

Incremental Model Transformation with Epsilon in Model-Driven Engineering

Marzieh Ghorbani, Mohammadreza Sharbaf , Bahman Zamani 

MDSE Research Group, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran

Corresponding author: Mohammadreza Sharbaf (m.sharbaf@eng.ui.ac.ir)

Abstract

Model-Driven Engineering (MDE) is a software development paradigm that uses models as the main artifacts in the development process. MDE uses model transformations to propagate changes between source and target models. In some development scenarios, target models should be updated based on the evolution of source models. In such cases, it is required to re-execute all transformation rules to update the target model. Incremental execution of transformations, which partially executes the transformation rules, is a solution to this problem. The Epsilon Transformation Language (ETL) is a well-known model transformation language that does not support incremental executions. In this paper, we propose an approach to support the incremental execution of ETL transformations. Our proposal includes a process, as well as a prototype, to propagate changes to the target model. In the proposed approach, all the changes in the source model are detected to identify and re-execute the rules which deal with computing the required elements for updating the target model. We evaluated the correctness and performance of our approach by means of a case study. Compared to the standard ETL, the results are promising regarding the correctness of target models as well as faster execution of the transformation.

Keywords

Model-Driven engineering; Model transformation; Change propagation; Incremental transformation; Epsilon transformation language.

1 Introduction

Nowadays, software development approaches seek a solution to cope with the complexity of software systems. Model-driven engineering (MDE) (Brambilla et al., 2017) is one of the latest software development paradigms, which proposes the idea of increasing the abstraction level to facilitate the software development process. Models are first-class citizens of MDE, which describe software systems at a high level of abstraction (Balsamo et al., 2004). Model transformation is an integral part of modelling, which is used to produce new target models from existing source models (Sendall & Kozaczynski, 2003). Model transformation can be specified by various transformation languages such as QVT (OMG, 2016), VIATRA (Csertán et al., 2002), ETL (Kolovos et al., 2008), and ATL (Jouault & Kurtev, 2006). Each language is developed with different objectives and is based on various technologies.

Due to the new requirements in model-driven software development, models are updated and evolved like other artifacts during the system development life cycle. Therefore, it is necessary to update the target models which are derived from the updated source models. Model transformation can be used to propagate changes between models. However, the typical batch transformation method, which re-executes whole transformation rules to compute the updated target model, is not an appropriate method for change propagation between large models. In other words, for every change in the source model, all transformation rules are carried out to get the updated target model (Jouault & Tisi, 2010). Hence, for the source model, which contains too many elements, the transformation process takes a long time to apply changes to the target model.

In this case, a solution is the incremental execution of model transformation to update the target model only based on the changes in the source model (Hearnden et al., 2006). In the incremental execution approach, the transformation is only executed for the elements that have been changed in the source model to generate the corresponding elements in the new target model. For example, if the name property of an element in the source model is changed, only the name of the corresponding element in the target model will be changed. In other words, in this approach, changes are propagated from the source model to the target model. This approach covers batch transformation drawbacks and is commonly known as incremental transformation (Jouault & Tisi, 2010). Incrementality is a key option for the execution of transformation that can enhance the power of batch transformation engines (Johann & Egyed, 2004). The Epsilon framework (Paige et al., 2009) provides a set of languages and tools for model management. The Epsilon Transformation Language (ETL) is a well-known model-to-model transformation language in MDE built upon the Epsilon framework. However, the standard ETL runs transformations in batch mode (Kolovos et al., 2008), and it does not support incremental execution of transformations. Moreover, with the advent of model-based software projects, efficient execution of ETL transformations for evolution of large models is important.

In this paper, we propose a process for incremental execution of ETL transformations to support change propagation by ETL and achieve efficient execution of ETL transformations for model evolution. We aim to perform model transformations incrementally to only execute ETL transformations for changes applied to the source model. The ETL incremental execution process consists of four steps. In the first step, both initial and updated versions of the source model are compared to detect changed elements in the source model. In the second step, using the changed elements and given the input parameters of the rules, the process identifies the transformation rules which are related to the changed elements. In the third step, identified rules are executed separately. Finally, in the fourth step, the target model is updated based on the rule execution results by adding or removing elements. To evaluate the correctness and performance of our process, we run a case study. In the case study, we investigate the execution time and scalability of our approach to supporting incremental transformation over different sizes of source models and different numbers of changes. The results show that the proposed process is scalable for increasing the size of the source model and the number of applied changes.

The paper is organized as follows. Section 2 briefly provides background information on model transformation strategies and incremental execution techniques. This section also introduces the Epsilon Transformation Language. Section 3 introduces the research method used to carry out our research. Section 4 presents a running example of incremental transformation. Section 5 describes the steps of the proposed approach, which includes an overview of the approach and a brief description of its implementation. Section 6 discusses the experimental results of a descriptive case study and assesses the correctness, performance and scalability of our approach. Section 7 reports the related work; and we conclude the paper and outline some future lines of work in Section 8.

2 Background

In this section, we first review different strategies for the execution of a model transformation, including batch, incremental, lazy and reactive. Then, we investigate the characteristics of existing methods for incremental execution of model transformation in the case of change propagation from the source model into the target model. Finally, we introduce ETL, a hybrid model transformation language that supports model-to-model transformation in the Epsilon framework.

2.1 Execution strategies for model transformation

Software evolution is possible by propagating changes from one or more developed artifacts into other related artifacts. In model-driven engineering, model transformation plays an essential role in the development process, but efficiently propagating changes between artifacts is significant, which has been the subject matter of several studies (Hearnden et al., 2006; Jouault & Tisi, 2010). In *batch* strategy, as a common model transformation approach, all of the source model elements are loaded, and by matching and executing all the transformation rules, the target model is produced. However, in the software evolution process, re-execution of the transformation for all elements of the model is not efficient (Le Calvar et al., 2019).

Therefore, the *incremental* transformation strategy has been introduced, working based on the changed elements and ignoring the re-execution of transformation rules for other elements. In the last decade, several incremental approaches to model transformation have been proposed, focusing on the efficient propagation of changes in different transformation languages. Most of these approaches can be characterized in three distinct phases, including *change detection*, *impact analysis* and *change propagation*, which is executed dedicatedly (Kusel et al., 2013).

Another execution strategy is the *lazy* transformation strategy, which uses the idea of lazy evaluation. It is an approach that calculates the value of the expression if needed. It will speed up the execution of the transformation program. The lazy evaluation approach is used when a lot of data are modified. Applying a lazy evaluation strategy to transformation causes the transformation engine to delay accessing the model element until the transformation does not need that element. When a lazy transformation is executed, the transformation engine only creates the required elements or properties, which reduces the transformation execution time for large models. The lazy transformation strategy is usually used to create a part of the target model in other transformation strategies (Tisi et al., 2011).

Another well-known execution strategy is the *reactive* transformation strategy, in which a transformation engine reacts to each change applied to the source model at the moment (Martínez et al., 2017). In the reactive strategy, the transformation engine instantly detects any changes applied to the source model, then it identifies and executes the desired transformation rule. Sometimes, to execute the transformation rules, references to other elements and some information about elements in the target model are required. To this end, the reactive engine ran a "request" query on the target model to take related elements and required information.

Therefore, a reactive transformation engine only executes rules that are absolutely required for responding to the "request" queries or updating elements in the target model (Martínez et al., 2017). The reactive transformation strategy combines the lazy and incremental strategies (Kolovos et al., 2013).

2.2 Incremental execution techniques

The incremental transformation is used when the target model already exists (Jouault & Tisi, 2010). Based on the transformation execution method, the incremental techniques are divided into three types (Czarnecki & Helsén, 2006):

- **Target incrementally:** in this technique, first, all transformation rules are executed, then the new version of the target model is created. After that, the new target model is merged with the previous target models.
- **Source incrementally:** in this technique, change propagation is made based on the changed elements, and only some rules of the transformation are executed. In this method, when a change occurs in the source model, the rule which is related to that changed element is detected first. Then, by executing the detected rule, the corresponding elements in the target model are generated.
- **Preservation of user edits in the target:** in this technique, it is up to the user to detect and execute the required transformation rules. The user only re-runs the transformation for changes that have occurred in the source model. The user must synchronize the existing target model with a changed source while maintaining user edits.

In this paper, the proposed process to execute ETL transformation incrementally is based on the incremental source technique, which will be discussed in its entirety in Section 5.

2.3 Epsilon Transformation Language (ETL)

ETL is one of the languages of the Epsilon family, which is provided for model-to-model transformations. ETL has been designed based on the hybrid language style, which brings both declarative and imperative advantages (Kolahdouz-Rahimi et al., 2020; Kolovos et al., 2008). The rule-based execution scheme of ETL supports a task-specific rule definition for model-to-model transformation that can transform an arbitrary number of source models into an arbitrary number of target models. Each module of ETL transformation has many rules which are specified by a unique name. Also, each rule consists of an input parameter and several destination parameters. ETL transformation can be used to propagate changes between source and target models conforming to different modelling languages. However, ETL only supports the batch transformation strategy, which is generally less efficient than the incremental strategy for change propagation.

3 Research Method

The main objective of our research is to build a process to execute ETL model transformation incrementally. Therefore, we followed the design science research methodology (DSRM) presented by Peffers et al. (2007). This methodology starts with recognizing the problem that should be tackled, then proposes the solution as a design artifact and finally, assesses the utilization of the created solution with an appropriate scenario. From a top-level methodological perspective, our research methodology is based on five steps, in which we utilize different actions to support our overall objectives. In the following, we investigate the actions necessary to accomplish each step of this research.

- **Problem identification and motivation:** We investigated the problem space and background to overview requirements for incremental execution of a model transformation and have a detailed description of the problem using an illustrative example.
- **Define the objectives of a new solution:** We translated the obtained requirements to define our objectives for the design of a new process for incremental execution of the ETL transformations.
- **Design and development:** We designed an execution process based on the structure of ETL by following the changes that are applied to the source model and identified different transformation rules that are appropriate for each change regarding the previously derived requirements and objectives.
- **Demonstration:** We developed a graphical toolkit for the designed incremental execution process of the ETL transformations.
- **Evaluation:** We evaluated the correctness, execution time and scalability of the incremental execution process of the ETL transformations using a case study following the guidelines proposed by Runeson and Höst (2008).

4 Running Example

In this section, to illustrate more precisely the proposed approach to incremental transformation, we introduce "OO2DB" as a short running example of ETL model transformation, which translates an object-oriented class model into a database model. The ETL transformations should be defined based on the source and destination meta-models. The partial view of OO and DB meta-models, which describe the definition of source and destination models of the OO2DB transformation, are shown in Figure 1 and Figure 2, respectively.

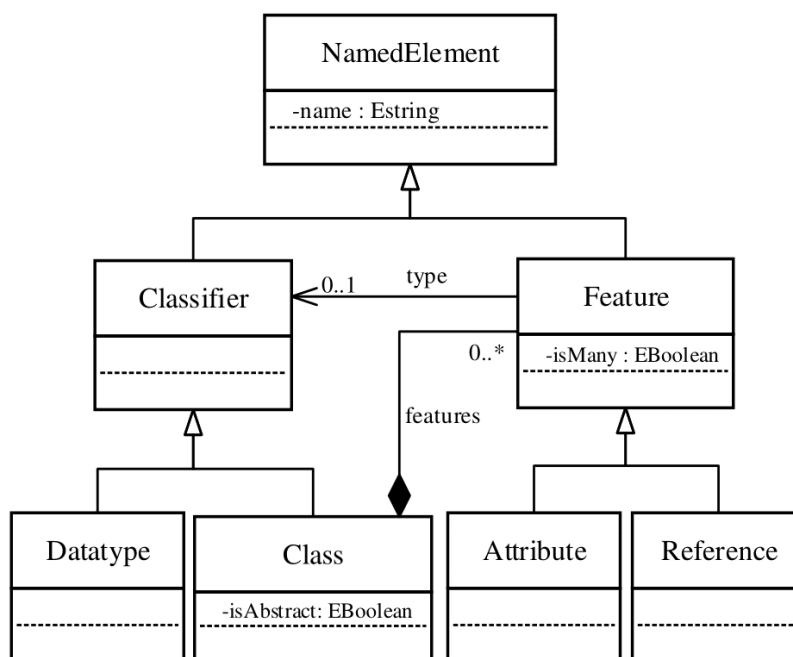


Figure 1. Excerpt of OO metamodel.

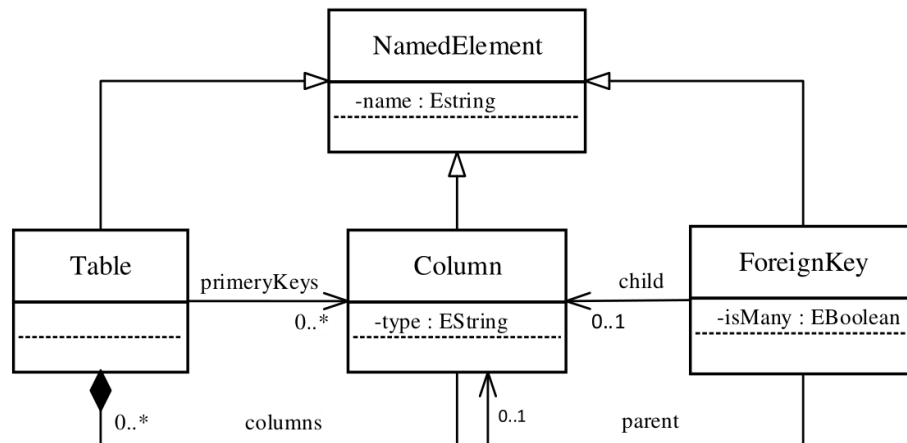


Figure 2. Excerpt of DB meta-model.

The OO meta-model consists of the basic elements of a class model, including *classes*, *attributes*, *references* and *datatypes*. A *class* contains a set of *features*, including *attributes* and *references*, which can be associated with a referenced *classifier*, which is a *class* or *datatype*. Each *feature* can have many values, which means it can be associated with an unbounded number of elements. The DB meta-model describes a minimal database model, which consists of *tables*, *columns*, and *foreign keys*. A *table* contains a set of *columns*, and an arbitrary number of them can be identified as the primary key. Moreover, each *foreign key* can be defined as a multivalued relationship that is used to associate *tables* based on their *columns*.

The simplified ETL transformation to translate a class model to a database model is composed of four rules. Each ETL rule associates elements in the source model to elements in the target model. The *transform* part defines source elements, and the *to* part specifies target elements. For instance, Listing 1 shows the *Reference2ForeignKey* rule, which is used to translate object-oriented references into foreign keys of the database and their related columns.

```

1  rule Reference2ForeignKey
2      transform r : OO!Reference
3      to fk : DB!ForeignKey, fkCol : DB!Column {
4          fkCol.table := r.type;
5          fkCol.name = r.name + "ID";
6          fkCol.type = "Integer";
7          fk.name = r.name;
8          fk.child = fkCol;
9          fk.parent = r.owner.equivalent().primaryKeys.first();
10 }
```

Listing 1. Sample rule: *Reference2ForeignKey*.

An ETL rule can specify the number of *binding assignments* that are used to compute the value of the generated element attributes. In the *Reference2ForeignKey* rule, three bindings are used to assign values to the *table*, *name* and *type* properties of a generated *column*, which is referenced using the variable *fkCol*. Lines 7 to 9 also specify three bindings to set the values of the *name*, *child* and *parent* properties for a generated *foreign key*, which is referenced using the variable *fk*.

The *Class2Table* rule in Listing 2 is responsible for transforming object-oriented *classes* to corresponding database *tables* with associated *columns* as primary keys. This rule generates a *table* and a *column* for each *class*. The attributes of the generated *table* and *column*, which are referenced using the variables *t* and *pk*, are valued in lines 4 to 8 of Listing 2. Also, the generated *table* and *column* should be associated with each other. This rule describes the connection between *t* and *pk* in lines 8 and 9, which show two bindings for performing the association of them to each other.

```

1  rule Class2Table
2      transform c : OO!Class
3      to t : DB!Table, pk : DB!Column {
4          t.name = c.name;
5          pk.name = t.primaryKeyName();
6          pk.type = "Integer";
7          t.columns.add(pk);
8          t.primaryKeys.add(pk);
9  }
```

Listing 2. Sample rule: *Class2Table*.

The *SingleValuedAttribute2Column* rule in Listing 3 translates a single-valued *attribute* into a corresponding database *column*. The input pattern of this rule expresses a *guard* in line 4 to filter the single-valued attributes based on the value of their *isMany* properties. This rule has to generate a *column* for every single-valued attribute, which is referenced using the variable *c*. Lines 5 to 7 specify required bindings to compute the value of the *name*, *table* and *type* properties for the generated *column*, according to the corresponding value of properties for the input attribute and some transformation rules and operations.

```

1  rule SingleValuedAttribute2Column
2      transform a : OO!Attribute
3      to c : DB!Column {
4      guard : not a.isMany
5          c.name = a.name;
6          c.table := a.owner;
7          c.type = a.type.name.toDbType();
8  }
```

Listing 3. Sample rule: *SingleValuedAttribute2Column*.

Finally, the *MultiValuedAttribute2Table* in Listing 4 transforms a multi-valued *attribute* into a corresponding *table* with a database *foreign key* and two associated *columns* as *primary* and *foreign keys*. The multi-valued attributes are specified by the input pattern guard in line 6. For each multi-valued attribute, the rule first creates a new *table* (variable *t*) and computes its name by executing the *valuesTableName()* operation in line 7. Then, the rule creates two new *columns* (variables *pkCol* and *fkCol*) and specifies a set of bindings in lines 9 to 14 that assign values of the *name*, *table* and *type* properties for the generated columns. After that, the rule creates a new *foreign key* (variable *fk*) and associates the generated *foreign key* to the generated *columns*, *pkCol*, and *fkCol*, according to the bindings in lines 16 and 17 for the *parent* and *child* properties, respectively.

```

1  rule MultiValuedAttribute2Table
2      transform a : OO!Attribute
3      to t : DB!Table, pkCol : DB!Column,
4          fkCol : DB!Column, fk : DB!ForeignKey {
5
6      guard : a.isMany
7          t.name = a.valuesTableName();
8
9          pkCol.name = "ID";
10         pkCol.table = t;
11         pkCol.type = "Integer";
12         fkCol.name = a.name + "ID";
13         fkCol.table ::= a.owner;
14         fkCol.type = "Integer";
15
16         fk.parent = pkCol;
17         fk.child = fkCol;
18 }

```

Listing 4. Sample rule: *MultiValuedAttribute2Table*.

In the first execution of the described transformation, all rules should be run to generate the target model elements. To keep the target model synchronized with the source model, we need to propagate changes between models. However, re-executing whole transformation rules to compute all target model elements is time-consuming and inefficient. In this case, the incremental execution of transformation can be useful to achieve the updated target model without wasting time and efficiently.

As an example, consider Figure 3, which shows the source and target models for the *OO2DB* transformation. The initial version of the source model (Figure 3.a) is an instance of the *OO* meta-model, including *Student*, *Course*, *Class* and *Professor* classes, that describes the structure of a Course Management System. Using the *OO2DB* transformation, the initial version of the source model will be translated into the initial version of the target model (Figure 3.c). The initial version of the target model is an instance of a *DB* meta-model, including *Student*, *Course*, *Class* and *Professor* tables, that describes the scheme of a relational database for the mentioned Course Management System.

For instance, suppose that the initial version of the source model is changed to the updated version of the source model (Figure 3.b) by adding the *courseDesc* attribute to *Course*, removing the *classDesc* attribute from *Class*, and adding the *taught-by* reference between *Class* and *Professor*. In this case, the initial version of the target model requires to be updated only according to the source model changes. Since the mentioned changes are related to the update of references and single attributes in the source model, an optimal solution to update the initial version of the target model is only re-execution of the rules *Reference2ForeignKey* and *SingleValuedAttribute2Column* for the updated elements.

Consequently, based on the added *courseDesc* attribute, the *courseDesc* column is added to the target model. Also, due to the addition of the *taught-by* reference, the *taught-by* foreign key and *taught-byID* column are added to the target model. Finally, based on the deletion of the *courseDesc* attribute, the corresponding column is deleted from the initial version of the target model to create the updated version of the target model (Figure 3.d).

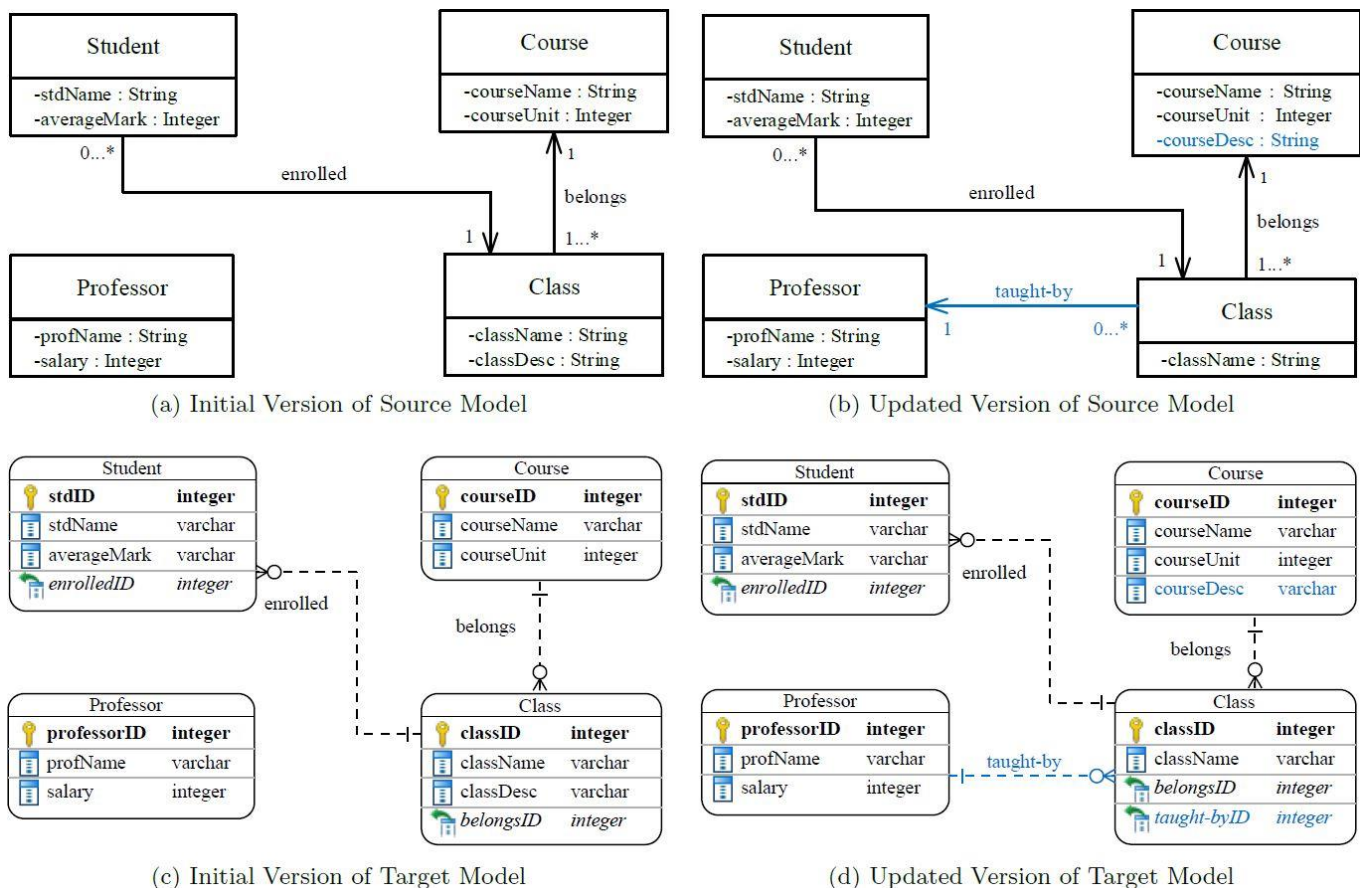


Figure 3. Models of running example.

5 ETL Incremental Execution Process

To address the lack of an approach to incremental execution of ETL transformations, we have introduced a new process to execute only the required ETL transformation rules based on the changes that are applied to the source model. The main objective of this process is to improve ETL transformation performance to propagate changes from source to target model in the evolution of models with a large number of elements. The ETL incremental execution process consists of four complementary phases: *Change Detection*, *Rule Selection*, *Rule Execution* and *Target Extraction*. Figure 4 shows the proposed process that needs the updated, the initial version of the source model, the initial version of the target model and the ETL transformation script. The source and target models are arbitrary meta-models, which are named Source Metamodel and Target Metamodel, respectively. These meta-models are required throughout the process. This process, which is built as an Eclipse plug-in on top of the Epsilon framework, automatically executes the required transformation rules.

In the following, we explain all consecutive phases of the ETL incremental execution process. For each phase, we sketch the *OO2DB* running example to investigate in more detail how that phase works. Furthermore, at the end of this section, we demonstrate the plug-in developed to support the proposed incremental ETL process.

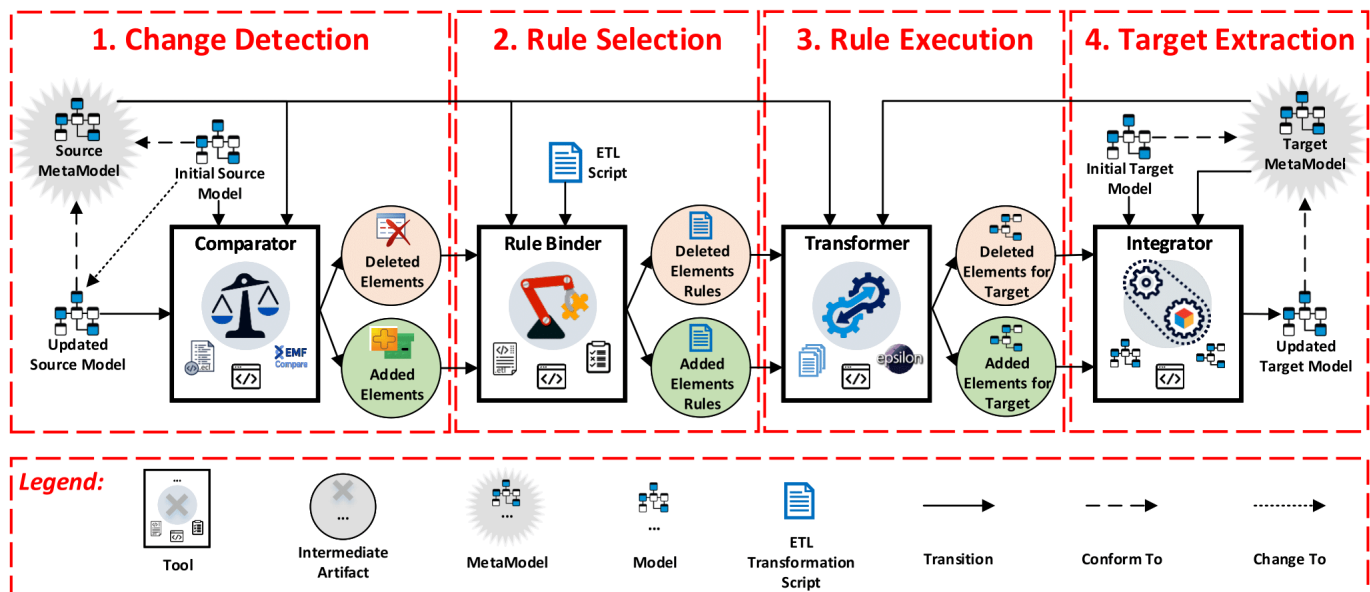


Figure 4. ETL incremental execution process.

5.1 Change Detection

The *Change Detection* phase is started when the user submits the updated version of the source model. In the first step, we need to detect changes applied to the initial source model to re-execute only the required ETL transformation rules. This phase is in charge of detecting elements that are changed (added or deleted) in the initial source model using an *atomic* change operation. We assume that the update of an element is equivalent to deleting an old element and adding a new one.

To detect the changes, a model comparison using Epsilon Comparison Language (ECL) (Kolovos, 2009) or EMF Compare¹ will be performed against elements of both initial and updated versions of the source model according to Algorithm 1. In the comparison, the type of elements, the value of attributes for each element, and the relationships between elements can be regarded as identity metrics. By removing null or unmatched pairs at the comparison output, a match trace consisting of the matched elements of both versions of the source model is obtained. The obtained match trace only contains common elements between initial and updated source models, and the added and deleted elements in the updated source model are missing. Hence, the match trace can be used to find elements that are added to the updated version or deleted from the initial version, respectively, by subtracting common elements from the updated version or initial version.

The model comparison will be performed based on Algorithm 1. This algorithm receives two versions of the same model as inputs (e.g., *Left* and *Right*) and returns a list of matched elements named *MatchTrace* as output. In this algorithm, the *isMatch* function checks the similarity of two elements of the same type, according to the EMF Compare or ECL rules. For every two elements in the *Left* and *Right* models that are identified as matched elements, the algorithm adds a new couple of (*left element*, *right element*) to the *MatchTrace*.

¹ See, <https://www.eclipse.org/emf/compare/>

Algorithm 1. Model comparison algorithm

```

Input initial version of source model as Left, updated version of source
model as Right
Output list of matched elements as MatchTrace
procedure Matching Method
  for each  $e_1 \in Left$  do
    for each  $e_2 \in Right$  and  $e_2 \rightarrow isTypeOf(e_1)$  do
      if isMatch( $e_1, e_2$ ) then
        add couple( $e_1, e_2$ ) to MatchTrace
      end if
    end for
  end for
end procedure

```

To extract the added and deleted elements, we proposed Algorithm 2 and Algorithm 3; respectively. The *MatchTrace* is an input of both algorithms. According to the computed *MatchTrace*, Algorithm 2 searches all elements of the updated source model to detect the missing elements, which determine the *added elements*. Conversely, Algorithm 3 searches all elements of the initial source model to detect the missing elements, which determine the *deleted elements*.

Algorithm 2. Addition detection algorithm

```

Input updated source model as Updated, the match list of initial and
updated source models as MatchTrace
Output list of added elements as AddedElements
procedure Add Detection Method
  for each  $e_1 \in Updated$  do
     $e_2 = findMatch(MatchTrace \rightarrow right, e_1)$ 
    if isEmpty( $e_2$ ) then
      add  $e_1$  to AddedElements
    end if
  end for
end procedure

```

Algorithm 3. Removal detection algorithm

```

Input initial source model as Initial, the match list of initial and
updated source models as MatchTrace
Output list of deleted elements as DeletedElements
procedure Delete Detection Method
  for each  $e_1 \in Initial$  do
     $e_2 = findMatch(MatchTrace \rightarrow left, e_1)$ 
    if isEmpty( $e_2$ ) then
      add  $e_1$  to DeletedElements
    end if
  end for
end procedure

```

Figure 5 shows the result of performing the *Change Detection* phase for the running example (see Section 4). The double-headed arrows show the matched elements, which is the result of Algorithm 1 for initial and updated source models. In this example, the *taught-by* reference and the *courseDesc* attribute are two new elements in the updated source model, which are detected and added to the *Added Elements* list by Algorithm 2. Moreover, *classDesc* is the only removed element, which is detected and added to the *Deleted Elements* by Algorithm 3.

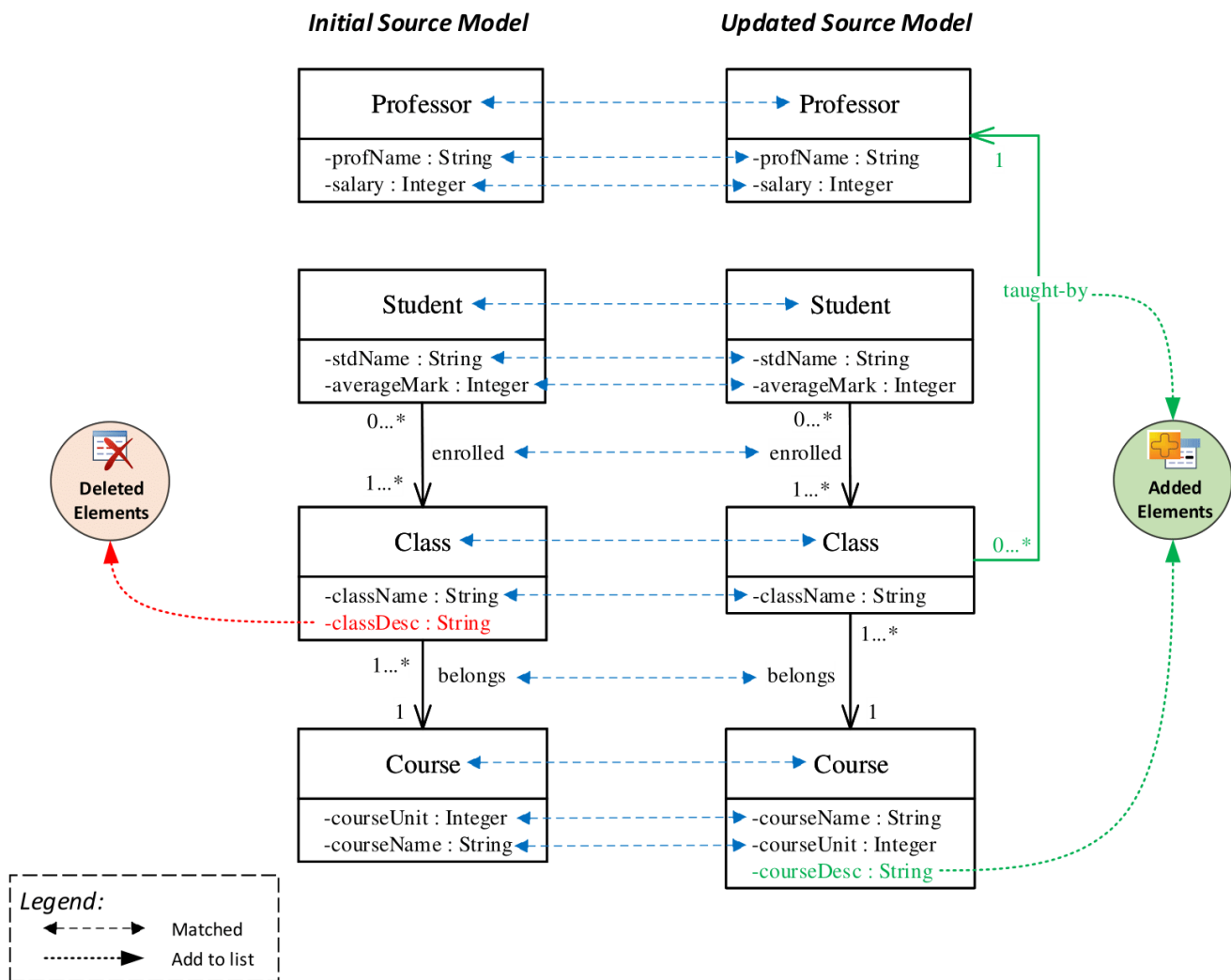


Figure 5. Detection of added elements and deleted elements for running example.

5.2 Rule Selection

In order to incrementally apply changes of the updated source model to the initial target model, we have to detect the changes in the source model and determine the changes that should be applied to the target model along with the corresponding transformation rule. In the *Rule Selection* phase, each added and deleted element detected in the *Change Detection* phase is mapped to a specific rule of ETL transformation. ETL rules can be mapped to the changes based on the type and properties of their source elements, which are respectively defined in the *transform* and *guard* parts of each transformation rule. To this end, Algorithm 4 receives all transformation rules and the added and the deleted elements as changes. Then, it maps each change to the relevant transformation rule by analysing the following conditions. First, the compliance of the *transform* part with the type of element which is changed, and after that, the evaluation result of the *guard* part for the changed element. Therefore, a transformation rule is mapped to a changed element if both conditions are satisfied by that element. Finally, Algorithm 4 returns a map list of changed elements and their relevant rules as a result.

By applying Algorithm 4 to the detected changes of the running example, the *Reference2ForeignKey* rule is mapped to the added *taught-by* reference. Also, as the *isMany* property is false for the added *courseDesc* and the deleted *classDesc* attributes, the *SingleValuedAttribute2Column* rule is selected for both of them.

Algorithm 4. Rule selection algorithm

```

Input all of the Added Elements and Deleted Elements as ChangedElements,
ETL transformation rules as Rules
Output list of relevant rules to changes as Result
procedure Selection Method
  for each  $e_0 \in \text{ChangedElements}$  do
     $mRules = \text{findMatchRules}(\text{Rules} \rightarrow \text{transform}, e_0)$ 
    for each  $rule \in mRules$  do
      if  $e_0 \rightarrow \text{isSatisfied}(rule \rightarrow \text{guard})$  then
        add  $\text{map}(rule, e_0)$  to Result
      end if
    end for
  end for
end procedure

```

5.3 Rule Execution

In the incremental execution of a transformation, only rules which are mapped to changed elements should be executed. Hence, the *Rule Execution* phase executes only ETL transformation rules based on the mapping list created in the *Rule Selection* phase. To this end, we create two intermediate target models, one for the *Added Elements* and the other for the *Deleted Elements*. After that, we automatically create the *context module* and prepare the required information based on the mapping list and initial version of the target model for each selected rule. Then, we execute the transformation rule for the changed element, and finally, we save the result to the *Added* or *Deleted* intermediate target model according to the type of changed (*Added* or *Deleted*) element.

The output of the *Rule Execution* phase for the running example is the execution of three transformation rules and the creation of four elements. The *SingleValuedAttribute2Column* rule is executed for the *courseDesc* and *classDesc* attributes that create the *courseDesc* and *classDesc* columns. The *courseDesc* column is created for an added element, so it is saved in the *Added* intermediate target model. Instead, the *classDesc* column is the result of the removed attribute and saved in the *Deleted* intermediate target model. As well, the *Reference2ForeignKey* rule is executed for the *taught-by* reference, which creates the *taught-by* foreign key and the *taught-byID* column as results. Both created elements are saved in the *Added* intermediate target model since the *taught-by* reference was an *Added* element.

5.4 Target Extraction

After the execution of relevant transformation rules, the target model should be extracted based on the created results, which are saved in the intermediate target models. To this end, we first subtract the *Deleted* intermediate target model from the initial target model. In addition, for some deleted elements, dependencies such as relationships and references have to be deleted in the updated target model. After that, we merge the result with the *Added* intermediate target model to create the updated version of the target model. In order to implement the *Target Extraction* phase, we use Algorithm 5. This algorithm receives both the *Added* and *Deleted* intermediate target models and the initial version of the target model as input. Then, the initial target model is manipulated according to the *Added* and *Deleted* target elements with change model operations. Finally, the updated version of the initial target model is stored as output.

Figure 3.d shows the updated version of the target model, which is the output of the *Target Extraction* phase for the running example. As shown in this figure, instead of creating 22 elements using batch transformation, which needs the execution of 15 transformation rules, we create only four elements by incremental execution of only three transformation rules according to the new changes. The ETL incremental execution process also reduced the execution time by about 73% for the running example.

Algorithm 5. Target extraction algorithm

```

Input added intermediate target model as TargetAdded, deleted intermediate
target model as TargetDeleted, and initial target model as InitialTarget
Output updated target model as UpdatedTarget
procedure Integration Method
  copy InitialTarget to UpdatedTarget
  for each  $e_0 \in \text{TargetDeleted}$  do
     $\text{depList} = \text{findDependency}(\text{InitialTarget}, e_0)$ 
    for each  $e_1 \in \text{depList}$  do
      delete  $e_1$  from UpdatedTarget
    end for
    delete  $e_0$  from UpdatedTarget
  end for
  for each  $e_0 \in \text{TargetAdded}$  do
     $e_1 = \text{findOwner}(\text{UpdatedTarget}, e_0)$ 
    if  $\neg \text{isEmpty}(e_1)$  then
      add  $e_0$  to below of  $e_1$  in UpdatedTarget
    else
      add  $e_0$  to UpdatedTarget
    end if
  end for
end procedure

```

5.5 Toolkit prototype

To show the applicability of the ETL incremental execution process, an Eclipse toolkit plug-in is implemented on the Epsilon framework. The source code of our plug-in is available from GitHub² under the EPL 2.0 license. The implemented plug-in, as shown in Figure 6, can be started from the Eclipse menu by clicking on "Run Transformation" and opening the "Model Selection" window (Figure 7). The Model Selection window allows the user to select the initial and updated versions of the source model and the initial version of the target model. By going to the next window (Figure 8), the user should select the source and target meta-models, ETL transformation file and comparison strategy, and go to the next window (Figure 9) to see the address of the updated target model and the transformation execution time.

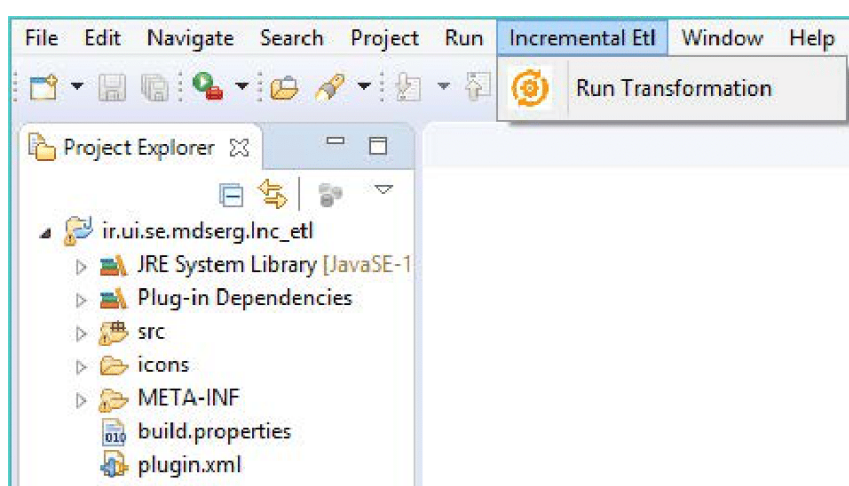


Figure 6. Implemented toolkit: start menu.

² See, <https://github.com/Marziah-Ghorbani/IncrementalETL>

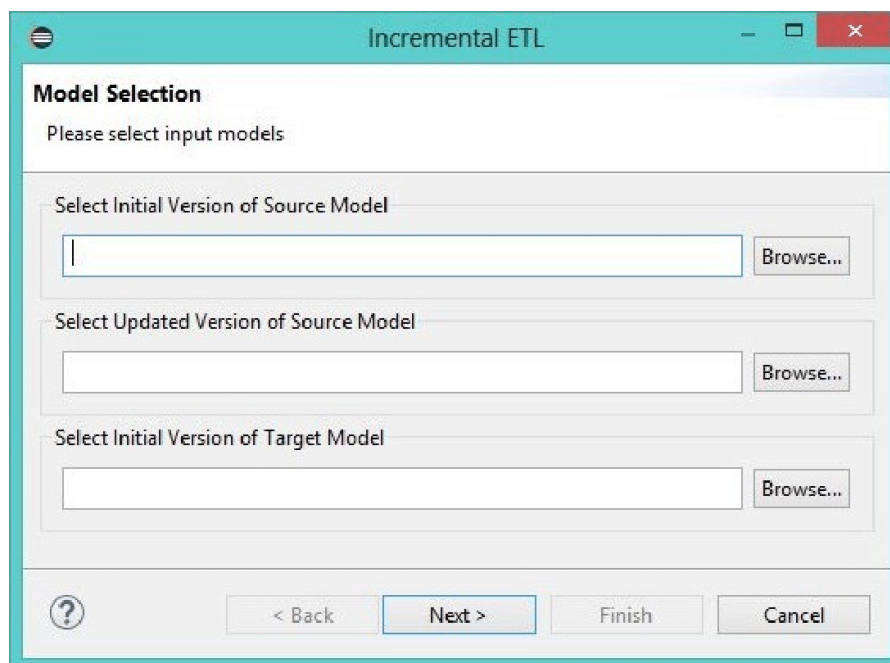


Figure 7. Implemented toolkit: model selection.

All GUIs and the proposed process are coded using the Java language; however, algorithms more specifically rely on the Epsilon Object Languages (EOL). Our prototype is able to incrementally transform UML and EMF-based models. Moreover, the proposed process can work well for any standard ETL transformation, which strongly depends on the Epsilon transformation interpreter.

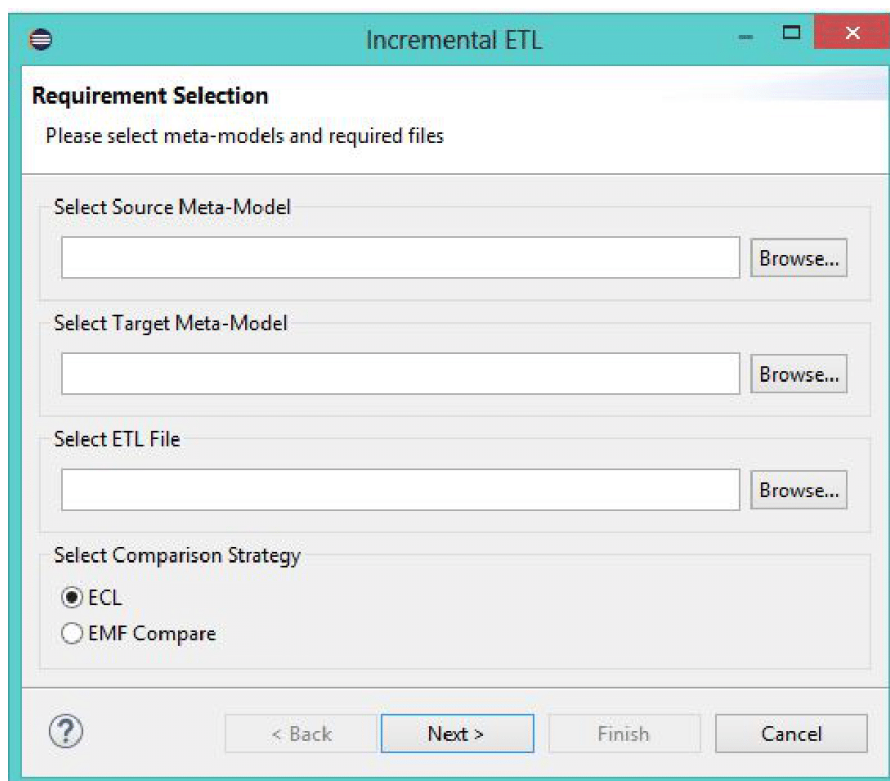


Figure 8. Implemented toolkit: requirement selection.

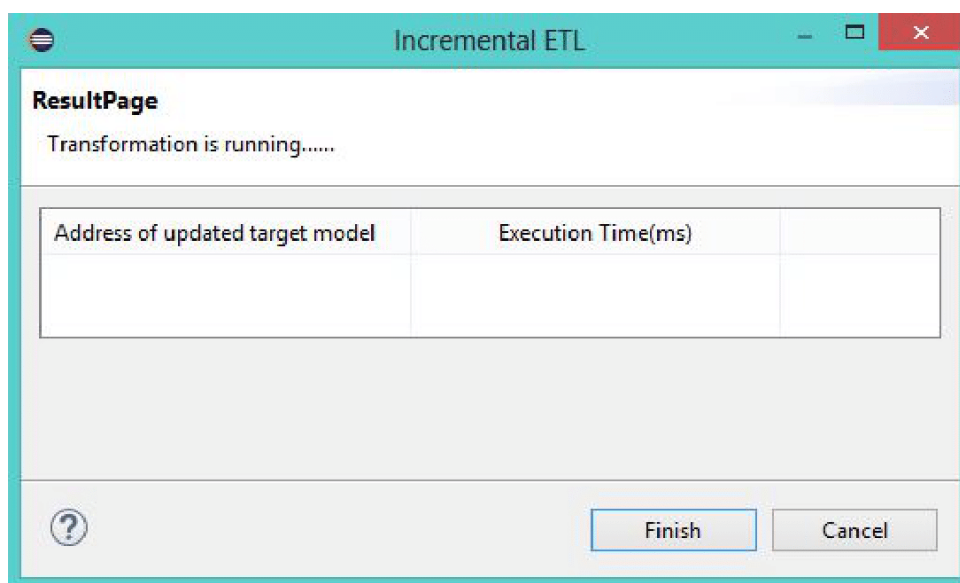


Figure 9. Implemented toolkit: requirement selection.

6 Evaluation

The proposed incremental model transformation introduced in Section 5 aims to execute ETL transformation more efficiently than standard ETL execution for updating source models with some new changes. Therefore, we conducted a descriptive case study following the guidelines defined by Runeson and Höst (2008) to assess the *correctness*, *execution time* and *scalability* of our approach. To this end, we performed both an ETL incremental execution process and a standard batch (non-incremental) ETL strategy for *Families2Persons* and *OO2DB*, which are two well-known examples in the model transformation area. In the following, we first introduce the research questions that we aimed to answer with the case study. Then, we focus on the design of the case study. Finally, we present and discuss the results of the case study concerning the mentioned research questions.

6.1 Research questions

The first goal of this study is to evaluate the correctness of our incremental execution process and the implemented tool when used for a real-world scenario. Another goal is to quantitatively assess the execution time of our approach compared to the standard batch execution of ETL. It also aims to explore the scalability of our approach when expanding the size of models and increasing the number of changes. Therefore, we conducted an experiment in order to investigate the following research questions.

RQ1. Correctness: Does the proposed incremental execution process as well as the implemented toolkit produce a valid model?

RQ2. Execution time: Does the proposed process reduce the time of transformation re-execution for updated source models?

RQ3. Scalability: What are the effects of expanding the size of the source model and increasing the number of changes on the incremental transformation execution time?

6.2 Case study design

As an appropriate input to this case study, we needed EMF models for our transformations. These models must have an extensive evolution history. We further required information on the change operations that have been applied to each source model. According to the source models, the correctness of the incremental execution process and resulting target models can be ensured manually by spot-checking. To this end, we first performed *Families2Persons* and *OO2DB* transformations step by step for two real-world models using the incremental execution process. After each step, we immediately checked the accuracy of the intermediate outputs. Subsequently, we executed the standard ETL transformation. Finally, we compared target models for both incremental and standard strategies using EMF Compare. The same results for both strategies show that the incremental process operates correctly.

Moreover, to investigate execution time and scalability, we required large-scale source models with a specific number of change operations that had been applied to source models. To achieve this, we used Ecore Mutator³ to randomly mutate EMF models conforming to the source meta-models (i.e., the Family and OO meta-models). Consequently, we produced several models of small to large size, including source models with 100 to 45,000 elements. We also created five test models with 8, 16, 32, 64, and 128 changes for each source model. All the created test models were transformed to the target models using both incremental and standard ETL execution strategies. During the experiments, the time for doing each transformation was measured independently. Therefore, by comparing the data obtained from our experiments, the research questions related to the execution time and scalability could be answered appropriately. The results are discussed in the next section.

6.3 Results

In this section, we present the results of our case study. In particular, we indicate the collected data, then we proceed with analysing them, and finally, we discuss and draw some conclusions for each research question raised in Section 6.1. Note that all the experiments were carried out on a standard Windows 8 laptop using an Intel®Core™ i7 with a 3.6 GHz processor and 8GB RAM. Moreover, all the reported times were computed as the average for ten independent executions.

RQ1. Correctness: With the first research question, we aimed to investigate the correctness of the incremental ETL process as well as an implemented toolkit. To this end, we rely on the result of our experiment based on the experience gained from the *Families2Persons* and *OO2DB* transformations for real-world modelling scenarios.

In the first scenario, we focused on *Families2Persons* transformation, which includes two independent rules. As an initial version of the source model, we used an extended version of the “Families to Persons”⁴ source model, including six families. We applied all the different change cases introduced in the “TTC 2017 Families to Persons” case study (Anjorin et al., 2017) to the forward direction. After each change, we ran the incremental ETL process to check the target model and ensure the correctness of the transformation. After applying all changes to the initial source model, we also executed standard ETL to produce the target model. Finally, we used EMF Compare to check the similarity of the target model produced by standard ETL, and the target model was updated with an incremental ETL process. This test scenario indicated that the incremental ETL process produced the expected output for each change, and the final target model was valid and quite similar to the target model produced by standard ETL. For our purposes, a valid target is a model that only consists of the respective elements without any missing elements and conforms to the target meta-model.

³ See, <https://code.google.com/archive/a/eclipse-labs.org/p/ecore-mutator>

⁴ See, <https://www.eclipse.org/at/atTransformations/>

In the second scenario, we followed a procedure similar to the first scenario for *OO2DB* ETL transformation. The *OO2DB* transformation has a different structure with more complex and interdependent rules that help us indicate the correctness of our approach for any ETL transformations. In this scenario, we used the original model, and changes were applied in the evolution of the “Order Management System” case study (Sharbaf & Zamani, 2020) to prepare the initial and updated versions of the source model for *OO2DB* transformation. The initial source model had more than 50 elements, including classes, attributes, references and datatypes. We modified the initial source model using 11 changes consisting of five additions, three updates and three deletions. Then we transformed the update source model using standard ETL and incremental ETL processes. The manual spot-checking shows that the incremental ETL process has correctly found eight additions and six deletions since this process considers each update as a sequential deletion and addition. Moreover, EMF Compare confirms that the resulting target model for both incremental and standard ETL are the same. However, the incremental ETL process re-executed transformation rules for only 14 elements, while standard ETL executed transformation rules for 51 elements.

Regarding the results of these two scenarios, we can conclude that, in general, our approach is able to support incremental execution for different ETL transformations correctly. However, more advanced transformation should also be investigated, and we will continue to conduct more experiments and evaluations.

RQ2. Execution time: The second research question concerns the overall time of executing the incremental ETL process compared to standard ETL transformation execution time. To this end, we conducted an experiment including two parts to investigate execution time for incremental and standard ETL in two different ETL transformations.

In the first part of our experiment, we considered various versions of the source model for *Families2Persons* transformation, including models with 100 to 45,000 elements. We duplicated each version and applied eight change operations on one version and 128 change operations on another version. We ran the incremental ETL process and standard ETL transformation for all updated versions of the source model to measure the overall execution time for every transformed model. The results of our experiment for increasing the source model size for 8 or 128 change operations are illustrated in Figure 10. The results show that the execution times for the incremental ETL process are approximately less than or equal to the runtimes of standard ETL. Moreover, the execution time for incremental ETL has a linear growth compared with the execution time for standard ETL, which has a polynomial (quadratic) complexity.

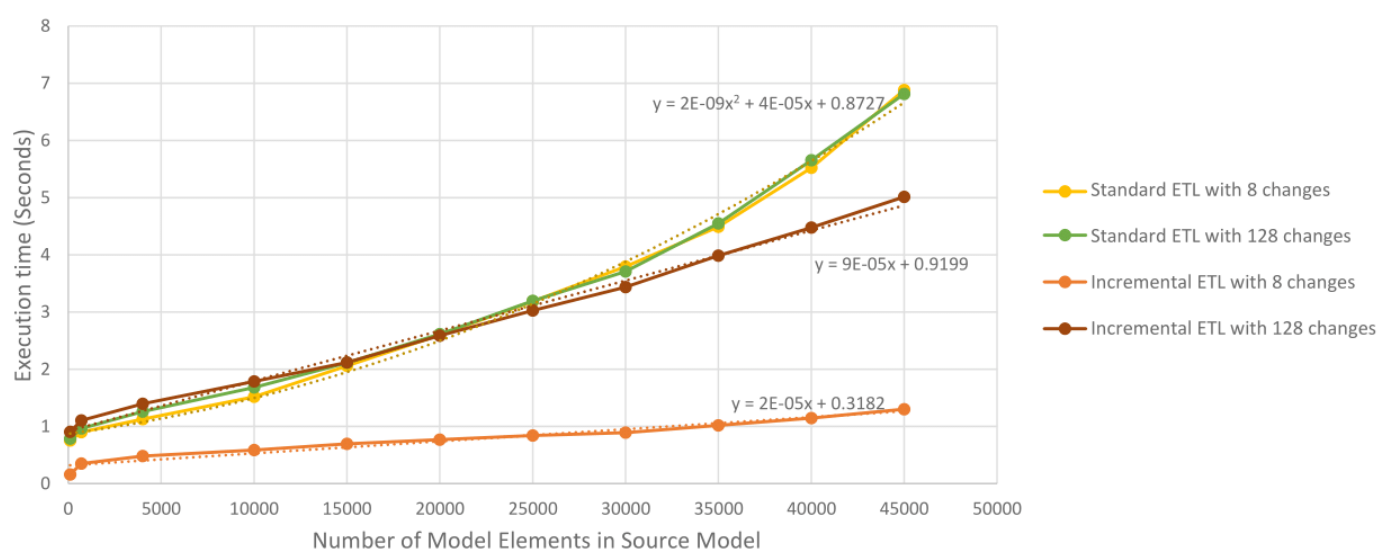


Figure 10. Comparison of execution time for incremental vs. standard ETL in *Families2Persons* transformation.

In the second part of our experiment, we followed the same path for *OO2DB* transformation, including more complex rules. The results of our experiment for increasing the source model size for 8 or 128 change operations are depicted in Figure 11. The results again show very clearly that the time taken for the incremental ETL process does not increase more than the execution time for standard ETL as the size of the source model increases. Moreover, the time complexity for the incremental ETL process is linear again, while the differences between the runtime of standard ETL and incremental ETL for large models are growing more quickly. However, the results show that the incremental ETL process for source models with less than 25,000 elements has a high overhead when the size of change operations increases.

Overall, the results of this experiment indicate that for source models that have been updated with a small number of change operations, the incremental ETL process is a better choice than the standard ETL. However, for small source models which are updated with a large number of change operations, the standard ETL works better. But when we aim to propagate changes over large-scale models or execute complex transformations with many interdependent rules, it is more reasonable to use the incremental ETL process and re-execute the transformations.

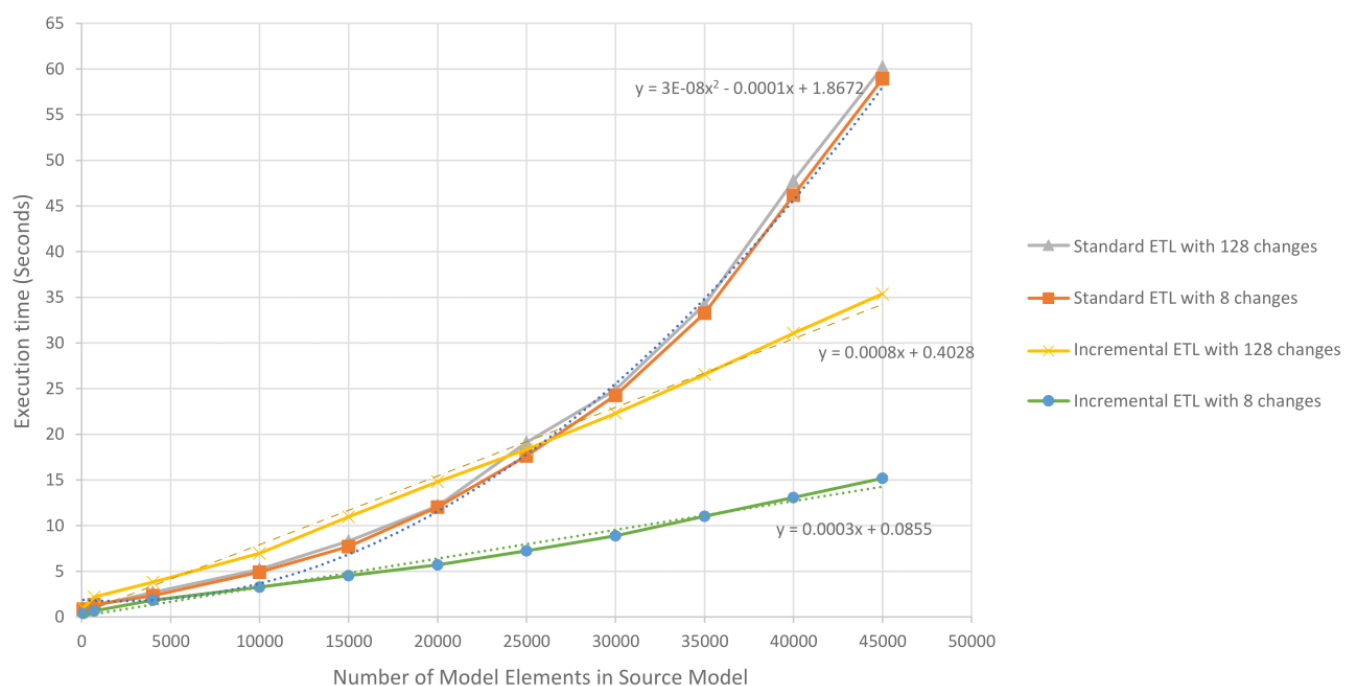


Figure 11. Comparison of execution time for incremental vs. standard ETL in *OO2DB* transformation.

RQ3. Scalability: The last research question aims to investigate the scalability of our incremental ETL process when expanding the size of models and increasing the number of change operations. To assess the scalability of our approach, we measured the required overall time to re-execute *Families2Persons* and *OO2DB* ETL transformations for several source models and different changes. To achieve this, we used models from small to large sizes for both mentioned ETL transformations, including 100; 700; 4000; 10,000; 15,000 20,000; 25,000; 30,000; 35,000; 40,000 and 45,000 elements in the source model. We also updated each source model with 8, 16, 32, 64 and 128 change operations. This configuration led to five test sets of source models for *Families2Persons* transformation and five other test sets for *OO2DB* transformation.

The results of our experiment for increasing the source model size in *Families2Persons* test sets are shown in Figure 12. Also, Figure 13 represents the results for increasing the source model size for *OO2DB* transformation. These figures indicate that, when stepwise increasing the model size in the different numbers of change operations, the execution time for the incremental ETL process in both transformations has a linear complexity.

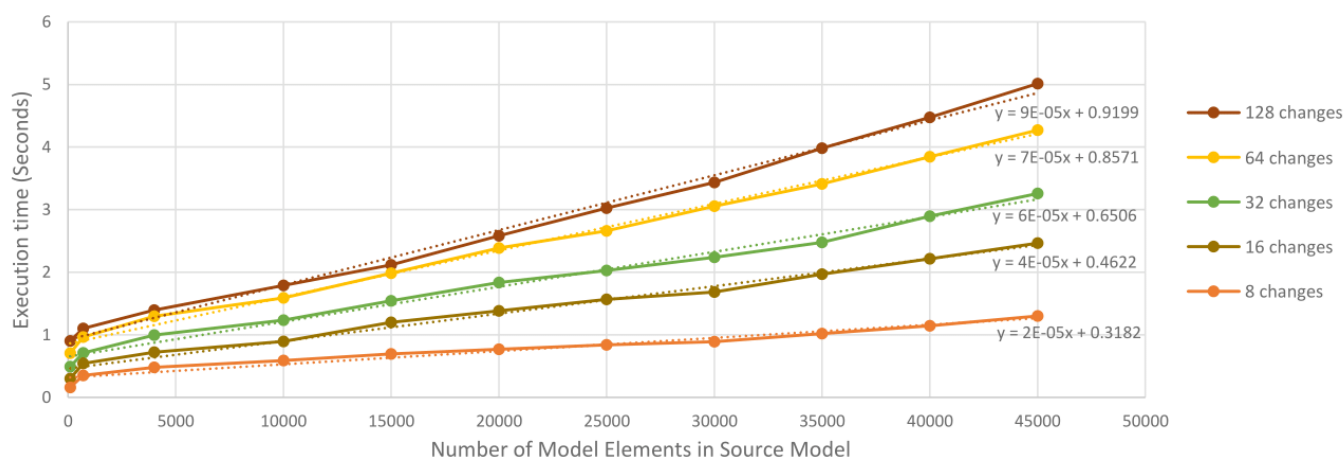


Figure 12. Execution time of incremental ETL for different number of source model elements in *Families2Persons*.

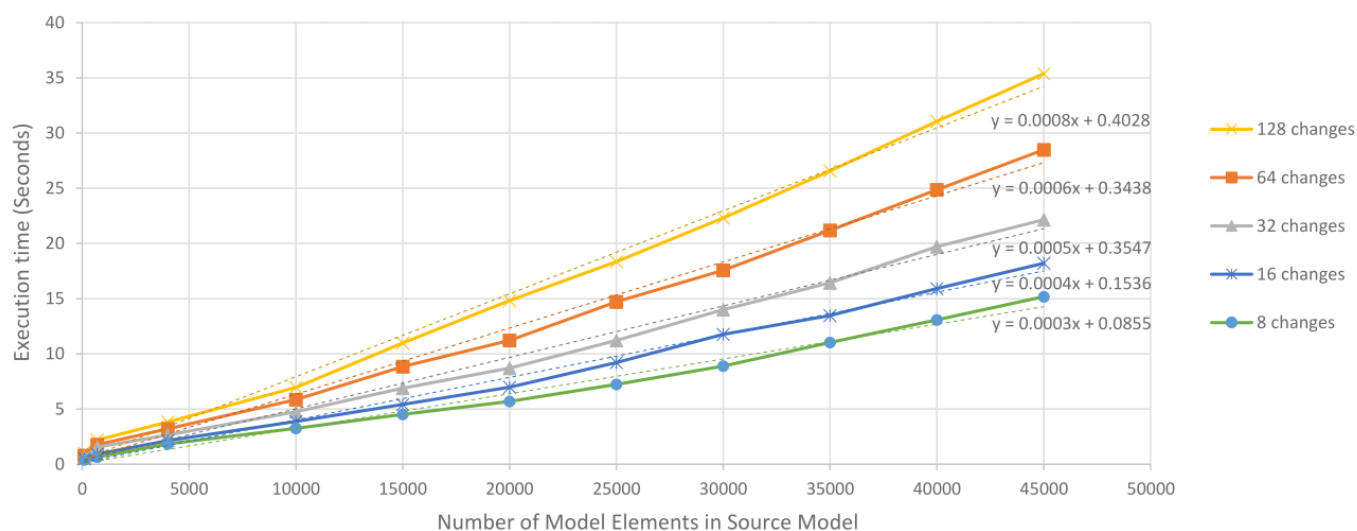


Figure 13. Execution time of incremental ETL for different number of source model elements in *OO2DB*.

Moreover, the results of our experiment for increasing the number of change operations applied to source models for *Families2Persons* transformation are illustrated in Figure 14. Also, Figure 15 depicts the overall execution times that are required for the re-execution of *OO2DB* transformation for all the test sets. Regarding the results, the measured execution times for *OO2DB* transformation are longer than the execution times for *Families2Persons* transformation since the *OO2DB* has some interdependent rules that call each other. However, as the number of change operations increases, the execution time follows a linear growth rate for both transformations.

Consequently, we conclude that the incremental ETL process is scalable for increasing the size of source models. Moreover, the smooth growth of the execution time with increasing the number of change operations indicates that the proposed process is scalable when increasing the number of change operations. However, the change detection in the incremental ETL process depends on ECL or EMF Compare, the limitations of which can affect the scalability of incremental ETL.

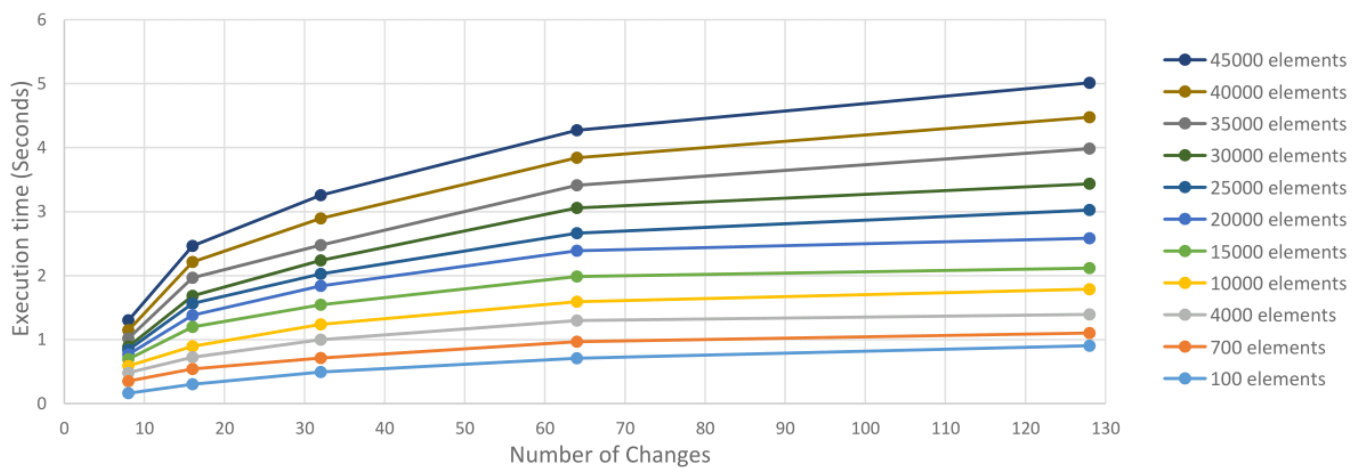


Figure 14. Execution time of incremental ETL for different number of changes applied to the source in Families2Persons.

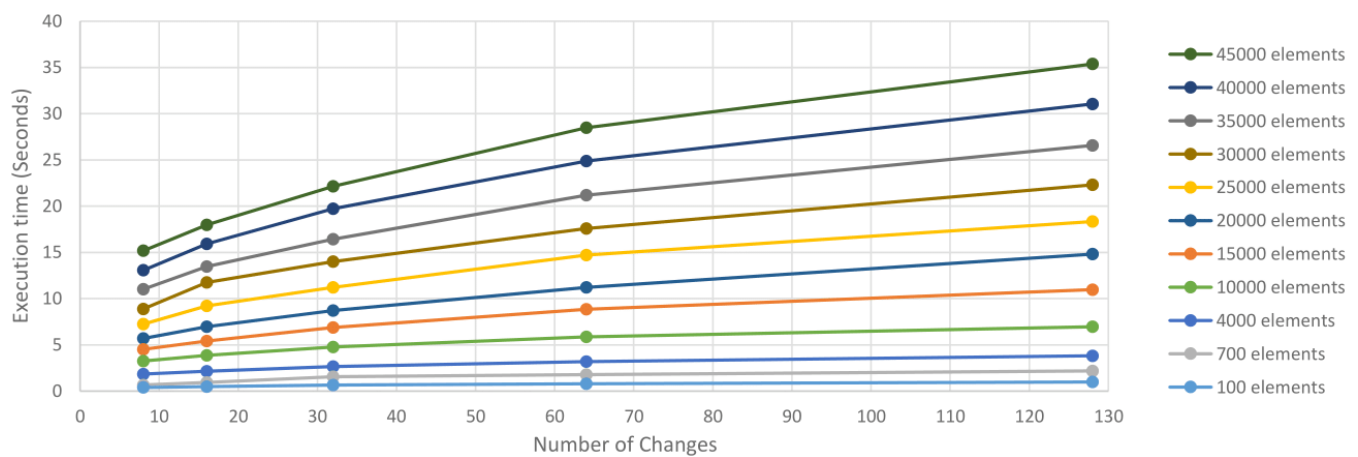


Figure 15. Execution time of incremental ETL for different number of changes applied to the source in OO2DB.

7 Related Work

In recent years, various approaches have been proposed to improve the performance of model transformation execution. However, we could not find any research on incremental execution of ETL transformations. Therefore, we will restrict ourselves to efforts that introduce approaches to propagating changes from the source to the target for model transformation. In the following, we first investigate works proposing approaches to execute model transformation incrementally. Then, we introduce some other existing approaches to propagating changes based on incremental strategies, which include methods such as incremental model-to-text transformations, reactive model transformations, and incremental bidirectional transformations.

7.1 Incremental model transformation

This section reviews the most relevant approaches to our work which have implemented an incremental transformation engine or proposed a technique to incrementally re-execute different model transformations such as ATL, QVT and VIATRA.

In the context of incremental model transformation, the first approach was proposed by Hearnden et al. (2006). They present a live propagation mechanism for changes to a source model of a declarative rule-

based transformation. To this end, they proposed to extend a transformation engine based on the interpretation of logic languages. They used a tree structure to represent the trace of a transformation execution. The created tree is also used to detect rule dependences to re-transform source model changes to a mapped element in the target models. Unlike our approach, this work is not applicable to large-scale models and only supports live change propagation for simple model transformations without interdependent rules.

Jouault and Tisi (2010) presented a solution for supporting live and incremental model transformations in the ATL engine. To achieve this, they proposed an incremental execution algorithm for ATL, which relies on dependency tracking of OCL expressions and individual rule execution control mechanisms. Tracking of dependencies helps determine which OCL expressions are impacted when a change takes place in the source model. Finally, the collected information is used only to control rule execution based on the parts of the model that are impacted by incremental changes. Although this work is very similar to our approach, the proposed execution algorithm is based on the ATL transformation engine, which has a structure different from the ETL engine.

Jouault and Beaudoux (2016) proposed the use of an active operation framework to implement OCL-based incremental model transformations. The active operation provides fine-grained change propagation and the preservation of collection ordering that supports transformation incrementality. The proposed approach has been implemented as an incremental framework and optimized to support large-scale models. While our approach focuses on ETL model transformation, their approach is applicable to OCL-based model transformation approaches such as QVT and ATL.

Le Calvar et al. (2019) presented efficient ATL incremental transformations by compiling an ATL transformation script to Java code. They proposed the ATOL compiler using the active operations library. The ATOL compiler can handle a significant subset of ATL. This compiler also supports incremental refining in-place ATL transformations, which are used to compute derived properties. The authors also proposed another approach (Le Calvar et al., 2020) to augment incremental transformation rules with constraints. This approach provides a mechanism to explore a set of constraints corresponding to a specific view model when executing transformation incrementally. However, both the proposed approaches only focused on ATL transformation and did not support ETL transformation.

Ráth et al. (2008) proposed an approach to support live model transformation using an incremental pattern matching engine (Bergmann et al., 2008). The RETE-based pattern matcher detects source model changes using a graph pattern. The RETE network tracks changes and executes the action sequences as triggers to update the target model. Although this approach has been proposed for both declarative and imperative transformations, it is only implemented for the VIATRA 2 model transformation framework and does not support ETL transformation.

Boronat (2021) presented incremental execution of rule-based model transformations based on a dependency injection and standardized model changes. In this work, a model change propagation mechanism for execution of change-driven model transformations was implemented in YAMTL. The change propagation mechanisms are designed to support consistency between large models. Each source model change is directly performed by injecting the dependency, which transforms changes to corresponding changes in the target model. However, unlike our approach, the proposed mechanism depends on model changes recorded using EMF change and cannot support change propagation for models that are updated in different model editors.

7.2 Incremental change propagation

In this section, we provide a brief overview of works that have focused on using incremental strategies for model synchronization and other transformation mechanisms (i.e., bidirectional model transformation or reactive model transformation), as well as model-to-text transformations.

Giese and Wagner (2006) presented an incremental model synchronization based on the Triple Graph Grammars (TGG). They proposed a declarative specification formalism that was employed to achieve an incremental execution of the transformation rules by exploiting the known dependencies between the transformation rules. The proposed approach was implemented as a plug-in in the Fujaba Tool Suite. Moreover, the authors extended their work and proposed a new approach (Giese & Wagner, 2009) to support incremental bidirectional model synchronization using update propagation attributes and corresponding dependencies. However, both approaches are proposed for graph transformation, which has a different structure in comparison to the ETL transformation.

Samimi et al. (2018) proposed a new approach to bidirectional transformation based on Epsilon Validation Language, which can synchronize source and target models at the same time. This approach works based on a specific tracing meta-model that allows change propagation according to the stored references to corresponding elements. Weidmann et al. (2019) proposed another incremental bidirectional model transformation based on TGG. They combine an incremental graph pattern matcher and an integer linear programming (ILP) solver to determine a set of solution rules and choose the optimal one from the candidate solutions. In these approaches, models are not created from scratch, and both employ incremental strategies to support bidirectional transformation.

Ogunyomi et al. (2014) presented the use of a signature for incremental model-to-text transformation. The signature is a lightweight mechanism to determine a subset of model-to-text transformation rules that must be re-executed for each change in the source model. Ogunyomi et al. (2019) proposed another incremental model-to-text transformation approach using property access traces. The approach used runtime analysis to identify changes in source models and provide a property access trace for determining which subset of the transformation must be re-executed. Although both proposed approaches are implemented upon the Epsilon platform, they are not applicable to the ETL engine that supports model-to-model transformations.

Bergmann et al. (2015) proposed a reactive model platform built on top of incremental graph queries for EMF models, where transformations are executed continuously as reactions to changes in the underlying model. In this context, Varró et al. (2016) introduced three generations of the VIATRA framework to support reactive and incremental model transformation using incremental graph pattern matching techniques over an imperative programming language such as Java. Martínez et al. (2017) presented a reactive model transformation engine for ATL, which defines a persistent data flow among models to take care of activating only the strictly needed computation in response to updating of source models. Horváth et al. (2020) recently proposed multi-tenant and parallel reactive model transformation as new research lines to generate reactive model transformation on a low-code platform. Similar to our approach, these approaches also use continuous re-execution of transformation rules to propagate source model changes. However, they aim to provide reactive model transformation.

8 Conclusion and Future Work

In this paper, we proposed the incremental ETL process, an approach to re-executing an ETL transformation to propagate changes from the source models to the target models. We introduced algorithms to identify the source model changes, select and only execute the relevant transformation rules, and update the target model based on the corresponding target elements. We described the incremental ETL process step by step using a running example. The proposed process was implemented as a toolkit

on top of the Epsilon platform. We assessed the correctness of our approach for real-world modelling scenarios and different ETL transformations with a descriptive case study. We also evaluated the performance of incremental ETL using an experiment assessing the speed of our process in comparison to standard ETL. Moreover, the results of our experiment are promising regarding the scalability of incremental ETL for increasing the size of source models and the number of applied changes.

As future work, we plan to extend the change detection phase to support composite change operations such as refactoring. We will carry out more detailed experiments to investigate further transformation cases to evaluate the generalizability of the incremental ETL process and the implemented toolkit. We also intend to integrate the incremental execution process with the Epsilon platform to provide a new engine for incremental transformation. Furthermore, complementary work is required to provide support to reactive model transformation on top of the Epsilon platform.

Additional Information and Declarations

Conflict of Interests: The authors declare no conflict of interest.

Author Contributions: M. G.: Methodology, Software, Investigation, Visualization, Writing - Original Draft; M. S.: Methodology, Software, Investigation, Visualization, Writing - Original Draft, Writing - Review & Editing, Paper Bureaucracy Organizer; B. Z.: Conceptualization, Methodology, Writing - Review & Editing.


Data Availability: The data that support the findings of this study are available from the corresponding author.

References

- Anjorin, A., Buchmann, T. & Westfechtel, B. (2017). The Families to Persons Case. In A. Garcia-Dominguez, F. Krikava and G. Hinkel (eds.), *Proceedings of the 10th Transformation Tool Contest* (pp. 27–34). CEUR-WS. <http://ceur-ws.org/Vol-2026/paper2.pdf>
- Bergmann, G., Dávid, I., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z. & Varró, D. (2015). VIATRA 3: A reactive model transformation platform. In D. Kolovos & M. Wimmer (Eds.), *Theory and Practice of Model Transformations (ICMT)*. (pp. 101–110). Springer. https://doi.org/10.1007/978-3-319-21155-8_8
- Bergmann, G., Ökrös, A., Ráth, I., Varró, D. & Varró, G. (2008). Incremental pattern matching in the VIATRA model transformation system. In *Proceedings of the Third International Workshop on Graph and Model Transformations*, (pp. 25–32). ACM. <https://doi.org/10.1145/1402947.1402953>
- Boronat, A. (2021). Incremental execution of rule-based model transformation. *International Journal on Software Tools for Technology Transfer*, 23, 289–311. <https://doi.org/10.1007/s10009-020-00583-y>
- Brambilla, M., Cabot, J. & Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*. Second Edition. Morgan Claypool Publishers.
- Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A. & Varró, D. (2002). VIATRA – visual automated transformations for formal verification and validation of UML models. In *Proceedings of 17th IEEE International Conference on Automated Software Engineering*, (pp. 267–270). IEEE. <https://doi.org/10.1109/ASE.2002.1115027>
- Czarnecki, K. & Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM System Journal*, 45(3), 621–645. <https://doi.org/10.1147/sj.453.0621>
- Giese, H. & Wagner, R. (2009). From model transformation to incremental bidirectional model synchronization. *Journal of Systems and Software*, 8(1), 21–43. <https://doi.org/10.1007/s10270-008-0089-9>
- Giese, H. & Wagner, R. (2006). Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel & G. Reggio (Eds.), *Model Driven Engineering Languages and Systems (MODELS)*, (pp. 543–557). Springer. https://doi.org/10.1007/11880240_38
- Hearnden, D., Lawley, M. & Raymond, K. (2006). Incremental Model Transformation for the Evolution of Model-Driven Systems. In O. Nierstrasz, J. Whittle, D. Harel & G. Reggio (Eds.), *Model Driven Engineering Languages and Systems (MODELS)*, (pp. 321–335). Springer. https://doi.org/10.1007/11880240_23
- Horváth, B., Horváth, Á. & Wimmer, M. (2020). Towards the next Generation of Reactive Model Transformations on Low-Code Platforms: Three Research Lines. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings* (Article No.: 65). ACM. <https://doi.org/10.1145/3417990.3420199>

- Johann., S. & Egyed, A. (2004). Instant and incremental transformation of models. In *Proceedings of 19th International Conference on Automated Software Engineering*, (pp. 362–365). IEEE. <https://doi.org/10.1109/ASE.2004.1342765>
- Jouault, F. & Beaudoux, O. (2016). Efficient OCL-based Incremental Transformations. In *16th International Workshop in OCL and Textual Modeling @ MODELS*, (pp. 121–136). https://oclworkshop.github.io/2016/papers/OCL16_paper_14.pdf
- Jouault, F. & Kurtev, I. (2006). Transforming Models with ATL. In J.-M. Bruel (Ed.), *Satellite Events at the MODELS 2005 Conference*, (pp. 128–138). Springer. https://doi.org/10.1007/978-3-540-69927-9_4
- Jouault, F. & Tisi, M. (2010). Towards Incremental Execution of ATL Transformations. In L. Tratt & M. Gogolla (Eds.), *Theory and Practice of Model Transformations (ICMT)*, (pp. 123–137). Springer. https://doi.org/10.1007/978-3-642-13688-7_9
- Kolahdouz-Rahimi, S., Lano, K., Sharbaf, M., Karimi, M., & Alfraihi, H. (2020). A comparison of quality flaws and technical debt in model transformation specifications. *Journal of Systems and Software*, 169, 110684. <https://doi.org/10.1016/j.jss.2020.110684>
- Kolovos, D. S. (2009). Establishing Correspondences between Models with the Epsilon Comparison Language. In R. F. Paige, A. Hartman & A. Rensink (Eds.), *Model Driven Architecture - Foundations and Applications*, (pp. 146–157). Springer. https://doi.org/10.1007/978-3-642-02674-4_11
- Kolovos, D. S., Paige, R. F. & Polack, F. A. C. (2008). The Epsilon Transformation Language. In A. Vallecillo, J. Gray & A. Pierantonio (Eds.), *Theory and Practice of Model Transformations (ICMT)*, (pp. 46–60). Springer. https://doi.org/10.1007/978-3-540-69927-9_4
- Kolovos, D. S., Rose, L. M., Matragkas, N., Paige, R. F., Guerra, E., Cuadrado, J. S., De Lara, J., Ráth, I., Varró, D., Tisi, M. & Cabot, J. (2013). A Research Roadmap Towards Achieving Scalability in Model Driven Engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, (pp. 1–10). ACM. <https://doi.org/10.1145/2487766.2487768>
- Kusel, A., Etzlstorfer, J., Kapsammer, E., Langer, P., Retschitzegger, W., Schoenboeck, J., Schwinger, W. & Wimmer, M. (2013). A survey on incremental model transformation approaches. In *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013)*, (pp. 4–13). CEUR-WP. <http://ceur-ws.org/Vol-1090/1.pdf>
- Le Calvar, T., Jouault, F., Chhel, F., & Clavreul, M. (2019). Efficient ATL Incremental Transformations. *The Journal of Object Technology*, 18(3), 1–17. <https://doi.org/10.5381/jot.2019.18.3.a2>
- Le Calvar, T., Jouault, F., Chhel, F., Saubion, F. & Clavreul, M. (2020). Intensional View Definition with Constrained Incremental Transformation Rules. In *Proceedings of 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, (pp. 395–402). IEEE. <https://doi.org/10.1109/MODELS-C.2019.00061>
- Martínez, S., Tisi, M., & Douence, R. (2017). Reactive model transformation with ATL. *Science of Computer Programming*, 136, 1–16. <https://doi.org/10.1016/j.scico.2016.08.006>
- Ogunyomi, B., Rose, L. M., & Kolovos, D. S. (2018). Incremental execution of model-to-text transformations using property access traces. *Software & Systems Modeling*, 18(1), 367–383. <https://doi.org/10.1007/s10270-018-0666-5>
- Ogunyomi, B., Rose, L. M. & Kolovos, D. S. (2014). On the use of signatures for source incremental model-to-text transformation. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão & E. Insfran (Eds.), *Model Driven Engineering Languages and Systems (MODELS)* (pp. 84–98). Springer. https://doi.org/10.1007/978-3-319-11653-2_6
- OMG. (2016). MOF2 Query/View/Transformation (QVT) Specification, V1.3. <https://www.omg.org/spec/QVT/1.3>
- Paige, R. F., Kolovos, D. S., Rose, L. M., Drivalos, N. & Polack, F. A. C. (2009). The design of a conceptual framework and technical infrastructure for model management language engineering. In *Proceedings of 14th IEEE International Conference on Engineering of Complex Computer Systems*, (pp. 162–171). IEEE. <https://doi.org/10.1109/ICECCS.2009.14>
- Peffer, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, 24(3), 45–77. <https://doi.org/10.2753/mis0742-1222240302>
- Ráth, I., Bergmann, G., Ökrös, A. & Varró, D. (2008). Live model transformations driven by incremental pattern matching. In A. Vallecillo, J. Gray & A. Pierantonio (Eds.), *Theory and Practice of Model Transformations (ICMT)*, (pp. 107–121). Springer. https://doi.org/10.1007/978-3-540-69927-9_8
- Runeson, P., & Höst, M. (2008). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2), 131–164. <https://doi.org/10.1007/s10664-008-9102-8>
- Samimi-Dehkordi, L., Zamani, B., & Kolahdouz-Rahimi, S. (2018). EVL+Strace: a novel bidirectional model transformation approach. *Information and Software Technology*, 100, 47–72. <https://doi.org/10.1016/j.infsof.2018.03.011>
- Sendall, S., & Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5), 42–45. <https://doi.org/10.1109/ms.2003.1231150>
- Sharbaf, M., & Zamani, B. (2020). Configurable three-way model merging. *Software: Practice and Experience*, 50(8), 1565–1599. <https://doi.org/10.1002/spe.2835>
- Balsamo, S., Di Marco, A., Inverardi, P., & Simeoni, M. (2004). Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5), 295–310. <https://doi.org/10.1109/tse.2004.9>
- Tisi, M., Martínez, S., Jouault, F. & Cabot, J. (2011). Lazy Execution of Model-to-Model Transformations. In J. Whittle, T. Clark & T. Kühne (Eds.), *Model Driven Engineering Languages and Systems (MODELS)*, (pp. 32–46). Springer. https://doi.org/10.1007/978-3-642-24485-8_4

-
- Varró, D., Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., & Ujhelyi, Z.** (2016). Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling*, 15(3), 609–629. <https://doi.org/10.1007/s10270-016-0530-4>
- Weidmann, N., Anjorin, A., Fritsche, L., Varró, G., Schürr, A. & Leblebici, E.** (2019). Incremental Bidirectional Model Transformation with eMoflon::lBeX. In *Proceedings of the Eighth International Workshop on Bidirectional Transformations (BX 2019)*, (pp. 45–55). CEUR-WS. <http://ceur-ws.org/Vol-2355/paper4.pdf>
-

Editorial record: The article has been peer-reviewed. First submission received on 27 February 2022. Revision received on 5 April 2022. Accepted for publication on 24 April 2022. The editor in charge of coordinating the peer-review of this manuscript and approving it for publication was Stanislava Mildeova .

Acta Informatica Pragensia is published by Prague University of Economics and Business, Czech Republic.

ISSN: 1805-4951
