

# A BPMN-Based Design and Maintenance Framework for ETL Processes

Zineb El Akkaoui, Esteban Zimányi

Université Libre de Bruxelles, Belgium

Jose-Norberto Mazón, Juan Trujillo

University of Alicante, Spain

---

## ABSTRACT

*Business Intelligence (BI) applications require the design, implementation, and maintenance of processes that extract, transform, and load suitable data for analysis. The development of these processes (known as ETL) is an inherently complex problem that is typically costly and time consuming. In a previous work, we have proposed a vendor-independent language for reducing the design complexity due to disparate ETL languages tailored to specific design tools with steep learning curves. Nevertheless, the designer still faces two major issues during the development of ETL processes: (i) how to implement the designed processes in an executable language, and (ii) how to maintain the implementation when the organization data infrastructure evolves. In this paper, we propose a model-driven framework that provides automatic code generation capability and ameliorate maintenance support of our ETL language. We present a set of model-to-text transformations able to produce code for different ETL commercial tools as well as model-to-model transformations that automatically update the ETL models with the aim of supporting the maintenance of the generated code according to data source evolution. A demonstration using an example is conducted as an initial validation to show that the framework covering modeling, code generation and maintenance could be used in practice.*

**Keywords:** data warehouses, ETL process, conceptual model, code generation, maintenance, model-driven development

---

## INTRODUCTION

Organizational data used by BI applications come from heterogeneous and distributed sources that are integrated into a data warehouse (DW) (Inmon, 2002). To achieve this integration, the data warehousing process includes the extraction of the data from the sources, the transformation of these data (e.g., to correct syntactic and semantic inconsistencies) and the loading of the warehouse with the cleansed, transformed data. This process is known as ETL (standing for Extraction, Transformation, Load). It has been widely argued that the ETL process development is complex, error-prone, and time-consuming (Simitsis, 2008; Vassiliadis, 2009; Wyatt, 2009). Actually, ETL process development constitutes the most costly part of

a data warehouse project, in both time and resources.

One of the main reasons for this is that, in practice, ETL processes have been traditionally designed by considering a specific vendor tool from the very beginning of the data warehouse project lifecycle. Unfortunately, commercial ETL tools have a steep learning curve, due to a lack of standard capabilities to be provided, e.g., they all provide different underlying languages with a wide spectrum of functionality features or complex wizards.

Some existing approaches address this problem by proposing a conceptual modeling stage for developing ETL processes in a vendor-independent manner (Skoutas 2009; Trujillo, 2003; Vassiliadis, 2005). These proposals successfully support the designer tasks, although they lack of effective

mechanisms for automatically generate vendor-specific code of the ETL process to be executed into different platforms. Moreover, the increasing need of fresher analysis data and the evolving nature of organizational data pose new challenges for these proposed approaches. The lack of systematic technique for continuous update of ETL process increases significantly the development effort (Papastefanatos, 2009). Indeed, during the ETL process lifecycle, both the data sources and the analysis requirements are likely to evolve, the latter implying an evolution of the data warehouse. Such changes may lead to inaccurate ETL processes: (i) syntactically invalid ETL model and code; and (ii) inconsistent data output generated by the process to feed the data warehouse. To avoid this situation, the ETL process should be automatically updated to accommodate the evolution. However, in general, schema evolution is done manually and remains an error-prone and time-consuming undertaking, because the designer lacks the methods and tools needed to manage and automate this endeavor by (i) predicting and evaluating the effects of the proposed schema changes, and (ii) rewriting queries and applications to operate on the new schema.

To overcome these problems, the present work proposes a Model-Driven Development (MDD) framework for ETL processes. This framework aims at covering the overall ETL development process, including the automatic generation of vendor-specific code for several platforms. Further, the framework supports an automated maintenance capability of the process and its code in order to accommodate evolution of organizational data.

For creating and managing ETL processes, in addition to the traditional graphical languages, current platforms generally provide programming capabilities through specific languages, which can be scripting languages (e.g. Oracle Metabase or OMB) or imperative languages (e.g. C# for SQL Server Integration Services). In our framework, transformations between a vendor-independent model and such

vendor-specific code are formally established by using model-to-text (M2T) transformations, an OMG standard for transformations from models to text (i.e. code). For evolving ETL processes, a set of model-to-model (M2M) transformations are iteratively applied on the original model to automatically derive the updated one. Finally, by applying our M2T transformations, the updated code can be derived.

The present framework relies on our previous work: the BPMN4ETL metamodel for designing ETL processes described in (El Akkaoui et al., 2012; El Akkaoui & Zimányi, 2009). The rationale behind this metamodel is the characterization of the ETL process as a combination of two perspectives: (i) A control process, responsible of synchronizing the transformation flows; (ii) A data process, feeding the data warehouse from the data sources. In this way, designers are able to specify conceptual models of ETL processes together with the business process of the enterprise (Wilkinson, 2010).

Furthermore, the model-driven approach has been customized from a generic data warehouse approach (Mazón & Trujillo, 2008) into a concrete implementation for the ETL component. Hence, our framework is motivated by the facilities provided by MDD technologies to support designers in their development and maintenance tasks by means of models and transformations. Importantly, model-to-text and model-to-model transformations enhance, respectively, the automatic generation and maintenance of executable code associated with the ETL process.

This paper describes our model-driven framework by illustrating its two main contributions:

- A code generation capability ensured by M2T transformations to any ETL programming language (via the definition of transformation patterns).
- An update capability to preserve the correctness of the generated code, as this

code may evolve due to data source or data warehouse changes.

- In order to provide an initial validation, an illustration of the latter framework capabilities is conducted by using a toy example as recommended by (Wieringa 2010), when the focus on the paper is explaining the framework usability.

The remainder of this article is structured as follows. The next section discusses related work in ETL modeling, implementation, and maintenance. Then, we provide an overview on our model-driven framework. The subsequent section describes the code generation mechanism. Then, we discuss how the framework copes with the evolution of the generated code. With the aim of providing an initial validation of our framework an example is used through these two sections. The last section concludes the article and points to some future perspectives.

## RELATED WORK

Existing approaches for developing ETL processes address three main issues: (i) designing ETL processes independently of a specific vendor, (ii) producing, based on the design, an executable code tailored to a specific technology, and (iii) maintaining the model and its code. We discuss next these issues.

Regarding the first issue, Vassiliadis (2005) and Papastefanatos (2009) propose modeling ETL processes using workflow and graph-based models that represent, respectively, data source relations and ETL objects. In order to facilitate ETL design, some automation mechanisms are proposed requiring additional semantics to be added to ETL objects. Other work in this direction describes the semantics of source and target schemas as well as their mappings using ontologies. For example, in Skoutas & Simitsis (2009) an application ontology is built, yielding a semi-automated construction of ETL processes based on graph operation rules. Another related approach (Romero, 2011) adds user requirements to the data source ontology, and provides an

algorithm for producing the conceptual ETL design and the data warehouse design. Unfortunately, a main drawback of these approaches is the enormous effort required to build the ontology comprising all the required information. On the contrary, in the present paper we advocate an ETL development approach that starts from a model based on a rich workflow language which does not require the definition of any supplementary ontology.

Regarding the implementation of ETL design, UML-based physical modeling of the ETL processes was proposed in Tziovara et al. (2007). This approach formalizes the data storage logical structure and the ETL hardware and software configurations. Further, it focuses on the optimization of the physical ETL design through a collection of algorithms. Although relevant to implementation, none of these proposals automatically produce code for executing ETL processes. An ETL programming approach using the Python language has been proposed by Thomsen & Pedersen (2009). Yet, this approach does not provide a vendor-independent design, limiting the reusability of the provided framework.

Another line of work takes into account both ETL development axes: design and code generation. For example, a conceptual metamodel for designing ETL processes based on BPMN and an implementation approach to its corresponding BPEL code is described in El Akkaoui & Zimányi (2009) and El Akkaoui et al. (2012). In Muñoz et al. (2009), the authors present a Model-Driven Architecture approach to design the ETL processes by means of the UML Activity Diagram. Again, none of these proposals provide a multi-vendor code generation utility.

The other related research on ETL processes studies data warehouse maintenance. For example, Golfarelli (2006) suggests a formalization of the data warehouse, its changes and versioning strategy. An intersection operator is proposed to specify the effect of the changes on the data warehouse and to state the validity of current OLAP queries among different data warehouse

versions. Specific ETL process maintainability approaches focus on ETL queries rather than OLAP ones. Papastefanatos (2009) identifies a set of structural changes on data sources and their associated replication algorithm on the ETL process. This algorithm takes into account user policies that define the ETL objects behavior towards a change (e.g. propagate and block). Moreover, quality in ETL process design is assessed according to maintainability purposes. Papastefanatos (2008) propose measures to compare alternative design techniques according to their tolerance to evolution events. Assessed ETL objects are either internal to the data warehouse, such as dimension tables, or external such ETL objects. Similarly, Muñoz et al., (2010) propose and validate a set of design measures related to ETL design maintainability.

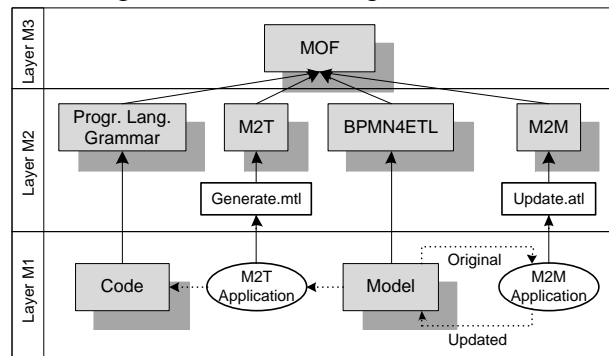
Along these lines, and building from our previous work (El Akkaoui et al., 2011), the contributions of this paper are as follows: (i) Improvement and extension of the code generation capability to ETL programming languages; and (ii) Addition of maintainability capabilities for the generated code to cope with data source or data warehouse evolution. The advisability of using models has been highlighted in other related complex domains such as data mining (Cuzzocrea et al., 2011; Cuzzocrea, 2011; Cuzzocrea, 2010).

## MODEL DRIVEN FRAMEWORK

Model Driven Development (MDD) is a paradigm for software development where extensive models are created before source code is generated from them. The architecture of MDD is depicted in a set of layers with different abstraction levels in which transformations are applied to refine models (based on metamodels) into the corresponding code (based on grammars). As shown in Fig. 1, the M2 layer contains vendor-independent description of concepts, i.e., metamodels, grammars and transformations among them. From these, vendor-specific representations are instantiated at the M1 layer, such as models,

while others are automatically generated such as code programs.

In order to improve the support of ETL process development, the BPMN4ETL metamodel (El Akkaoui et al., 2012) is used within a two-fold MDD-based framework. First, the framework enables implementation of ETL models (i.e. ETL process design) through executable code generation. Second, it handles the automatic updates of these models according to data store changes.



**Figure 1: MDD framework for ETL implementation and maintenance.**

As depicted in Fig. 1, at the M2 Layer, transformations are established using model-to-text (M2T) and model-to-model (M2M) transformations. M2T transformations, depicted by the Generate.mtl file, are responsible for code generation and consist of mapping the BPMN4ETL Metamodel to the Progr. Lang. Grammar. M2M transformations, depicted by Update.atl file, are used to update models for maintenance purposes and are created on the BPMN4ETL Metamodel.

Moreover, Fig. 1 shows the transformations at the M1 layer. The M2T Application on the Original Model, original BPMN4ETL instance, derives the corresponding code, Original Code. On the other hand, following to a modification on the Original Model, the Updated Model is automatically derived by applying the M2M transformations. The updated models can hence be automatically derived as many times as changes occur in the data sources. Consequently, the updated code can be produced by further applications of the same set of M2T transformations.

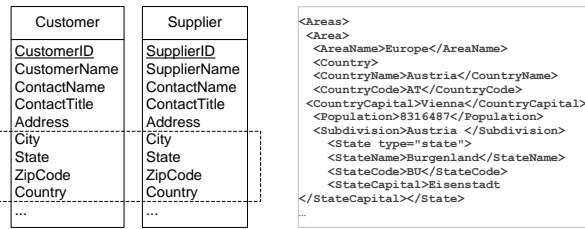


Figure 2: Excerpt of operational data sources.

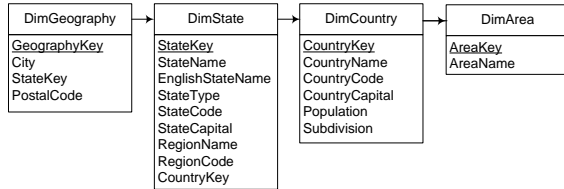


Figure 3: Excerpt of the data warehouse schema.

In the following, we describe our framework using a running example. Excerpts of the used data sources and the data warehouse schemas are respectively shown in Figs. 2 and 3. Operational data reside in both a relational database and an XML file. Decisional data reside in a hierarchy dimension about location data. Thus, the data contained in the sub-hierarchy DimState→DimCountry→DimArea come from the XML file called Territories.xml. The data contained in the DimGeography level is brought from Customer and Supplier tables.

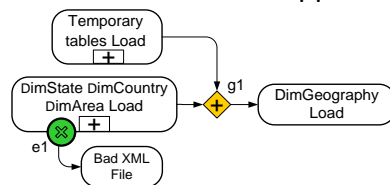


Figure 4: Control model.

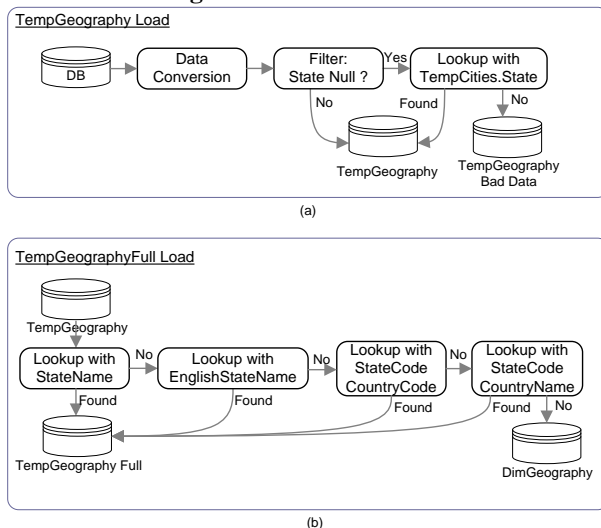


Figure 5: DimGeography Load data model.

The BPMN4ETL model designed to provide the DimGeography dimension with necessary data from the data sources are depicted in Figs. 4 and 5. It combines two perspectives: (i) a data process view that tracks data from the operational databases or other data sources to the data warehouse, providing precise information about the input and output data of each (data) process element; and (ii) a control process view that enables the orchestration of data processes together with adjacent applications.

Fig. 4 shows the example control process including load tasks and subprocesses of the location dimension levels, e.g., the DimArea DimCountry DimState Load subprocess and the DimGeography Load task. Other task kinds are considered by the control model such as Temporary Tables Load which creates temporary tables in the database useful for the DimGeography Load task. Fig. 5 depicts the example data process that populates the DimGeography dimension. Attribute State may be null in the Customer and Supplier tables. In these cases, data should be filled with its corresponding value using the TempCities table, see Fig. 5a. Referential integrity in the temporary TempGeography table is checked previously to the final loading using lookup tasks. For example, the StateName could be written in the original language or in its English translation (e.g., Karnten or Carinthia, respectively, for a state in Austria). Also, the state and/or country name can be abbreviated (AZ for Arizona and USA for United States). Fig. 5b shows the sequence of lookup tasks for these cases.

As described in Akkaoui & Zimányi (2009) and El Akkaoui et al. (2012), the BPMN4ETL language provides customized elements for representing ETL operators by extending BPMN ones. Next, we briefly outline the main ETL process elements respectively belonging to control and data process views.

**Control container.** A control container is a *control process/subprocess, swimlane, or loop*. A subprocess represents semantically coupled

adjacent elements that accomplish a significant portion or stage of the ETL process, e.g. the subprocess DimState DimCountry DimArea in Fig. 4. A swimlane enables the organization and the hierarchization of large ETL processes. For example, it divides the ETL process by business entities such as a department or company. Finally, it is usual that one or more tasks need to be executed multiple times which is addressed by the loop container;

**Control task.** A control task includes *data process* and *foreign control tasks*, which are respectively depicted in Fig. 4 by DimCategory Load and Temporary Tables Load;

**Control sequence.** A *gateway* and *control connection* represent relationships between control process elements. A Gateway has the specificity to merge and split the process. Gateways that simultaneously merge the flow are designed as a *parallel merge gateway*, e.g. g1 gateway in Fig. 4;

**Control event.** A control event represents something that happens and affects the sequence and timing of the ETL process. Events attached to tasks are designed as *boundary events*, e.g. the subprocess DimState DimCountry DimArea has a boundary event e1 in Fig. 4;

**Control artifact.** *Annotation* can be associated to any process element to add semantics;

Moreover, the data process elements are almost analogous to the control ones except for data tasks that we expose in the following:

**Data task.** Seven major categories of data tasks are (see Fig. 5).

- *Multi-field derivation* includes any kind of variable manipulation and computation, e.g. Data Conversion;
- *Filter* filters the input rows based on one or multiple conditions, e.g. Filter: State Null?;
- *Lookup* has two functionality: filtering the input rows based on their matching with a reference *fieldSet* (table) and including new fields from the reference

*fieldSet* to the input rows, e.g. Lookup with TempCities.State;

- *Split* splits the input fields (columns) into two field sets;
- *Merge* includes tasks that combine multiple row sets into a single one. It involves *join* and *union* tasks;
- *Aggregate* includes the application of standard, analytical and other custom aggregation functions;
- *Sort* orders the input rows;
- *Pivot & unpivot* transpose the input rows to columns;
- *Data input* is the entry point of data into the process from any possible data source: a database, file or web service, e.g. DB is a *column data input* that refers to a database. It has an associated *fieldSet* determining the extracted table into the process;
- *Data output* loads data into the data warehouse, e.g. DimGeography.

Each data task has particular behavior within the data process which is driven by its properties. For instance, input and output data tasks refer to resource *fieldSet* used to retrieve or load the data. The *resource* element, including this *fieldSet* determines the data source or warehouse. Also, the stream pipelining from the resources to data tasks is characterized by a group of input and output *fields*, denoted *InputSet* and *OutputSet* properties. Finally, *Condition*, *computation*, and *query* properties are used to address expressions applied by data tasks.

## MODEL-DRIVEN ETL CODE GENERATION

In this section, we describe the code generation capability of our framework. It is based on a vendor-independent pattern for M2T transformations.

## Transformation Pattern

The transformations consist of matching statements between the input metamodel and the output grammar. They are expressed using a set of templates, each of which is responsible of translating one element of the metamodel into a snippet of code. The templates are applied according to the pattern shown in Fig. 6. Note that this pattern is independent from the specific ETL programming language. Hence, it could be provided as a guideline for developing code generators for any ETL tool.

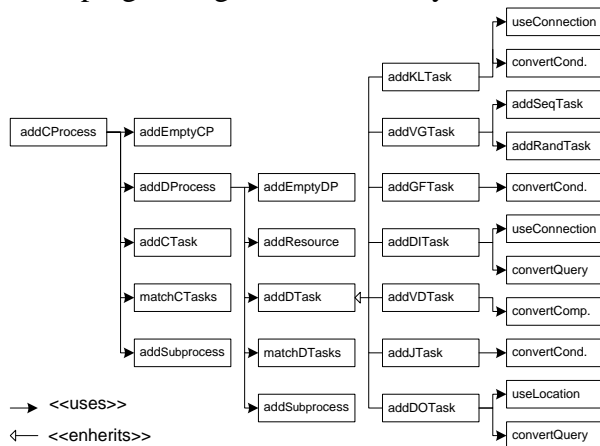


Figure 6: M2T transformation pattern.

Fig. 6 highlights the pattern of code generation for BPMN4ETL models. This pattern starts by creating the main control process using the addEmptyCP template and then iteratively creates its components, e.g. data processes using addDProcess. The latter requires the creation of a new empty data process using the addNewDP template as well as its resources (data sources and warehouses), and are linked to each other by using respectively addResource, addDTask, and matchDTasks templates. According to the data task type, a specific template inheriting from addDTask is applied such as addDITask. Similarly, foreign control tasks templates of the control model are created and matched, as well as subprocesses, loops, and events' components.

## Transformation Implementation

The M2T transformations for code generation are implemented by using the above

pattern. During execution, the BPMN4ETL model is provided as input argument, where each element is converted by a specific template to the target tool language. A template contains static and dynamic code. The static code is replicated literally during the execution. Dynamic code corresponds to OCL expressions specified using the model elements. Our set of M2T transformations are implemented and executed within the Acceleo transformation engine.

We illustrate next the transformations from the BPMN4ETL to the Oracle MetaBase (OMB), the language used by Oracle Warehouse Builder (OWB) for implementing ETL processes. Equivalence between BPMN4ETL and OMB objects is established through these transformations. For example a control process, control task, data process, and data task respectively correspond to a PROCESS FLOW, ACTIVITY, MAPPING, and OPERATOR in OMB.

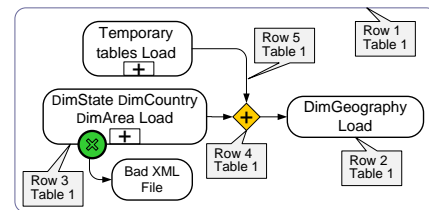


Figure 7: Control model code generation.

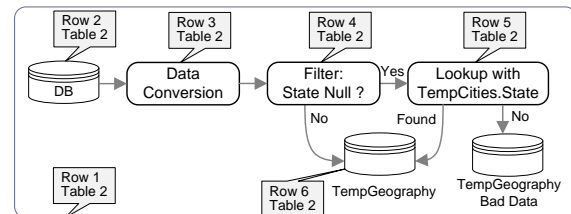


Figure 8: Data model code generation.

In the following, we show the code generation mechanism on the running example. The transformation details are shown in Figs. 7 and 8. Each element in the figures indicates the row number in Tables 1 or 2 containing the corresponding code and the applied template. For example, Fig. 7 states that the generated code for the control connection is depicted in Row 5 of Table 1.



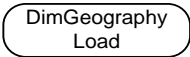
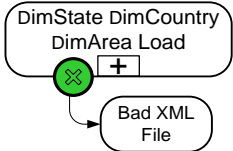
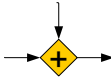

#	Object Name	OMB Code	Description
1	The overall control process	OMBCONNECT OWB OWNER/OWB OWNER@OWB SYSTEM:1521:PROD USE REPOSITORY 'OWB OWNER' ### CREATE MODULE ### OMBCREATE PROCESS FLOW MODULE 'M_PF_NWETL' ### CREATE FLOW PACKAGE ### OMBCREATE PROCESS FLOW PACKAGE 'P PF FACTSALES' ### POSITION ON THE PACKAGE ### OMBCC 'P PF FACTSALES' ### TRANSFORM CONTROL PROCESS ### OMBCREATE PROCESS FLOW 'CP FACT SALES LOAD'	Transforms a Control Process into a PROCESS FLOW – using the addCProcess template, Listing 1.
2	Control Task 	### TRANSFORM DATA PROCESS TASK ### OMBALTER PROCESS FLOW 'CP FACT SALES LOAD' ADD MAPPING ACTIVITY 'CT DIMCATEGORY LOAD'	Transforms a Control Task of type Data Process into a MAPPING activity – using the addCTask template, Listing 6 (Appendix).
3	Control Subprocess & Boundary Event 	### TRANSFORM SUBPROCESS ### OMBALTER PROCESS FLOW 'CP FACT SALES LOAD' ADD SUBPROCESS ACTIVITY 'CT DIMAREA DIMCOUNTRY DIMSTATE LOAD'	Transforms a Control Task of type Control Subprocess into a SUBPROCESS activity – using the addCTask template, Listing 7 (Appendix).
4	Gateway 	### TRANSFORM PARALLEL MERGE GATEWAY ### OMBALTER PROCESS FLOW 'CP FACT SALES LOAD' ADD AND ACTIVITY 'G AND'	Transforms a Boundary Event into END ERROR activity, the related Compensation Task into USER DEFINED activity – using Listing 8 (Appendix).
5	Control Connection 	### TRANSFORM CONNECTION ### OMBALTER PROCESS FLOW 'CP FACT SALES LOAD' ADD TRANSITION 'C TEMPORARYTABLES LOAD G AND' FROM ACTIVITY 'CT TEMPORARYTABLES LOAD' TO 'G AND'	Transforms a Parallel Merge Gateway into an AND activity – using Listing 9 (Appendix).

Table 1: Generated OMB code for the control elements in our running example.

*Control Model Transformation.* The control model involves several elements: control task, control subprocess, boundary event, gateway, and control connection. Table 1 shows that a data process is matched to a MAPPING activity in OMB (Row 2), whereas, a control subprocess is translated to a SUBPROCESS activity (Row 3). A boundary event and its compensation task are mapped to an END ERROR and USER DEFINED activities, respectively. The control connection between these elements is transformed into a TRANSITION between the associated activities (Row 3). A parallel merge gateway is mapped to the AND activity (Row 4). Finally, the control connection is mapped to a TRANSITION in OMB (Row 5). We describe next the control process transformations corresponding to the

control model element. The other templates are provided in Appendix.

```

1 [template addControlProcess(cprocess :
2 ControlProcess)]
3 [cprocess.setContext()/]
4 ### TRANSFORM CONTROL PROCESS ###
5 OMBCREATE PROCESS_FLOW 'CP_[cprocess.name/]'
6 ### TRANSFORM CONTROL OBJECTS ###
7 [for (c : ControlObject |
8 cprocess.controlObjects)]
9 [if (c.ocIsKindOf(ControlTask))]
10 [c.ocAsType(ControlTask).addControlTask()/]
11[/if]
12[if (c.ocIsKindOf(ControlEvent))]
13[c.ocAsType(ControlEvent).addControlEvent()/]
14[/if]
15 [if (c.ocIsKindOf(Gateway))]
16 [c.ocAsType(Gateway).addGateway()/]
17[/if][/for]
18 ### TRANSFORM CONNECTIONS ###
19 [for (co : ControlObject |
20 cprocess.controlObjects)]
21 [if not (co.outConnections->isEmpty())]
22 [for (c : Connection | co.outConnections)]
23 [c.addConnection()/][/for][/if][/for]
24 [/template]


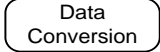



```

Listing 1: Control process template.



Listing 1 depicts the Acceleo template code `addControlProcess` that transforms a control element. First, in Line 2, the `setContext` template is invoked to set general information about the OWB project, e.g., the connection to OWB repository, module, and package associated with the process. Second, in Line 4, the template `addControlProcess` creates the OMB control process counterpart, called a PROCESS FLOW, by using the `OMBCREATE` command. This statement allows creating any OWB process object. In order to uniquely

identify the control process, we assume that the name property is unique for all control processes. The generated OWB name is composed of this property preceded by the CP prefix. The same logic has been used for naming all the generated OWB model elements each time a particular prefix needs to be added, as it is shown in the generated OMB code in Table 1. The Acceleo engine applies iteratively the `addControlProcess` template over all the control elements.

#	Object Name	OMB Code	Description
1	The overall data process	OMBCREATE ORACLE MODULE 'SALES DW' OMBCREATE LOCATION 'MY LOCATION' SET PROPERTIES (TYPE, VERSION) VALUES ('ORACLE DATABASE', '11.2') OMBCREATE MAPPING 'DIMGEOGRAPHY LOAD'	Creates and configures a project context, and transforms a data process into a MAPPING – using <code>addDataProcess</code> in Listing 2.
2	Column data input task 	OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ADD VIEW OPERATOR 'GEOGRAPHIES' SET PROPERTIES (QUERY) VALUES 'SELECT CITY, POSTALCODE, REGION AS STATE, COUNTRY FROM CUSTOMERS'	Transforms the column data input task into a VIEW operator – using Listing 10 (Appendix).
3	Multi-field derivation task 	OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ADD EXPRESSION OPERATOR 'DATA CONVERSION'  OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ADD CONNECTION FROM GROUP 'INOUTGRP1' OF OPERATOR 'GEOGRAPHIES' TO GROUP 'INGRP1' OF OPERATOR 'DATA CONVERSION'  OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ALTER ATTRIBUTE 'CODEPOSTAL' OF GROUP 'OUTGRP1' OF OPERATOR '[DATA CONVERSION]' SET PROPERTIES (EXPRESSION) VALUES ( 'To Number(INGRP1.CODEPOSTAL)')	Transforms the Data Conversion task into an EXPRESSION operator and the conversion expression into the EXPRESSION property – using Listing 11 (Appendix).
4	Filter task 	OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ADD FILTER OPERATOR 'NULL STATE' SET PROPERTIES (FILTER CONDITION) VALUES ('INGRP1.STATE = NULL')	Transforms the filter task into a FILTER operator – using Listing 12 (Appendix).
5	Lookup task 	OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' OMBCREATE FLATFILE "CITIES" SET PROPERTIES(DATA FILE NAME, RECORD DELIMITER, FIELD DELIMITER) VALUES('C:\nn Cities.txt', 'nn n',';')  OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ADD LOOKUP OPERATOR 'LOOKUP STATE' SET PROPERTIES (LOOKUP CONDITION) VALUES ('INGRP1.STATE = TEMPCTIES.STATE') BOUND TO TABLE 'TEMPCTIES'	Transforms the lookup task into a LOOKUP operator and the lookup condition property into the LOOKUP CONDITION – using <code>addKLTASK</code> in Listing 13 (Appendix).
6	Data output task  TempGeography	OMBALTER MAPPING 'DIMGEOGRAPHY LOAD' ADD TABLE OPERATOR 'TEMPGEOGRAPHY' BOUND TO TABLE 'TEMPGEOGRAPHY'	Transforms (column) data output task into the TABLE operator – using Listing 14 (Appendix).

**Table 2: Generated OMB code for the data objects in our running example.**

*Data Model Transformation.* Data model code generation consists mainly on transforming different types of data tasks. Table 2 shows some of these transformations.

Column data input is transformed into a VIEW operator. The `selectQuery` property from the data input is assigned to the `QUERY` property of the operator (Row 2). Also, a multi-field derivation task, e.g. data conversion, is transformed into the `EXPRESSION` operator (Row 3). These elements are linked using a `CONNECTION`, that allows the input attributes (i.e. `GROUP`) definition. Then, it modifies the type of the `PostalCode` attribute by applying the `To_Number` function. For the filter task, a `FILTER` operator is used with a `FILTER CONDITION` transforming the filter condition into SQL (Row 4). Finally, the lookup task uses a file resource as a lookup reference, which is loaded into the `TempCities` temporary table using a record data input task. The record data input is translated into a `FLATFILE` operator and the lookup task into the `LOOKUP` operator. The lookup is bound to the reference table `TEMPCITIES` (Row 5). Next we describe the data process transformations.

```
1[template public addDataProcess(dprocess :
2 DataProcess)]
3 [dprocess.setContext()/]
4 OMBCREATE MAPPING '[dprocess.name/]' \
5 [for (t : DataTask | dprocess.dataTasks)]
6 [OMBALTER MAPPING '[t.dataProcess.name/]' \
7 [t.addDTasks()]/]/for]
8 [for (ds: DataTask |
9 t.inputSets.source.dataTask]
10[if not (ds.ocIsUndefined())]
11[if not
12(ds.ocIsKindOf(MultiFieldDerivation))]
13[ds.addDConnection()]/]/if]/if]/for]
14[/template]
```

**Listing 2: Data process template.**

We mentioned, while explaining Listing 1, that the transformation of a control task of type data process invokes the `addDataProcess` template, whose code is shown in Listing 2. Line 4 creates a `MAPPING`. Then, the data tasks code is generated interactively by invoking `addDTasks` in Line 7. Except for the multi-field derivation task, Lines 8-13 add the connections between the tasks and their

predecessors using `addDConnection`. The multi-field derivation task requires a particular technical treatment, which we omit for the sake of conciseness.

It is worth mentioning that some custom templates, not mentioned in the template pattern, may be added during the implementation. These templates differ among ETL tools, thus they need to be specified for each target tool.

## MODEL-DRIVEN ETL CODE EVOLUTION

Likewise the ETL process, the generated code may evolve over the time due to data source and/or data warehouse update. In this section we show how our model-driven framework can automatically maintain the ETL process in order to generate an evolving code, which correctly answers the data warehouse requirements.

On the first hand, the structure of data sources is continuously updated which may have implications on data warehouse consistency. On the other hand, the data warehouse structure can also be updated due to new analysis requirements. In both cases, evolution mechanisms should be established to handle such updates and adapt the ETL process. However, we focus in this work on the data source updates.

For this purpose, our MDD-based framework follows a typical three-step approach for automating process evolution: (i) identify the source updates; (ii) determine their potential implications over data process elements; and (iii) specify evolution strategies to automatically handle the updates.

### Update Identification

In our approach, the data source and warehouse schemas are captured into a simplified metamodel referred to as the *resource metamodel* (El Akkaoui et al., 2012). It contains two main classes: *Field* (e.g. column) and *FieldSet* (e.g. table) with corresponding properties. The resource

metamodel provides these abstraction mechanisms in order to cope with several types of data models. Thus, it is suitable for record-based, column-based, and XML-based data source types.

Structural updates of the resource metamodel are identified based on a formalization of schema modifications proposed in Curino et al. (2008).

Updates	Description
Create table	Introduces a new, empty table to the database (Add_FieldSet)
Drop table	Removes an existing table from the schema and deletes the data in the table (Drop_FieldSet)
Rename table	Renames a table, without affecting the data (Rename_FieldSet)
Distribute table	Distributes table tuples into two newly generated tables (Horizontal_Split)
Merge table	Creates a new table by merging data of two tables with the same schema (N/D)
Copy table	Creates a duplicate of a table (N/D)
Add column	- Introduces a new column into the specified table (Add_Field) - Changes column semantics: conversion, concatenation and split (Alter_Field)
Drop column	Removes an existing column from a table (Drop_Field)
Rename column	Renames an existing column from a table (Rename_Field)
Copy column	Copies a column into another table (N/D)
Move column	Copies a column but the original column is dropped (N/D)

**Table 3: Curino et al. (2008) schema updates.**

Table 3 describes the schema updates identified in Curino et al. (2008) (e.g. Create table) along with their corresponding updates in the resource metamodel (e.g. Add\_FieldSet). The updates depicted with N/D are not considered since they constitute a composition of others. The Merge Table update comes to several Add\_Field's on the table from the structural viewpoint. The Copy Table/Column does not affect the data process models. Finally, the Move Column consists of a Drop\_Field composed with an Add\_Field.

Using the resource metamodel syntax, Add\_Field, Drop\_Field, Alter\_Field, and

Rename\_Field respectively correspond to add, drop, alter, and rename an instance of the Field class. Add\_FieldSet, Drop\_FieldSet, and Rename\_FieldSet consist respectively of add, drop and rename an instance of the FieldSet class. Finally, Horizontal\_Split is a structural update that breaks down an instance of FieldSet into two instances. Since the changes are identified on the common field-based structure, they can be applicable for all data source and warehouse types.

## Evolution Strategies

Each resource *Field* or *FieldSet* update has a specific implication over the elements of the ETL process. This section shows how our framework handles this impact through a set of evolution strategies. Only data process elements are concerned of the evolution because of their direct relation with the data source (see Section 3).

In BPMN4ETL, *Field* and *FieldSet* constitute properties frequently associated to data process elements, specifically tasks, resources, and expressions. Hence, any *Field* or *FieldSet* modification is directly reflected on these elements. Namely, field is a property of an InputSet/OutputSet, Query, Condition, and Computation classes, while a fieldSet is a property of a Data Input, Lookup, and Data Process classes. Table 4 shows the handling actions to be effected at each of these elements. A mark \* is used when no actions is required. For instance, an Add\_Field update does only affect the data input task by adding and configuring an extraction query with named fields without the new added one.

Next, we provide an overview of the evolution strategy for each data source update:

Add\_Field should be handled, as mentioned, by adjusting all data input tasks: create extraction query (if does not exist) with named fields excluding the new one. This action maintains the ETL process and the data warehouse unchanged, as initially specified by business users.

Updates	Handling action on elements associated with field				Handling action on elements associated with fieldSet		
	InputSet/ OutputSet	Query	Condition	Computation	Data Input	Lookup	Data Process
Add_Field	✖	✖	✖	✖	Add extraction query with named fields excluding new one	✖	✖
Drop_Field	Drop field	Drop field	Remove field related cond. or remove all	Field value to null in comp. or remove all	✖	✖	✖
Alter_Field (type, length)	✖	✖	✖	✖	✖	✖	Add multi-field deriv.
Rename_Field	Rename	Rename	Rename	Rename	✖	✖	✖
Rename_FieldSet	✖	✖	✖	✖	Rename	Rename	✖
Add_FieldSet	✖	✖	✖	✖	✖	✖	✖
Drop_FieldSet	✖	✖	✖	✖	Remove data input (& tasks to first encountered merge /data output )	Remove lookup	Remove a part or all data process
Horizontal_Split	✖	✖	✖	✖	Replace with two generated data inputs	✖	Add union of the two inputs

Table 4: Handling action on field and fieldSet associated elements.

Drop\_Field must be handled by removing the field property instance from the associated elements, as shown in Table 4. Removing a field for a condition means deleting one of its operands, because the derived meaningless condition part should be removed. For example, removing  $f3$  from  $(f1 = f2)$  or  $(f3 <> f4)$  condition drives into  $(f1 = f2)$  condition. For computation, removing a field implies attributing a null value to its occurrences. When the field constitutes the left operand of the computation, this latter should be removed. It is worth mentioning that removing some elements may lead to 'inactive' tasks (see Table 5); thus, requiring to be deleted after a designer workaround.

Alter\_Field consists of altering the field type or length. Simple cases entail conversion between equivalent structure (i.e. type and length) or from one structure to a sub-one. For example, a simple Alter\_Field converts the field from character to string or byte to integer without increasing its length. To cope with such update, a re-conversion is applied using a multi-field derivation task immediately after the data input task extracting the altered field.

The rest of the model is preserved. The same evolution strategy can be required for conversions to super structures but risking information lost.

Rename\_Field requires renaming the field among the data process elements.

Rename\_FieldSet requires renaming the field among the data process elements.

Add\_FieldSet does not require any evolution.

Drop\_FieldSet raises two close possibilities: dropping either the fieldSet associated with a data input (extraction table) or with a lookup (reference table) both implying a task deletion.

Horizontal\_Split implies replacing the existing data input task into two data input tasks, where each extracts one splitted fieldSet, which are then merged using a union task.

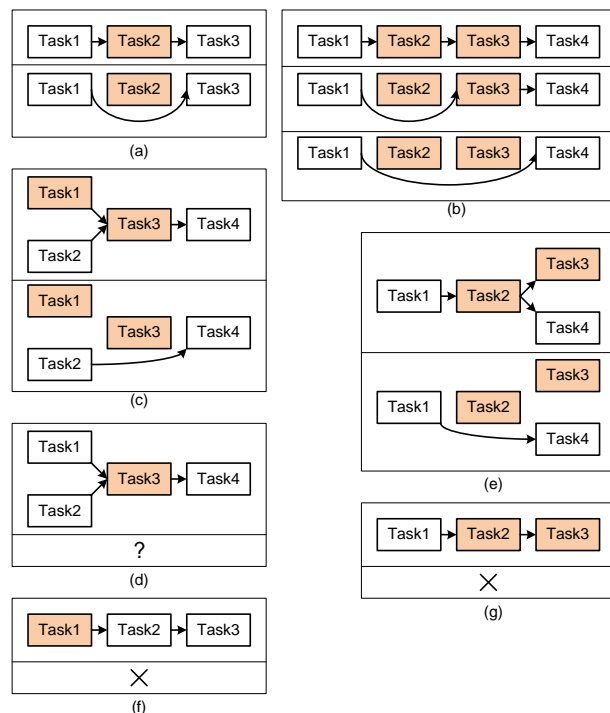
It is worth mentioning that even if the evolution strategies are described using BPMN4ETL, they stay valid for any ETL language due to equivalence between ETL operators. Further, as said, task deletions may induce further updates on the rest of the process, as it is studied next.

**Task Remove Scenarios.** When a mandatory property is eliminated due to a drop update, the associated task is taken away because it becomes meaningless.

Inactive Task	Circumstances
Any task	No field in task input or output
Aggregation	No field in (group by) fields
Sort	No field in the task field
Pivot	No field in task fields
Multi-field derivation	No computation in task
Loukup/ Filter/ Join	No condition in task
Lookup	No fieldSet in task reference relation
Data input	No extraction query in task and no extraction relation
Data process (subprocess)	No data input or data output tasks

**Table 5: Circumstances for inactive tasks.**

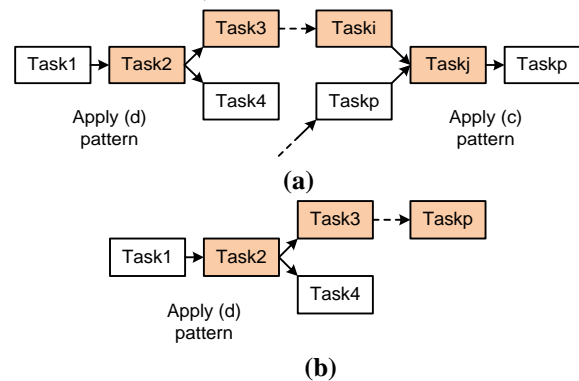
Table 5 captures the circumstances where a task should be removed. For example, any task loses sense by removing its input or output fields. Also, the aggregation task is not applicable without the group by fields. Besides, the data process (or subprocess) has no meaning with no data input or output tasks.



**Figure 9: Remove inactive tasks scenarios.**

In practice, removing a task follows one of the following scenarios, depicted in Fig. 9:

- Scenario (a) and (b) drop one-to-one links connecting the task to its neighbors. Scenario (a) assumes only one task is to be removed. Scenario (b) assumes more neighbor tasks to be removed, which drives the application of scenario (a) multiple times;
- Scenario (c) drops a merge task with one of its incoming task. The other incoming task is then linked with the merge successor task;
- Scenario (d) holds no incoming task to the merge is to be dropped. In this case, the process cannot be linked automatically and the designer is involved;



**Figure 10: Drop split strategy with: (a) no merge task; (b) with merge task.**

- Scenario (e) removes a split task which necessary induces the remove of one of its outgoing tasks. The merge task is deleted according to scenario (a) or (b). Tasks from one splitted stream are to be removed until a merge task, or a data output task is met, see Fig. 10(1) and Fig. 10(2). If a merge is met, it is deleted according to scenario (c).
- Scenario (f) removes the data input task and its outgoing tasks until a merge or data input task is met. If a merge is crossed apply scenario (c) else drop all the process because the process has no more input stream.
- Scenario (g) removes all predecessor tasks to the data output until a split or a

data input is met. In case the split is met, apply scenario (d).

Other possible scenarios are not identified in Fig. 9 because they do not happen in "real-world" situations, such as drop data output task without its predecessor task, or unlink a split task without one of its outputs.

## Evolution Implementation

This section shows how our MDD framework enables an easy implementation of the highlighted evolution strategies. Specifically, the ATL language is used for establishing a set of M2M transformations in a formal manner.

*ATL Language Preliminaries.* Model evolutions are implemented in the Atlas Transformation Language (ATL) language, a hybrid declarative-imperative language for implementing M2M transformations. Declarative rules are preferred over imperative ones, since they enable to match output elements with input ones. They are typically called matched rules. Imperative rules are typically invoked by declarative ones in order to allow the use of control statements, e.g. if then else and for statements. Typical imperative rules are called helpers.

Moreover, ATL proposes an advanced capability called *refactoring* or *refining* mode. This capability avoids the necessity of creating rules and bindings for each element and property in the model, only modified elements require rules (as in our evolution scenario).

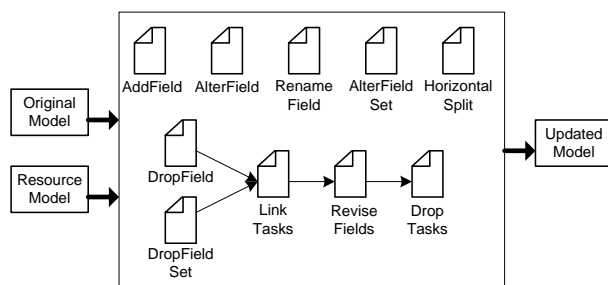


Figure 11 : ATL evolution modules.

*Evolution Modules.* Some update strategies should be performed in steps. However, in

contrast with M2T template, the M2M rules are applied simultaneously and independently from the input model hierarchy. Modules are thus used to encompass simultaneous rules and progressively apply sequential ones.

Modules implementing the specified update strategies are depicted in Fig. 11. According to the update, one or a sequence of modules is applied on the Original Model to produce the Updated Model. The Resource Model determines the updated part of the data source. For example, a DropField event is addressed by four modules: (i) a DropField module is applied to drop the field from directly associated elements; (ii) a LinkTasks module creates a new link from the previous to the successor task in order to get around the task to be removed; (iii) DropTasks module is applied to actually drop these tasks; finally (iv) ReviseFields module removes the fields generated by the dropped task from the successor ones. An additional module Common groups the helpers to be used by the other modules.

*Implementation Illustration.* Suppose that in the source table Customer of Fig. 2 the field City is removed. By applying the Drop\_Field evolution strategy, the data process model of Fig. 5 holds the following changes: (i) all the City field occurrences are removed; (ii) the derived useless computation ca is removed from the multi-field derivation tasks; (iii) unlinks the Data\_Conversion task by linking its previous and successor tasks; and (iv) Data\_Conversion task is removed.

The evolution is performed in steps by successively executing the aforementioned ATL modules, where each module partially contributes to the process evolution. For instance the DropField module drops the City field occurrences and the ca computation, while the LinkTasks gets around the Data\_Conversion task to be removed.



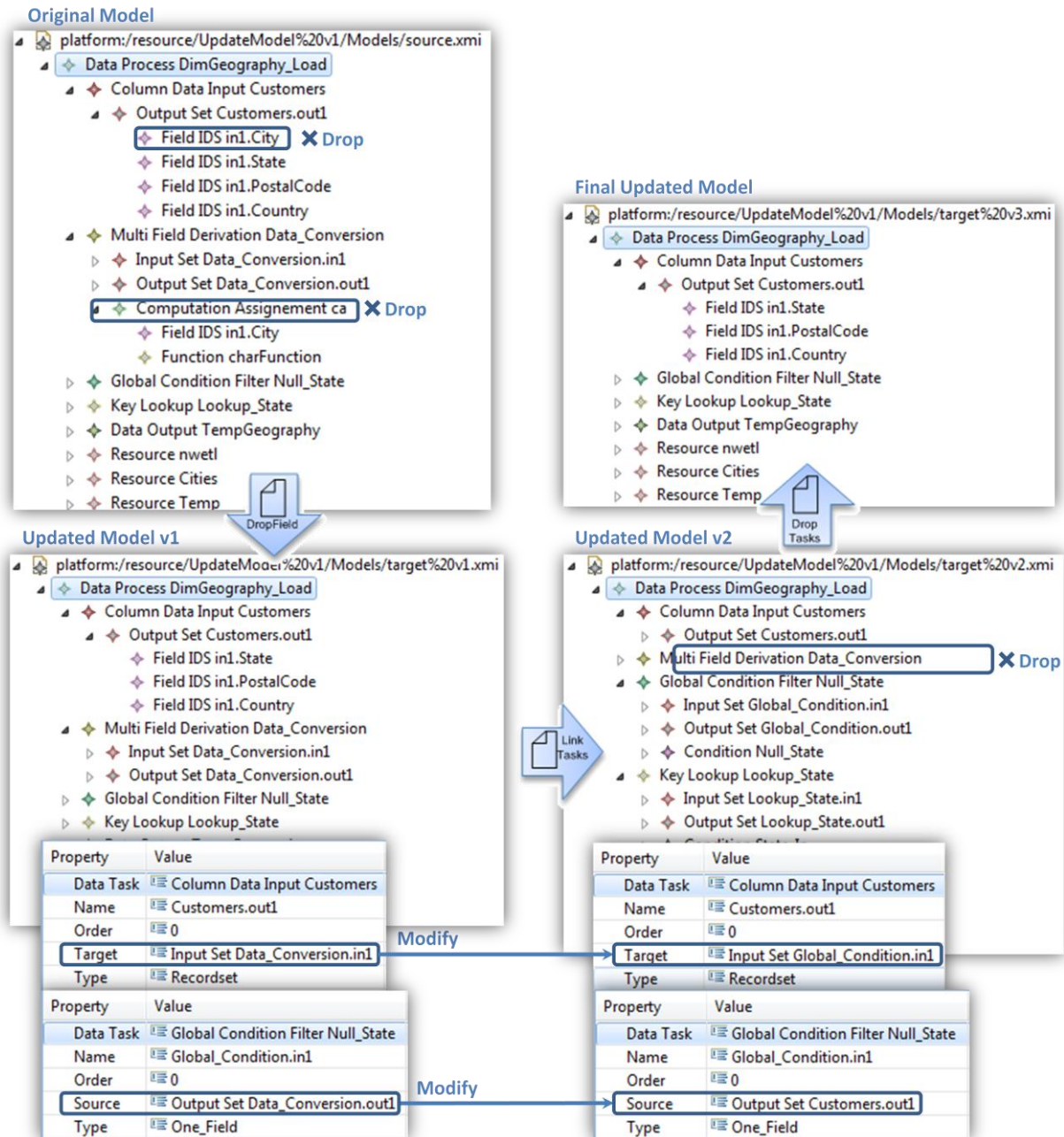


Figure 12: Drop\_Field change on DimGeography\_Load example (Fig. 5).

We explain next how the evolution modules are implemented in ATL. Fig. 12 shows the example process designed by using the Eclipse Ecore tree editor<sup>i</sup> in different evolution steps.

```
-- @atlcompiler atl2010
-- @path BPMN4ETL=../Models/DP3.4.ecore
-- @path Resource=../Models/Resource.ecore
module DropField;
create UpdatedModel : BPMN4ETL refining
OriginalModel : BPMN4ETL, modified :
Resource;

-- drop field from all process elements
rule DropField{
  from old_df : BPMN4ETL!FieldIDS,
```

```
df: Resource!FieldIDS(old_df.name=df.name)
to drop
}
-- drop unitary condition if one operand is
the dropped field or not defined
rule DropFieldUCondition{
  from old_uc : BPMN4ETL!UnaryCondition,
  df : Resource!FieldIDS(old_uc.field.name=
df.name or old_uc.field.oclIsUndefined())
to drop
}
-- drop binary condition if one operand is
the dropped field or not defined
rule DropFieldBCondition{
  from old_bc : BPMN4ETL!BinaryCondition,
  df : Resource!FieldIDS(old_bc.lField.name=
df.name or old_bc.rField.name=df.name or
old_bc.lField.oclIsUndefined() or
old_bc.rField.oclIsUndefined())
```



```

    to drop
}
-- drop computation assignment fields if
the left operand is the dropped field or not
defined
rule DropFieldComputation{
  from old_ca:
BPMN4ETL!ComputationAssignment,
  df : Resource!FieldIDS(old_ca.lValue.name=
df.name or old_ca.lValue.oclIsUndefined())
  to drop
}

```

**Listing 3: ATL code of the DropField module.**

Listing 3 shows the DropField module that applies the first evolution step transforming the Original Model to Updated Model v1, depicted in Fig. 12. First, the original, updated and resource models, as well as the applied refining mode are indicated in the create module statement. Second, rules are established for each element being modified by the module. For example, in order to remove all City field occurrences, the DropField rule matches field elements to null, using the drop keyword. The fields to be removed are indicated in the resource model, by using a filtering condition e.g. `old_df.name=df.name`. Moreover, in order to remove derived useless conditions and computations elements, rules such as DropFieldComputation are applied. In our example, this rule implies the remove of the ca computation.

```

-- @atlcompiler atl2010
-- @path BPMN4ETL=../Models/DP3.4.ecore
module LinkTasks;
create UpdatedModel : BPMN4ETL refining
OriginalModel : BPMN4ETL;
uses Common;

rule LinkTaskIS{
  from is : BPMN4ETL!InputSet(is.source.
dataTask.isToDrop() and not is.source.
dataTask.oclIsTypeOf(BPMN4ETL!DataInput))
  to update_is: BPMN4ETL!InputSet(
    source <- is.source.dataTask.inputSets->
collect(is1|is1.source)->first()
  )
}
rule LinkTaskOS{
  from os : BPMN4ETL!OutputSet(os.target.
dataTask.isToDrop() and not os.target.
dataTask.oclIsTypeOf(BPMN4ETL!DataOutput))
  to update_os: BPMN4ETL!OutputSet(
    target<- os.target.dataTask.outputSets->
collect(os1|os1.target)->first()
  )
}

```

**Listing 4: ATL code of the LinkTasks module.**

Listing 4 depicts the LinkTasks module part applying the remove task Scenario (a), see Fig. 9. This module is in charge of the second evolution step transforming the Updated Model v1 to Updated Model v2, see Fig. 12. It first detects the inactive tasks to be removed using `isToDrop()` helper. For example, Data\_Conversion task should be removed since no computations remains in this task. Second, it updates links between previous and successor tasks to the tasks to be removed, using the LinkTaskIS and LinkTaskOS rules. Particularly, these rules are responsible of respectively modifying the target properties of inputSet and outputSet elements of neighbor tasks. Fig. 12 shows for example that after the module execution, the target property of Customers outputSet points on Global Condition inputSet.

```

-- @atlcompiler atl2010
-- @path BPMN4ETL=../Models/DP3.4.ecore
module DropTasks;
create UpdatedModel : BPMN4ETL refining
OriginalModel : BPMN4ETL;
uses Common;

rule DropDataTask{
  from dt : BPMN4ETL!DataTask(dt.isToDrop())
  to drop
}

```

**Listing 5: ATL code of the DropTasks module.**

Listing 5 shows the DropTasks module responsible of the last evolution step by translating the Updated Model v2 to Final Updated Model, depicted in Fig. 12. It is responsible of actually removing the Data\_Conversion task element.

## CONCLUSION

In this paper we discussed a BPMN-based, vendor-independent framework for implementing ETL processes that copes with evolution of data sources. Using a Model-Driven Development (MDD) approach, ETL models built using our BPMN4ETL metamodel can be translated into vendor-specific code supported by any ETL tool, using a suite of Model-to-Text transformations. Further, in the case of data

source evolution, the generated ETL code can be automatically updated using Model-to-Model transformations.

Several research challenges arise from the work presented in this paper. Even though an initial validation of our framework has been conducted, demonstrating the usability of the framework by means of an exhaustive validation procedure is still missing. Further, code generation still takes significant development effort since each target ETL tool requires a particular suite of transformations. We believe that this problem can be addressed in two ways: (a) a set of technology-independent patterns can be defined to guide the transformation development; (b) ETL tools can be categorized according to three paradigms from the data processing perspective (i.e., procedural, imperative, and hybrid), and in two paradigms from the control process perspective (i.e., imperative and workflow). This suggests that we could define ‘pivot’ metamodels for these ETL paradigms, and then, using the MDD approach, an automatic mapping from our metamodel to the pivot metamodels could be built. The main effort will be then restricted to define M2T transformations from a pivot metamodel to the target tool metamodel.

## APPENDIX

### Control M2T Transformations

```
[template addControlTask(ctask:ControlTask)]
[if (ctask.ocIsKindOf(DataProcess))]
  ### TRANSFORM DATA PROCESS TASK ###
[ctask.ocIsType(DataProcess)].
addDataProcess()/]
  OMBALTER PROCESS_FLOW
  'CP_[ctask.controlProcess.name/]'
  ADD MAPPING ACTIVITY
  'CT_[ctask.name/]'[/if]

  [if (not
ctask.ocIsKindOf(ForeignControlTask))]
  ### TRANSFORM FOREIGN CONTROL TASK ###
  OMBALTER PROCESS_FLOW
  'CP_[ctask.controlProcess.name/]'
  ADD WEB SERVICE ACTIVITY
  'CT_[ctask.name/]'[/if]

  [if (ctask.ocIsKindOf(DataSubProcess))]
  ### TRANSFORM SUBPROCESS ###
```

```
[ctask.ocIsType(DataProcess)].addDataProcess
()/]
  OMBALTER PROCESS_FLOW
  'CP_[ctask.controlProcess.name/]'
  ADD SUBPROCESS ACTIVITY
  'CT_[ctask.name/]'[/if]
[/template]
```

### Listing 6: Control task transformation

```
[template
addControlEvent(cevent:ControlEvent)]
  [if (cevent.eventType.toString() = 'Error')
  or (cevent.eventType.toString() =
  'Cancel')]
  [if (cevent.ocIsKindOf(StartEvent))]
  ### TRANSFORM START EVENT GATEWAY ###
  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD START ACTIVITY 'CE_[cevent.name/]'[/if]

  [if (cevent.ocIsKindOf(EndEvent))]
  ### TRANSFORM END EVENT GATEWAY ###
  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD END ACTIVITY 'CE_[cevent.name/]'[/if]

  [if (cevent.ocIsKindOf(NonBoundaryEvent))]
  ### TRANSFORM NONBOUNDARY EVENT GATEWAY ###
  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD MANUAL ACTIVITY
  'CE_[cevent.name/]'[/if]

  [if (cevent.ocIsKindOf(BoundaryEvent))]
  ### TRANSFORM BOUNDARY EVENT GATEWAY ###
  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD END ACTIVITY 'CE_[cevent.name/]'

  [if not (cevent.outConnections.target-
>isEmpty())]
  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD USER_DEFINED ACTIVITY
  'CT_[cevent.outConnections.target.name/]'

  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD TRANSITION 'C_[cevent.name/]_
[cevent.name/]'
  FROM ACTIVITY 'CT_[cevent.name/]'
  TO 'CT_[cevent.name/]'

  OMBALTER PROCESS_FLOW
  'CP_[cevent.controlProcess/]'
  ADD TRANSITION 'C_[cevent.name/]_
[cevent.outConnections.target.name/]'
  FROM ACTIVITY 'CE_[cevent.name/]'
  TO
  'CT_[cevent.outConnections.target.name/]'
  [/if][/if][/if]
[/template]
```

### Listing 7: Control event transformation.

```
[template addGateway(g:Gateway)]
  [if (g.ocIsKindOf(ParallelMergeGateway))]
  ### TRANSFORM PARALLEL MERGE GATEWAY ###
  OMBALTER PROCESS_FLOW
  'CP_[g.controlProcess.name/]'
  ADD AND ACTIVITY 'G_[g.name/]'[/if]

  [if (g.ocIsKindOf(InclusiveMergeGateway))]
  ### TRANSFORM EXCLUSIVE MERGE GATEWAY ###
```

```

OMBALTER PROCESS_FLOW
'CP_[g.controlProcess.name/]'
ADD OR ACTIVITY 'G_[g.name/]' [/if]

[if (g.ocIsKindOf(ParallelSplitGateway))]
### TRANSFORM PARALLEL SPLIT GATEWAY ###
OMBALTER PROCESS_FLOW
'CP_[g.controlProcess.name/]'
ADD FORK ACTIVITY 'G_[g.name/]' [/if]

[if (g.ocIsKindOf(ExclusiveSplitGateway))]
### TRANSFORM EXCLUSIVE SPLIT GATEWAY ###
OMBALTER PROCESS_FLOW
'CP_[g.controlProcess.name/]'
ADD ROUTE ACTIVITY 'G_[g.name/]' [/if]

[if (g.ocIsKindOf(InclusiveSplitGateway))]
### TRANSFORM INCLUSIVE SPLIT GATEWAY ###
[for (con : Connection | g.outConnections)]
OMBALTER PROCESS_FLOW
'CP_[g.controlProcess.name/]'
ADD TRANSITION 'C_[con.name/]'
FROM ACTIVITY
'[g.inConnections.source.getPrefix()/]_
[g.inConnections.source.name/]'
TO '[con.target.getPrefix()/]_
[con.target.name/]'
OMBALTER PROCESS_FLOW
'CP_[g.controlProcess/]'
MODIFY TRANSITION 'C_[con.name/]'
OF ACTIVITY
'[g.inConnections.source.getPrefix()/]_
[g.inConnections.source.name/]'
SET PROPERTIES (CONDITION)
VALUES ('[con.condition/]') [/for][if]
[/template]

```

**Listing 8: Gateway transformation.**

```

[template addCConnection(c:Connection)]
[if (c.target.ocIsUndefined()) and
not (c.ocIsKindOf(InclusiveSplitGateway))
and not (c.target.ocIsKindOf(
InclusiveSplitGateway))]
### TRANSFORM CONNECTION ###
OMBALTER PROCESS_FLOW
'CP_[c.source.controlProcess.name/]'
ADD TRANSITION
'C_[c.source.name/]_[c.target.name/]'
FROM ACTIVITY
'[c.source.getPrefix()/]_[c.source.name/]'
TO
'[c.target.getPrefix()/]_[c.target.name/]'

[if (c.source.ocIsKindOf(BoundaryEvent))]
### MODIFY TRANSITION CONDITION ###
OMBALTER PROCESS_FLOW
'CP_[c.source.controlProcess/]'
MODIFY TRANSITION
'C_[c.source.name/]_[c.target.name/]'
OF ACTIVITY 'CE_[c.source.name/]'
SET PROPERTIES (CONDITION)
VALUES ('ERROR') [/if][if]
[/template]

```

**Listing 9: Control connection transformation.**

## Data M2T Transformations

```

[template public addCDITask(t:
ColumnDataInput)]
[for (f: Field | t.ocAsType(ColumnDataInput).
selectQuery.fields)]
# Set a connection with required data
resources

[f.fieldset.resource.useConnection()/][for]
[if t.ocAsType(ColumnDataInput).
selectQuery.ocIsUndefined()]
ADD TABLE OPERATOR '[t.fieldSet.name/]'
BOUND TO TABLE '[t.fieldSet.name/]' [/if]

[if not t.ocAsType(ColumnDataInput).
selectQuery.ocIsUndefined()]
ADD VIEW OPERATOR '[t.fieldSet.name/]'
SET PROPERTIES (QUERY)
VALUES '[t.selectQuery.queryToSQL()/]'
[/if]
[/template]

```

**Listing 10: Column data input task transformation.**

```

[template public addMFDTask(t :
MultiFieldDerivation)]
ADD EXPRESSION OPERATOR '[t.name/]'
[t.addConnection()/]
[for (f: Field | t.outputSets.fields)]
ALTER ATTRIBUTE '[f.name/]' OF GROUP
'OUTGRP1' OF OPERATOR '[t.name/]'
SET PROPERTIES (EXPRESSION) VALUES
('[t.computations->at(i).
rValue.computationToSQL()/]')
[/ for]
[/template]

```

**Listing 11: Multi-field derivation task transformation.**

```

[template public addKLTTask(t : KeyLookup)]
ADD LOOKUP OPERATOR '[t.name/]'
SET PROPERTIES (LOOKUP_CONDITION)
VALUES
('[t.lookupCondition.ConditionToSQL()/]')
BOUND TO TABLE '[t.lookupTable/]'
[/template]
[template public addCDITask(t :
ColumnDataInput)]
[for (f : Field |
t.ocAsType(ColumnDataInput).
selectQuery.fields)]

```

**Listing 12: Lookup task transformation.**

```

[template public addFiTask(t :
GlobalConditionFilter)]
ADD FILTER OPERATOR '[t.name/]'
SET PROPERTIES (CONDITION)
VALUES
'[t.filterCondition.ConditionToSQL()/]'
[/template]

```

**Listing 13: Filter task transformation.**

```

[template public addDOTask(t:DataOutput)]
ADD TABLE OPERATOR '[t.name/]'
BOUND TO TABLE '[t.resource/]'
[/template]

```

**Listing 14: Data output task transformation.**

<sup>i</sup> <http://wiki.eclipse.org/Ecore>

## REFERENCES

- Curino, C. A., Moon, H. J., & Zaniolo, C. (2008). Graceful database schema evolution: the prism workbench. *PVLDB*, 1(1), 761–772.
- Cuzzocrea, A., Mazón, J.-N., Trujillo, J.-C. & Zubcoff J.J. (2011). Model-driven data mining engineering: from solution-driven implementations to 'composable' conceptual data mining models. *International Journal of Data Mining, Modelling and Management*, 3(3), 217–251.
- Cuzzocrea A. (2011). A UML-extended Approach for Mining OLAP Data Cubes in Complex Knowledge Discovery Environments. In I. Song & E. Zimányi (Eds.), *Proceedings of the 13th International Conference on Enterprise Information Systems, ICEIS'11* (pp. 281–289). Beijing, China: SciTePress.
- Cuzzocrea A., F. Francesco, & Pontieri L. (2010). Effective Analysis of Flexible Collaboration Processes by Way of Abstraction and Mining Techniques. Filipe J., & Cordeiro J. (Eds.). *Proceedings of the twentieth International Conference on Enterprise Information Systems, ICEIS'11* (pp. 157–166). Funchal, Madeira, Portugal: SciTePress.
- El Akkaoui, Z., & Zimányi, E. (2009). Defining ETL workflows using BPMN and BPEL. In I. Song & E. Zimányi (Eds.), *Proceedings of the 12th ACM International Workshop on Data Warehousing and OLAP, DOLAP'09* (pp. 41–48). Hong Kong, China: ACM Press.
- El Akkaoui, Z., Zimányi, E., Mazón, J.-N., & Trujillo, J.-C. (2011). A model-driven framework for ETL process development. In I. Song, A. Cuzzocrea & K.C. Davis (Eds.), *Proceedings of the 14th ACM International Workshop on Data Warehousing and OLAP, DOLAP'11* (pp. 45–52). Glasgow, Scotland, UK: ACM Press.
- El Akkaoui, Z., Mazón, J.-N., Vaisman, A., & Zimányi, E. (2012). BPMN-based conceptual modeling of ETL processes. In A. Cuzzocrea & U. Dayal (Eds.), *Proceedings of the 14th International Conference on Data Warehousing and Knowledge Discovery, DAWAK'12*. Vienna, Austria: Springer.
- Golfarelli, M., Lechtenbörger, J., Rizzi, S., & Vossen, G. (2006). Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation. *Data Knowl. Eng.* 59(2), 435–459.
- Inmon, W. (2002). Building the Data Warehouse. Wiley.
- Mazón, J.-N., & Trujillo, J.-C. (2008). An MDA approach for the development of data warehouses. *Decision Support Systems*, 45(1), 41–58.
- Muñoz, L., Mazón, J.-N., & Trujillo, J.-C. (2009). Automatic generation of ETL processes from conceptual models. In I. Song & E. Zimányi (Eds.), *Proceedings of the 12th ACM International Workshop on Data Warehousing and OLAP, DOLAP'09* (pp. 33–40). Hong Kong, China: ACM Press.
- Muñoz, L., Mazón, J.-N., & Trujillo, J.-C. (2010). A family of experiments to validate measures for UML activity diagrams of ETL processes in data warehouses. *Information & Software Technology*, 52(11), 1188–1203.
- Papastefanatos, G., Vassiliadis, P., Simitsis, A., & Vassiliou, Y. (2008). Design Metrics for Data Warehouse Evolution. In Q. Li, S. Spaccapietra, E. Yu, & A. Olivé, (Eds.), *Proceedings of the 27th International Conference on Conceptual Modeling, ER '08*, (pp. 440-454). Berlin, Heidelberg: Springer.
- Papastefanatos, G., Vassiliadis, P., Simitsis, A., & Vassiliou, Y. (2009). Policy-regulated management of ETL evolution. *In Journal on Data Semantics XIII*, 146–176.
- Romero, O., Simitsis, A., & Abelló, A. (2011). GEM: Requirement-driven Generation of ETL and Multidimensional

Conceptual Designs. In A. Cuzzocrea & U. Dayal (Eds.), *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery DaWaK '11*, (pp. 80–95). Toulouse, France: Springer.

Simitsis, A., & Vassiliadis, P. (2008). A method for the mapping of conceptual designs to logical blueprints for ETL processes. *Decision Support Systems*, 45(1), 22–40.

Skoutas, D., & Simitsis, A. (2009). Ontology-driven conceptual design of ETL processes using graph transformations. In *Journal on Data Semantics XIII*, 122–149.

Thomsen, C., & Pedersen, T. B. (2011). Easy and effective parallel programmable ETL. In I. Song, A. Cuzzocrea & K.C. Davis (Eds.), *Proceedings of the 14th ACM International Workshop on Data Warehousing and OLAP, DOLAP'11* (pp. 37–44). Glasgow, Scotland, UK: ACM Press.

Trujillo, J.-C., & Luján-Mora, S. (2003). A UML based approach for modeling ETL processes in data warehouses. In I.Y. Song, S.W. Liddle, T.W. Ling & P. Scheuermann (Eds.), *Proceedings of the 22nd International Conference on Conceptual Modeling, ER'03* (pp. 307–320). Chicago, IL, USA: Springer.

Tziovara, V., Vassiliadis, P., & Simitsis, A. (2007). Deciding the physical implementation of ETL workflows. In I. Song & T. Pedersen (Eds.), *Proceedings of the 10th ACM International Workshop on Data Warehousing and OLAP, DOLAP'07* (pp. 49–56). Lisbon, Portugal: ACM Press.

Vassiliadis, P., Simitsis, A., Georgantas, P., Terrovitis, M., & Skiadopoulos, S. (2005). A generic and customizable framework for the design of ETL scenarios. *Information Systems*, 30(7), 492–525.

Vassiliadis, P., Simitsis, A., & Baikous, E. (2009). A taxonomy of ETL activities. In I. Song & E. Zimányi (Eds.), *Proceedings of the 12th ACM International Workshop on*

*Data Warehousing and OLAP, DOLAP'09* (pp. 25–32). Hong Kong, China: ACM Press.

Wilkinson, K., Simitsis, A., Castellanos, M., & Dayal, U. (2010). Leveraging Business Process Models for ETL Design. . In J. Parsons, M. Saeki, P. Shoval, C. Woo & Y. Wand (Eds.), *Proceedings of the 29th International Conference on Conceptual Modeling, ER'10* (pp. 15–30). Vancouver, BC, Canada: Springer.

Wieringa, R. (2010). Design science methodology: principles and practice. In J. Kramer, J. Bishop, P. T. Devanbu & S. Uchitel (Eds.), *Proceedings of the 32nd International Conference on Software Engineering ICSE'10* (pp. 493–494). Cape Town, South Africa: ACM Press.

Wyatt, L., Caufield, B., & Pol, D. (2009). Principles for an ETL benchmark. In R. Nambiar & M. Poess (Eds.), *Proceedings of the First TPC Technology Conference, TPCTC 2009* (pp. 183–198). Lyon, France: Springer.