Incremental MTL vs. GPLs: Class into Relational Database Schema

Sandra Greiner
sandra.greiner@unibe.ch
University of Bern
Bern, Switzerland

Stefan Höppner
stefan.hoeppner@uni-ulm.de
University of Ulm
Ulm, Germany

Frédéric Jouault
Frederic.JOUAULT@eseo.fr
ESEO
Angers, France

Théo Le Calvar
theo.le-calvar@imt-atlantique.fr
IMT Atlantique, LS2N (UMR CNRS 6004)
Nantes, France

Mickael Clavreul
Mickael.CLAVREUL@eseo.fr
ESEO
Angers, France

ABSTRACT

Model transformation languages (MTLs) are domain-specific languages tailored to express model-to-model transformation programs. They typically offer higher-level syntactic constructs, such as rules, and specific features, such as automatic traceability support, than general-purpose languages (GPLs). 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 better understand whether GPLs are up to incremental tasks, or whether MTLs are significantly more appropriate.

KEYWORDS

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

ACM Reference Format:
Sandra Greiner, Stefan Höppner, Frédéric Jouault, Théo Le Calvar, and Mickael Clavreul. 2023. Incremental MTL vs. GPLs: Class into Relational Database Schema. In *Proceedings of TTC, STAF (TTC23)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [7]. One of the main issues is the lack of understanding the benefits and functionality of MTLs compared to GPLs [5]. 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 [8]. For the purpose of comparison, we require submissions to label their transformation code with the syntactic complexity of each statement (see Section 3.3) and information on what 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 is the size of incremental transformations written in GPLs distributed over different parts of the transformation, such as the model loading and saving or the transformation rules, compared to MTLs?

RQ 2: How high is the error rate in GPL transformations compared to MTL transformations for incremental use-cases?

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, that using MTLs reduces the amount of errors that are introduced during development [5, 7]. Using 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**. As explained earlier, the overall goal of our work is to provide results on the usefulness of MTLs and GPLs for writing incremental model transformations. **RQ3** verbalises this goal in a question that we will try to answer based on the insights gained from evaluating the solution submissions for our transformation case.

To attract a large number of solution submissions, we propose to use an *incremental* variant of the well-known '*Class2Relational*' transformation [9] as transformation case. We measure the success of each submitted solution by the facts 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 [12], or as part of language specifications [11] and examples thereof [3]. 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 [2, 4] as appropriate 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 [9] as transformation case similar to the definition proposed in the ATL zoo [1]. In contrast to the variant of the ATL zoo, we do not consider the batch execution mode but its behavior in incremental transformations. 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.

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.

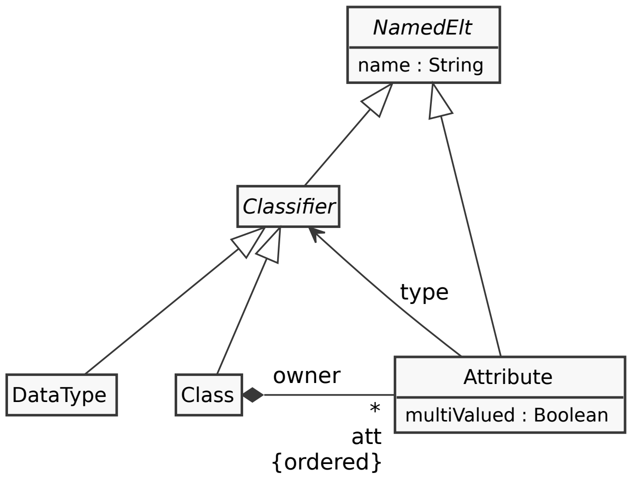

Figure 1: Class metamodel.

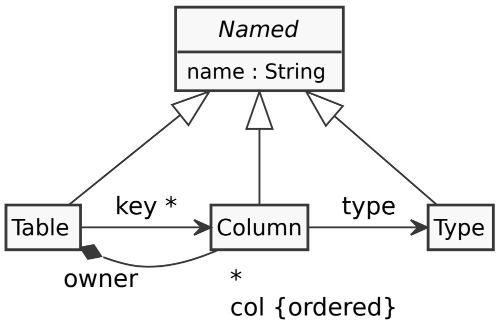

Figure 2: Relational schema metamodel.

Relational Database Schema Metamodel. Figure 2 depicts the target metamodel. It consists of several tables that own an ordered list of typed columns. A column may serve as key for a table. The relational schema metamodel does not differentiate between foreign and primary keys. Tables, columns, and types are named elements and can be identified accordingly.

Notes on Challenges. Some points can be regarded as specific to the transformation case. Firstly, neither the source or 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 model or target model but an `id` column may serve as such for each table created for a source class.

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

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

Furthermore, for single-valued attributes, a column is created 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 the type of the owner and with name of the owner, followed by 'id' (i.e., `"a.owner.name + '_' + a.name"`) is created as well as a second column typed and named as the attribute itself. All 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 in the transformation scenario as they are not part of the class metamodel.

Incremental Behavior. In the incremental transformation, 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, the typical create, update, and delete changes may occur at all levels of the input model. For instance, 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 the name can be deleted entirely.

Secondly, different behaviors can be supported through the transformation engines and the definitions in the incremental execution mode. Handling null-values represents one such behavior. Typically, accessing a null-value may throw a null-pointer exception. However, for instance, when concatenating a null-value with an existing String, it may either throw an exception, or it can be tolerated. In the latter case, for instance, ATOL transformations [10] allow strings concatenation and replace the null-reference with the verbatim String "`null`". As such, we consider the handling of null-values as variants of the incremental transformation. A fast solution is to ignore null-values and to assume that the input is correct. A more sophisticated variant will replace null-values by (default) values. Handling dangling references represents another example of behavior that results from deleting or adding objects. Sec. 3.2 presents the entire list of incremental behavior.

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 refers to further quality aspects of transformations 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.

Consequently, the correctness depends on the behavior of the batch transformation. As ground truth transformation, 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 may be considered not as correct as the transformation of 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 due to incremental updates to the source model. We will consider a transformation as *complete*, if it passes the correctness criteria defined above. If the transformations fail at any point, we gradually consider it as incomplete by manually inspecting the reasons of their failure. The following criteria will be evaluated.

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

(1) fail (due to an 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 type

The reference batch solution assumes behavior (3). However, it may depend on the concrete transformation case and the specific rule which strategy is optimal in the respective situation. 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.

Dangling 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 only a single object is added without a reference in the target model as consequence of a missing reference in the source model, it depends on its kind whether references in the target model are needed or not. A table and a type can be integrated into the target

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

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 its kind which transformation behavior occurs. For a multi-valued attribute, a table may be created but accessing the owner of the attribute will be an access of a null-value and will have to be solve 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 in the table created for its owner. As the link to the owner is missing, the transformation can assume the following behavior:

(1) do not create the dangling object (roll-back)
(2) ignore the missing link and leave the dangling object
(3) add the object to the first object of the right type

Again, the potential three 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 may not scale in terms of execution and storage time.

Dangling References. Similar as with dangling objects, due to deleted objects or solely added references, references may be missing one end. If one end is deleted 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 the strategy (2). If a solution assumes strategy (1), however, we will manually score it as correct and complete in the same way.

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. We plan to consider those in the prospective journal extension.

3.3 Syntactic Complexity: Labeling of Transformations

To compare solutions developed in different languages, we rely on 'syntactic complexity'. This size 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 (()[]{}) [8]. 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 consider relevant 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 must only include 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 though the entire submitted project.

Figure 3 depicts an excerpt of a Java implementation of the Class2Relational case which is labeled as we expect solutions to be. 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 43-48 (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. On top, we require authors to label the solution in order for us to evaluate its complexity as explained inSec. 3.3. The score of a submitted solution will be computed from these three criteria.

However, we welcome authors to elaborate on further features of their solution. Although not explicitly required, the authors can report on further quality aspects their solution contributes to. 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. involve the completeness of the transformation as well as the correctness. The latter is implicitly used 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 self evaluate the correctness of their solution and to improve their transformation.

Self-Reported. measures regard the *performance* in terms of execution time, the syntactic complexity in terms of labeling the solution specification as well as further optionally reported solution-specific properties which may benefit any of the both aspects.

Performance: 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.

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 (()[]{}) [8]. 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. While we will not score them, 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 are available on the repository of the case https://github.com/ATL-Research/incremental-class2relational. Source code of the batch ATL transformation, the incremental ATOL transformation, sources 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 [6] for detailed explanations.

The benchmark validates the correctness and completeness of proposed solutions.

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 instead 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`[2] 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.

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

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

(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.

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] Freddie Allilaire. 2023. *ATL Transformations | The Eclipse Foundation.* https://www.eclipse.org/atl/atlTransformations/ Modified: April 6, 2023 at 13:23:55 GMT+2.

[2] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. 2017. The Families to Persons Case. In *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017) (CEUR Workshop Proceedings, Vol. 2026)*, Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.). CEUR-WS.org, 27–34. https://ceur-ws.org/Vol-2026/paper2.pdf

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

[4] Antonio García-Domínguez and Georg Hinkel. 2019. The TTC 2019 Live Case: BibTeX to DocBook. In *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 (CEUR Workshop Proceedings, Vol. 2550)*, Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.). CEUR-WS.org, 61–65. https://ceur-ws.org/Vol-2550/paper8.pdf

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

[6] Georg Hinkel. 2018. The TTC 2018 Social Media Case. In *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 (CEUR Workshop Proceedings, Vol. 2310)*, Antonio García-Domínguez, Georg Hinkel, and Filip Krikava (Eds.). CEUR-WS.org, 39–43. https://ceur-ws.org/Vol-2310/paper5.pdf

[7] Stefan Höppner, Yves Haas, Matthias Tichy, and Katharina Juhnke. 2022. Advantages and disadvantages of (dedicated) model transformation languages. 27, 6 (2022), 159. https://doi.org/10.1007/s10664-022-10194-7

[8] Stefan Höppner, Timo Kehrer, and Matthias Tichy. 2021. 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). https://doi.org/10.1007/s10270-021-00937-3

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

[10] Théo Le Calvar, Frédéric Jouault, Fabien Chhel, Frédéric Saubion, and Mickael Clavreul. 2019. 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).* 395–402. https://doi.org/10.1109/MODELS-C.2019.00061

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

[12] Bernhard Westfechtel. 2016. A Case Study for a Bidirectional Transformation Between Heterogeneous Metamodels in QVT Relations. In *Evaluation of Novel Approaches to Software Engineering*, Leszek A. Maciaszek and Joaquim Filipe (Eds.). Springer International Publishing, Cham, 141–161. https://doi.org/10.1007/978-3-319-30243-0_8

```
public class Class2RelationalIncremental {
    // Setup 8
    private static final RelationalFactory RELATIONALFACTORY = RelationalFactory.eINSTANCE;
    ...
    // Helper 4
    private static Type objectIdType() {
        ...
        // Helper 2
        return objectIdType;
    }
    ...
    // Setup 8
    public static Resource start(String inPath, String outPath) {
        ...
        // Incrementality 5
        Adapter adapterIn = new AdapterImpl() {
            // Incrementality 5
            public void notifyChanged(Notification notification)
            ...
        }
    }
    // Traversal 8
    public static List<Named> transform(List<EObject> input) {
        // Traversal 5
        for (EObject namedElt : input) {
            ...
        }
    }
    ...
    // Tracing 6
    public static void Class2TablePre(Class c) {
        // Tracing 5
        TRACER.addTrace(c, RELATIONALFACTORY.createTable());
        ...
    }
    // Transformation 6
    public static void Class2Table(Class c) {
        // Tracing 8
        var out = TRACER.resolve(c, RELATIONALFACTORY.createTable());
        ...
        // Transformation 12
        // Tracing 16
        out.getCol().addAll(
            c.getAttr().stream()
                .filter(e -> !e.isMultiValued())
                .map($ -> TRACER.resolve($, RELATIONALFACTORY.createColumn()))
                .filter($ -> $ != null)
            .collect(Collectors.toList()));
    }
    ...
}
```

Figure 3: Example labeling for Java