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

Thomas Buchmann[1,*,†]

[1]Deggendorf Institute of Technology, Dieter-Görlitz-Platz 1, 94469 Deggendorf

Abstract

This paper presents a solution to the Case at TTC 2023 using BXtendDSL. BXtendDSL is declarative language for bidirectional and incremental model transformations, built on top of an imperative framework. The transformation developer may extend the transformation on the imperative layer whenever the expressive power of the declarative language is not enough to tackle the transformation problem at hand. Thus, BXtendDSL provides a flexible and powerful tool for all possible transformation problems.

Keywords

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

1. Introduction

The "Incremental MTL vs. GPLs: Class into Relational Database Schema" case [1] from the 2023 edition of the Transformation Tool Contest (TTC) 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.

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. The research question in the transformation case is to determine whether MTLs perform better than GPLs in incremental transformation scenarios.

In this paper, we present our solution to the proposed transformation case using BXtendDSL [2, 3, 4] – our hybrid language for bidirectional and incremental model transformations. BXtendDSL is a dedicated language for bidirectional and incremental model transformations, i.e., the transformation developer is relieved from addressing tracing and incrementality, as it is handled completely by the underlying framework. Besides a declarative language for specifying relations between source and target model elements, BXtendDSL provides an imperative layer, which may be used whenever parts of the transformation problem at hand can not be expressed on the declarative layer.

The paper is structured as follows: In Section 2, we provide an overview about BXtendDSL. Section **??** describes both the declarative and imperative parts of our solution to the transformation case, followed by a detailed evaluation according to different criteria in Section 4. Section 5 concludes the paper.

2. BXtendDSL

BXtendDSL [2, 3, 4] is a state-based framework for defining and executing bidirectional incremental model transformations that is based on EMF [5] and the programming language Xtend[1]. It builds upon BXtend [6], 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.

TTC'23, 15th Transformation Tool Contest, July 20, 2023, Leicester, UK
thomas.buchmann@th-deg.de (T. Buchmann)
https://tbuchmann.github.io/ (T. Buchmann)
0000-0002-5675-6339 (T. Buchmann)

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

The stand-alone BXtend framework is completely integrated (and slightly exented, c.f., [3]) into BXtendDSL, i.e. no additional dependencies are required.

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.

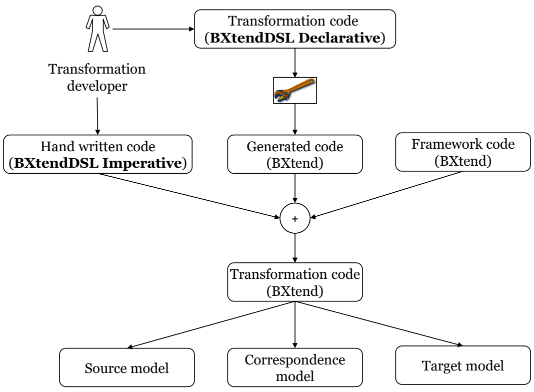

Figure 1: Layered approach used in BXtendDSL

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 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. The source code of our solution is publicly available on GitHub.

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. Although BXtendDSL supports bidirectional transformations, the current transformation case requires an unidirectional transformation only. Thus, all mappings are directed from source (Class) to target (Relational) model, indicated by the `-->` symbol.

```
sourcemodel "Class"
targetmodel "Relational"

rule DataType2Type
src DataType dt;
trg Type t;

dt.name --> t.name;

rule SingleAttribute2Column
src Attribute att | filter;
trg Column col;

att.name --> col.name;
{att.type : DataType2Type} --> {col.type : DataType2Type};

rule MultiAttribute2Table
src Attribute att | filter;
trg Table tbl;

att.name att.owner --> tbl.name;
att.name att.type att.owner --> tbl.col;

rule SingleClassAttribute2Column
src Attribute att | filter;
trg Column col;

att.name att.type --> col.name;
att.name att.type --> col.type;

rule MultiClassAttribute2Column
src Attribute att | filter;
trg Table t;
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 declaring 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 of 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. Similar filter implementations are used for single valued attributes and attributes whose type is a datatype. This also works in an incremental way, e.g. if the multi-valued property of an attribute is changed, or if the type of an attribute changes. Please note that all manual changes are retained in case the declarative file changes and code is regenerated.

Listing 2: Hook method for mapping filtering attributes

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

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

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

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 execution 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. Please note that both utility method do not modify the model, they are only used to retrieve the matching values. The (incremental) transformation of types is handled by the rule DataType2Type on the declarative layer (see Listing 1, lines 4-8).

Listing 4: Hook method for feature mapping

```
override protected colFrom(String attName,
Classifier type, Class owner) {
  val colList = newArrayList
  val columnName = (owner === null
    || owner.name === null
    || owner.name === "")? "tableId"
     : owner.name.toFirstLower + "Id"
  val idCol = RelationalFactory.eINSTANCE
   .createColumn() => [name = columnName
    type = Utils.getType(findIntegerDatatype())
  ]
  val valCol = RelationalFactory.eINSTANCE
   .createColumn() => [
    name = attName
    type = Utils.getType(type)
   ]
  colList += idCol
  colList += valCol
  return new Type4col(colList)
}
```

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 multi-valued 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. Please note that in the current implementation of the hook method, it does not work in an incremental way. I.e., the columns are not reused, rather they are recreated.

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

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

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

```
override protected colFrom(List<Column> attSinCol,
List<Column> attSinCol_2, List<Table> attMulT,
Table parent) {
  val columnsList = newArrayList
  if (!parent.col.empty) {
   var key = parent.col.get(0)
   columnsList += key
  }

  for (Column c : attSinCol) {
   var obj = unwrap(c.corr.
    source.get(0) as SingleElem) as Attribute
   if (obj.type !== null) {
    columnsList += c
   } else {
    c.owner = null
    EcoreUtil.delete(c, true)
   }
  }

  for (Column c : attSinCol_2) {
   var obj = unwrap(c.corr.
   source.get(0) as SingleElem) as Attribute
   if (obj.type !== null)
    columnsList += c
   else EcoreUtil.delete(c, true)
  }

  for (Table t : attMulT) {
   var obj = unwrap(t.corr.
    source.get(0) as SingleElem) as Attribute
   if (obj.type === null)
    EcoreUtil.delete(t, true);
  }
  new Type4col(columnsList)
}
```

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

The implementation of the solution to this transformation case was pretty straightforward using BXtendDSL. Since only one transformation direction was required, directed mappings could be used. While incrementality comes for free, not every aspect of the transformation at hand can be expressed on the declarative layer of BXtendDSL only. Thus, a significant portion of the transformation code had to be supplied via filters and hook methods on the imperative layer using the Xtend programming language.

In our GitHub repository (see Appendix ??), the project `BXtendDSLSolutionRunner` is used to integrate the BXtendDSL solution into the evaluation framework provided by the case authors. In order to execute it, an executable JAR file has to be created from the BXtendDSLSolutionRunner project, which can then be called from the shell scripts used for evaluation in the framework.

The results show that the BXtendDSL solution is correct (i.e. commuting batch and incremental transformations) in every of the provided test cases. In a second test criterion (completeness), the resulting target models are compared against predefined expected models. Only in three out of thirteen cases, the obtained model after the transformation does not match any of the predefined expected models. See table 1 for a detailed analysis.

Test	Correctness	Completeness
correctness1	ok	expected1.xmi
correctness2	ok	no match
correctness3	ok	expected1.xmi
correctness4	ok	expected1.xmi
correctness5	ok	expected1.xmi
correctness6	ok	expected1.xmi
correctness7	ok	expected1.xmi
correctness8	ok	expected1.xmi
correctness9	ok	expected1.xmi
correctness10	ok	expected2.xmi
correctness11	ok	expected2.xmi
correctness12	ok	no match
correctness13	ok	no match
correctness_couple	ok	no expected
correctness_full	ok	no expected
scale1	ok	no expected
scale200	ok	no expected
scale2000	ok	no expected

Table 1
Correctness and Completeness of our solution.

We labeled our solution according to the requirements stated in the case description. However, we also determined the specification effort in terms of LOC metrics as used e.g. in [7]. Furthermore, we obtained separate numbers for the declarative and the imperative layer. Our solution is concise and requires only a moderate specification effort, due to the fact that incrementality and tracing is automatically provided by our framework and does not need to be addressed explicitly by the transformation developer. The results are depicted in table 2.

	Declarative Layer	Imperative Layer
Number of lines	48	117
Number of words	131	394
Number of characters	974	3343

Table 2
Size of the transformation definition.

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 [7, 3].

5. Conclusion

In this paper, we described our BXtendDSL solution to the Incremental MTL vs. GPLs Case. The transformation case aims at investigating the benefit of dedicated MTLs specifically in terms of the incremental nature of the transformation problem at hand. BXtendDSL is a dedicated language for bidirectional and incremental model transformations which provides tracing and incremental functionality automatically.

The transformation developer may focus only on the current transformation problem without taking into account these technical details. Using the declarative part of BXtendDSL, the transformation developer specifies relations between source and target model elements, and whenever the expressive power of the declarative layer is not enough to tackle parts of the transformation problem, the developer may switch to the imperative layer to specify the algorithmic details. Thus, the overall solution is very concise while it completely fulfills the commutativity criterion and almost every completeness criterion of the evaluation framework.

The transformation case 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] S. Greiner, S. Höppner, F. Jouault, T. Le Calvar, M. Clavreul, Incremental mtl vs. gpls: Class into relational database schema (2023).

[2] M. Bank, T. Buchmann, B. Westfechtel, Combining a declarative language and an imperative language for bidirectional incremental model transformations, in: S. Hammoudi, L. F. Pires, E. Seidewitz, R. Soley (Eds.), Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021, Online Streaming, February 8-10, 2021, SCITEPRESS, 2021, pp. 15–27. URL: https://doi.org/10.5220/0010188200150027. doi:10.5220/0010188200150027.

[3] T. Buchmann, M. Bank, B. Westfechtel, Bxtenddsl: A layered framework for bidirectional model transformations combining a declarative and an imperative language, J. Syst. Softw. 189 (2022) 111288. URL: https://doi.org/10.1016/j.jss.2022.111288. doi:10.1016/j.jss.2022.111288.

[4] T. Buchmann, M. Bank, B. Westfechtel, Bxtenddsl at work: Combining declarative and imperative programming of bidirectional model transformations, SN Comput. Sci. 4 (2023) 50. URL: https://doi.org/10.1007/s42979-022-01448-8. doi:10.1007/s42979-022-01448-8.

[5] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks, EMF Eclipse Modeling Framework, The Eclipse Series, 2nd ed., Addison-Wesley, Boston, MA, 2009.

[6] T. Buchmann, 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., 2018, pp. 336–345. URL: https://doi.org/10.5220/0006563503360345. doi:10.5220/0006563503360345.

[7] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H.-S. 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.