A BXtendDSL Solution to the TTC2023 Incremental MTL vs. GPLs Case

Thomas Buchmann
thomas.buchmann@th-deg.de
Deggendorf Institute of Technology
Deggendorf, GER

ABSTRACT

This paper presents a solution to the Case at TTC 2023 using BXtendDSL. BXtendDSL is hybrid language for bidirectional and incremental model transformations, allowing transformation developers to specify model transformations on the declarative and imperative level, allowing for maximum expressive power to tackle all possible transformation problems.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

incremental transformations, Model Transformation Language, GPL, Class model, relational data schema

ACM Reference Format:
Thomas Buchmann. 2023. A BXtendDSL Solution to the TTC2023 Incremental MTL vs. GPLs Case. In *Proceedings of 15th Transformation Tool Contest (TTC'23).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The transformation case addresses a comparison between dedicated model transformation languages (MTLs) and general purpose programming languages (GPLs) in the context of an incremental transformation of Class models into Relational Data Schemas. Since model transformation languages typically are domain-specific languages tailored to efficiently express model-to-model transformations, they comprise high-level constructs like rules and automatic support for traceability which are missing in GPLs. Furthermore, MTLs often provide different modes of execution: In a batch transformation, the input model is transformed and an output model is produced. An incremental transformation on the other hand is able to propagate changes from the input model to the output model while retaining changes in the output model. Some MTLs also support for bidirectional transformations, i.e., the output model maybe transformed back into the input model and vice versa.

TTC'23, July 20, 2023, Leicester, UK

ACM ISBN 978-1-4503-XXXX-X/18/06…$0.00
https://doi.org/XXXXXXX.XXXXXXX

During the last decades, a wide range of MTLs and accompanying tool support has been proposed, however, many model transformations in practice are still written in GPLs. While there are reasons for this situation in the context of the batch execution of a transformation, an incremental transformation has different requirements and should shift the focus towards dedicated MTLs.

The proposed case addresses an incremental transformation scenario of class diagrams into relational data schemas with the aim to compare solutions written in MTLs with solutions written in GPLs.

In this paper, we present our solution to the proposed transformation case using BXtendDSL [2, 4, 5] – our hybrid language for bidirectional and incremental model transformations.

2 BXTENDDSL

BXtendDSL [2, 4, 5] is a state-based framework for defining and executing bidirectional incremental model transformations on demand that is based on EMF [6] and the programming language Xtend[1]. It builds upon BXtend [3], a framework that follows a pragmatic approach to programming bidirectional transformations, with a special emphasis on problems encountered in the practical application of existing bidirectional transformation languages and tools.

When working with the stand-alone BXtend framework, the transformation developer needs to specify both transformation directions separately, resulting in BXtend transformation rules with a significant portion of repetitive code.

To this end, BXtendDSL adds a declarative layer on top of the BXtend framework, which significantly reduces the effort required by the transformation developer. Figure 1 depicts the layered approach of our tool: First, the external DSL (BXtendDSL Declarative) is used to specify correspondences declaratively. Second, the internal DSL (BXtendDSL Imperative) is employed to add algorithmic details of the transformation that can not be expressed on the declarative layer adequately.

The handwritten code and the generated code are combined with framework code to provide for an executable transformation. The transformation developer is relieved from writing repetitive routine parts of the transformation manually using a code generator. The generated code ensures roundtrip properties for simple parts of the transformation. Since the declarative DSL usually is not expressive enough to solve the transformation problem at hand completely, the generated code must be combined with handwritten imperative code. Certain language constructs of the declarative DSL define the interface between the declarative and the imperative parts of the transformation. From these constructs, *hook methods* are generated, the bodies of which must be manually implemented. Hook methods

[1] https://eclipse.dev/Xtext/xtend/

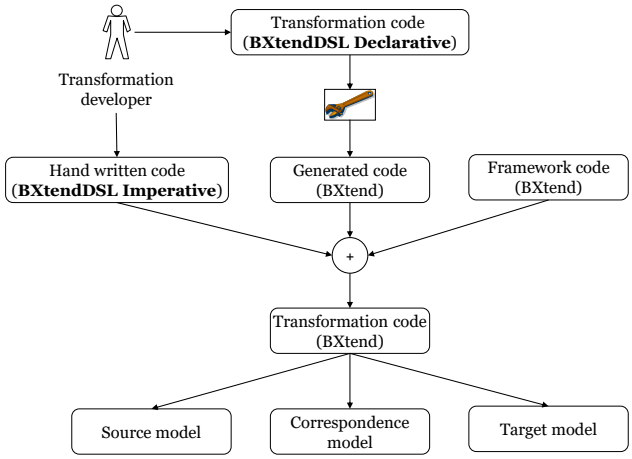

Figure 1: Layered approach used in BXtendDSL

are used, e.g. for implementing filters or actions to be executed in response to the deletion or creation of objects, etc.

Incremental change propagation relies on a persistently stored *correspondence model*, which allows for m : n correspondences between source and target model elements. A powerful *internal DSL* may be used at the imperative level, to retrieve correspondence model elements associated with a given element from the source and target models, respectively. Please note that the transformation developer does not have to deal with managing correspondences at the declarative level, rather all the algorithmic details of managing the correspondence model are handled by our framework automatically.

3 SOLUTION

In this section, we explain the details of our BXtendDSL solution for the Class into Relational Data Schema case. We will discuss the different layers in separate subsections. Please note that incremental behavior is provided automatically by our framework, so the transformation developer does not need to address it specifically.

3.1 Declarative Layer

BXtendDSL code at the declarative layer is used to define transformation rules between elements of source and target models respectively. Listing 1 depicts the code for the transformation at the declarative layer. Since the transformation is unidirectional, all mappings are directed from source (Class) to target (Relational) model, indicated by the –> symbol.

```
1  sourcemodel "Class"
2  targetmodel "Relational"
3
4  rule DataType2Type
5    src DataType dt;
6    trg Type t;
7
8    dt.name --> t.name;
9
10 rule SingleAttribute2Column
11   src Attribute att | filter;
12   trg Column col;
13
14   att.name --> col.name;
15   {att.type : DataType2Type} --> {col.type : DataType2Type};
16
17 rule MultiAttribute2Table
18   src Attribute att | filter;
19   trg Table tbl;
20
21   att.name att.owner --> tbl.name;
22   att.name att.type att.owner --> tbl.col;
23
24 rule SingleClassAttribute2Column
25   src Attribute att | filter;
26   trg Column col;
27
28   att.name att.type --> col.name;
29   att.name att.type --> col.type;
30
31 rule MultiClassAttribute2Column
32   src Attribute att | filter;
33   trg Table t;
34    Column id | creation;
35    Column fk | creation;
36
37   att.name att.owner --> t.name;
38   att.name att.owner --> id.name;
39   att.name att.owner --> fk.name;
40
41 rule Class2Table
42   src Class clz;
43   trg Table tbl | creation;
44
45   clz.name --> tbl.name;
46   {clz.attr : SingleAttribute2Column, SingleClassAttribute2Column,
         MultiAttribute2Table} --> tbl.col;
```

Listing 1: BXtendDSL code at the declarative layer

The declarative transformation specification comprises rules for all required model elements. Each rule is composed of src and trg patterns with elements of source and target models, respectively. Some patterns make use of modifiers, such as filter and creation. Those modifiers are transformed into hook methods, whose bodies need to be implemented by the transformation developer on the imperative layer (see, Section 3.2). After declarting src and trg patterns, the mapping of attributes and references is specified by *mappings*. As explained above, we only use directed mappings in this transformation (–>). Lines 4-8 depict the transformation rule for DataTypes and Types. A DataType object from the class model is mapped to a Type object in the relational model and the datatype name is assigned to the attribute name of the Type.

Rule singleAttribute2Column employs a filter modifier on the source pattern. This is required to indicate that the rule should only be applied to Attributes that are singlevalued and whose type refers to a DataType. Please note that no algorithmic details for the filter are specified on the declarative level, since this would have required a much more expressive and thus complex language. Rather a hook method is generated and the behavior is specified on the imperative layer using the Xtend programming language (see Section 3.2).

Furthermore, the mapping in Line 15 is enclosed in curly brackets. This indicates, that references to already transformed elements should be used and retrieved from the correspondence model. The execution of the rules follows the textual order as specified in the declarative specification, i.e. the rule DataType2Type is actually executed before the rule SingleAttribute2Column, which means that when we want to apply the mapping, we can be sure that the respective types already exist in the target model and we can easily retrieve them from the correspondence model (i.e., the trace model).

In case that the types of structural features used in the mapping is not compatible, a hook method is also generated. As well in cases where more than one structural feature is used on either side of the arrow symbol (e.g. in Line 21 of Listing 1).

Please note that source or target patterns may consist of more then one element, as shown e.g. in Lines 33-35. If a multivalued attribute with a type reference that is not a datatype is transformed, a new table consisting an objectID and a foreign key should be created. For the two columns a *creation* modifier is used, which allows the transformation developer to add additional imperative code that is executed after new elements have been created (in our case, the columns get the required type reference and are added to the parent table).

The last rule that is executed is Class2Table. When this rule is executed, all columns that have been transformed because other rules have been applied, actually exist and can be assigned to the proper tables in the mapping depicted in Line 46.

3.2 Imperative Layer

On the imperative layer, the bodies for hook methods must be supplied. This holds for the specification of modifiers (e.g., filter or creation), as well as for mappings where further information is required, which cannot be supplied using the declarative language only.

Listing 2: Hook method for mapping filtering attributes

```
1 override protected filterAtt(Attribute att) {
2   (att.isMultiValued) && (att.type instanceof Class)
3 }
```

Listing 2 depicts the implementation of a filter, specified on the declarative layer in the rule MultiAttribute2Column (see Line 32, Listing 1). The rule should only consider attributes which are multivalued and whose type is a Class. Similar implementations have been supplied for the other filter modifiers.

Listing 3: Creation hook

```
1 override protected onIdCreation(Column id) {
2   id.type = Utils.getType(findIntegerDatatype())
3   id.corr.target().t.col += id
4 }
```

Listing 3 depicts the implementation of a creation hook method. Using creation modifiers on the declarative layer results in the generation of respective methods, that need to be implemented on the imperative layer. The method shown in Listing 3, is called when the id Column is created during the exeuction of rule MultiClassAttribute2Column (see Line 34 in Listing 1). The id column has Integer type and the respective Object is retrieved by the utility methods getType() and findIntegerDatatype(), which have been added to the imperative layer manually. Finally, the column is added to its parent table's reference col.

Listing 4: Hook method for feature mapping

```
1  override protected colFrom(String attName,
2     Classifier type, Class owner) {
3   val colList = newArrayList
4   val idCol = RelationalFactory.eINSTANCE
5     .createColumn() => [
6    name = owner.name.toFirstLower + "Id"
7    type = Utils.getType(findIntegerDatatype())
8   ]
9   val valCol = RelationalFactory.eINSTANCE
10    .createColumn() => [
11   name = attName
12   type = Utils.getType(type)
13  ]
14  colList += idCol
15  colList += valCol
16  return new Type4col(colList)
17 }
```

Listing 4 depicts the hook method that is created as a result of the feature mapping defined in Line 22 of Listing 1. The rule MultiAttribute2Table is called, when a multivalued attribute with a *primitive* type is transformed into a Table with id-Column and value-Column. Please note that in the declarative specification, only the target table is created, the corresponding columns are then created in the hook method. The required information to create the columns is passed to the hook method as input parameters. The hook method is required to return a predefined Xtend @Data-class. When creating the columns and assigning the respective types, the Utility methods explained above are reused.

Listing 5: Hook method for mapping attribute type + name to table name

```
1 override protected tblNameFrom(String attName,
2    Class owner) {
3   var tblName = owner.name
4   if (tblName === null || tblName === "") tblName = "Table"
5   new Type4tblName(owner.name + "_" + attName)
6 }
```

Listing 5 depicts another hook method which is created because two features on the source side (Attribute.name and Attribute.owner) are mapped to a single feature on the target side (Table.name). The method stub is generated as a result of the statement specified in Line 21 of Listing 1. In the imperative implementation of the hook, we check if the owner has a name value. If this is the case it is concatenated with the attribute name, otherwise we use the prefix "Table" and concatenate it with the attribute name.

Listing 6: Hook method for adding all columns to the right tables

```
1  override protected colFrom(List<Column> attSinCol,
2     List<Column> attSinCol_2, List<Table> attMulT,
3     Table parent) {
4   val columnsList = newArrayList
5   if (!parent.col.empty) {
6    var key = parent.col.get(0)
7    columnsList += key
8   }
9
10  for (Column c : attSinCol) {
11   var obj = unwrap(c.corr.
12     source.get(0) as SingleElem) as Attribute
13   if (obj.type !== null) {
14    columnsList += c
15   } else {
16    c.owner = null
17    EcoreUtil.delete(c, true)
18   }
19  }
20
21  for (Column c : attSinCol_2) {
22   var obj = unwrap(c.corr.
23     source.get(0) as SingleElem) as Attribute
24   if (obj.type !== null)
25        columnsList += c
26   else EcoreUtil.delete(c, true)
27  }
28  for (Table t : attMulT) {
29   var obj = unwrap(t.corr.
30    source.get(0) as SingleElem) as Attribute
31   if (obj.type === null)
32    EcoreUtil.delete(t, true);
```

```
33    }
34    new Type4col(columnsList)
35  }
```

Finally, the last Listing discussed in this paper is shown in Listing 6. The method stub is generated as a result of the feature mapping depicted in Line 46 of Listing 1. It is used to assign all columns to their respective parent tables. Furthermore, we address handling the dangling references in the code specified in the imperative layer. Lists of columns and tables, that have been transformed when the other rules have been applied are passed as method parameters. Before adding the respective column to the resulting data object (Type4col), we make sure that its associated source object actually has a non-null type-reference. If the associated type is null, we delete the column.

4 EVALUATION

In order to evaluate the solution, solution providers are requested to integrate it into the given framework. For technical reasons, it was not possible (yet) to integrate the BXtendDSL solution directly to the framework. Thus, we provide an additional project BXtendDSLSolutionRunner in our GitHub repository, which programmatically initializes the requires source models, applies the changes specified in the changes models, executes the transformation and saves the obtained target models in the respective folders. We compared the obtained models with the provided expected models and they seem to be both correct and complete according to the evaluation criteria requested in the case description.

We labeled the solution as much as possible. Please note that most of the code that deals with model traversal is generated and does not need to be supplied manually by the transformation developer. The same holds for code that ensures incrementality.

Regarding performance, the provided models are too small to obtain sounding results for execution times, as they are around several milliseconds. In other (and larger performance tests), BXtendDSL has already proven to scale excellent with growing model sizes [1, 4].

5 CONCLUSION

In this paper, we described our BXtendDSL solution to the Incremental MTL vs. GPLs Case. Due to technical reasons, we were unable to run the automated tests, but we thoroughly tested each of the sample models provided, and manually compared the results obtained from our transformation with the expected ones.

The transformation case also helped to reveal a bug in our code generation engine, which will be fixed in the upcoming release of BXtendDSL. Please follow the instructions given in the README file of the public Git repository in order to get the BXtendDSL solution to compile without errors.

RESOURCES

The BXtendDSL solution may be obtained from a public GitHub repository, which can be found at https://github.com/tbuchmann/Incremental-class2relational.

REFERENCES

[1] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. 2020. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling* 19, 3 (May 2020), 647–691. https://doi.org/10.1007/s10270-019-00752-x

[2] Matthias Bank, Thomas Buchmann, and Bernhard Westfechtel. 2021. Combining a Declarative Language and an Imperative Language for Bidirectional Incremental Model Transformations. In *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021*, Slimane Hammoudi, Luís Ferreira Pires, Edwin Seidewitz, and Richard Soley (Eds.). SCITEPRESS, 15–27. https://doi.org/10.5220/0010188200150027

[3] Thomas Buchmann. 2018. BXtend - A Framework for (Bidirectional) Incremental Model Transformations. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*. 336–345. https://doi.org/10.5220/0006563503360345

[4] Thomas Buchmann, Matthias Bank, and Bernhard Westfechtel. 2022. BXtendDSL: A layered framework for bidirectional model transformations combining a declarative and an imperative language. *J. Syst. Softw.* 189 (2022), 111288. https://doi.org/10.1016/j.jss.2022.111288

[5] Thomas Buchmann, Matthias Bank, and Bernhard Westfechtel. 2023. BXtendDSL at Work: Combining Declarative and Imperative Programming of Bidirectional Model Transformations. *SN Comput. Sci.* 4, 1 (2023), 50. https://doi.org/10.1007/s42979-022-01448-8

[6] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF Eclipse Modeling Framework* (2nd ed.). Addison-Wesley, Boston, MA.