Inf Syst E-Bus Manage (2017) 15:425–460
DOI 10.1007/s10257-016-0325-8

ORIGINAL ARTICLE

Automating ETL processes using the domain-specific modeling approach

Marko Petrović[1] · Milica Vučković[1] · Nina Turajlić[1] · Slađan Babarogić[1] · Nenad Aničić[1] · Zoran Marjanović[1]

Received: 15 June 2015 / Revised: 1 May 2016 / Accepted: 30 June 2016 /
Published online: 9 July 2016

Abstract The development of Extract–Transform–Load (ETL) processes is the most complex, time-consuming and expensive phase of data warehouse development. Yet, the dynamics of modern business systems demand a more agile and flexible approach to their development. As a result, current research in this area is focused on ETL process conceptualization and the automation of ETL process development. This paper proposes a novel solution for automating ETL processes using the domain-specific modeling (DSM) approach. The proposed solution is based on the formal specification of ETL processes and the implementation of such formal specifications. Thus, in accordance with the DSM approach, several new domain-specific languages (DSLs) are introduced, each defining concepts relevant for a specific aspect of an ETL process. The focus of this paper is the actual implementation of the formal specification of an ETL process. To this end, a specific ETL platform (ETL-PL) is introduced to technologically support both the modeling of ETL processes (i.e., the creation of models in accordance with the introduced DSLs) and the automated transformation of the created models into the

✉ Marko Petrović
petrovic.marko@fon.bg.ac.rs

Milica Vučković
vuckovic.milica@fon.bg.ac.rs

Nina Turajlić
turajlic.nina@fon.bg.ac.rs

Slađan Babarogić
babarogic.sladjan@fon.bg.ac.rs

Nenad Aničić
anicic.nenad@fon.bg.ac.rs

Zoran Marjanović
marjanovic.zoran@fon.bg.ac.rs

[1] Faculty of Organizational Sciences, University of Belgrade, Belgrade, Serbia

executable code of a specific application framework (representing ETL-PL's execution environment). It should be emphasized that ETL-PL actually presumes the dynamic execution of ETL models or, more precisely, the executable code is generated at runtime. Thus the execution environment consists of code generator components and the components implementing the application framework. ETL-PL has been implemented as an extension of the .NET platform.

Keywords Extract–transform–load (ETL) · Model-driven development (MDD) · Domain-specific modeling (DSM) · Domain-specific languages (DSL) · Runtime models · Execution platform

1 Introduction

Data warehouse systems, as a specific type of information system, should enable the acquisition of business data, its transformation into appropriate strategic business information and the subsequent storage of the transformed data in a format that facilitates business analysis. These systems support the decision-making process i.e., their aim is to support the decision-makers in making better and faster decisions. They are expected to have the right information in the right place at the right time with the right cost in order to support the right decision (Jarke et al. 2003).

Research and practice in the field of data warehousing have already significantly promoted the understanding of this domain and have led to considerable progress being made with regard to the formalization and automation of data warehouse development. Both the static and dynamic aspects of data warehouses have been studied. In addition, research and practice have also focused on one of the most demanding phases in the data warehouse development process—the development of the process for the acquisition and integration of business data, its transformation into appropriate strategic business information and the subsequent storage of the transformed data in a format that facilitates business analysis (the extract–transform–load process—ETL).

The development of appropriate ETL processes (which adequately fulfill their purpose) requires overcoming several challenges as discussed in (Turajlić et al. 2014). First, it is necessary to integrate the available business data coming from diverse data sources which are usually very heterogeneous in nature (i.e., they may be based on different technologies, use various data models etc.). In order to resolve the numerous structural and semantic conflicts that may exist, a wide array of transformations must be performed. Furthermore, such transformed data must then be translated into a form suitable for its further analysis. On the other hand, the sheer volume of data that is to be gathered, processed, transformed, stored and delivered, imposes strict constraints not only regarding the way the data must be structured but also with regard to the requirements related to the performance and scalability of data warehouse systems. Finally, these processes must be designed to be flexible so that they are able to respond to the constant changes not only in the state and structure of existing data sources (while at the same time allowing for the

inclusion of new data sources) but also to the changes in business requirements (imposed by the dynamic business environment). A change in business requirements calls for new business analysis to be conducted, which in turn means that new strategic information must be provided. In other words, according to (El Akkaoui et al. 2012) agile and flexible ETL tools are needed which can quickly produce and modify executable code based on constantly changing needs of the dynamic business environment.

Taking this into consideration, it could be said that the manner in which these processes are designed and implemented significantly impacts the quality of the obtained information, and consequently the usability and success of the system as a whole. Moreover, it has been estimated that as much as 70 % of the time and effort invested in the development of data warehouses is spent on the development of ETL processes (Kimball and Caserta 2004; Kimball et al. 2010). Therefore, it is evident that an appropriate methodological approach to ETL process development must be adopted.

The methodological approaches, developed during the past couple of decades, are aimed at resolving some of the problems inherent to this process such as: high development and maintenance expenses, low productivity, failure to adequately satisfy user requirements, etc. As previously stated, these problems stem from the complexity of modern business systems, frequent changes in the organizational and technological environment and the emerging need for businesses to adapt to these changes, especially in light of the of the pervasiveness of the Internet and the transition to e-business. Consequently, the main premise of this paper is that ETL process development must be based on abstractions as they are the only valid methodological means for overcoming complexity. Moreover, it is argued that semantically richer abstractions are desired because they can encapsulate greater knowledge thereby increasing productivity and efficiency (Greenfield et al. 2004; Kelly and Tolvanen 2008). Furthermore, since a greater level of automation is sought, it is necessary to formalize the existing knowledge and experience in such a form that would allow for its reuse (Greenfield et al. 2004; Kelly and Tolvanen 2008). The possibility of reuse additionally increases productivity and efficiency, while at the same time lowering the cost of data warehouse system development. By elevating the semantic level and supporting it technologically development can be significantly automated and fewer steps will be needed to implement the abstract specifications.

In this paper a novel approach to ETL process development is proposed, and the aim of the proposed solution is to automate the development to a significant extent.

Hence, the proposed solution was developed in accordance with the leading approach to software development today—Model-driven development (MDD). The main goal of MDD is to enable the automation of software development in order to increase development productivity, reduce development time and cost, and improve the quality and flexibility of the obtained solution. To this end it promotes the use of abstractions which enable the analysis of a problem at different levels of detail. MDD is based on the premise that the most important product of software development is not the source code itself but rather the models representing the knowledge about the system that is being developed. In other words, in MDD,

models are primary software artifacts and the development process is automated through appropriate model transformations, which should ultimately result in a concrete implementation i.e., executable code. In light of the complexity of ETL processes and the problems related to their development it can be stipulated that they should be developed in accordance with the MDD approach.

More specifically, in this paper the Domain-Specific Modeling (DSM) approach has been chosen for the development of ETL process. Given that DSM introduces models as primary software artifacts it promotes the use of abstractions. Furthermore, in the DSM approach, software development can be fully automated through the application of model transformations (Kelly and Tolvanen 2008). In order to enable such automatic transformations the models must be formal. Thus, the DSM approach has been adopted not only because it allows for the formalization of semantically rich abstractions, in a form which can be reused, but also because it enables the generation of executable code from models representing the specification of the system.

The proposed solution, for the development of ETL processes in the context of a DSM approach, is based on the formal specification of ETL processes and the implementation of such formal specifications. Several new domain-specific languages (DSLs) are introduced, for the formal specification (i.e., modeling) of ETL processes, which define concepts relevant for different aspects of this particular domain. In order to reduce the complexity of the ETL process specification the different aspects of an ETL process would be modeled separately using the appropriate DSLs.

However, the focus of this paper is the actual implementation of the formal specification of an ETL process. To this end, a specific ETL platform (ETL-PL) has been developed to technologically support both the specification (i.e., creation of models using the graphical or textual notation of the introduced DSLs) and the automated transformation of these models into the executable code of a specific application framework (representing the execution environment of an ETL process). The application framework defines specific implementation concepts which are close to the real domain concepts of the DSLs introduced for the specification of ETL processes. By defining implementation concepts which are close to the real domain concepts the semantic level of the solution is significantly elevated. Furthermore, if both the specification and the application framework use concepts close to the real ETL domain concepts the transformation between them can be fully automated, thus significantly increasing development productivity and efficiency while lowering the development and maintenance costs. In other words, by elevating the semantic level, and supporting it technologically, development can be significantly automated and fewer steps will be needed to implement the abstract specifications. Moreover, the obtained solutions would have good performances and be scalable and maintainable yet, at the same time, flexible (i.e., they could be easily extended to adapt to the constant changes in the environment and new requirements).

The proposed software architecture for implementing ETL-PL is presented, specifying the main components of ETL-PL. The components of ETL-PL are divided into two layers: the development environment and the execution

environment. The development environment is comprised of tools which support the modeling of ETL processes. More specifically, it contains tools for defining the abstract and concrete syntax (in both a graphical and textual notation) of a DSL and tools (syntax editors, graphical and textual) for creating models in accordance with the defined DSL. The execution environment is responsible for the automatic generation of executable code from the models as well as the actual execution of the generated code. Thus the execution environment consists of code generator components and the components implementing the application framework.

The paper is structured as follows: Sect. 2 gives an analysis of the related in work on ETL process development. Section 3 gives a brief overview of the conceptual framework for the formal specification of ETL processes. Section 4 explains the motivation behind the proposed implementation of ETL process specifications. The proposed ETL platform is outlined in Sect. 5, while the implementation details are presented in Sect. 6. Finally, Sect. 7 concludes the paper and discusses future work.

2 Related work

There is a growing need for the formalization and automation of ETL process development, due to the fact that it is extremely complex and time-consuming and that it requires significant financial resources, and a good deal of research effort has thus far been dedicated to this issue. A detailed analysis of these approaches can be found in a previous paper (Turajlić et al. 2014), and only a brief discussion will be presented here in order to justify the proposed approach.

It should first be emphasized that only a few approaches exist which enable the automated development of ETL processes in the context of MDD. In order to enable automated development, MDD requires that the models be formally expressed. Thus far, two distinctive approaches have emerged for realizing MDD in general, which differ primarily in the languages used for the specification of the models. One advocates the use of general purpose modeling languages (GPMLs) and their extension, while the other advocates the use of specially designed domain-specific languages (DSLs). The existing body of research on ETL process development could be classified along the same lines.

The modeling of ETL processes using existing general purpose modeling languages (such as Unified Modeling Language—UML or Business Process Model and Notation—BPMN), which have been extended to incorporate the concepts specific to the ETL process domain, has been proposed in (Trujillo and Luján-Mora 2003; Luján-Mora et al. 2004; Muñoz et al. 2008, 2009; El Akkaoui and Zimányi 2009; El Akkaoui et al. 2011, 2012).At the same time, the use of DSLs which are tailored to a particular domain has also been proposed in (Vassiliadis et al. 2002a, b, 2003, 2005; Simitsis and Vassiliadis 2003, 2008; Simitsis et al. 2005; Simitsis 2005).

It can be argued that, since GPMLs were envisaged to support the description of the various aspects of any given business process in any given domain (in order to promote standardization), they include a large number of domain-neutral concepts which are defined at a low level of abstraction. According to (Kelly and Tolvanen

2008) GPMLs do not raise the level of abstraction above code concepts. According to the same authors, the main benefit of DSLs is that, unlike GPMLs, they raise the level of abstraction beyond current programming languages and their abstractions, by specifying the solution in a language that directly uses the concepts and rules from a particular problem domain. Furthermore, the complexity of GPMLs (i.e., too many concepts whose semantics are imprecise) along with the fact that they are often too technical for domain-experts to master, lead to a number of issues related to the acceptance, utilization and value of these languages. Moreover, to extend these languages it is necessary to be familiar with their concepts in order to be able to identify those which can be specialized. Finally, it is up to the designer to know the semantic rules (e.g., the legal connections and structures, the necessary data etc.) and ensure that they are fulfilled when defining the specification. On the other hand, the aim of DSLs is to provide only a minimal set of domain-specific concepts, with clear and precise semantics, along with a set of strict rules controlling their usage and the way in which they can be composed. Since DSLs allow for the inclusion of domain rules (in the form of constraints) both the syntax and the semantics of the concepts can be controlled, thus incorrect or incomplete designs can be prevented by making them impossible to specify.

Therefore, in comparison with GPMLs, DSLs are more expressive (i.e., they enable a precise and unambiguous specification of the problem) while at the same time being more understandable and easier to use by domain experts (since they do not include unnecessary general purpose concepts). In addition, the use of such languages facilitates communication among the various stakeholders (from both the business as well as the technical communities) thereby promoting teamwork which is one of the main principles of current agile approaches to software development.

As a final point it should be noted that some of the approaches do not provide explicit concepts which allow for the formal definition of the semantics of the data transformations. For example, in (Vassiliadis et al. 2002a, b; Simitsis and Vassiliadis 2003; Luján-Mora and Trujillo 2004) notes or annotations are used for the explanation of the semantics of the transformations (e.g., type, expression, conditions, constrains etc.), while in (Trujillo and Luján-Mora 2003) even the actual attribute mappings are defined through notes. Since in these approaches the authors allow for the notes to be given in a natural language (and often without any restrictions on their content) they do not represent a formal specification.

However, in order to enable automated development, it is necessary to provide the means for formally specifying the data transformation semantics, and the approaches proposed in (Muñoz et al. 2008; El Akkaoui and Zimányi 2009; El Akkaoui et al. 2011, 2012) have, to some extent, managed to address this issue.

The way in which the actual automation of software development is achieved (Model-driven architecture—MDA or Domain-Specific Modeling—DSM) is another point of difference between the general purpose approach and the domain-specific approach. Generally, in order to enable the automation of the development in accordance with MDD, it is necessary to first map the domain concepts to design concepts and then on to programming language concepts.

In the MDA approach, software development can be partially or fully automated through the successive application of model transformations, starting from the

model representing the specification of the system (i.e., the conceptual model) and ending in a model representing the detailed description of the physical realization, from which the executable code can ultimately be generated. The development of ETL processes in accordance with the MDA approach is proposed in (Muñoz et al. 2008; Mazón and Trujillo 2008). Thus, the conceptual models are defined as platform independent models—PIM which are then automatically transformed into platform specific models—PSM (through a set of formally defined transformations) from which the code (necessary to create data structures for the ETL process in the corresponding platform) can be derived. However, since the PSMs must be specially designed for a certain technology of ETL processes (i.e., each PSM must be based on the resources of a specific technology) the proposed approach presumes that a metamodel must be manually defined for each specific tool in order to create the transformations from the proposed conceptual model to each deployment platform. Furthermore, the MDA approach in general, is based on the refinement of models through successive model transformations, yet this process usually also requires that the automatically generated models be manually extended with additional details. These manual extensions could lead to a discrepancy between the original and generated models (i.e., the original models would become obsolete). This discrepancy is further emphasized when the modification of models, previously created by partial generation, is required. Since the correct modification of these models remains an unresolved issue, MDA advocates using a single GPML, namely UML, at all the levels (thereby lowering the abstraction levels of models) which not only entails all of the previously discussed issues regarding the use of GPMLs for modeling ETL processes, but also brings additional complexity to the development of model transformations (Fowler 2010). Thus, an improvement of the proposed approach has been suggested in (El Akkaoui et al. 2011) to directly obtain the code corresponding to the target platform, bypassing the need for the defining of an intermediate representation (metamodel) of the target tool. The conceptual model can then be automatically transformed into the required vendor-specific code to execute the ETL process on a concrete platform.

Conversely, in the DSM approach the implementation is automatically generated from the specification (which can be modeled using domain-specific concepts) by code generators which specify how the information is extracted from the models and transformed into code. In other words, the generator reads the model based on the metamodel of the language and maps it to code. The generators are also domain-specific (i.e., they produce the code according to the solution domain) since, according to (Kelly and Tolvanen 2008), this is the only way to enable full code generation i.e., the generation of code that does not need to be additionally modified. Usually the code generation is further supported by a domain-specific framework which provides implementation concepts, closer to the domain concepts used in the specification, thus narrowing the gap between the solution domain and the problem domain that would otherwise need to be handled by the code generator. The main benefit of DSM according to (Kelly and Tolvanen 2008) is that generators, along with framework code, provide an automated direct mapping to a lower abstraction level (i.e., there is no need to make error-prone mappings from domain concepts to design concepts and on to programming language concepts)

thus providing full code generation instead of resulting in a partial implementation. Because the generated code can be compiled to a finished executable without any additional manual effort, the specification (i.e., model) in fact becomes truly executable.

In summary, it can be concluded that, if the goal is to formalize and automate the development of ETL processes to a significant extent, the DSM approach should be adopted not only, because it allows for the formalization of semantically rich abstractions in a form which can be reused, but also because it enables the automatic generation of executable code from models representing the specification of the system. More precisely put, since DSLs allow for the formalization of semantically rich abstractions (which capture the existing knowledge and experience in the ETL domain) they are more appropriate for the formal specification of ETL processes.

On the other hand, the modeling concepts of GPMLs do not relate to any specific problem domain on the modeling side while on the implementation side, they do not relate to any particular software platform, framework, or component library. However, it is argued that the application framework (supporting the implementation of the ETL process specification in the DSM approach) should define specific implementation concepts which are more close to the real domain concepts introduced in the DSLs used for the specification of ETL processes. If both the specification and the application framework use formal concepts close to the real ETL domain concepts the transformation between them can be fully automated. Furthermore, MDA assumes the existence of several models at different levels of abstraction obtained through progressive refinement (which can be both automatic and manual) thus automation is usually only partially achieved. An additional benefit of the DSM approach is that, both the models and the code generators, can be easily changed (and the code then only needs to be regenerated) which makes the development process more agile. As a final point, according to (Kelly and Tolvanen 2008) domain-specific approaches are reported to be on average 300–1000 % more productive than GPMLs or manual coding practices.

The approach, closest to fulfilling all of the posed requirements, is proposed in (El Akkaoui and Zimányi 2009; El Akkaoui et al. 2011, 2012) in which the authors have even provided built-in mechanisms to validate the syntactic and semantic correctness of the created models. However it is based on the use of a single modeling language which is built by extending a general purpose modeling language, namely BPMN.

Finally, it should be noted, that there is a large number of commercial ETL tools (e.g., Microsoft SQL Server Integration Services, Oracle Data Warehouse Builder, Pentaho Data Integration, etc.) offered by both ETL vendors and Database vendors, and the main rationale for purchasing such a tool is to minimize the development and deployment time and, consequently, development costs. Yet one of the major drawbacks of vendor ETL tools is their extremely high price. Even when it is possible to buy only some of the necessary components of a tool at a smaller price, or even obtain the tool for free (usually from Database vendors as part of a DBMS license) it, almost always, entails some further expenses such as: extra charges for running the tool on a different platform or even on additional CPUs; buying other necessary components; support and maintenance costs; additional fees for providing

training, documentation, etc. Furthermore, it takes a significant amount of time for developers to become proficient with the acquired tool even when they are skilled programmers. As stated in (El Akkaoui and Zimányi 2009) each one of these tools provides its own language which often involves implementation level considerations hence they are difficult to understand, optimize, and maintain. Another crucial limitation of most vendor ETL tools is that they do not offer adequate support for complex custom transformations. Since the proprietary source code is rarely available modifications and extensions cannot be easily made. Even if the code is made available, its modification or extension requires significant technical knowledge. Moreover, it is also necessary to ensure that the alterations do not affect the existing functionality. In order to be competitive the vendors constantly improve their tools, and new versions of the tools are deployed. However, this can also present a liability, since it is a question whether the previously implemented functionality will be compatible with the new version. Furthermore, vendors can cease to provide support for previous versions of a tool, or even go out of business.

3 Conceptual ETL framework

One of the most important and demanding phases in the data warehouse development process is the development of the process for transforming the business data into strategic information i.e., the ETL process. An ETL process is comprised of a number of activities which are to be executed in a particular order with the aim of transforming business data into strategic information. The activities in this process represent the actual data operations (i.e., the data flow), while the control flow represents the execution order of these activities.

In this paper the DSM approach has been adopted for the development of ETL process because the aim of the proposed solution is to automate the development to a significant extent and DSM, as discussed in previous section, enables the generation of executable code from models representing the specification of the system. Furthermore, the main premise is that ETL development must be based on semantically rich abstractions which encapsulate the existing knowledge and experience in the ETL domain and DSM allows for the formalization of such abstractions in a form which can be reused.

The first phase in ETL process development is the specification (i.e., modeling) of ETL processes. The main goal of this phase is to define "what" the software solution should provide in terms of its basic functionality. Several points where taken into consideration regarding the manner in which the specification should be given. Firstly, since domain experts play a key role in this phase (as they possess an in-depth understanding of the domain i.e., the semantics of the data that is to be transformed) the models should be expressed in terms of concepts specific to the particular domain (i.e., the concepts and terms used by the domain experts). Secondly, the modeling languages should be as simple as possible (i.e., they should provide a minimal set of necessary concepts) but at the same time semantically rich to enable the specification of the various aspects of the problem domain at the appropriate level of abstraction. Thirdly, they should be formal in order to enable

automatic model transformations. Thus, for the formal specification of ETL processes, the use of DSLs is preferred over the extension of GPMLs, since they provide only a minimal set of semantically rich domain-specific concepts which makes them more approachable to domain experts.

Furthermore, taking into account the complexity of ETL processes, it is obvious that the various aspects of an ETL process (e.g., the control flow, the data flow, etc.) should be modeled separately else the specification would lead to an overly complex, convoluted model, in which all of the various aspects of an ETL process are interwoven. However, it can be argued that the use of a single modeling language (be it an extended GPML or a DSL) would not be conducive since it would include a vast amount of disparate concepts. It is therefore stipulated that each aspect of an ETL process should be modeled by a separate language, which should include only the concepts which are relevant for that particular aspect, thereby keeping the languages straightforward and easy to use. These languages would then constitute a conceptual framework for ETL process specification.

The introduced conceptual framework for ETL process specification defines several novel DSLs for the specification of the different aspects of ETL processes. Primarily, a language for the specification of data operations, i.e., the data flow, (ETL-O) and a language for specification of their execution order, i.e., the control flow, (ETL-P). Three supplementary languages are also provided: a language for the specification of various logical and arithmetic expressions (ETL-E), a language for the specification of transformation operation templates (ETL-T) and a language for the specification of source and target data models (ETL-D). The introduced DSLs define concepts which are relevant for the respective aspects of the ETL processes and they fulfill all of the previously stated requirements regarding modeling languages. Moreover, they are developed as new languages rather than as extensions of generic modeling languages (such as BPMN and UML) which, in light of the discussion in the previous section, makes them extremely appropriate for solving problems in the given domain.

The conceptual framework defining the fundamental concepts of ETL processes and their relationships is given in Fig. 1.

It should be emphasized that the metamodels of the introduced DSLs are described using the concepts of a meta-language defined by *Microsoft* (as part of *DSL Tools*) which has thus, as such, been incorporated into the implementation of the proposed ETL platform. Therefore, the validity of the DSL metamodels is verified against this meta-language. On the other hand, the DSL metamodels, describing concrete models, are used for validating the concrete models.

The *ETL Base* package introduces the set of abstract concepts which enable the integration of the various ETL modeling languages. The proposed ETL DSLs are then defined as extensions of this abstract metamodel. In other words, each of the proposed DSLs introduces its own concepts (the concepts relevant for the particular aspect which is to be modeled) by extending the base concepts. The metamodel of the base classes for all of the proposed DSLs is given in Fig. 2. The abstract concepts of the this metamodel are used for specifying the ETL DSL models (the *Model* concept) and their elements (the *ModelElement* concept), as well as for referencing other models (the *ModelRef* concept) and the elements of those models

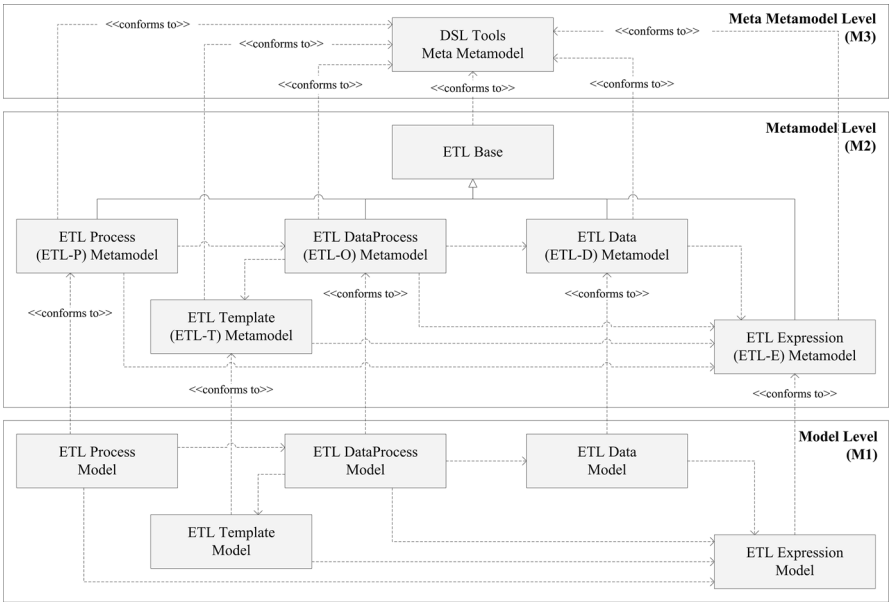

Fig. 1 Conceptual ETL framework

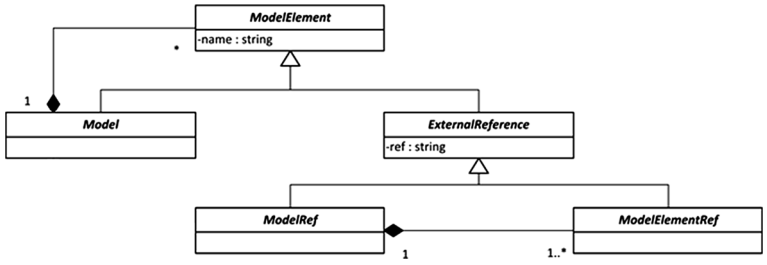

Fig. 2 ETL Base metamodel

(the *ModelElementRef* concept) in order to track the dependencies (relationships) among various models, wherein the definition of a particular model uses certain definitions given in another model.

An *ETL process* is comprised of a number of activities (*Data processes*) which are to be executed in a particular order with the aim of transforming business data into strategic information. The activities in this process represent the actual data operations (which are specified using ETL-O), while the execution order of these activities is specified using ETL-P. In other words, ETL-P defines the fundamental concepts necessary for specifying the execution semantics of an ETL process (i.e., the control flow), while ETL-O defines the fundamental concepts necessary for the specification of ETL process activities, or more precisely, for defining the semantics of the data operations and the order in which they are to be executed (i.e., the data

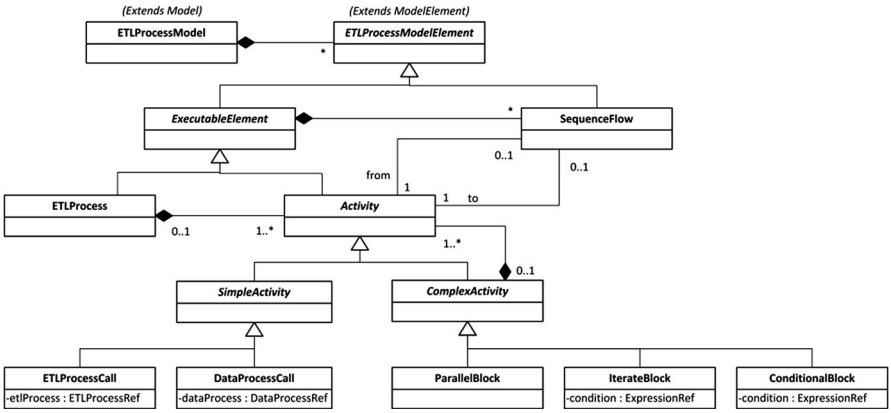

Fig. 3 ETL-P metamodel (abridged version)

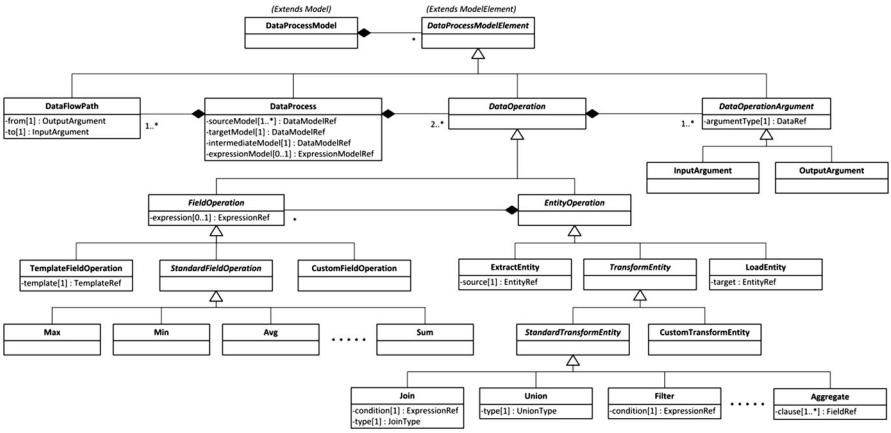

Fig. 4 ETL-O metamodel (abridged version)

flows of an ETL process). The main concepts of the ETL-P and ETL-O metamodels are given in Figs. 3 and 4, respectively.

Thus, the control flow of an ETL process (*ETLProcess*) defined using ETL-P, is comprised of a number of actual tasks (*Activities*). The activities can be *SimpleActivities*, such as *ETLProcessCall* and *DataProcessCall* which enable the invocation of other ETL processes (defined in separate ETL-P models and referenced via *ETLProcessRef*) and data processes (defined in separate ETL-O models and referenced via *DataProcessRef*), respectively, or *ComplexActivities* composed of other activities (simple and/or complex). For specifying the execution order of activities, three concepts are provided representing the basic control structures (i.e., *SequenceFlow*, *ConditionalBlock* and *IterateBlock*). An additional concept, the *ParallelBlock*, representing the concurrent execution of activities, is introduced to provide the possibility for optimizing the process execution. Through

the hierarchical composition of these concepts the execution semantics of the overall ETL process can be represented. The expressions for specifying the conditions and constraints, pertaining to the execution of an ETL process, are given in the form of ETL-E expressions (defined in a separate ETL-E model and referenced via *ExpressionRef*).

The data flow of a single ETL process activity (i.e., the *DataProcess*), defined using ETL-O concepts, consists of *DataOperations,* representing the atomic data operations, with *DataFlowPaths* composing these operations into a single data flow. In other words, the flow of data between the operations, i.e., the execution order of the atomic operations, is represented by *DataFlowPaths* (whereby the *OutputArgument* of one operation is related to an *InputArgument* of the following operation). Since the data involved in the transformations is represented, in separate ETL-D models, by entities which are comprised of fields (referenced via *EntityRef* or *FieldRef*), the transformation of entities also requires the transformation of the corresponding fields i.e., each entity transformation (*EntityOperation*) includes one or more field transformations (*FieldOperations*). The semantics of the actual transformation operations are another crucial element of the specification. Therefore, transformation operations at the entity level are further refined in order to introduce more specific concepts i.e., the operations pertaining to the ETL process domain such as (*ExtractEntity*, *TransformEntity* and *LoadEntity*), while the semantics of the transformation operations at the field level are expressed using ETL-E. Furthermore, in order to enable a more precise specification of the semantics of the actual transformations, specific concepts for representing both standard/common (e.g., *Join*, *Union*, *Filter*, *Aggregate,* etc., at the entity level, or *Max*, *Min*, *Avg*, *Sum,* etc., at the field level, in accordance with the taxonomy given in (Vassiliadis et al. 2009)) and custom transformation operations are introduced. For each of these operations the conditions and constraints are specified using ETL-E expressions. In addition *TemplateFieldOperations* provide the necessary support for using transformation templates defined in separate ETL-T models.

ETL-T enables the creation of new field transformation operations in the form of *Templates* which can be reused (via a *TemplateRef*). For example, the *ExtractFirstName* template could be defined, for which two *TemplateArguments* would be specified (the full name as the *InputTemplateArgument* and the first name as the *OutputTemplateArgument*), while the actual operation semantics are given in the form of an ETL-E expression.

ETL-E supports the formal specification (in a textual notation) of the semantics of the data operations, as well as the various conditions and restrictions pertaining to the execution of an ETL process. To this end, it provides a number of expression types for: primitive expressions (representing variable declarations, numeric and textual constants, method invocations, etc.), methods (which are defined using ETL-E and referenced by the concepts of the other ETL DSLs), statements (corresponding to the basic control structures: sequence, iteration and condition) and operators (for building complex expressions, where the operands can be primitive and/or complex expressions). It should be noted that the evaluation result of every expression must be of a certain data type. A defined ETL-E expression will actually be referenced in other ETL models via an *ExpressionRef*.

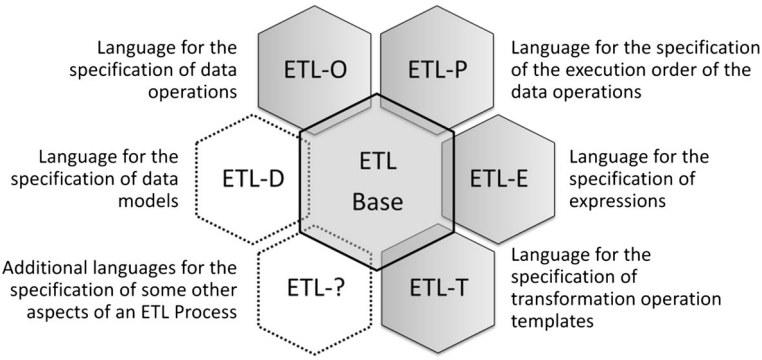

Fig. 5 ETL domain languages

Finally, ETL-D provides a uniform representation of the various data models involved in the transformations (either as their inputs or the results of their execution). In order to reconcile the heterogeneity of the different data sources, the various local schemata (describing these heterogeneous data sources) must be transformed into equivalent schemata which are uniformly described using the concepts of a consolidated model.

It should be emphasized that the problem of the design of the consolidated model, in order to reconcile the heterogeneity of the different data sources and represent their concepts in a uniform manner, is outside the scope of this paper, since it is handled in the analysis phase of the data warehouse development process.

In the remainder of this paper a simplified ETL-D metamodel is utilized (containing the *Entity*, *Field* and *Relationship* concepts) and it is assumed that all of the concrete data models conform to this metamodel. The defined ETL-D model and its elements will be referenced in other ETL models via *DataModelRef* and *EntityRef* or *FieldRef*, respectively.

The introduced ETL DSLs will not be further elaborated, since the focus of this paper is on the automated implementation of ETL processes. However, a brief description of their fundamental concepts was included in order to give a comprehensive overview of the proposed ETL platform.

As a final point, it should be noted that, though the presented ETL DSLs do not constitute a complete set of languages necessary for the specification of every aspect of the ETL process domain, this set could be easily extended to include new languages (Fig. 5). Furthermore, the languages themselves were also envisaged to be easily extensible through specialization.

3.1 ETL process modeling

The proposed approach to ETL process modeling involves the creation of a separate model for each of the different aspects of an ETL process. By separating the different aspects into different models the complexity of an ETL process model is significantly reduced. Each model is created using the concepts of the appropriate

DSL (e.g., ETL-O for the specification of data operations—ETL process activities, ETL-P for the specification of the execution order of these activities etc.). Since these models are created independently the actual order of their creation is not predetermined. Thus it is left to the designers to decide (in accordance with their experience, knowledge of the system that is being developed and preferences) which aspect should be modeled first.

However, the models formed in accordance with the proposed ETL-O specification may still be very complex, depending on the complexity of the ETL process that is being modeled, i.e., the number of actual transformations it requires. In order to facilitate the creation of such models it is proposed that it should be done gradually, at different levels of abstraction, with each subsequent level progressively refining the previous i.e., giving a more detailed description of the given ETL process. The hierarchical description of a data process is accomplished by introducing a set of diagrams. Consequently three types of diagrams are introduced: data process diagrams, complex transformation operation diagrams and simple transformation operation diagrams.

Therefore, the proposed approach to the modeling of an ETL process (Fig. 6) entails the creation of these diagrams (which are specified using ETL-O concepts) along with an additional diagram (ETL Process execution diagram) depicting the execution of the ETL process (specified using ETL-P concepts). The concrete syntax of the proposed ETL-O and ETL-P DSLs provides the necessary graphical elements (representing the concepts of these languages) for constructing these diagrams. In addition, a complete specification of an ETL process also requires the creation of the data models (both source and target) using the ETL-D DSL, which also provides appropriate graphical elements, as well as the specification of the necessary ETL expressions which are expressed in textual notation using the ETL-E DSL.

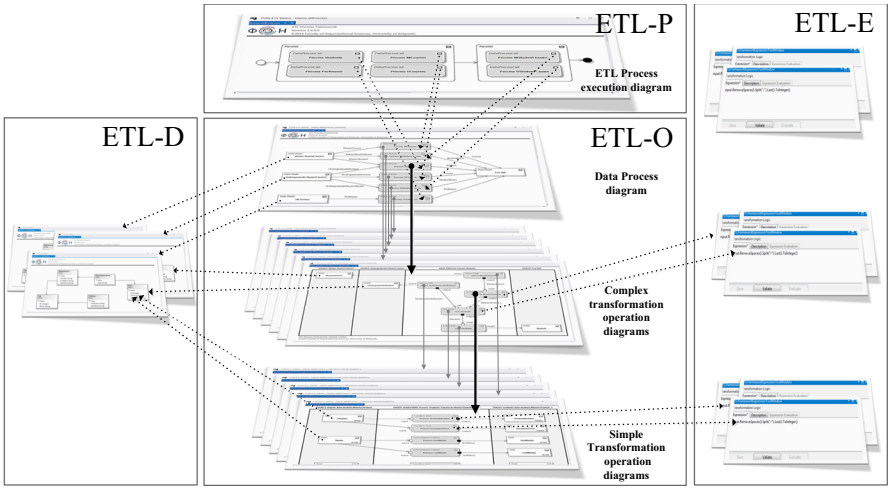

Fig. 6 Hierarchical specification of an ETL process

In the following subsection an example illustrating the modeling of a concrete ETL process through these diagrams is given.

3.2 An illustrative example

This section illustrates the creation of the models in accordance with the introduced ETL DSLs. A simplified example is given for the development of an ETL process in the context of a data warehouse for the Faculty of Organizational Sciences which integrates data coming from different departments of the faculty. As previously explained, the specification of the ETL process involves the creation of a number of diagrams. For each type of diagram a representative example will be given. The diagrams have been additionally annotated in this paper in order to clarify the correspondence between the graphical elements and the concepts of the metamodels given in Figs. 3 and 4.

The specification of the process commences with the creation of the relevant data process diagrams. In Fig. 7 an example of a data process diagram is given in which several data processes have been identified.

ProcessStudents, for example, processes data pertaining to students at different study levels i.e., it extracts the *MasterStudents* and *UndergraduateStudents* data from the relevant data sources (*MasterService* and *UndergraduateStudentService*, respectively), then transforms the extracted data into the *Student* entity and finally loads the transformed data into the *FonDW*. For each identified data process a corresponding complex transformation operation diagram is created, representing the semantics of the data process (i.e., the necessary entity transformation operations and the order in which they are to be executed). For example, in Fig. 8, the execution of

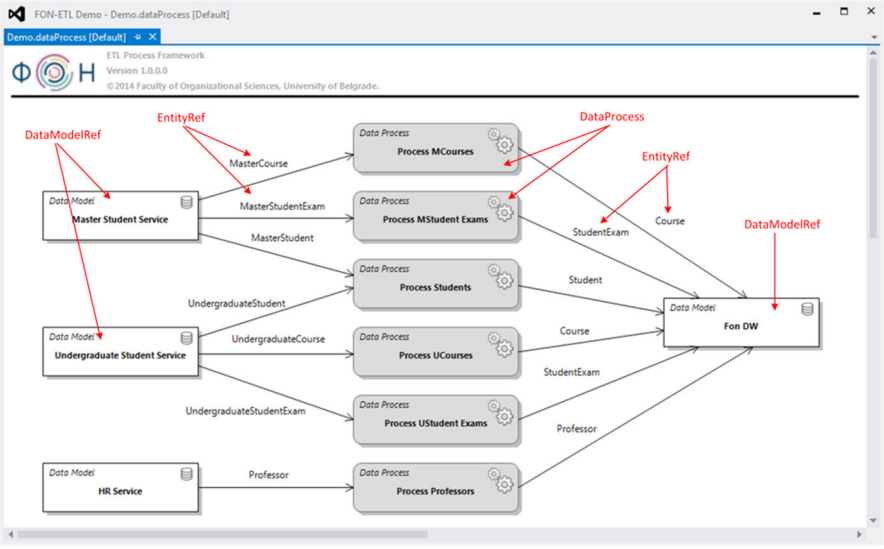

Fig. 7 *DataProcess* diagram

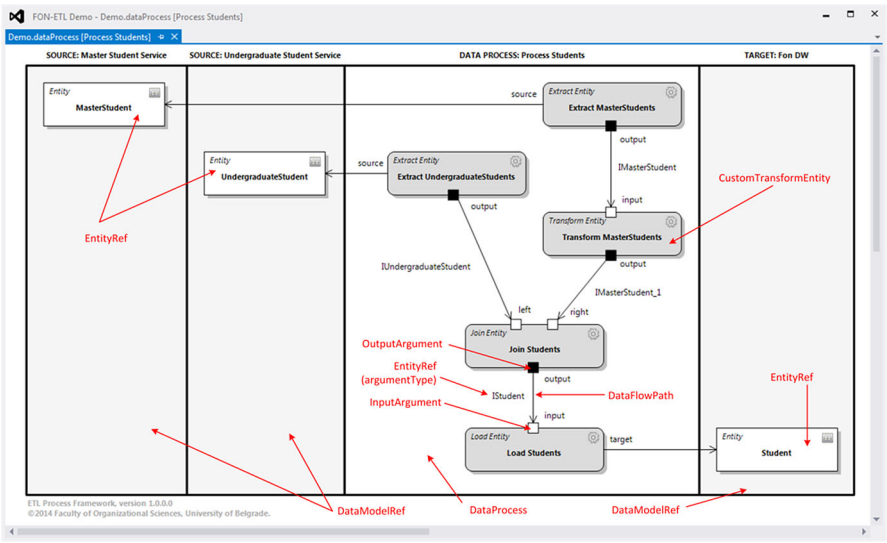

Fig. 8 Complex transformation operation diagram for *ProcessStudents*

ProcessStudents commences with the *ExtractMasterStudents* and *ExtractUndergraduateStudents* operations which extract the necessary data (i.e., *MasterStudents* and *UndergraduateStudents*, respectively) from the *MasterService* and *UndergraduateStudentService* data sources. The result of the execution of these operations is depicted by the *IMasterStudents* and *IUndergraduateStudents* entities.

The *TransformMasterStudents* operation performs the necessary transformations on the *IMasterStudent* entities resulting in *IMasterStudent_1* entities which, along with the *IUndergraduateStudent* entities, represent the input of the *JoinStudents* operation. The execution of this operation then results in *IStudent* entities which are finally loaded into the *FonDW* through the *LoadStudents* operation. It should be noted that the *IMasterStudent*, *IUndergraduateStudent*, *IMasterStudent_1* and *IStudent* entities are the intermediate results of this data process.

The semantics of the entity transformation operations are represented through simple transformation operation diagrams. In Fig. 9a diagram representing the *TransformMasterStudent* operation is given. For each field of the output entity (*IMasterStudent_1*) the required transformation of one or more fields of the input entity (*IMasterStudent*) is defined. For example, the value of the *StudentNumber* field is obtained by executing the *ProcessStudentNumber* operation which takes the value of the *Number* field as its input.

The semantics of the *ProcessStudentNumber* and *ProcessEnrolmentYear* operations are specified by an ETL-E expression (Fig. 10), while the semantics of the *ProcessFirstName* and *ProcessLastName* operations are given by referencing the appropriate templates (i.e., *GetFirstNameTemplate* and *GetLastNameTemplate*, respectively).

The specification of the execution of the ETL process is given by the ETL process execution diagram in which the identified data processes are represented as

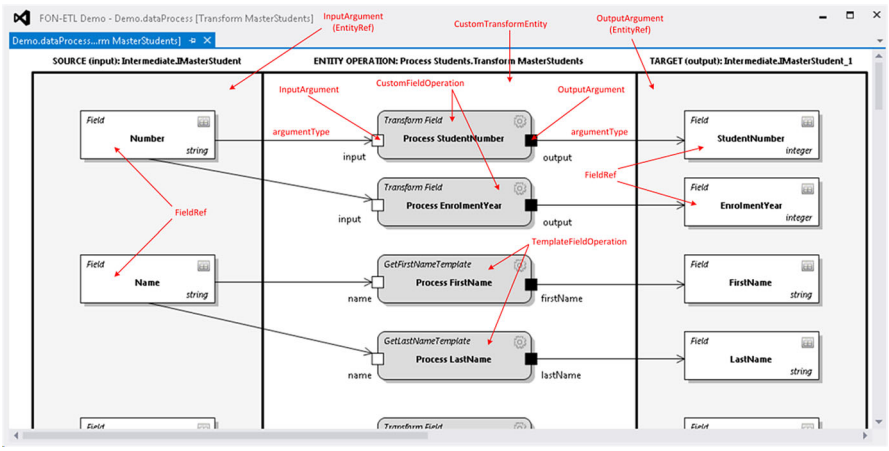

Fig. 9 Simple transformation operation diagram for *TransformMasterStudents*

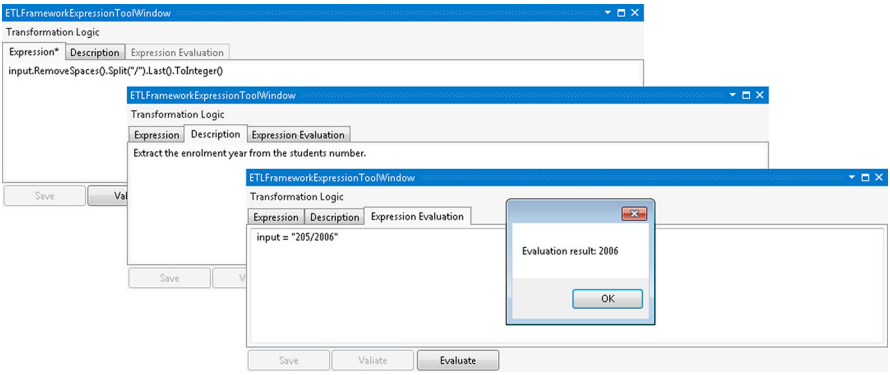

Fig. 10 Transformation expression for the *ProcessStudentNumber* operation

ETL process activities. In Fig. 11 an ETL process execution diagram for this particular ETL process, which begins with the parallel execution of the *ProcessStudents*, *ProcessProfessors, ProcessUCourses* and *ProcessMCourses* activities and ends with the parallel execution of the *ProcessUStudentExams* and *ProcessMStudentExams* activities, is given.

4 The implementation of ETL process specifications

Once a DSL has been specified the next step is to provide its actual implementation. The implementation of a DSL is obtained through the automatic transformation of its specification into executable code.

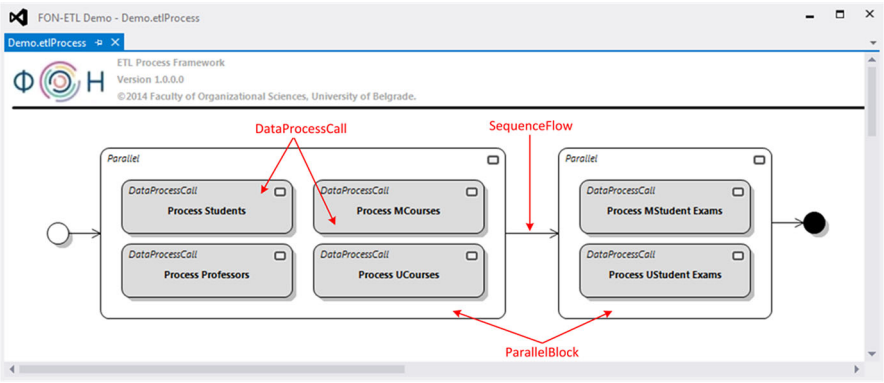

Fig. 11 ETL process execution diagram

On the one hand the aim of the paper is to provide a means for fully automating ETL process development in order to significantly increase development productivity and efficiency and lower the development and maintenance costs. In light of the discussion in Sect. 2. It can be concluded that, since the MDA approach (which is based on the refinement of models through successive model transformations) typically requires that the automatically generated models be manually extended with additional details, automation is usually only partially achieved with this approach. Conversely, in the DSM approach the implementation is automatically generated from the specification by code generators which specify how information is extracted from the models and directly transformed into code. Since the code generators are also domain specific, no manual modifications of the generated code are necessary. The code generators should further be supported by a domain-specific framework in order to narrow the gap between the problem domain and the solution domain. If both the specification and the framework use formal concepts close to the real ETL domain concepts, the transformation between them can be fully automated thus the specification becomes indeed executable.

On the other hand, it is stipulated that, given the nature of ETL processes, several additional requirements should also be fulfilled by the software solution implementing the proposed DSLs:

- It should enable the dynamic execution of ETL process specifications or, more precisely, the automated transformation of ETL models into an executable form at runtime.
- It should provide the necessary flexibility to rapidly respond to changes in business requirements or data sources, by allowing for ETL process specifications to be easily adapted (i.e., modified, extended or even created anew) and immediately executed.
- It should support model versioning, i.e., the execution of different versions of a model.

- It should be easily deployable and scalable without affecting the operation of the execution environment.
- It should enable the execution of ETL processes in a distributed environment and allow for the possibility of parallelizing the execution of different data processes.

In order to meet these requirements, the service-oriented approach to software development (SOA) should be adopted for the development of the supporting software solution, because it results in extremely scalable and flexible solutions and allows for parallelization and distributed execution. In addition, in order to achieve the desired flexibility, the execution of the ETL process should be driven by the relevant metadata, i.e., the ETL-P or ETL-O models that are to be executed (as well as the supplementary ETL-E, ETL-T and ETL-D models) which would all be stored in a model repository.

Moreover, the services, which are to be responsible for the control flow and data flow of an ETL process, should be developed as generic services which would be capable of executing any concrete control flow or data flow model, respectively, and which could easily be installed on each available hardware node (so that every node could handle the execution of any concrete service instance). The functionality of these generic services would then be augmented, at runtime, by the execution semantics, given in the relevant models and interpreted or compiled on demand. For this to be possible a specific application framework should be provided which would include: a set of *implementation concepts* corresponding to each of the introduced language concepts, a specifically developed *Generator* component which would be responsible for interpreting ETL-P and ETL-O models and creating corresponding executable in-memory object models, and a *Compiler* component which would be responsible for dynamically generating executable code from ETL-E models.

The execution semantics would then be obtained as follows (Fig. 12): for each concept, specified in an ETL-P or ETL-O model, the corresponding implementation

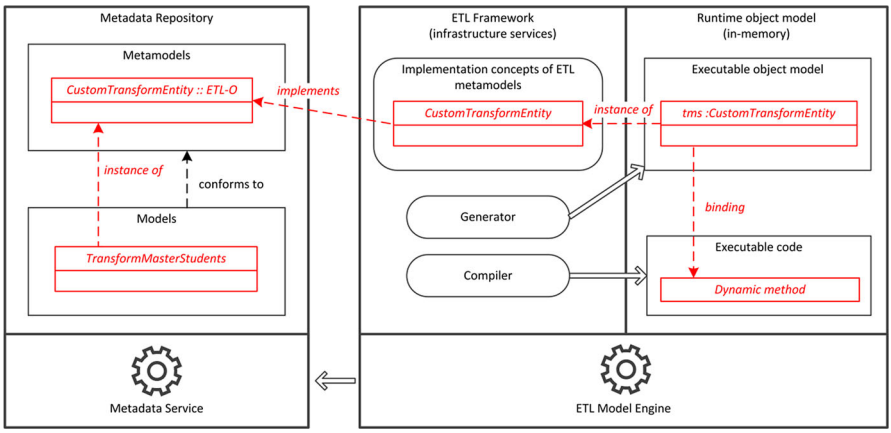

Fig. 12 Proposed implementation of the generic services

concept is retrieved, instantiated and added to an in-memory object model by the Generator component. If the implementation concepts are close to the DSL concepts, the retrieval of the implementation concept which corresponds to a particular DSL concept would be trivial and could be accomplished by introducing appropriate naming conventions. However, in order to enhance the overall performances of an ETL process it is argued that ETL-E models (giving the specific execution logic) should be dynamically compiled into executable code rather than interpreted. Hence, the ETL-E models, which are referenced in ETL-P and ETL-O models, should be compiled at runtime, resulting in a set of dynamic methods which could then be bound to the corresponding objects, along the lines of the Adaptive model notion (Fowler 2010).

By providing such generic services, the dynamic execution of ETL process models (i.e., the automatic generation, compilation and execution of ETL processes at runtime) would be made possible. This would also allow for ETL process specifications to be easily adapted (i.e., models could be modified, extended or even created anew, and the corresponding executable code would be promptly generated and executed) which would provide the necessary flexibility to quickly respond to the constant changes in business requirements or data sources.

Furthermore, the deployment of a developed ETL process would be straightforward, i.e., it would be accomplished simply by storing the relevant models in the repository. Moreover, the modification of existing models, or the creation of new ones, wouldn't affect the operation of the execution environment. Consequently, such an approach would support model versioning and significantly facilitate the testing of the created ETL processes.

Finally, such an approach would inherently enable the parallelization and distributed execution of an ETL process, thereby making it possible to fully exploit the existing hardware resources. Furthermore, the hardware infrastructure could be easily augmented at runtime by adding additional hardware nodes on which only the generic services need be installed. Those hardware nodes would then be instantly operational thus increasing the available processing capability.

It can be concluded that by adopting such an approach not only would full automation be supported, but the automation would actually take place in real-time.

The concrete implementation details of a solution which fulfills the posed requirements are given in the following sections.

5 ETL platform

In this paper a specific ETL platform is proposed to technologically support the ETL process specifications as well as to enable the automated development of ETL processes in accordance with the DSM approach and their subsequent execution. The proposed ETL platform (Fig. 13) would be an extension of a general purpose platform such as Microsoft.NET or J2EE.

The bottom layer (*ETL Framework*) represents the execution environment and is comprised of a set of services which are responsible for the execution and management of ETL processes. The introduction of the application framework

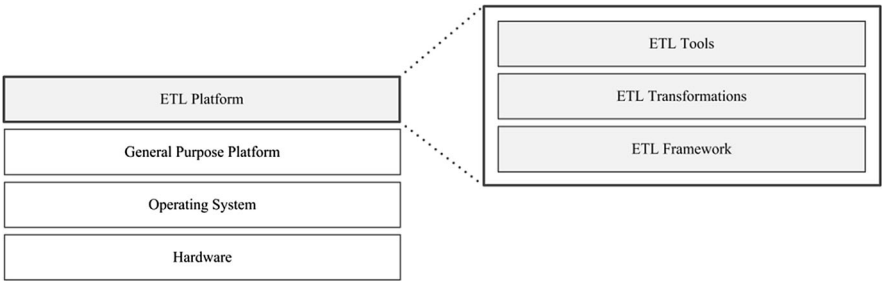

Fig. 13 ETL platform architecture

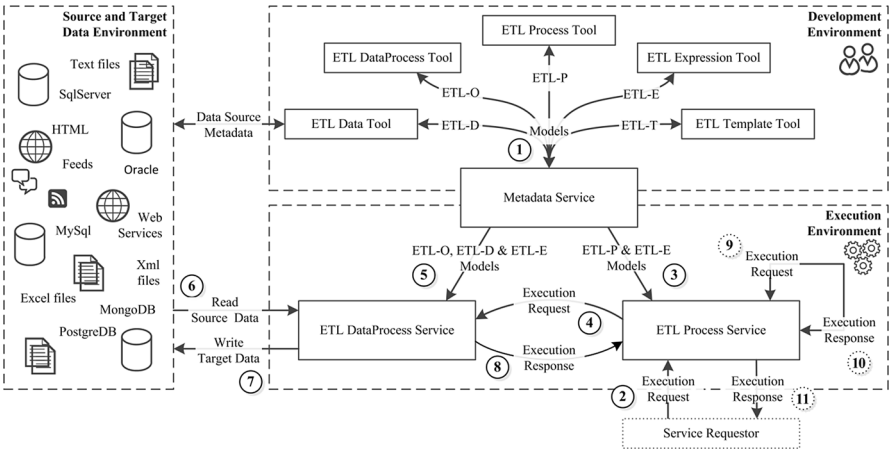

Fig. 14 Overview of the ETL platform

significantly elevates the semantic level of the solution and supports its automated implementation (i.e., the automatic generation of executable code from the given models).

The middle layer (*ETL Transformations*) is responsible for the automatic transformations of models (which have been created in accordance with the defined DSLs) into executable code. These transformations are supported by specially developed generators.

The final layer (*ETL Tools*), representing the development environment, is comprised of a number of software tools which have been developed to technologically support the modeling of ETL processes.

An overview of the proposed ETL platform is given in Fig. 14.

The *Data Environment* represents the relevant, usually very heterogeneous, data sources. All of the data sources (sources as well as targets) are uniformly represented, using the concepts of the ETL-D language.

The *Development Environment* is dedicated to domain experts. It provides the supporting infrastructure for efficient ETL process development. Thus, the first step

in ETL process development using ETL-PL, would be the modeling of the different aspects of an ETL process using the specially developed tools for each of the introduced ETL DSLs (as depicted in Sect. 3.2.), and the created models would then be stored in the metadata repository **(1)**.

The *Metadata Repository* (governed by the *Metadata Service*) is the central component of ETL-PL. The *Metadata Service* thus represents the communication channel for the automated exchange of metadata (i.e., models) between the development and execution environments. In accordance with contemporary methodological approaches to software development, based on models and MDD standards, models are the central elements of the repository. In other words, models represent the main concept which is to be stored, maintained and searched for by users or software agents (such as components, programs, services). The repository also stores information about model referencing (i.e., when one model uses the definitions given in another model) to track the dependencies among the models. Moreover, in order to achieve the desired flexibility and adaptability of ETL process solutions, ETL-PL also uses the repository to provide model-driven execution. In effect, ETL process execution is actually driven by the repository contents, i.e., the models representing the ETL process. Consequently, changes in business requirements are realized through the adaptation of existing models, or creation of new ones, instead of hard-coding the business logic.

The *Execution Environment* consists of a set of services responsible for the automatic generation of executable code from the models, as well as the actual execution of the generated code along with a number of supporting infrastructure services (e.g., *ETLNotifyingService*, *ETLLoggingService*, *ETLSchedulingService*, etc.).

The execution of an ETL process is set into motion upon the receiving an *Execution Request* message **(2)**. The execution can be instigated either by one of the components of ETL-PL (e.g., the *ETLSchedulingService*) or by an external system. Furthermore, the execution can be scheduled or else triggered in response to an event in the environment. In addition, it can also be initiated by a top-level ETL process **(9)**, as is the case when an ETL process coordinates the execution of other ETL processes. Both the Request-Response and One-Way Message Exchange Patterns are supported for requesting the execution of an ETL process. In the case of a Request-Response message exchange, the initiator will receive an *Execution Response* upon the completion of the execution (**10** or **11**, depending on who initiated the execution).

The execution of an ETL process is driven by the actual models that are to be executed. Thus, following the reception of an execution request, the relevant models (ETL-P and ETL-E models) will be retrieved from the *Metadata Repository* **(3)**. From these models the corresponding executable code will be generated, compiled and finally, executed.

An ETL process is comprised of a number of activities (i.e., data processes) and the execution of data processes is supported by a separate service. Similarly to the execution of an ETL process, the execution of a data process commences upon receiving an *Execution Request* **(4)**. The relevant models (ETL-O, ETL-E, ETL-D and ETL-T) will be retrieved from the *Metadata Repository* **(5)** and once again the

corresponding executable code will be generated, compiled and finally, executed. In the course of the execution of the data process the data is extracted from the data sources **(6)**, transformed and then loaded into the targets **(7)**. Only Request-Response message exchange is supported for data process execution, thus the execution concludes by creating and sending an *Execution Response* **(8)**.

6 ETL platform implementation

In order to develop a high-quality software solution first a stable software architecture must be defined. Since software development can be extremely complex and time-consuming it is necessary to raise the level of abstraction, in order to manage the complexity, and view the solution as a set of components each providing part of the required functionality. The identified components are then organized into layers, on the basis of the functionality they provide, thereby simplifying the solution design. In addition to identifying the necessary structural components, a software architecture also defines the behavior of the system in terms of the collaboration among the identified components. The communication between the components is realized via interfaces (through which the components expose the functionality they provide).

The nature of ETL processes imposes strict requirements regarding the performances and scalability of the supporting software solutions. It is, thus, imperative to define a stable, yet flexible, software architecture which will, on the one hand, fulfill the necessary requirements, while on the other hand, be easily extensible to respond to the constant changes in business requirements.

In accordance with these requirements, a software architecture is proposed, which defines the main components of ETL-PL, their roles and responsibilities, along with a set of rules controlling the way in which they can interact. The main components of ETL-PL are organized into two layers: the development environment (*ETLDevelopment*) and the execution environment (*ETLExecution*).

ETLDevelopment

The development environment is comprised of tools which support the modeling of ETL processes in accordance with the introduced ETL DSLs. More specifically it contains tools for defining the abstract and concrete syntax (in both a graphical and textual notation) of a DSL and tools (syntax editors, graphical and textual) for creating models in accordance with the defined DSL. More precisely, it consists of a set of tools (*ETLProcessTool*, *ETLDataProcessTool*, *ETLExpressionTool, ETLDataTool* and *ETLTemplateTool*) which were developed to technologically support both the introduced DSLs (ETL-P, ETL-O, ETL-E, ETL-D and ETL-T, respectively) and the modeling of an ETL process using these DSLs (Fig. 15).

The development environment is also supported by several infrastructure components, among which the *ETLDocumentationTool* (which can automatically generate the ETL process documentation in a.*docx* format) can be singled out.

The specification of the abstract and concrete syntax of the proposed DSLs is accomplished using *Microsoft DSL Tools* (Microsoft 2013), along with the open source *Irony* parser generator framework (Ivantsov 2009) for languages which have

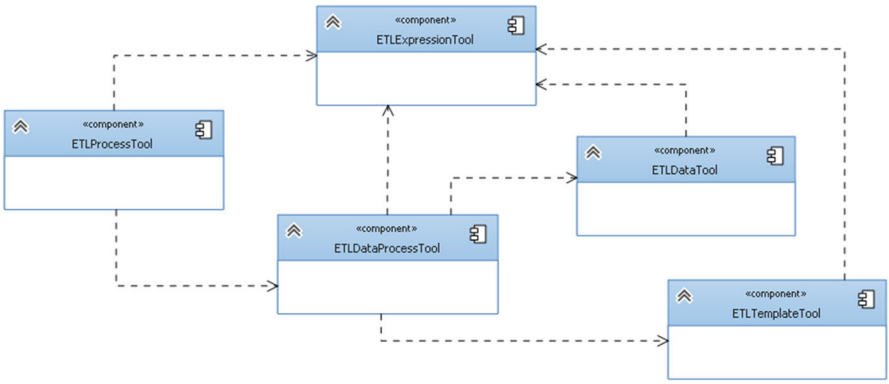

Fig. 15 *ETLDevelopment* component diagram

a textual concrete syntax, while the creation of models using these DSLs is supported by specially developed software tools (primarily graphical editors). The usage of the developed tools for the creation of the models, in accordance with the proposed ETL DSLs, was illustrated in Sect. 3.2.

ETLExecution

The execution environment is responsible for the automatic generation of executable code from the models as well as the actual execution of the generated code. Thus the execution environment consists of code generator components and the components implementing the application framework.

It is developed in accordance with the service-oriented approach to software development (SOA). Hence it is comprised of a number of independent services (communicating with one another asynchronously, via messages) responsible for the execution and management of the developed ETL processes (or more precisely the executable code generated from the defined models). The SOA approach was adopted because it results in extremely scalable and flexible solutions, which is imperative in light of the frequent changes in business requirements. Since it promotes the loose-coupling of services, the solution can be easily extended, simply by adding new services, or modified without affecting the existing services.

The execution environment consists of four core services (*ETLProcessService*, *ETLDataProcessService*, *ETLCompilerService* and *ETLMetadataService*) which are responsible for the execution and management of ETL processes (Fig. 16). More precisely, *ETLProcessService* and *ETLDataProcessService* are responsible for executing the control flows and data flows of an ETL process, respectively. The *ETLCompilerService* handles the generation and compilation of executable code, while the *ETLMetadataService* manages the ETL metadata repository. In addition to these core components, the execution environment also contains a number of supporting infrastructure components such as: *ETLNotifyingService*, *ETLLoggingService*, *ETLTracingService*, *ETLSchedulingService*, etc.

Three crucial characteristics of the execution environment should be specially emphasized. First, the developed services enable the dynamic execution of ETL

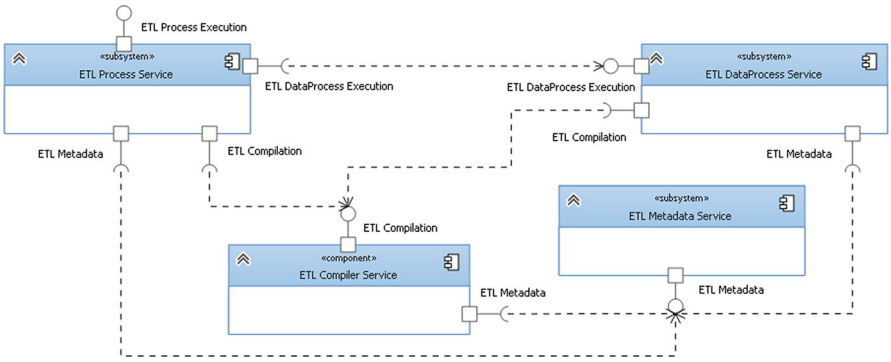

Fig. 16 *ETLExecution* component diagram

processes or, more precisely, the automated generation and execution of ETL processes at runtime. The execution of a *Process* (be it an *ETLProcess* or an *ETLDataProcess*) is driven by the relevant metadata (i.e., the ETL-P or ETL-O models that are to be executed, respectively, as well as the supplementary ETL-E, ETL-T, ETL-D models). The specially developed *ETLCompilerService* generates the executable code, with the *ETLMetadataService* providing the relevant metadata. Second, the execution environment was developed with the possibility of parallelizing the execution of the different services (responsible for the actual processing of data) in mind. Namely, ETL-PL allows for the independent execution (and specification) of the different aspects of an ETL processes by providing separate services for the execution of the control flows and data flows of an ETL process (*ETLProcessServices* and *ETLDataProcessServices*, respectively). It should be emphasized that these services are actually developed as generic services which are capable of interpreting any concrete control flow and data flow model, respectively. The functionality of these generic services is then augmented, at runtime, by the concrete transformation logic (which is compiled on demand). This opens up the possibility of parallelizing the execution of the different concrete services comprising an ETL process. Finally, that it was designed to enable the execution of ETL processes in a distributed environment. ETL-PL therefore presumes that the generic services are installed on each of the available hardware nodes so that every node can handle the execution of any concrete service instance. By parallelizing the execution of the services, instead of executing them sequentially, the performances of an ETL process are significantly increased. Distributing the execution of the services over the different available hardware nodes leads to yet a further increase in performances.

Since the focus of this paper is on the implementation of ETL processes only the *ETLExecution* environment will be elaborated in the following subsections while the *ETLDevelopment* environment will not be further discussed.

6.1 ETL process implementation

An *ETL process* is comprised of a number of activities (data processes) which are to be executed in a particular order with the aim of transforming business data into

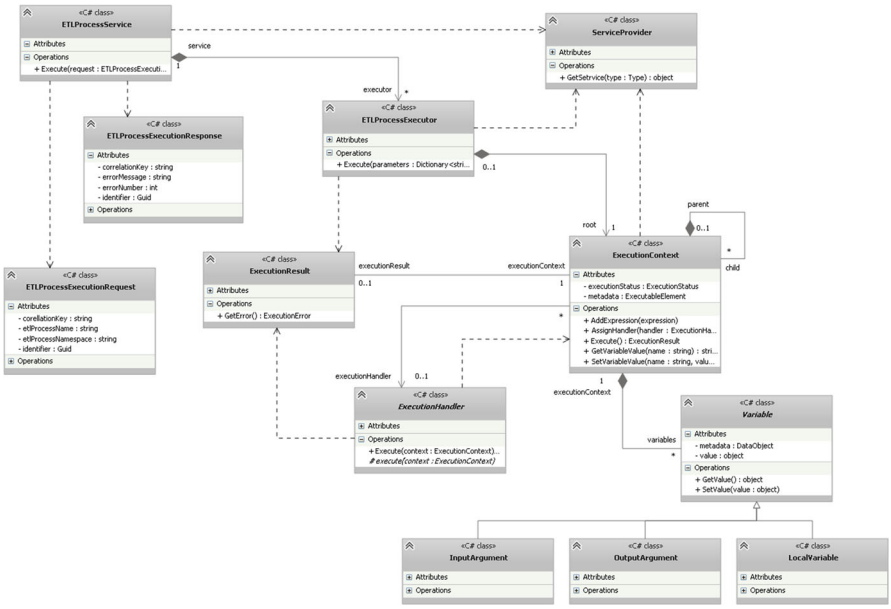

Fig. 17 *ETLProcessService* class diagram

strategic information. The specification of the control flow (i.e., execution order of the activities) is supported by four main control structures *SequenceFlow*, *ConditionalBlock*, *IterateBlock* and *ParallelBlock.* The activities can be either *SimpleActivities*, such as actual data processes (defined in separate ETL-O models and invoked via the *DataProcessCall*) or other ETL processes (defined in separate ETL-P models and invoked via the *ETLProcessCall)*, or *ComplexActivities* (i.e., *ConditionalBlock*, *IterateBlock* and *ParallelBlock*) composed of other activities, simple and/or complex. In this section the different mechanisms, by which the control structures have been implemented, will be elaborated, while the implementation details of the data processes will be given in the next subsection.

The main concepts of the proposed application framework (related to the ETL process implementation) are depicted in Fig. 17.

The *ETLProcessExecutor* class is responsible for executing a particular ETL process, while the *ServiceProvider* class is responsible for providing and managing the various components (such as *ETLMetadataService*, *ETLCompilerService*, *ExecutionContextService*, *ExecutionHandlerService*, *ETLDataProcessService*, *ETLProcessService*, etc.) that are used during the execution of an ETL process.

The *ExecutionContext* class is used for representing the execution context of an ETL process or an ETL process activity. It stores the state of the process or activity that is being executed. A separate execution context is created for the execution of each of the activities comprising an ETL process. Furthermore, if an activity is composite (i.e., *ConditionalBlock*, *IterateBlock* and *ParallelBlock*) separate execution contexts must also be created for the execution of each of its subactivities, and

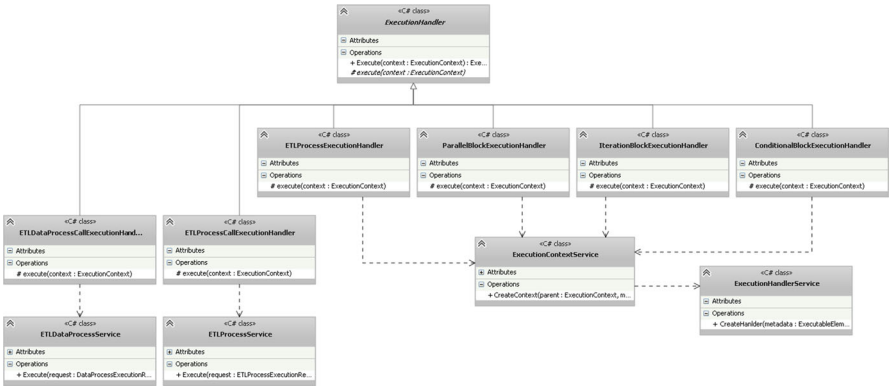

Fig. 18 *ExecutionHandlers* class diagram

so forth if the subactivities are themselves also composite. Hence, these execution contexts form a parent–child hierarchy, in which the execution context of each composite activity contains the execution contexts of its subactivities, with the execution context of the ETL process as a whole at the root.

The states are maintained through a set of *InputArguments*, *OutputArguments*and*LocalVariables.* The three different types of *Variables* where introduced in order to provide more control over their usage e.g., the value of an *OutputArgument* is available only after the execution of the activity has been brought to an end (i.e., when its *executionStatus* has been set to *Executed*).

The execution of the activities is supported by the *ExecutionHandler* class. The *ExecutionHandler* class is specialized (Fig. 18) to support the different types of activities inherent to the ETL process. Each subclass implements the *execute* method to provide the desired behavior. The public *Execute* method acts as a wrapper for protected *execute* method and implements the behavior common to all of the activities (such as error handling, logging etc.). Both methods expect an *ExecutionContext* as an input parameter, and result in an *ExecutionResult* instance.

As depicted in the sequence diagram, given in Fig. 19, the execution of an *ETLProcess* is initiated by invoking the *Execute* method of an *ETLProcessService* instance. In other words the execution commences upon receiving an *ETLProcessExecutionRequest* message and completes by creating and sending an *ETLProcessExecutionResponse* message.

Subsequently, a specific application domain is created (Hazzard and Bock 2013; Troelsen 2012) and the *ETLProcessExecutor* is instantiated. It should be emphasized that the specific application domain was introduced to support the generation and compilation of executable code during the execution of a process.

Once the *ETLProcessExecutor* has been created, its *Execute* method will be invoked. As previously stated, the execution of ETL processes is metadata driven, so the first step is to retrieve the relevant metadata (i.e., the ETL-P model which is to be executed as well as the ETL-E models it references). To this end the *LoadMetadata* method of an *ETLMetadataService* instance is invoked.

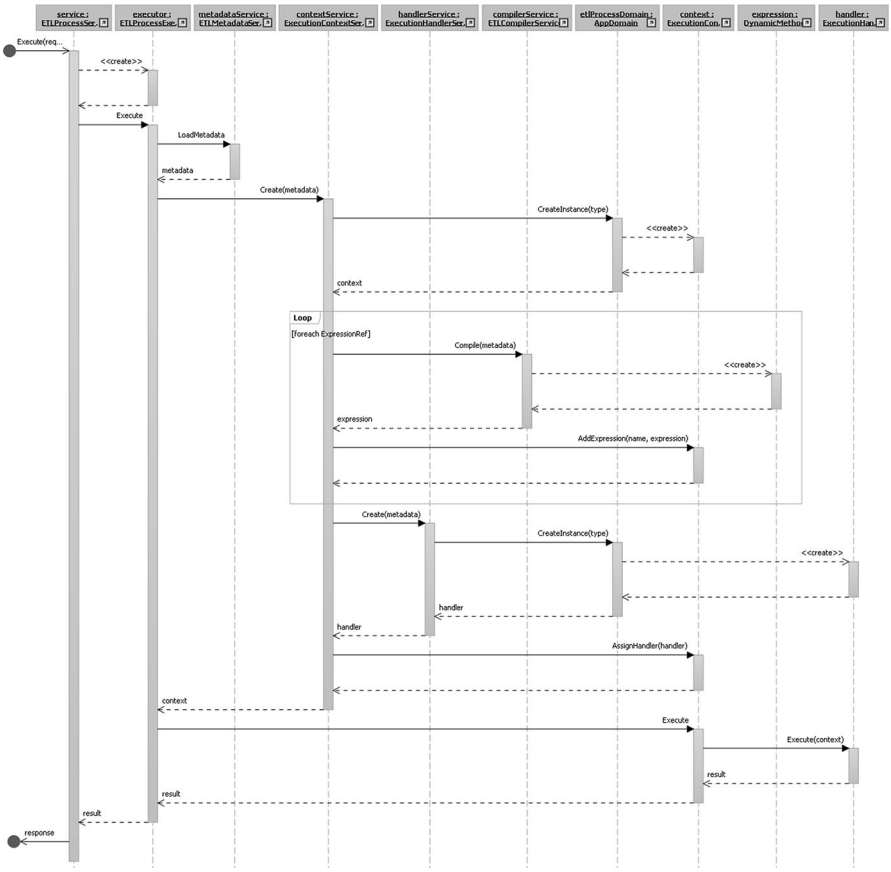

Fig. 19 *ETLProcessService* sequence diagram

The next step is to create the root execution context (as an instance of the *ExecutionContext* class) for the ETL process, based on the relevant metadata, which is accomplished by invoking the *Create* method of an *ExecutionContextService* instance. Finally, the *Execute* method of the root *ExecutionContext* (or more precisely, the *ExecutionHandler* assigned to that context) is invoked to execute the ETL process which actually sets off the execution of the sequence of activities comprising the process. The activities are thus executed one by one. The execution of each activity entails the creation of a new *ExecutionContext* for that activity and the invocation of its *Execute* method. However, if the activity is composite, its execution presumes that individual *ExecutionContexts* will be created for the execution of each of its subactivities. Depending on the type of activity (*ConditionalBlock,IterateBlock* or *ParallelBlock*) the actual execution will be either sequential or parallel.

In order to create an *ExecutionContext* the *Create* method of an *ExecutionContextService* instance is invoked. First the appropriate *ExecutionContext* is

instantiated. Then, for each referenced ETL-E expression, the corresponding executable code must be generated and compiled, which is accomplished by invoking the *Compile* method of an *ETLCompilerService* instance. The compilation process (Hazzard and Bock 2013; Troelsen 2012; Microsoft 2014a, b) results in a dynamic method (*DynamicMethod*) which is then attached to the *ExecutionContext* instance. Finally, the appropriate *ExecutionHandler* instance is created (by invoking the *Create* method of an *ExecutionHandlerService* instance) and assigned to the *ExecutionContext*. It should be emphasized that a single *ExecutionHandler* is created for each particular type of activity thus different execution contexts pertaining to the same type of activity will be assigned the same *ExecutionHandler* instance.

6.2 ETL DataProcess implementation

A data process consists of a number of simple data operations (i.e., the data extraction, transformation and loading operations) which are composed into a data flow. The flow of data between the operations (i.e., the execution order of the data operations) is defined by *DataFlowPaths*, with output (*OutputArgument*) of one operation providing the input (*InputArgument*) of the following operation. Thus, the execution flow of a data process is driven by the interdependence of the data operations. However, the actual scheduling of the execution time of these operations is predetermined by the availability of the relevant data. To this end a *push mechanism* has been adopted in the proposed application framework to ensure that each data operation, upon completion, transfers the relevant data to the next operation. The implementation of this mechanism is based on the well-known *Observer* pattern with the output argument of an operation taking the role of the *Subject* and the input argument of the following operation taking the role of the *Observer*. The binding of these arguments is accomplished via the *Subscribe* method of the *IOperationOutputArgument* interface.

An abridged model depicting the main concepts of the proposed application framework (related to the implementation of ETL data processes) is given in Fig. 20.

The *DataProcessExecutor* class defines two methods: *Initialize* for configuring the execution environment and *Execute* for initiating the execution of a data process. The data operations comprising a *DataProcess* are represented by the *ExtractEntity*, *TransformEntity* and *LoadEntity* abstract classes, which have been further specialized to introduce concrete data operations (*ExtractEntityFromXml*, *ExtractEntityFromSqlServer*, *ExtractEntityFromOracle*, *JoinEntity*, *UnionEntity*, *SplitEntity*, *SortEntity*, *FilterEntity*, *AggregateEntity*, *LoadEntityIntoText*, *LoadEntityIntoSqlServer*, *LoadEntityIntoOracle,* etc.). The classification of data operations is in accordance with the taxonomy of ETL operations given in (Vassiliadis et al. 2009; Petrović 2014).

The inputs and outputs of a data operation are represented by the *InputArgument* and *OutputArgument* classes, respectively, which, in accordance with the adopted data transfer mechanism, implement the required interfaces (*IOperationInputArgument* and *IOperationOutputArgument*).

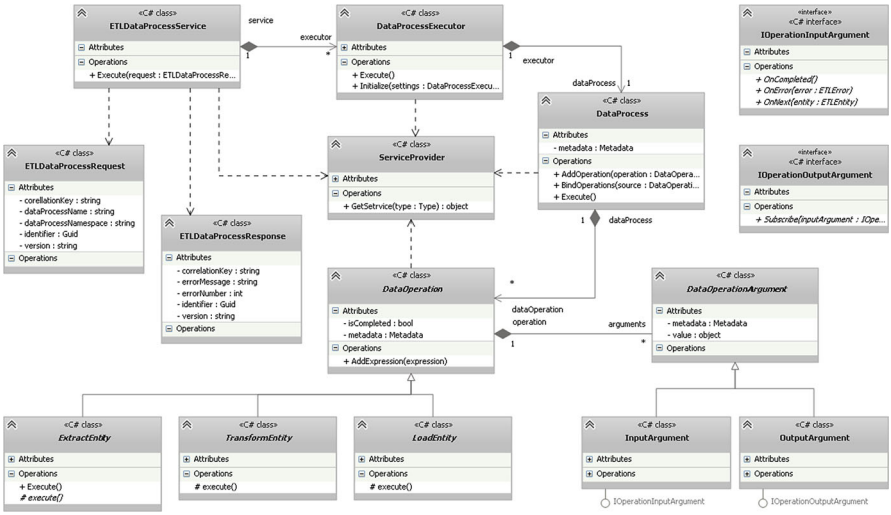

Fig. 20 *ETLDataProcessService* class diagram

The *ServiceProvider* class is responsible for providing and managing the various components (such as *ETLMetadataService*, *ETLCompilerService*, *ETLDataOperationService*, etc.) that are used during the execution of a data process.

The *ETLDataProcessService* sequence diagram is given in Fig. 21.

Similarly to the execution of an ETL process, the execution of a *DataProcess* commences upon receiving an *ETLDataProcessExecutionRequest* message and completes by creating and sending an *ETLDataProcessExecutionResponse* message (depicted in Fig. 21 by the invocation of the *Execute* method of a *ETLDataProcessService* instance). A specific application domain is then created and the *DataProcessExecutor* is instantiated.

However, contrary to the execution of an ETL process, all of the data operations comprising the data process must be instantiated and bound to each other, before the execution of the data process can commence. Thus once the *DataProcessExecutor* has been created, its *Initialize* method will be invoked to configure the execution environment. Since the execution of a data processes is also metadata driven, it is first necessary to retrieve the relevant metadata (i.e., the ETL-O model which is to be executed as well as the ETL-E models it references) by invoking the *LoadMetadata* method of an *ETLMetadataService* instance. In accordance with the obtained metadata, the appropriate *DataProcess* is instantiated and then configured. The configuration of a data process instance entails the creation of all of the involved data operations.

In order to create a *DataOperation* the *Create* method of a *DataOperationService* instance is invoked. First the appropriate *DataOperation* is instantiated. Then, for each referenced ETL-E expression, the corresponding executable code must be generated and compiled, which is accomplished by invoking the *Compile* method of an *ETLCompilerService* instance. The compilation process (Hazzard and Bock

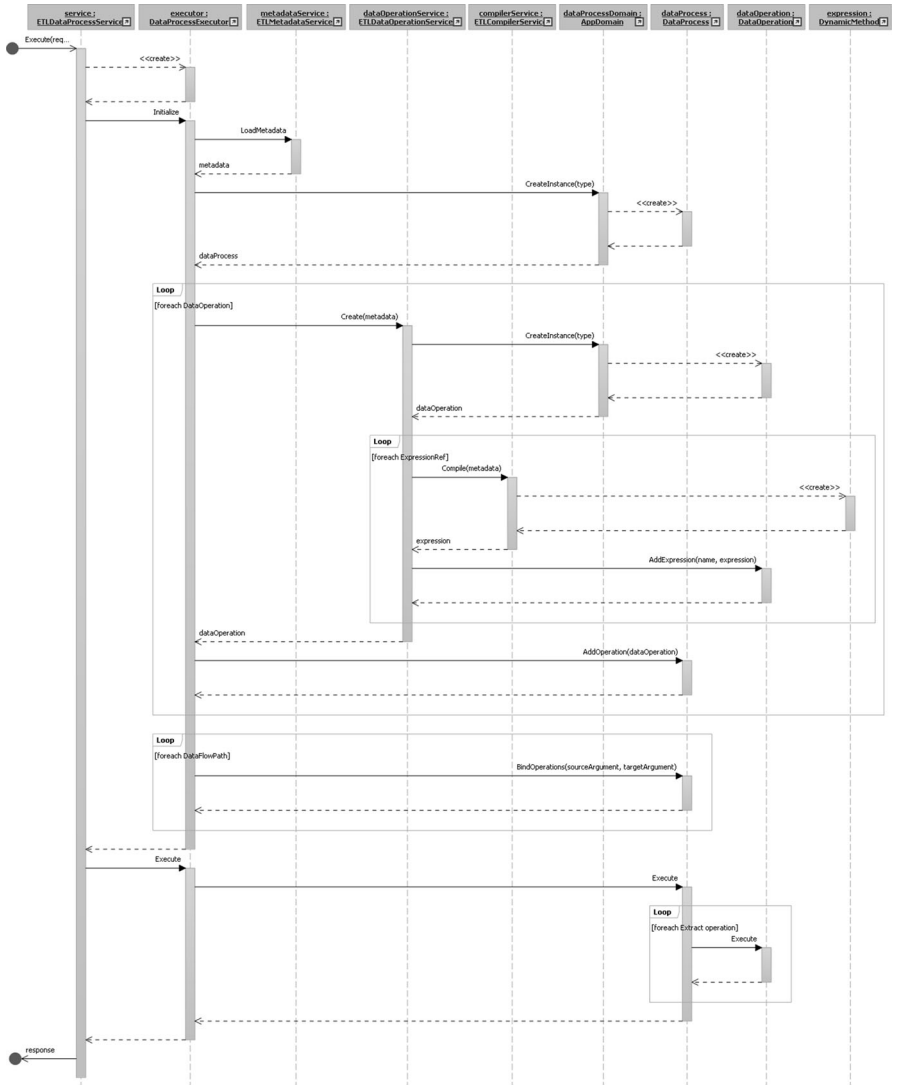

Fig. 21 *ETLDataProcessService* sequence diagram

2013; Troelsen 2012; Microsoft 2014a, b) results in a dynamic method (*DynamicMethod*) which is then attached the *DataOperation* which is subsequently attached to the to the *DataProcess* instance through the *AddOperation* method.

When all of the *DataOperations* have been created, the final step is to bind the operations to each other in order to create the defined data flow. This is accomplished by invoking the *BindOperations* method of the *DataProcess* instance for every *DataFlowPath* in the obtained model.

Once the execution environment has been configured, the *Execute* method of the *DataProcessExecutor* instance is invoked to execute the data process. This entails the creation of a collection of the *Extract* operations and the concurrent invocation of their *Execute* methods. In accordance with the adopted data transfer mechanism, the remaining data operations (i.e., the data transformation and load operations) will automatically be executed as soon as they receive the necessary data. The execution of the data process is completed when all of the data operations have been executed (i.e., once the *isComplete* attribute of every single operation is set to true).

7 Conclusion

A novel solution, based on the Domain-Specific Modeling (DSM) approach to software development, is proposed for the conceptualization and automation of ETL process development. A specific platform (ETL-PL) is proposed to technologically support, not only the automated development of ETL processes, but also their subsequent execution.

In comparison with the existing methodological approaches, reviewed in Sect. 2, it should first be emphasized that, as previously stated, only a few approaches exist which enable the automated development of ETL processes in the context of MDD (i.e., which support both the formal specification of ETL processes and the automated transformation of such specifications into executable code), while the remaining approaches only deal with the first aspect i.e., the modeling ETL processes.

On the specification side, building on the identified strengths and weaknesses of the analyzed approaches, the proposed solution provides a means for the formal specification of the different aspects of an ETL process (e.g., the control flow, the data flow, the data structures, etc.), using an extensible set of independent DSLs (each providing only a minimal, yet extensible, set of semantically rich domain-specific concepts pertaining to the relevant aspect) and even more importantly (in order to enable automated development) for the formal definition of the semantics of the actual data transformations. Moreover, the syntax and the semantics of the DSL concepts are controlled, hence incorrect or incomplete designs are prevented by making them impossible to specify. Therefore, one of the main advantages of ETL-PL is that, since it is geared towards ETL domain experts, it doesn't require skilled programmers, or even technical knowledge. The fact that it is based on a minimal set of semantically rich abstractions, which encapsulate the existing knowledge and experience in the ETL domain, makes it possible to fully exploit the knowledge and expertise of domain experts. Consequently, it is easier to learn and use compared to vendor ETL tools and GPML based approaches. Furthermore, by providing a graphical development environment, the ETL processes (which can contain very complex custom transformation operations) are easily specified by domain experts (who need not be technically proficient) and the actual implementation will be automatically obtained from the specification, thus significantly increasing development productivity and efficiency, while lowering the development and maintenance costs. Moreover, by separating the different aspects into

different models, and allowing for complex data processes to be gradually defined (through several diagrams at different levels of abstraction) the development of very complex ETL processes is significantly facilitated.

On the implementation side, it should be noted that, to the best of our knowledge, only two groups of authors deal with this aspect in the context of MDD, yet their specifications are based on extensions of GPMLs. The introduction of ETL DSLs as a means for the formal specification of ETL processes, as well as the automated transformation between the specification and the application framework, significantly elevates the semantic level of the solution whose implementation is supported by the introduced application framework. Since both the specification and the application framework use concepts close to the real ETL domain concepts the transformation between them can be fully automated, thus significantly increasing development productivity and efficiency while lowering the development and maintenance costs. Moreover, the obtained solutions would have good performances and be scalable and maintainable yet, at the same time, flexible (i.e., they could be easily extended to adapt to the constant changes in the environment and new requirements).

It should also be emphasized that an additional advantage of ETL-PL is that it presumes the dynamic execution of ETL process models i.e., the automatic generation, compilation and execution of ETL processes at runtime. More importantly, in light of the constant changes in business requirements, the proposed solution provides the necessary flexibility to quickly respond to these changes since the process specification can easily be adapted (i.e., modified, extended or even created anew) and the corresponding executable code will be promptly generated and executed. In addition, the entire ETL process is well-documented.

The deployment of the developed ETL process is straightforward, since the proposed solution presumes that the generic services (which are capable of interpreting any concrete ETL model and whose implementation is stable) are installed on each of the available hardware nodes, so that every node can handle the execution of any concrete ETL process or data process model. The specific implementation of an ETL process is given in the models, which are stored in the metadata repository, thus the actual deployment is accomplished by simply storing the models in the repository. Furthermore, the modification of existing models, or the creation of new ones, doesn't affect the operation of the execution environment.

Finally, ETL-PL is developed in accordance with the service-oriented approach to software development (SOA). The SOA approach was adopted because it results in extremely scalable and flexible solutions, which is imperative in light of the frequent changes in business requirements. Since it promotes the loose-coupling of services, the solution can be easily extended, simply by adding new services, or modified without affecting the existing services. It was thus designed to enable parallel and distributed execution of an ETL process. By parallelizing the execution of the services, instead of executing them sequentially, the performances of an ETL process are significantly increased. Distributing the execution of the services over the different available hardware nodes leads to yet a further increase in performances. The ever increasing volume of data that is to be processed can,

thus, be handled by simply increasing the hardware capabilities (by adding new hardware nodes) and parallelizing the execution.

In order to validate the proposed solution a number of tests have been conducted yielding promising results thus future work would be aimed at testing it in different domains. Further work would also be aimed at exploring the possibility of enhancing the ETL-PL execution environment by introducing a specific *ExecutionOptimizator* service which would be responsible for determining the best possible execution plan for an ETL process based on the defined ETL process model and the actual hardware infrastructure available at runtime.

References

El Akkaoui Z, Zimányi E (2009) Defining ETL worfklows using BPMN and BPEL. In: Proceedings of DOLAP '09, (China), pp 41–48

El Akkaoui, Zimányi E, Mazón J-N, Trujillo J (2011) A model-driven framework for ETL process development. In: Proceedings of DOLAP '11, (UK), pp 45–52

El Akkaoui Z, Mazón J-N, Vaisman A, Zimányi E (2012) BPMN-based conceptual modeling of ETL processes. In: Data warehousing and knowledge discovery, LNCS 7448. Springer, Berlin, pp 1–14

Fowler M (2010) Domain-specific languages. Addison-Wesley Professional, Boston

Greenfield J, Short K, Cook S, Kent S (2004) Software factories: assembling applications with patterns, models, frameworks, and tools. Wiley, Hoboken

Hazzard K, Bock J (2013) Metaprogramming in.NET. Manning Publications, Greenwich

Ivantsov R (2009) Irony—.NET language implementation kit. [Online] CodePlexProject Hosting for Open Source Software: http://irony.codeplex.com/

Jarke M, Lenzerini M, Vassiliou Y, Vassiliadis P (2003) Fundamentals of data warehouses. Springer, Berlin

Kelly S, Tolvanen JP (2008) Domain-specific modeling: enabling full code generation. Wiley, Hoboken

Kimball R, Caserta J (2004) The data warehouse ETL toolkit: practical techniques for extracting, cleaning, conforming, and delivering data. Wiley, Hoboken

Kimball R, Ross M, Thornthwaite W, Mundy J, Becker B (2010) The Kimball group reader: relentlessly practical tools for data warehousing and business intelligence. Wiley, Hoboken

Luján-Mora S, Trujillo J (2004) A data warehouse engineering process. In: Advances in information systems, LNCS 3261. Springer, Berlin, pp 14–23

Luján-Mora S, Vassiliadis P, Trujillo J (2004) Data mapping diagrams for data warehouse design with UML. In: Conceptual modeling-ER 2004, LNCS 3288. Springer, Berlin, pp 191–204

Mazón J-N, Trujillo J (2008) An MDA approach for the development of data warehouses. Decis Support Syst 45(1):41–58

Microsoft (2013) Modeling SDK for Microsoft Visual Studio 2013. [Online] http://www.microsoft.com/en-us/download/details.aspx?id=40754

Microsoft (2014a) Emitting dynamic methods and assemblies. [Online] https://msdn.microsoft.com/en-us/library/8ffc3x75%28v=vs.110%29.aspx

Microsoft (2014b) Expression trees (C# and Visual Basic). [Online] https://msdn.microsoft.com/en-us/library/bb397951.aspx

Muñoz L, Mazón JN, Pardillo J, Trujillo J (2008) Modelling ETL processes of data warehouses with UML activity diagrams. In: On the move to meaningful internet systems: OTM 2008 workshops, LNCS 5333. Springer, Berlin, pp 44–53

Muñoz L, Mazón JN, Trujillo J (2009) Automatic generation of ETL processes from conceptual models. In: Proceedings of DOLAP '09, (China), pp 33–40

Petrović M (2014) A model driven development approach for the data warehouse extract, transform and load process. Ph.D. Thesis final version (in Serbian), Faculty of Organizational Sciences, University of Belgrade, Serbia

Simitsis A (2005) Mapping conceptual to logical models for ETL processes. In: Proceedings of DOLAP '05, (Germany), pp 67–76
Simitsis A, Vassiliadis P (2003) A methodology for the conceptual modeling of ETL processes. In: Proceedings of the decision systems engineering—DSE '03, (Austria), pp 305–316
Simitsis A, Vassiliadis P (2008) A method for the mapping of conceptual designs to logical blueprints for ETL processes. Decis Support Syst 45(1):22–40
Simitsis A, Vassiliadis P, Terrovitis M, Skiadopoulos S (2005) Graph-based modeling of ETL activities with multi-level transformations and updates. In: Data warehousing and knowledge discovery, LNCS 3589. Springer, Berlin, pp 43–52
Troelsen A (2012) Pro C# 5.0 and the.NET 4.5 Framework. Apress
Trujillo J, Luján-Mora S (2003) A UML based approach for modeling ETL Processes in data warehouses. In: Conceptual modeling-ER 2003, LNCS 2813. Springer, Berlin, pp 307–320
Turajlić N, Petrović M, Vučković M (2014) Analysis of ETL process development approaches: some open issues. In: Proceedings of SYMORG'14, pp 45–51
Vassiliadis P, Simitsis A, Skiadopoulos S (2002) Modeling ETL activities as graphs. In: Proceedings of DMDW'02, pp 52–61
Vassiliadis P, Simitsis A, Skiadopoulos S (2002) Conceptual modeling for ETL processes. In: Proceedings of DOLAP '02, (USA), pp 14–21
Vassiliadis P, Simitsis A, Georgantas P, Terrovitis M (2003) A framework for the design of ETL scenarios. In: Advanced information systems engineering, LNCS 2681. Springer, Berlin, pp 520–535
Vassiliadis P, Simitsis A, Georgantas P, Terrovitis M, Skiadopoulos S (2005) A generic and customizable framework for the design of ETL scenarios. Inf Syst 30(7):492–525
Vassiliadis P, Simitsis A, Baikousi E (2009) A taxonomy of ETL activities. In: Proceedings of DOLAP'09, (China), pp 25–32