of the number of errors?*

RQ 3: *In which situations are transformation DSLs better than GPLs for incremental transformations, and in which parts of the development process?*

With **RQ1** we aim to investigate how well aspects of incremental transformations are abstracted in dedicated MTLs and how much 'effort' it takes to reimplement these abstractions in a general purpose programming language. Literature and language developers often claim,

TTC'23: 15th Transformation Tool Contest, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. A. Boronat, A. García-Domínguez, and G. Hinkel, 20 July 2023, Leicester, UK.

sandra.greiner@unibe.ch (S. Greiner);
stefan.hoeppner@uni-ulm.de (S. Höppner);
frederic.jouault@eseo.fr (F. Jouault);
theo.le-calvar@imt-atlantique.fr (T. Le Calvar);
Mickael.CLAVREUL@eseo.fr (M. Clavreul)
0000-0001-8950-0092 (S. Greiner)

that using MTLs reduces the amount of errors that are introduced during development [2, 1]. Through the submissions for this case and our benchmarking framework (see Section 3.1&4.2), we aim to provide empirical data for this discussion by answering **RQ2**. The overall goal of our work is to assess the usefulness of MTLs and GPLs for specifying incremental model transformations. **RQ3** addresses this goal. We try to answer it based on examining the solution submissions for this transformation case.

To attract a large number of solution submissions, our transformation case is an *incremental* variant of the well-known '*Class2Relational*' transformation [4]. We measure the success of each submitted solution by assessing whether it produces a correct target model for a given source model, and whether it is properly labeled.

Different variants of the Class2Relational transformation exist in literature, such as a transformation of entire Ecore models into self-defined relational database schema [5], or as part of language specifications [6] and examples thereof [7]. While these transformations may be beneficial to address real-world scenarios, it is complex to define them as batch transformation and, thus, even harder to define proper incremental behavior. To pursue our goal of attracting a multitude of solutions in diverse languages, we decided to reduce the transformation size and complexity to focus on concise and key incremental scenarios apparent in the Class2Relational transformation. For the same reason, we do not consider former cases of bidirectional, incremental transformations [8, 9] because they are trimmed for MTLs and do not ask for labeling the solution. Furthermore, our case is easy to specify in one direction and does not have to deal with information loss, which is an additional challenge that bidirectional transformations are confronted with, on top of propagating changes from a source to a target model.

The rest of this paper is structured as follows: Sec. 2 and Sec. 3 introduce the transformation case and tasks we would like participants to solve. In Sec. 4, we describe how we evaluate the submitted solutions. Lastly, Sec. 5 details how we will value the contributions and which rewards we propose to give to the participants.

2. Transformation case

As we aim for many solutions, we propose the Class2Relational transformation [4] as our transformation case similar to the definition proposed in the ATL zoo [10]. In contrast to the variant of the ATL zoo, we do not regard the batch execution mode but the incremental one. Since the scenario is well-known as a de-facto hello-world example for MTLs, we expect that an implemented variant of the case exists already in several MTLs. To provide the adequate variant for this transformation case, this section introduces the specific class and relational metamodels as well as the expected transformation behavior.

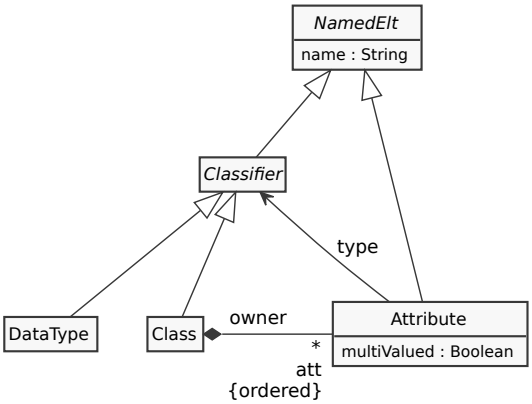

Figure 1: Class metamodel.

2.1. Metamodels

Class Metamodel Figure 1 depicts the source metamodel of the transformation. It comprises named classifiers, which are either datatypes or classes. Classes may contain several single- or multi-valued attributes, which are either typed with a primitive *DataType* or a complex type (i.e., of a specific *Class*). Explicit references do not exist between classifiers but can be expressed in terms of attributes of complex types. As such, any Ecore model can be translated into a simplified representation being an instance of this metamodel. A corresponding transformation (called `Ecore2Class`) is available besides the transformation in the ATL zoo.

Relational Database Schema Metamodel Figure 2 depicts the target metamodel. A model consists of several tables that each contain an ordered list of typed columns. A column may serve as key for a table. The metamodel does not distinguish foreign from primary keys. Tables, columns, and types are named elements and are identified accordingly.

Notes on Challenges Some points can be regarded as specific to the transformation case. Firstly, neither the source nor target metamodels possess a unique root element. Accordingly, the input and output models may comprise several top-level classes and tables, respectively. Additionally, primary keys are not explicitly present in the source or target model, but an `id` column may serve as such for each table created for a source class.

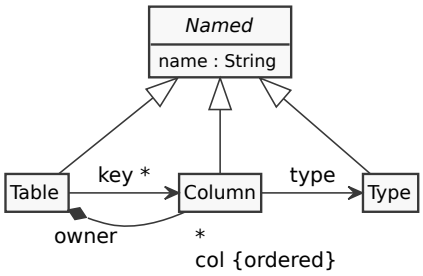

Figure 2: Relational schema metamodel.

2.2. Transformation Behavior

The transformation behavior may vary with respect to the execution mode. We describe the state-of-the-art *batch* behavior, as implemented in the ATL zoo, first. Second, we describe the variants of executing the *incremental* transformation correctly.

General (Batch) Mappings In the batch transformation defined in the ATL zoo, the transformation rules establish the following mappings:

- `Class` → `Table` and *objectID*-`Column`
- `DataType` → `Type`
- *single-valued, primitive* `Attribute` → `Column`
- *multi-valued, primitive* `Attribute` → `Table`, *id*-`Column`, and *value*-`Column`
- *single-valued, class* `Attribute` → *id*-`Column`
- *multi-valued, class* `Attribute` → `Table`, *id*-`Column`, and *foreign*-`Column`

Accordingly, for each class a table and a column serving as (implicit) primary key are created. The *objectID* column is a `Column` with `name` 'objectId' and of `type` `Integer`.

Furthermore, each single-valued attribute is transformed into a column whereas, for multi-valued attributes, a new table is created. In the latter case, one column of the new table serves as foreign key of the class owning the attribute and a second column represents the value of the attribute. Thus, a `Column` of type `Integer` and named as the owner followed by the attribute name (i.e., `"a.owner.name + '_' + a.name"`) is created, as well as a second column typed `Integer` and named as the attribute itself followed by 'Id'. Columns created for single-valued attributes are integrated into the table created for the containing class.

This behavior serves as a baseline for executing the transformation. Please note: (1) no dedicated root exists in the transformation; (2) primary keys are integrated implicitly when creating a table for a class; and (3) inheritance relationships are not covered as they are absent from the class metamodel.

Incremental Behavior In the incremental case, the behavior of the transformation can vary with respect to handling updates to the input model. We assume that, at any point in time of re-executing the incremental transformation, the input model is a valid model which does not violate the syntax defined through the metamodels. Firstly, create, update, and delete changes may occur at all levels of the input model. Thus, objects, such as classes and their attributes, can be deleted or added. Additionally, their structural features can change, for example, a class can be renamed or its name can be deleted.

Secondly, varying behaviors can be supported through the transformation engines and the definitions in the incremental execution mode. Handling null-values (e.g., of incomplete source models with unset references or deleted attribute values) represents one example. For instance, when concatenating a null-value of a missing name with an existing `String`, a transformation may either terminate (e.g., with an exception), or it can be tolerated. As an example, ATOL transformations [11] support concatenation of (potentially empty) `Strings` and replace a null-reference with the verbatim `String` "`null`". As such, we consider the handling of null-values as variants of the incremental transformation. A relatively simple solution is to ignore null-values and to assume that the input is correct. A more sophisticated variant will replace null-values with default values. Handling unset references represents another example of behavior that results from deleting or adding objects. Sec. 3.2 presents the entire list of incremental behaviors expected in the case.

3. Task

The task of the incremental Class2Relational case is to define the incremental Class2Relational transformation. As we require correct and labeled transformations as (minimal) solutions, this section introduces the criteria for successfully passing the transformation case. It explains how *correctness* is evaluated, elaborates on the *expected incremental behavior*, demonstrates how (manual) labeling can be accomplished, and finally points to further properties which solutions can report.

3.1. Correctness: Commutativity

As correctness criterion, we enforce *commutativity* of batch and incremental transformations. We will consider the incremental transformation correct if a batch transformation produces the same result for the same input model as the incremental one making them interchangeable.

Consequently, the correctness depends on the behavior of the batch transformation. As ground truth transfor-

mation, we propose the permissive incremental ATOL Class2Relational transformation[1]. The ATOL transformation (engine) is more permissive than the batch variant defined in the ATL zoo as it tolerates null-values. While it is more fault-tolerant, the ATOL variant could be considered incorrect compared to the transformation from the ATL zoo, which does not allow for null-values.

3.2. Completeness: Incremental Behavior

Variants of the incremental transformation may differ in how they handle missing or added elements resulting from incremental updates to the source model. We will consider a transformation as *complete* if it passes the correctness criteria defined above. If the transformation fails at any point, we gradually consider it as incomplete by manually inspecting the reasons for the failure. The following criteria will be evaluated.

Null Values The structural features of objects in the source model may assume `null` or other default values when their original value is deleted. When accessing a `null`-value, an incremental transformation can:

1. fail (e.g., due to a null-pointer exception)
2. ignore the access and continue with the next rule
3. resolve the problem by replacing the `null`-value with a default value

The reference batch solution assumes behavior (3). However, which strategy is optimal in each situation may depend on the concrete transformation case and a specific rule. Therefore, while we consider solutions which discontinue the transformation (1) as well as silently ignoring the failure (2) as incomplete, we still provide a reduced score for the solution which continues the execution (2), ideally with a (log-)message to the user.

Adding Objects Due to added objects (i.e., classes, datatypes, or attributes), references between objects may be missing. Ideally, the input model should be validated first to avoid such situation, however, due to human errors this situation may occur.

If a single object is added without a reference, as consequence of the missing reference in the source model, it will depend on the object type whether references in the target model are needed or not. A table and a type can be integrated into the target model as root elements of the target model without requiring a container. In contrast, a `Column` requires a table to be present. If a source *Attribute* is not contained in a *Class*, it depends on the type of the attribute which transformation behavior shall occur.

[1] https://github.com/ATL-Research/incremental-class2relational

For a multi-valued attribute, a table may be created, but accessing the owner of the attribute may be an access to a null-value, which will have to be solved as explained in the previous paragraph on null-values. On the contrary, for single-valued attributes, only a column is created, which should be added to the table created for its owner. When the link to the owner is missing, the transformation can assume the following behavior:

1. do not create the dangling `Column` object (roll-back)
2. ignore the missing link and leave the dangling object
3. add the `Column` object to the first object of the right type (i.e., `Table`)

Again, these three potential solutions can be scored in ascending order. Rolling-back will create a valid target model but misses to propagate the information added to the source model, in this case to integrate a new object. The second solution produces an invalid model but propagates the same information. The third solution guarantees a valid target model, however, it is possible that the object is added to the wrong container.

The reference ATOL transformation assumes the second behavior. While the solution may not be ideal, solution (3) would add the column to a potentially undesired wrong table. We do not consider a semi-automated alternative of computing all potential containers and proposing them to the user [12], which may not scale in terms of execution time and storage.

Removing Objects Similarly to adding objects, removing objects can provoke undesired side effects. Some references may point to obsolete target objects. If one end is missing while the other end and the reference remains, the transformation engine can either

1. remove the link, or
2. create a new default-object for the missing end

in the target model. Our reference batch transformation assumes strategy (2). If a solution assumes strategy (1), similarly, we will manually score it as correct and complete, as it is a valid approach, too.

Further Criteria Our list of solutions to updates and problems potentially occurring in incremental transformations may not be exhaustive. Particularly, depending on the transformation engine's properties, further automated solutions may be possible which are not available in ATOL. While we use the ATOL transformation as reference incremental transformation based on which we score solutions, we welcome further reference transformations as part of the solution submission, which we can respect in future work.

3.3. Syntactic Complexity: Labeling of Transformations

To compare solutions developed in different languages, we rely on 'syntactic complexity'. This metric measures the amount of words that are separated either by whitespaces or other delimiters used in the languages, e.g. dot(.) and different parentheses ((){}[]) [3]. As it is difficult to offer a tool which computes the metric generically for any MTL or GPL, we require submissions to self report the values of this measure for each line of their code. For comparing solutions, we are interested in how much code is written for different transformation *aspects*. The aspects we require for this purpose are:

- *Setup*, i.e., code required to make the transformation work
- *Model Traversal*, i.e., traversing the input to find model element(s) to transform
- *Helper/Expression Outsourcing*, i.e., modular code that outsources expressions that are used multiple times in a transformation
- *Tracing*, i.e., explicit code that establishes or resolves trace links between input and output elements
- *Incrementality*, i.e., explicit code that manually implements incrementality functionality
- *Transformation*, i.e., the code that actually transforms input model elements to output model elements

For each statement in the transformation code we require submissions to provide the transformation aspect that it implements as well as the complexity value in form of a comment *above* the statement. The 'labels' must be provided in the following format:

`<CommentDelimiter> TransformationAspect Value`.

Ideally, each label only includes one transformation aspect and one complexity value. If a single statement of code implements the functionality of several aspects, we will expect the complexity value to be split between these aspects respective to the share of the statement that implements them. Each label must be reported in a separate line within the code. Within each solution, the symbols used to comment the complexity labels must be consistent through the entire submitted project.

Figure 3 depicts an excerpt of a Java implementation of the Class2Relational case which is labeled as described above. Line 3 implements setup functionality to create model elements of the relational metamodel. It contains eight separate words and, thus, its syntactic complexity value is eight. Consequently, using the format we propose, it is labeled with $Setup$ 8.

As another example, lines 44-49 (split into multiple lines for readability) implement transformation functionality as well as trace resolution. Thus, the syntactic complexity of the statement is given as $Transformation$ 12 and $Tracing$ 16 because 12 words in the statement implement part of the transformation and 16 words implement trace resolution.

In the event that a solution is provided using a non-textual transformation language such as a graphical representation, we expect authors to count the number of model elements used to design the transformation and to label these elements with notes using the same format we propose in this section. This should be provided in a separate file following the described labeling schema.

3.4. Additional Features: Performance and Quality

The main criteria that are evaluated automatically for passing the case are the correctness and thereby indirectly the completeness of the proposed solution. Additionally, we require authors to label their solution so that we can evaluate its complexity as explained in Sec. 3.3. We compute the score of a submitted solution based on these three criteria.

Still, we welcome authors to elaborate on further features of their solution. Although not explicitly required, the authors can report on further quality aspects which their solution contributes. For instance, the performance in terms of execution times may be an additional upside of the transformation. Similarly, further criteria, such as strategies to efficiently compute and perform the incremental update or saving memory consumption may be regarded to compare the solutions.

4. Benchmark

This section introduces the evaluation criteria and the benchmark framework used to evaluate solutions submitted by participants.

4.1. Evaluation Criteria

The evaluation fosters two kinds of criteria, automatically measured and self-reported ones. In the sequel, we describe both types of criteria.

Automatically Measured Criteria regard the completeness of the transformation as well as the correctness. We use the latter implicitly to determine the completeness.

Completeness: Completeness describes for how many of the tasks described in Sec. 3 a solution is submitted. Since this call aims for a comprehensive comparison between the languages, completeness also encompasses analyzing the submitted solution to detect variants in the

transformation process and how does it compare with the reference implementation behavior.

Correctness: Correctness describes the degree to which the submitted solutions commute with the reference solution. The incremental contributed solution should commute with the provided batch reference transformation when the same change was applied to the source model. Authors can use the benchmark framework to evaluate the correctness of their solution and to improve their transformation.

Self-Reported Measures regard the syntactic complexity in terms of labeling the solution specification as well as further optionally reported solution-specific properties which may be beneficial, such as the means to increase the *performance* in terms of execution time.

Syntactic Complexity: For the purpose of this case, we are interested in how much code is written to implement different aspects of model transformation. The quality of a solution is measured in how much code, measured in the amount of words that are separated either by whitespaces or other delimiters used in the languages, e.g. dot(.) and different parentheses (()[]{}) [3]. We ask authors of submitted solutions to kindly provide the measures for their solutions separately using the labeling format introduced in Sec. 3.3. The quality of a submission is ranked based on how much of its code is focused on the actual transformation, i.e. *Transformation* and *Helper/Expression Outsourcing* definitions, compared to the transformation specific additional aspects *Setup*, *Model Traversal* and *Tracing*.

Additional Properties which may help to perform the tasks in a submitted solution may be reported. For instance, *performance* describes how timely the solution can produce a result for a given input task. The degree of correctness does not factor into the performance evaluation. Solutions may report on the execution time of a given task using a unit of time (e.g. in milliseconds, in seconds, in minutes). Specific values in the chosen unit of time are not required but can be reported if they are known to the authors.

While we will not score additional properties, they may still positively influence complexity and performance (e.g., explicit or implicit trace maintenance) and therefore, may be additional relevant information.

4.2. Benchmark Framework

We provide a benchmark framework to automate verification of correctness and completeness of solutions. Detailed information on the framework and how to use and integrate your solution is available in the repository of the case[2]. Source code of the batch ATL transformation, the incremental ATOL transformation, source models, change models, and expected models are also available in the repository.

We provide metamodels in Ecore and models in XMI formats. Change models use the same format as proposed in the TTC 2018 Social Network case, refer to Reference [13] for detailed explanations.

4.2.1. Correctness evaluation

We evaluate correctness, as defined in Sec. 3.1, by comparing two executions of a transformation. The first execution performs the following actions:

1. load a source model
2. apply the transformation
3. load and apply a change model to the source model
4. propagate the change to the target
5. save the target model

The second execution performs the following actions:

1. load a source model that already has the change applied
2. apply the transformation
3. save the target model

To pass the correctness test, solutions must return identical solutions for both executions. We use the `SimpleEMFModelComparator`[3] tool to compare the target models of the first and second execution.

4.2.2. Completeness evaluation

Unlike correctness, completeness, as described in Sec. 3.2, is evaluated using a set of source models and changes that test specific behaviors to determine which level of completeness the solution achieves. Using a script, the benchmark automatically executes proposed transformations on various test cases and compares the transformation outputs to the outputs we expected. We provide the expected target model besides the source and change model.

The procedure is as follow:

1. load a source model
2. apply the transformation
3. load and apply a change model that corresponds to the test case
4. propagate the change
5. save the target model

Like correctness testing, we compare each target model of a transformation test case to the expected model of this case using the `SimpleEMFModelComparator` tool.

[2] https://github.com/ATL-Research/incremental-class2relational

[3] https://github.com/ATL-Research/EMFModelFuzzer/blob/main/lib/src/main/java/io/github/atlresearch/emfmodelfuzzer/SimpleEMFModelComparator.xtend

4.2.3. Expected submission bundle

The benchmark framework provides automated tooling to check correctness and completeness of solutions. In order for the benchmark to support submissions, we expect solutions to implement the following calling interface.

Parameters of the transformation are passed using the following environment variables:

- SOURCE_PATH: path of the source model
- TARGET_PATH: path of the target model
- CHANGE_PATH: optional, path of the change model

All input and change models are given in XMI format, we expect target models to also be in XMI format. If the variable CHANGE_MODEL is not provided as we call the execution of a transformation, it should behave like a batch transformation.

Solutions can freely use `stdout` and `stderr` to print warning, information or debug messages. For EMF-based solutions, we provide an abstract runner that handles model loading and application of changes to the source model.

5. Evaluation

The benchmark framework will provide independent measurements of the completeness and the correctness of the solutions submitted by the participants. Attendees to the contest will also evaluate the performance and the quality of their own solution. To recognize contributions and give appeal to this contest, we propose to award five prizes:

- **"Best Overall in GPL"** to the GPL solution with the highest ranking over all four evaluation criteria.
- **"Best Overall in MTL"** to the MTL solution with the highest ranking over all four evaluation criteria.
- **"Most Complete"** to the solution with the highest ranking over the Completeness evaluation criteria.
- **"Best Quality"** to the solution with the highest ranking over the Quality evaluation criteria based on the information given by authors.
- **"Best Contributor"** to the author that submits the highest number of solutions in different languages. Authors which provide solutions in both GPL and MTL categories will be given extra points.

References

[1] S. Höppner, Y. Haas, M. Tichy, K. Juhnke, Advantages and disadvantages of (dedicated) model transformation languages 27 (2022) 159. doi:10.1007/s10664-022-10194-7.

[2] S. Götz, M. Tichy, R. Groner, Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review 20 (2021) 469–503. URL: https://doi.org/10.1007/s10270-020-00815-4. doi:10.1007/s10270-020-00815-4.

[3] S. Höppner, T. Kehrer, M. Tichy, Contrasting dedicated model transformation languages vs. general purpose languages: A historical perspective on atl vs. java based on complexity and size, Software and Systems Modeling (2021). doi:10.1007/s10270-021-00937-3.

[4] INRIA, ATL Transformation Example. Class to Relational, 2005. URL: https://www.eclipse.org/atl/atlTransformations/Class2Relational/ExampleClass2Relational[v00.01].pdf, modified: November 23, 2022 at 22:26:16 GMT+1.

[5] B. Westfechtel, A case study for a bidirectional transformation between heterogeneous metamodels in qvt relations, in: L. A. Maciaszek, J. Filipe (Eds.), Evaluation of Novel Approaches to Software Engineering, Springer International Publishing, Cham, 2016, pp. 141–161. doi:10.1007/978-3-319-30243-0_8.

[6] Object Management Group (OMG), Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.3, formal/2016-06-03 ed., Needham, MA, 2016. https://www.omg.org/spec/QVT/1.3/PDF.

[7] Daniel Strueber, Henshin/Examples/Ecore2RDB - Eclipsepedia, Modified: February 11, 2023 at 12:53:21 GMT+1. https://wiki.eclipse.org/Henshin/Examples/Ecore2RDB.

[8] A. Anjorin, T. Buchmann, B. Westfechtel, The families to persons case, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), volume 2026 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017, pp. 27–34. URL: https://ceur-ws.org/Vol-2026/paper2.pdf.

[9] A. García-Domínguez, G. Hinkel, The TTC 2019 live case: Bibtex to docbook, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 12th Transformation Tool Contest, co-located with the 2019 Software Technologies: Applications and Foundations, TTC@STAF 2019, Eindhoven, The Netherlands, July 19, 2019, volume 2550 of *CEUR*

Workshop Proceedings, CEUR-WS.org, 2019, pp. 61–65. URL: https://ceur-ws.org/Vol-2550/paper8.pdf.

[10] F. Allilaire, ATL Transformations | The Eclipse Foundation, 2023. URL: https://www.eclipse.org/atl/atlTransformations/, modified: April 6, 2023 at 13:23:55 GMT+2.

[11] T. Le Calvar, F. Jouault, F. Chhel, F. Saubion, M. Clavreul, Intensional view definition with constrained incremental transformation rules, in: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), 2019, pp. 395–402. doi:10.1109/MODELS-C.2019.00061.

[12] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, A. Zündorf, Benchmarking bidirectional transformations: theory, implementation, application, and assessment, Software and Systems Modeling 19 (2020) 647–691. doi:10.1007/s10270-019-00752-x.

[13] G. Hinkel, The TTC 2018 social media case, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 11th Transformation Tool Contest, co-located with co-located with the 2018 Software Technologies: Applications and Foundations (STAF 2018) 2018, Toulouse, France, July 29, 2018, volume 2310 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 39–43. URL: https://ceur-ws.org/Vol-2310/paper5.pdf.

```
1  public class Class2RelationalIncremental {
2     // Setup 8
3     private static final RelationalFactory RELATIONALFACTORY = RelationalFactory.eINSTANCE;
4     ...
5     // Helper 4
6     private static Type objectIdType() {
7        ...
8        // Helper 2
9        return objectIdType;
10    }
11    ...
12    // Setup 8
13    public static Resource start(String inPath, String outPath) {
14       ...
15       // Incrementality 5
16       Adapter adapterIn = new AdapterImpl() {
17          // Incrementality 5
18          public void notifyChanged(Notification notification)
19          ...
20       }
21    }
22    // Traversal 8
23    public static List<Named> transform(List<EObject> input) {
24       // Traversal 5
25       for (EObject namedElt : input) {
26          ...
27       }
28    }
29    ...
30    // Tracing 6
31    public static void Class2TablePre(Class c) {
32       // Tracing 5
33       TRACER.addTrace(c, RELATIONALFACTORY.createTable());
34       ...
35    }
36
37    // Transformation 6
38    public static void Class2Table(Class c) {
39       // Tracing 8
40       var out = TRACER.resolve(c, RELATIONALFACTORY.createTable());
41       ...
42       // Transformation 12
43       // Tracing 16
44       out.getCol().addAll(
45          c.getAttr().stream()
46             .filter(e -> !e.isMultiValued())
47             .map($ -> TRACER.resolve($, RELATIONALFACTORY.createColumn()))
48             .filter($ -> $ != null)
49          .collect(Collectors.toList()));
50    }
51    ...
52 }
```

Figure 3: Example labeling for Java