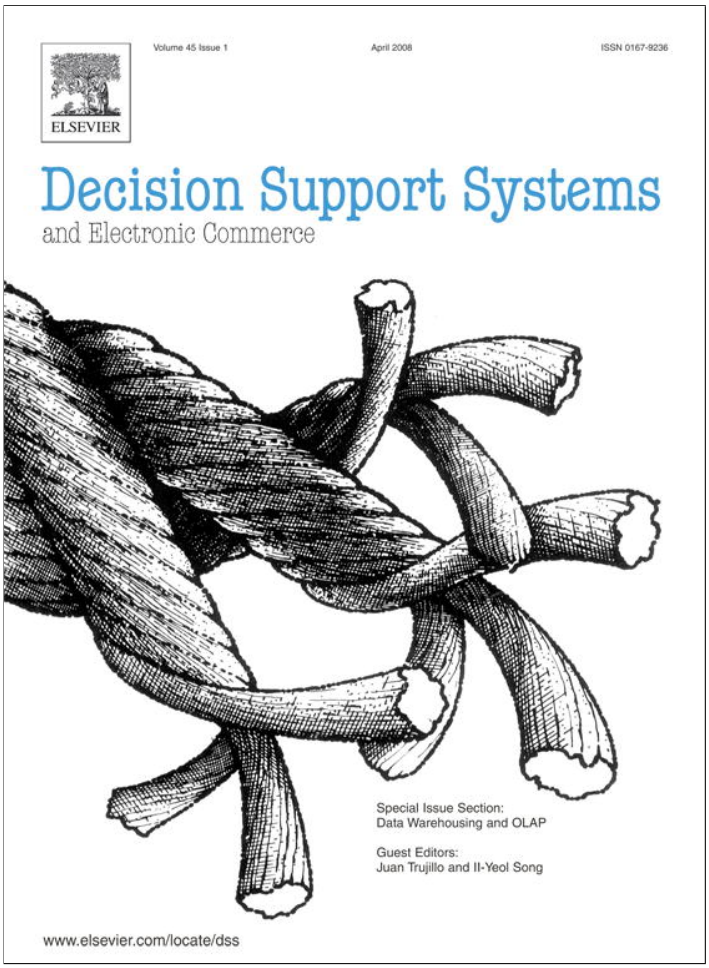

Available online at www.sciencedirect.com
ScienceDirect
Decision Support Systems 45 (2008) 22–40
Decision Support Systems
www.elsevier.com/locate/dss

A method for the mapping of conceptual designs to logical blueprints for ETL processes

Alkis Simitsis [a,*], Panos Vassiliadis [b]

[a] National Technical University of Athens, School of Electrical and Computer Engineering, Athens, Hellas, Greece
[b] University of Ioannina, Department of Computer Science, Ioannina, Hellas, Greece

Available online 5 February 2007

Abstract

Extraction–Transformation–Loading (ETL) tools are pieces of software responsible for the extraction of data from several sources, their cleansing, customization and insertion into a data warehouse. In previous work, we presented a modeling framework for ETL processes comprised of a conceptual model that concretely deals with the early stages of a data warehouse project, and a logical model that deals with the definition of data-centric workflows. In this paper, we describe the mapping of the conceptual model to the logical model. First, we identify how conceptual entities are mapped to logical entities. Next, we determine the execution order in the logical workflow using information adapted from the conceptual model. Finally, we provide a method for the transition from the conceptual model to the logical model.

Keywords: ETL; Conceptual modeling; Logical modeling; Data warehouses

1. Introduction

The design of the data warehouse back-stage has always been strenuous work, due to the inherent amount of complexity of this environment and the technical detail into which the designer must delve, in order to deliver the final design [6]. The task of the designer involves (a) the tracing of the existing data sources and the understanding of their hidden semantics, as well as (b) the design of a workflow that extracts data from these sources, cleans any inconsistencies they may have, transforms them in a suitable format and finally loads them into the target warehouse. The designer is aided by a conceptual model in order to complete the former task and a logical model in order to construct the latter. Also, a method that acts as a "bridge" that transforms the conceptual design to the logical one is also necessary.

In previous works, we presented a conceptual model [17] and a logical [18,19] model for the warehouse back-stage ETL (Extraction–Transformation–Loading) processes. In this paper, we bridge the different levels of our framework by presenting a semi-automatic transition from conceptual to logical model for ETL processes. By relating a logical to a conceptual model, we exploit the advantages of both worlds. On one hand, there exists a simple model, sufficient for the early stages of the data warehouse design. On the other hand, there exists a logical model that offers formal and semantically-founded concepts to capture the characteristics of an ETL process. Although there are several research approaches concerning the semi-automation of several tasks of logical DW design from conceptual

* Corresponding author.
E-mail addresses: asimi@dbnet.ece.ntua.gr (A. Simitsis), pvassil@cs.uoi.gr (P. Vassiliadis).

0167-9236/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.dss.2006.12.002

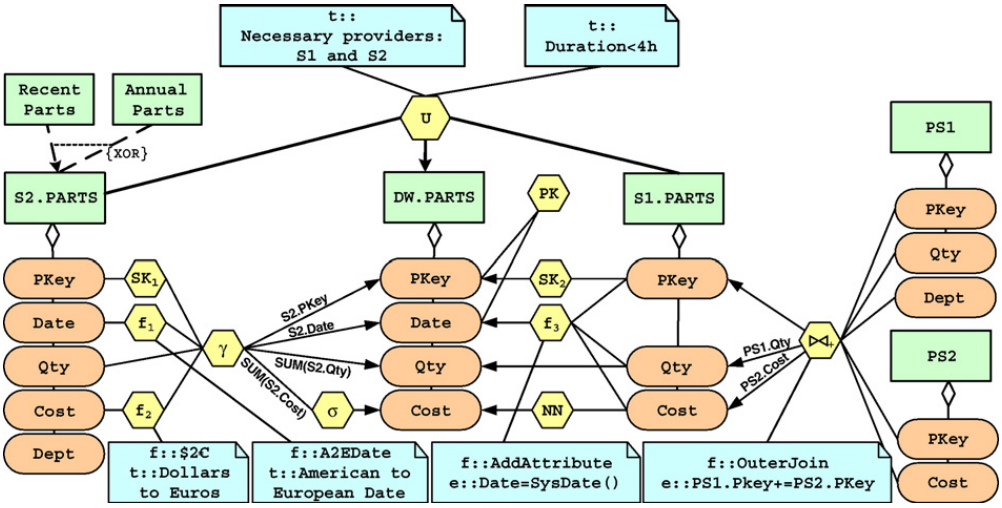

Fig. 1. Conceptual design of our example.

models [1,3–5,8–10,12], so far, we are not aware of any other research approach concerning a mapping from a conceptual to a logical model for ETL processes.

During the transition from one model to the other we have to deal with several issues. First, we need to identify the correspondence between the two models. Since the conceptual model is constructed in a more generic and high-level manner, each conceptual entity is mapped to a logical entity; however, the opposite does not hold. In the sequel, for each conceptual construct we provide its respective logical entity and we describe a method for the automatic transition from the former to the latter.

Moreover, we go beyond the simple one-to-one mapping, and we combine information for more than one conceptual construct in order to achieve a better definition for a logical entity. For example, the conceptual entity 'transformation' is mapped to a logical 'activity'. However, it is not obvious how the activity is fully described by the conceptual information provided. Using the conceptual schemata (input and output) of the transformation and its provider source, one can identify the schemata of the respective activity either directly (input, output and functionality) or indirectly (generated and projected-out). Still, this is insufficient, because we do not get any information about the instantiation of the appropriate template activity. As we demonstrate later in this paper, this issue can be addressed using extra information adapted from a note attached to the conceptual transformation.

The conceptual model is not a workflow; instead, it simply identifies the mappings and the transformations needed in an ETL process. The placement of the transformations into the conceptual design does not directly specify their execution order. However, the logical model represents a workflow and thus, it is very important to determine the execution order of the activities. To tackle this, we provide a method for the semi-automatic determination of a correct execution order of the activities in the logical model, wherever this is feasible, by grouping the transformations of the conceptual design into stages of *order-equivalent* transformations.

Finally, with the goal of formalizing the mapping between the two models and dealing with the aforementioned problems, we present a sequence of steps that constitutes the method for the transition from the conceptual to the logical model.

1.1. Outline

The paper is organized as follows: In Sections 2 and 3, we briefly describe the main characteristics of the conceptual and logical models respectively. In Section 4, we discuss the mapping of each conceptual entity to a logical one. In Section 5, we present a semi-automatic determination of the execution order of the activities in the logical workflow. In Section 6, we provide a method for the realization of the mapping between the two models. In Section 7, we present related work. Finally, in Section 8 we conclude our discussion with a prospect of the future.

2. Conceptual model

In this section, we focus on the conceptual part-of the definition of the ETL process. In previous work [13,17], we had proposed a graph-based conceptual model for ETL processes. Here, after presenting a reference example, we present the main characteristics of our model along with its formal definition.

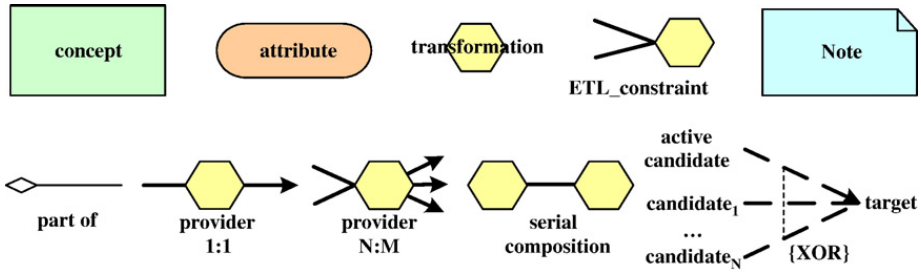

Fig. 2. Notation for the conceptual model.

2.1. Example

To motivate our discussion we introduce an example involving two source databases, S_1 and S_2, as well as a central data warehouse DW. The scenario involves the propagation of data from the concept PARTS of source S_1 as well as from the concept PARTS of source S_2 to the data warehouse. In the data warehouse, DW.PARTS stores daily (DATE) information for the available quantity (QTY) and cost (COST) of parts (PKEY). We assume that the first supplier is European and the second is American, thus the data coming from the second source needs to be converted to European values and formats. For the first supplier, we need to combine information from two different tables in the source database, which is achieved through an outer join of the concepts PS_1 and PS_2 respectively. Also, there exist two data sources – files RecentParts and AnnualParts – that contain information on daily and annual base respectively, for the population of the second supplier. In Fig. 1, we depict the full-fledged diagram of the example, in terms of our conceptual model. In Fig. 2, we graphically depict the different entities of the proposed model.

2.2. Model presentation

Each conceptual ETL scenario is represented as a graph G^C that we call the *Conceptual Graph*. The main entities of our model are the following:

Attributes, Ω^C. Attributes are the granular module of information. Their role is the same as in the standard ER/dimensional models (e.g., PKEY, DATE, COST).
Concepts, C^C. A concept represents an entity in the source databases or in the data warehouse (e.g., S1.PARTS, DW.PARTS).
Transformations, T^C. Transformations are abstractions that represent parts, or full modules of code, executing a single task and include two large categories: (a) filtering or data cleaning operations (e.g., not null (NN) check); and (b) transformation operations, during which the schema of the incoming data is transformed (e.g., surrogate key assignment (SK) transformation).
ETL Constraints, Cn^C. They are used in several occasions when the designer wants to express the fact that the data of a certain concept fulfill several requirements (e.g., to impose a PK constraint to DW.PARTS for the attributes PKEY and DATE).
Notes, N^C. Notes are used to capture extra comments that the designer wishes to make during the design phase or render constraints attached to an element or set of elements. A note in the conceptual model represents explanations of the semantics of the applied functions and/or simple comments explaining design decisions or constraints. We consider that the information for a note that is classified in the former category indicates either the type or an expression/condition of a function, and it is attached to a transformation or an ETL constraint. The information for a note of the second category is simple text without special semantics; these notes are used to cover different aspects of the ETL process, such as the time/event based scheduling, monitoring, logging, error handling, crash recovery and so on. Formally, a note is defined by: (a) a name, and (b) a content, which consists of one or more clauses of the form: ⟨type⟩::⟨text⟩.
We support three different types of information (clauses) in the content of a note using a simple prefix: (a) f:: for a function type; (b) e:: for an expression; and (c) t:: for simple text, before writing the text of the respective information. In order to keep our model simple, we do not obligate the designer to attach a note to every transformation or ETL constraint, or to fill all three types of information in every note.
In Fig. 1, observe several notes of the first category attached to transformations f_1, f_2, f_3, and $\bowtie_+$. For example, the note attached to the transformation $\bowtie_+$ indicates that the type of the transformation

template is `OuterJoin` and the expression needed for its instantiation is `PS1.PKEY+=PS2.PKEY`.

Part-of Relationships, $\mathbf{Po^C}$. Part-of relationships emphasize the fact that a concept is composed of a set of attributes, since we need attributes as first class citizens in the inter-attribute mappings.
Candidate relationships, $\mathbf{Cr^C}$. A set of candidate relationships captures the fact that a certain data warehouse concept (e.g., source table S_1) can be populated by more than one candidate source concepts (e.g., source files `AnnualParts` and `RecentParts`).
Active candidate relationships, $\mathbf{ACr^C}$. This relationship denotes the fact that out of a set of candidates, a certain one (e.g., source file `RecentParts`) has been selected for the population of a concept.
Provider relationships, $\mathbf{Pr^C}$. A 1:1 (N:M) provider relationship maps a (set of) input attribute(s) to a (set of) output attribute(s) through a relevant transformation.
Transformation Serial Composition, $\mathbf{T^C_{comp}}$. The composition is used when we need to combine several transformations in a single provider relationship (e.g., the combination of `SK` and γ).

The proposed model is constructed in a customizable and extensible manner, so that the designer can enrich it with his own re-occurring patterns for ETL activities, such as the assignment of surrogate keys or the check for null values.

2.3. Formal definition of the model

A model is much more than a diagrammatic notation. A conceptual model should be comprised of constructs and constraints expressed in a rigorous mathematical foundation, along with a user-friendly diagrammatic notation. In the rest of this section we present a rigorous definition for the constructs and constraints of our model to compensate for the shortcoming of previous work [17].

Assume the following infinitely countable pairwise disjoint sets of attribute names $\mathbf{\Omega}$, concept names $\mathbf{C}$, transformation names $\mathbf{T}$, constraint names $\mathbf{Cn}$, and note descriptions $\mathbf{N}$. A specific design, or *conceptual schema*, that obeys the proposed conceptual model comprises the following finite sets of constructs: attributes $\mathbf{\Omega^C} \subseteq \mathbf{\Omega}$, concepts $\mathbf{C^C} \subseteq \mathbf{C}$, transformations $\mathbf{T^C} \subseteq \mathbf{T}$, constraints $\mathbf{Cn^C} \subseteq \mathbf{Cn}$, and notes $\mathbf{N^C} \subseteq \mathbf{N}$.

The aforementioned constructs are related to each other in various forms. We define the following finite sets of pairs to capture the relationship among the constructs of a schema:

- Part-of relationships $\mathbf{Po^C}$ are first defined among attributes and their corresponding concepts, i.e., $\mathbf{Po^C} \subseteq (\mathbf{\Omega^C} \times \mathbf{C^C})$. Part-of relationships are also defined among notes and the transformations to which they correspond, i.e., $\mathbf{Po^C} \subseteq (\mathbf{N^C} \times \mathbf{T^C})$. Thus, $\mathbf{Po^C} \subseteq (\mathbf{\Omega^C} \times \mathbf{C^C}) \cup (\mathbf{N^C} \times \mathbf{T^C})$.
- Some concepts are acting as placeholders for more than one candidate concepts. Candidate relationships $\mathbf{Cr^C}$ are defined as pairs (*candidate*, *placeholder*) and formally this is captured by the set $\mathbf{Cr^C} \subseteq \mathbf{C^C} \times \mathbf{C^C}$. Some candidates are annotated as active ones, so we also assume a set of active relationships $\mathbf{ACr^C} \subseteq \mathbf{Cr^C}$.
- Finally, we need to interrelate source and target attributes. We employ provider relationships $\mathbf{Pr^C}$ for this task. Provider relationships are defined in the following four cases:
  - A source attribute is mapped to a target attribute.
  - A source attribute is mapped to a transformation or a transformation is mapped to a target attribute.
  - A transformation is mapped to a transformation (transformation serial composition).
  - A constraint is applied over the attributes of a concept.

  Formally, this is captured as $\mathbf{Pr^C} \subseteq (\mathbf{\Omega^C} \times \mathbf{\Omega^C}) \cup (\mathbf{\Omega^C} \times \mathbf{T^C}) \cup (\mathbf{T^C} \times \mathbf{\Omega^C}) \cup (\mathbf{T^C} \times \mathbf{T^C}) \cup (\mathbf{Cn^C} \times \mathbf{\Omega^C})$.

Formally, a conceptual schema is a directed graph $\mathbf{G^C} = (\mathbf{V^C}, \mathbf{E^C})$ that obeys the following constraints (we avoid overloading the mathematical notation to gain in readability):

1. $\mathbf{V^C} = \mathbf{\Omega^C} \cup \mathbf{C^C} \cup \mathbf{T^C} \cup \mathbf{Cn^C} \cup \mathbf{N^C}$.
2. $\mathbf{E^C} = \mathbf{Po^C} \cup \mathbf{Cr^C} \cup \mathbf{Pr^C}$.
3. Every attribute is connected to a single concept via a part-of edge.
4. Each member of a pair (*transformation*, *note*) does not participate in any other part-of relationship (i.e., for each transformation there exists exactly one note and for each such note, there exists exactly one transformation).
5. For each set of candidates that target the same placeholder relationship, exactly one active candidate can exist.

There are more correctness constraints that one can imagine. Still, these constraints mostly refer to the transformation process from the conceptual to the logical level. We choose to stop our consistency constraints here for reasons of flexibility. This decision based on our choice to provide a simple model with its main purpose

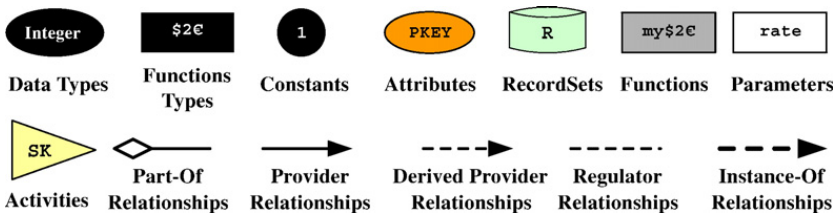

Fig. 3. Notation for the logical model.

of identification of the data stores and the transformations involved in the overall ETL process. This model is addressed not only to administrators, but also to managers and people with low expertise in data warehousing; thus, in order to facilitate the different groups in gaining an understanding of each other, we use a simple design language. Therefore, we allow cases of bad conceptual design, given that when the deliverable of this stage propagates to the next level (the logical design) it will be replenished and corrected wherever needed. Instead of discussing these kinds of cases here, we refer the reader to Section 5.1 where we provide an example of problematic design blended in the process of conceptual to logical transformation. This is also the reason for characterizing our process as semi-automatic.

3. Logical model

In this section, we present a condensed version of the logical model for ETL workflows that concentrates on the flow of data from the sources towards the data warehouse through the composition of activities and data stores. The full-blown version and the formal representation of the model can be found in [16,18,19].

The full layout of an ETL workflow, involving activities, recordsets and functions can be deployed along a graph $\mathbf{G^L}$ in an execution sequence that can be linearly serialized. We call this graph, the *Architecture Graph*. The graphical notation for the Architecture Graph is presented in Fig. 3.

As we have already stressed, since the conceptual model is constructed in a more generic and high-level manner, not all the logical entities have a mapping to a conceptual entity. In the logical modeling some entities are used in order to capture more detailed semantics of the logical design, e.g., the derived provider relationships, which can be determined only after the deliverable of this mapping has been produced. Thus, in this section, we present only the logical entities that can be mapped to a conceptual entity. In this setting, the components of our modeling framework are:

Attributes, $\mathbf{\Omega^L}$. They are characterized by their name and data type.

Recordsets, $\mathbf{RS^L}$. A recordset is characterized by its name, its (logical) schema and its (physical) extension (i.e., a finite set of records under the recordset schema).

Elementary Activities, $\mathbf{A^L}$. They are logical abstractions representing parts, or full modules of code. An *Elementary Activity* (simply referred to as *Activity* from now on) is formally described by the following elements:

- *Name*: a unique identifier for the activity.
- *Input Schemata*, $\mathtt{A^L.in}$: a finite list of one or more input schemata that receive data from the data providers of the activity.
- *Output Schemata*, $\mathtt{A^L.out}$: a finite list of one or more output schemata that describe the placeholders for the rows that pass the checks and transformations performed by the activity.
- *Functionality Schema*, $\mathtt{A^L.fun}$: a finite list of the attributes which take part in the computation performed by the activity (in fact, these are the parameters of the activity).
- *Generated Schema*, $\mathtt{A^L.gen}$: a finite list of attributes, belonging to the output schema(ta), that are generated due to the processing of the activity.
- *Projected-Out Schema*, $\mathtt{A^L}$.pro: a finite list of attributes, belonging to the input schema(ta), that are not further propagated from the activity.
- *Operational Semantics*: a program, in LDL++ [20], describing the content passing from the input schemata towards the output schemata. For example, the operational semantics can describe the content that the activity reads from a data provider through an input schema, the operation performed on these rows before they arrive to an output schema and an implicit mapping between the attributes of the input schema(ta) and the respective attributes of the output schema(ta).
- *Execution priority*. In the context of a scenario, an activity instance must have a priority of execution, determining when the activity will be initiated.

Provider relationships, $\mathbf{Pr^L}$. These relationships capture the mapping between the attributes of the schemata of the involved entities.

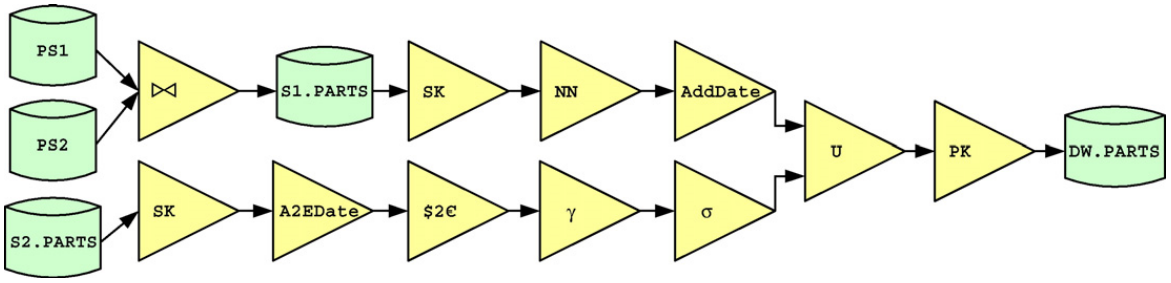

Fig. 4. Logical design of our example.

Part_of relationships, $\mathbf{Po}^{\mathbf{L}}$. These relationships involve attributes and parameters and relate them to their respective activity, recordset or function to which they belong.

Formally, the Architecture Graph is defined as $\mathbf{G}^{\mathbf{L}}=(\mathbf{V}^{\mathbf{L}},\mathbf{E}^{\mathbf{L}})$ where $\mathbf{V}^{\mathbf{L}}=\mathbf{\Omega}^{\mathbf{L}}\cup\mathbf{A}^{\mathbf{L}}\cup\mathbf{RS}^{\mathbf{L}}$ and $\mathbf{E}^{\mathbf{L}}=\mathbf{Pr}^{\mathbf{L}}\cup\mathbf{Po}^{\mathbf{L}}$. Fig. 4 depicts a simplified (due to the limited space) diagram of the example of Fig. 1, in terms of our logical model.

4. Mappings

In this section, we individually examine the constituents of the conceptual model, identify their respective logical entities, and describe how each conceptual entity is mapped to a logical one. Concepts and attributes are mapped to recordsets and attributes. Transformations and ETL constraints are mapped to activities. Notes are used for the determination and instantiation of the appropriate template activity. Moreover, we tackle two special design cases: the case of projected-out attributes and the convergence of two separate data flows at a common data store. Formally, we define a mapping from conceptual to logical models $\mathcal{M}_{\mathbf{CL}}:\mathbf{G}^{\mathbf{C}}\rightarrow\mathbf{G}^{\mathbf{L}}$. Table 1 summarizes how the conceptual entities are mapped to the logical ones.

4.1. Concepts and attributes

One of the main tasks of the conceptual model is to identify all data stores, along with their attributes, involved in the whole ETL process. For each concept in the conceptual model, a recordset is defined in the logical. The name and the list of attributes of the recordset are the same with those of the concept. There is one-to-one mapping from each attribute of the conceptual model to a respective one in the logical model; i.e., its name and data

Table 1
Mappings from conceptual to logical objects

Conceptual	$\mathbf{G}^{\mathbf{C}}(\mathbf{V}^{\mathbf{C}},\mathbf{E}^{\mathbf{C}})$	Logical	$\mathbf{G}^{\mathbf{L}}(\mathbf{V}^{\mathbf{L}},\mathbf{E}^{\mathbf{L}})$	$\mathcal{M}_{\mathbf{CL}}:\mathbf{G}^{\mathbf{C}}\rightarrow\mathbf{G}^{\mathbf{L}}$
Attributes	$\mathbf{\Omega}^{\mathbf{C}}$	Attributes	$\mathbf{\Omega}^{\mathbf{L}}$	$\mathbf{\Omega}^{L}=\{w\|\forall\omega\in\mathbf{\Omega}^{\mathbf{C}},w=\mathcal{M}_{\mathbf{CL}}(\omega)\}$
Concepts	$\mathbf{C}^{\mathbf{C}}$	Recordsets	$\mathbf{RS}^{\mathbf{L}}$	$\mathbf{RS}^{\mathbf{L}}=\{r\|\forall c\in\mathbf{C}^{\mathbf{C}},r=\mathcal{M}_{\mathbf{CL}}(c)\}$
Transformations	$\mathbf{T}^{\mathbf{C}}$	Activities	$\mathbf{A}^{\mathbf{L}}$	$\mathbf{A}^{\mathbf{L}}=\{a\|\forall t\in\mathbf{T}^{\mathbf{C}},\forall n\in\mathbf{N}^{\mathbf{C}},a=(\mathcal{M}_{\mathbf{CL}}(t),\mathcal{M}_{\mathbf{CL}}(n))\}\cup\{a\|\forall c\in\mathbf{Cn}^{\mathbf{C}},a=(\mathcal{M}_{\mathbf{CL}}(c)\}$
Notes	$\mathbf{N}^{\mathbf{C}}$			
Constraints	$\mathbf{Cn}^{\mathbf{C}}$			
Part-of	$\mathbf{Po}^{\mathbf{C}}$	Part-of	$\mathbf{Po}^{\mathbf{L}}$	$\mathbf{Po}^{\mathbf{L}}=\{y\|\forall x\in\mathbf{Po}^{\mathbf{C}},x=(c,\omega),c\in\mathbf{C}^{\mathbf{C}},\omega\in\mathbf{\Omega}^{\mathbf{C}},y=\mathcal{M}_{\mathbf{CL}}(x)=(\mathcal{M}_{\mathbf{CL}}(c),\mathcal{M}_{\mathbf{CL}}(\omega))\}$
Provider	$\mathbf{Pr}^{\mathbf{C}}$	Provider	$\mathbf{Pr}^{\mathbf{L}}$	$\mathbf{Pr}^{\mathbf{L}}=\{y\|\forall x\in\mathbf{Pr}^{\mathbf{C}},x=(\omega_1,\omega_2),\omega_1,\omega_2\in\mathbf{\Omega}^{\mathbf{C}},y=\mathcal{M}_{\mathbf{CL}}(x)=(\mathcal{M}_{\mathbf{CL}}(\omega_1),\mathcal{M}_{\mathbf{CL}}(\omega_2))\}$
Candidate	$\mathbf{Cr}^{\mathbf{C}}$	–	–	–
Active candidate	$\mathbf{ACr}^{\mathbf{C}}$	–	–	–

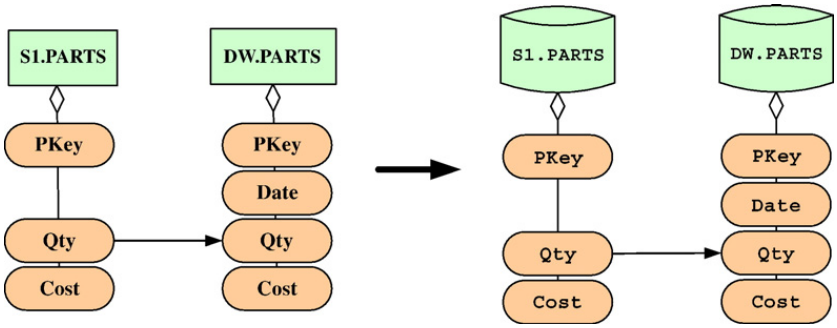

Fig. 5. Transition of a simple provider relationship.

type remain the same. Fig. 5 depicts the transition of concepts `S1.PARTS` and `DW.PARTS` to the respective recordsets along with its attributes.

Formally, for each concept attribute we introduce a recordset attribute and the mapping of attributes is defined as: $\boldsymbol{\Omega}^{\mathbf{L}} = \{w|\forall\omega\in\boldsymbol{\Omega}^{\mathbf{C}}, w = \boldsymbol{\mathcal{M}}_{\mathbf{CL}}(\omega)\}$. Additionally, for each concept we introduce a recordset and the mapping of concepts to recordsets is defined as: $\mathbf{RS}^{\mathbf{L}} = \{r|\forall c\in\mathbf{C}^{\mathbf{C}}, r = \boldsymbol{\mathcal{M}}_{\mathbf{CL}}(c)\}$.

4.2. Relationships

The conceptual model consists of four kinds of relationships: part-of, candidate, active candidate, and provider relationships.

The part-of relationships are used to denote the fact that a certain concept comprises a set of attributes; i.e., that we treat attributes as 'first-class citizens' in our model. We maintain this characteristic in the logical model too; thus, the conceptual part-of relationships are mapped to logical part-of relationships, with exactly the same semantics and characteristics. Observe the usage of part-of relationships in Fig. 5. Formally, for each part-of edge that connects an attribute with its container concept, we introduce a part-of edge and the respective mapping is defined as: $\mathbf{Po}^{\mathbf{L}} = \{y|\forall x\in\mathbf{Po}^{\mathbf{C}},\ x = (c,\omega), c\in\mathbf{C}^{\mathbf{C}}, \omega\in\boldsymbol{\Omega}^{\mathbf{C}}, y = \boldsymbol{\mathcal{M}}_{\mathbf{CL}}(x) = (\boldsymbol{\mathcal{M}}_{\mathbf{CL}}(c),\ \boldsymbol{\mathcal{M}}_{\mathbf{CL}}(\omega))\}$. The part-of edges that connect notes with transformations are not further propagated to the logical model.

The candidate and the active candidate relationships are not directly mapped to the logical model. Their introduction in the conceptual model covers the usual case that in the early stages of the data warehouse design there may exist more than one candidate concept (data stores) for the population of a certain concept. When we move on to the logical design, these problems have already been solved at the previous steps of the lifecycle [13]. Therefore, we only need to transform the active candidate concept; i.e., the one that is chosen, to a logical recordset.

The provider relationships intuitively represent the flow of data during an ETL process. Consequently, a conceptual provider relationship between a source and a target attribute involves all the transformations that should be applied according to design requirements. In the absence of any transformation between a source and a target, the conceptual provider relationship can be directly mapped to a logical provider relationship (Fig. 5). This is the case of provider relationships between attributes and the respective mapping is defined as: $\mathbf{Pr}^{\mathbf{L}} = \{y|x = (\omega_1,\omega_2), \omega_1,\omega_2\in\boldsymbol{\Omega}^{\mathbf{C}},\ y = \boldsymbol{\mathcal{M}}_{\mathbf{CL}}(x) = (\boldsymbol{\mathcal{M}}_{\mathbf{CL}}(\omega_1), \boldsymbol{\mathcal{M}}_{\mathbf{CL}}(\omega_2))\}$. The case where one or more transformations are needed between source and target attributes is covered in the next subsection.

4.3. Conceptual transformations

In the conceptual model, we use transformations in order to represent tasks that either: (a) maintain the schema of data (e.g., cleansing, filtering); in general, we name these transformations *filters*, or (b) change the schema of data (e.g., aggregation); we name these transformations *transformers*. In the logical level, we use activities to represent the same tasks. Thus, the mapping of conceptual to logical model includes the mapping of conceptual transformations to logical activities.

By observing the characteristics of the logical model and comparing the nature of its constructs to the constructs of the conceptual model, we understand that in order to fully describe the mapping between transformations and activities, we have to deal with three main issues:

- The specification of the *properties* of an activity (e.g., name, schemata, semantics).
- The *serial composition* of conceptual transformations.
- The definition of the *execution order* of activities in the logical model.

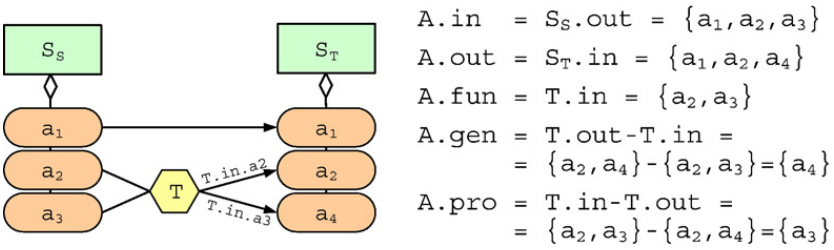

Fig. 6. Mapping of a transformer T to an activity A.

In this subsection, we deal with the first two issues. The latter is discussed later in the paper.

4.3.1. Properties of an activity

Formally, a transformation T in the conceptual level (either a filter or a transformer) is mapped to an activity A in the logical level. The input attributes of T (T.in) comprise the functionality schema of the respective activity A (A.fun). If there are attributes belonging to the output schema of T (T.out) that do not have a provider into the input schema of T, then these attributes comprise the generated schema of the activity (A.gen). Similarly, if there exist any attributes belonging to the input schema of T that do not have a consumer in the output schema of T, then these attributes comprise the projected-out schema of the activity (A.pro). In the simple case of an ETL process that involves only one transformer, the attributes of the source concept (S_S) of T comprise the input schema of activity A (A.in) and the attributes of the target concept (S_T) of T comprise the output schema of activity A (A.out). So the following formulae are valid:

$$A.in = S_S.out$$
$$A.out = S_T.in$$
$$A.fun = T.in$$
$$A.gen = T.out - T.in$$
$$A.pro = T.in - T.out$$

Obviously, for filters A.gen=∅ and A.pro=∅ hold. Fig. 6 depicts an example of the determination of activity's schemata for a transformer.

Note that in more complex cases, the input and output schemata are not computed so easily. In [14], we discuss how we confront such cases and present an algorithm called Schema Generation that automates the creation of all input and output schemata involved in the whole ETL process.

4.3.2. Serial composition

In the conceptual model, when we need to combine several transformations in a single provider relationship (e.g., the combination of SK_1 and γ in Fig. 1), we apply a serial composition. The mapping of the serial composition to the logical model is quite straightforward. The serial composition of the two transformations T_1 and T_2 is mapped to a sequence of two activities A_1 and A_2 in the logical level. The execution order of the two activities is determined from the order of the respective transformations in the serial composition. The schemata of the two activities are defined as we have already seen. The only difference is that the output schema of the initial activity A_1 will populate the input schema of the subsequent activity A_2, i.e., $A_2.in = A_1.out$.

4.4. Transformation of notes

So far, we have described how the schemata of an activity, along with the appropriate inter-attribute mappings are determined. The next step towards the complete description of an activity is the identification of its operational semantics; i.e., the appropriate LDL++ program that describes its operation. In previous work [19], we provided a generic and extensible palette of template activities, and presented how the expression of an activity, which is based on a certain simple template, is produced by a set of LDL++ rules of the following form that we call *Template Form*:

```
OUTPUT()
  <- INPUT(), EXPRESSION, MAPPING.
```

Clearly, the OUTPUT, INPUT and MAPPING parts of the above form have already been covered by the mapping of a conceptual transformation to a logical activity previously presented. In this subsection, we concretely describe: (a) how the designer chooses a template activity, and (b) what the EXPRESSION part is.

To succeed in this task, we take advantage of the usage of the notes attached to transformations. After parsing the information of a note, we get extra information concerning the transformation involved: (a) its function type (clause f::); (b) the expression(s) needed for the instantiation of its function type

Template	a_out(OUTPUT_SCHEMA, @OUTFIELD) <- a_in1(INPUT_SCHEMA), @OUTFIELD = @VALUE, DEFAULT_MAPPING.
Parameter instantiation	@OUTFIELD = DATE @VALUE = SYSDATE
Operational semantics of activity f_3	f3_out(PKEY_out,QTY_out,COST_out,DATE) <- f3_in1(PKEY_in,QTY_in,COST_in), DATE = SYSDATE, PKEY_out = PKEY_in, QTY_out = QTY_in, COST_out = COST_in.

Fig. 7. Operational semantics of activity f_3.

(clause e::); and (c) design decisions or constraints (clause t::).

Using the type provided by the f:: clause, we chose a template activity from a template library. In the case that no template of such type exists in the library, the designer can either register a new one or solve this problem later in the logical design. In each template there are one or more parts that should be determined during its instantiation. This job is facilitated by the e:: clause. A properly completed e:: clause indicates the requisite expression(-s) for a template. If more than one predicate is used in the template definition; i.e., it is a program-based template [19], then it is possible that more than one expression will be required. Also, even a single predicate template is possible to have more than one expression in its definition. In such cases, the respective note should provide all the expressions needed in the order they should be placed into the template definition. If a note does not contain all the appropriate expressions or if it does not contain them in the right order or even if it contains them improperly completed, then the template activity cannot be automatically constructed and further interference from the designer is required.

For instance, consider the transformation f_3 depicted in Fig. 1. This transformation represents a function tagged by a note that provides twofold information about: (a) its type: f::addAttribute; and (b) its required EXPRESSION: e::DATE=SYSDATE. Fig. 7 shows how to use this information to instantiate the operational semantics of activity f_3. We use the f:: information to choose the appropriate template for the logical activity. Thus, the conceptual transformation f_3 is mapped to the logical activity f_3 whose operational semantics are determined by the template activity addAttribute (1st row). The EXPRESSION part-of addAttribute is @OUTFIELD=@VALUE; i.e., the new attribute (OUTFIELD) takes an initial value (VALUE). For transformation f_3, the EXPRESSION is: e::DATE=SYSDATE. After the parameter instantiation (2nd row) we get the operational semantics of the activity f_3 (3rd row).

For further information concerning the usage of function types, templates, and expressions we refer the interested reader to [19].

Finally, the t:: information is not directly used in the logical model. The designer exploits such information, as long as it is still useful in the logical design, to annotate the logical workflow with informal notices concerning several aspects of the ETL process, such as the time/event based scheduling, monitoring, logging, error handling, crash recovery, and/or runtime constraints.

4.5. Transformation of ETL constraints

In the conceptual model, ETL Constraints are used to indicate that the data of a certain concept fulfill several requirements. For instance, in Fig. 1, we impose a PK constraint to DW.PARTS for the attributes PKEY and DATE. The functionality of an ETL constraint is semantically described from the single transformation that implements the enforcement of the constraint. Thus, we treat conceptual ETL constraints as conceptual transformations and we convert them into logical activities. More specifically, each ETL constraint enforced to some attributes of a concept S_i is transformed to an activity A that takes place in the logical workflow *exactly before* the respective recordset S_i that stands for the concept S_i. The schemata of the activity are filled in the same manner and under the same procedures as in the case of the mapping of conceptual transformations. The finite set of attributes of S_i, over which the constraint is imposed, constitutes the functionality schema of A. The input and output schemata of A comprise of all the attributes of S_i.

Formally, for each conceptual transformation, along with its attached note, and for each conceptual

constraint, we introduce a logical activity and the mapping is defined as: $\mathbf{A^L} = \{a|\forall t \in \mathbf{T^C},\ \forall n \in \mathbf{N^C},\ a = (\mathcal{M}_{\mathbf{CL}}(t), \mathcal{M}_{\mathbf{CL}}(n))\} \cup \{a|\forall c \in \mathbf{Cn^C}, a = (\mathcal{M}_{\mathbf{CL}}(c)\}$.

4.6. Special cases

In this subsection, we tackle two design issues that arise in the mapping of conceptual to logical model.

4.6.1. Rejection of an attribute

During an ETL process some source attributes may be projected-out from the flow. We discriminate two possible cases:

- *An attribute that belongs to the schema of a concept*. This case is covered by the conceptual model as an attribute that has not an output provider edge (e.g., in Fig. 1, the attribute S_2.PARTS.DEPT is not further propagated towards DW).
- *An attribute that participates in a certain transformation but it is not further propagated*. This case is covered by the conceptual model as an attribute that belongs to the input schema of a transformation, but there is no output edge tagged with the name of the discarded attribute from the transformation to any attribute (e.g., in Fig. 1, the attribute PS_1.DEPT participates to the outer join, but is not further propagated towards S_1.PARTS).

In the mapping of conceptual to logical, the first case is handled with the addition of an extra activity that projects-out the appropriate attribute(s); this activity should be placed immediately after the respective recordset. On the other hand, the second case should not be examined as a special case. The reason behind this is that the inter-activity discarding of attributes is captured by the semantics of the activity and the attributes discarded are simply belonging to the projected-out schema of the activity. Thus, this case is covered by the conversion of conceptual transformations to logical activities.

4.6.2. Convergence of two flows

In the conceptual model, the population of a concept (e.g., DW.PARTS) from more than one source is abstractly denoted by provider edges that simply point at this concept. The logical model, that is more rigorous, needs a more specific approach to cover the population of a recordset from more than one source. A solution to this is the addition of an extra activity that unifies the different flows. Therefore, the convergence of two (or more) flows is captured in the logical model with the usage of a union (U) activity. Obviously, before the addition of a union activity, each of the involved flows populates the same data store, i.e., the same schema. Thus, the input schemata of the union activity are identical. Since a union has empty functionality, generated and projected-out schemata, its output schema is the same with any of its input schemata and also, it is the same with the schema of the target recordset.

4.6.3. Presentation issue

For clarity of presentation, in what follows, we avoid using the formal notation previously presented. For instance, when we write 'a graph $\mathbf{G'}$ instead of 'a graph $\mathbf{G^{C\prime}}$ or 'a graph $\mathbf{G^{L\prime}}$, the related description will clarify whether we refer to a conceptual or to a logical graph.

5. Execution order

So far, we have clarified that from the conceptual model of an ETL process, one can: (a) identify the concerned data stores; (b) pick out the transformations that need to take place in the overall process; and (c) describe the inter-attribute mappings. But a further, more detailed, study of the data flow generates some questions concerning the execution order of the activities in the logical workflow. Consider the part-of Fig. 1 that involves the population of DW.PARTS from S2.PARTS. In order to design a logical workflow for this example, we have to answer questions like 'Q1: which of the activities SK_1 and γ should precede?' or 'Q2: which of the activities SK_1 and f_1 should precede?'

This section presents a method for determining the execution order of activities in the logical workflow. At first, without loss of generality, we examine the simple case of the population of one target concept from one source. With the intention of classifying the transformations according to their placement into the design, we define transformation stages and we give an algorithm for finding them. Also, we discuss issues concerning the ordering of transformations included in the same stage. Afterwards, we extend this method to capture more complex and realistic cases involving more than two data stores.

5.1. Stages

Assume the case of simple 'one source-one target' flow, like the example depicted in Fig. 8. The motivating question Q1 can be answered by observing the information presented in Fig. 8. Since the transformation γ has input attributes that belong to the output schema of SK_1, f_1 and f_2, it should follow them at the logical level. For a similar reason, the activity σ should follow γ. Moreover, after the presentation of the special

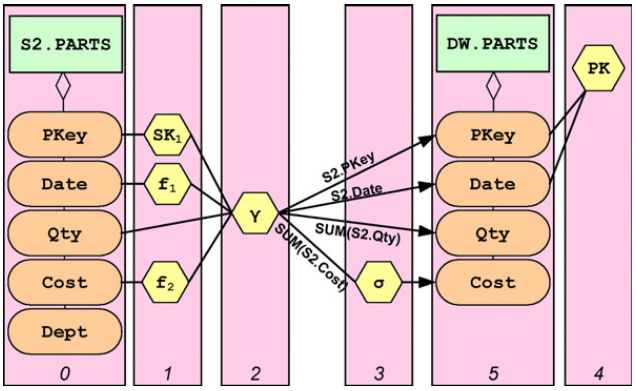

Fig. 8. Transformation stages.

cases in subsection 4.6, we should also take into account two more activities: (a) the ETL constraint PK; recall that when we transform an ETL constraint into an activity, we should place it exactly before the recordset involved (DW.PARTS); and (b) the 'hidden' activity that projects-out the attribute DEPT; recall that this activity should be placed exactly after the source concept (S_2.PARTS).

Intuitively, one can divide the design into several groups of transformations, (see Fig. 8), mainly taking into consideration the following observations: (a) the position of some activities in the workflow can be determined by detecting their providers and consumers; and (b) extra activities can be placed in the workflow by applying the mapping rules for ETL constraints and special cases.

5.1.1. Stages

The aforementioned observations guide us to divide the design into several *transformation stages*. A *transformation stage* (or a *stage*) is a visual region in a 'one source-one target' conceptual design that comprises: (a) a concept and its attributes (either the source or the target concept); or (b) a set of transformations that act within this region. A stage is shown as a rectangle labeled at the bottom with a unique stage identifier: an automatically incremented integer with initial value equal to 0, where stage 0 comprises only the source concept. The next subsection clarifies how stages are derived and exploited.

5.1.2. Position of activities in different stages

Assume two conceptual transformations T_i and T_j. Then, the execution order of their respective activities A_i and A_j in the logical model is computed as follows: When T_i belongs to a stage with smaller identifier than the stage of T_j, then A_i has smaller execution priority (i.e., it is executed prior) than A_j.

We have answered the question of determining the execution priorities of activities in different stages. Another issue which remains to be clarified is the determination of the execution priority among transformations that belong to the same stage. The transformations that belong to the same stage are called *stage-equivalent transformations*.

5.1.3. Position of activities within the same stage

We tackle this problem using the following thought: If all activities that stem from stage-equivalent transformations are swappable, then there is no problem in their ordering. Consequently, we answer questions concerning the ordering of activities of this kind, by studying which of these activities can be swapped with each other. For the rest of activities, extra action from the designer is needed. In [14] we resolve the issue in which two activities can be swapped in a logical ETL workflow, i.e., when we are able to interchange their execution priorities. Also, we provide a formal proof that the swapping of two activities A_1 and A_2 is allowed, when the following conditions hold:

1. A_1 and A_2 are adjacent in the graph; without loss of generality assume that A_1 is a provider for A_2.
2. Both A_1 and A_2 have a single input and output schemata and their output schema has exactly one consumer.
3. The functionality schema of A_1 and A_2 is a subset of their input schema, both before and after the swapping.
4. The input schemata of A_1 and A_2 are subsets of their providers, again both before and after the swapping.

All transformations belonging to the same stage can be adjacent and this is valid for their respective activities too; thus condition (1) holds. Since stages are defined between two concepts, in a single stage all transformations, as well as their respective activities, have a single input schema and a single output schema; thus condition (2) holds too. Therefore, we have to examine the validity of conditions (3) and (4), in order to decide if two transformations belonging to the same stage are swappable or not.

The transformations that belong to the same stage, and whose respective activities in the logical design are swappable, are called *order-equivalent transformations*. For two order-equivalent transformations A_i and A_{i-1} the following formulae hold:

$$A_i.fun \subseteq A_i.in$$

$$A_i.in \subseteq A_{i-1}.out.$$

The above formulae represent conditions (3) and (4) respectively. If these formulae do not hold for two transformations belonging to the same stage, then we

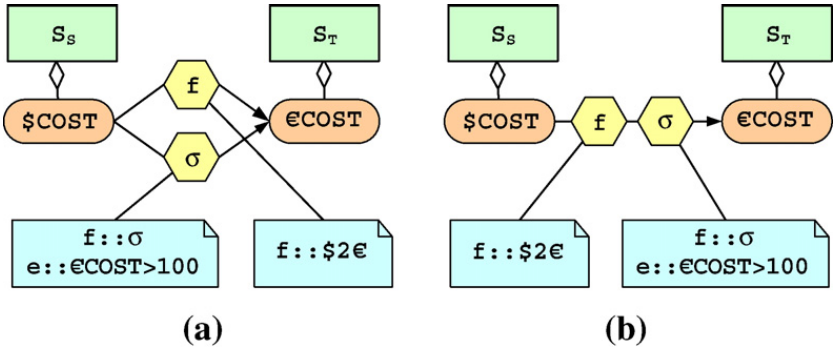

Fig. 9. Cases of bad (a) and correct (b) design.

cannot automatically decide their order and we need additional information from the user.

5.1.4. Cases of bad design

Clearly, this extra information is required only in cases of bad design. Consider the example of Fig. 9. In the left side, part (a), we discern two stage-equivalent transformations: f and σ. Assume that concept S_S contains costs in dollars. The first transformation converts dollar costs to euro values and the second filters out values below the threshold of 100€. Obviously, if the filter has greater execution priority than the function, then there is a semantic problem; the result of filtering dollars with a euro value threshold is not eligible. Besides, as we explain thoroughly in [14], the condition (3) fires: if the filter is applied first, then σ.fun={€COST} ≠ σ.in={$COST}. Therefore, these two transformations are not order-equivalent and the designer will be asked for the right order. If the designer creates the blueprint depicted in Fig. 9(b), then the ordering is explicitly defined, since both transformations belong to subsequent stages.

5.2. Stage derivation

We now present the FS algorithm (Fig. 10) that is used for the automatic determination of all stages of a conceptual design which involves a source and a target concept. The FS algorithm accomplishes the following tasks: (a) first, it finds out all the attributes that should be projected-out in the terms introduced in subsection 4.6; and (b) then, it divides the design into several stages, by exploiting the observations about the provider(-s) and

Algorithm Find Stages (FS)

1. **Input:** A graph $G_p=(V', E')$, where V' contains a source concept C_S, a target concept C_T, their attributes, and a set **AT** of all transformations between C_S and C_T.
2. **Output:** An array $Stage_{C_S,C_T}[id]$, id=0,…,n, where n is the number of all possible stages between C_S and C_T.
3. **Begin**
4. id = 0;
5. Stage[id] = {C_S};
6. id++;
7. **for each** attribute $a_j \in$ schema(C_S) {
8. **if** ($\forall x \in (V'-C_S), \neg \exists$edge($a_j$,x)) { // if a concept attribute does not have any consumer
9. Stage[id] = {π-out$_{a_j}$}; // then discard it from the flow (add a π-out activity into the current Stage
10. }}
11. **while** (AT ≠ ∅) {
12. id++; // a new id for a new stage
13. **for each** transformation $T_i \in$ AT {
14. **if** ($\forall (x,T_i) \in E'$, $x \in (V'-T_i)$, $x \in$ Stage[id-k] // if all the providers of T_i or $x \in$ schema(C_s), $C_s \in$ Stage[id-k], k=1,…,id) // belong to previous stages
15. AT = AT − {T_i}; // remove T_i from the list of transformations
16. Stage[id] = Stage[id] ∪ {T_i}; // add T_i into the current Stage
17. }}}
18. Stage[++id] = {C_T}; // the target concept is added to the last Stage
19. return Stage[]; // return the array Stage[ ]
20. **End.**

Fig. 10. The FS algorithm.

the consumer(-s) of each transformation. The input of the algorithm is a subgraph $\mathbf{G_p}=(\mathbf{V'},\mathbf{E'})$ of the whole conceptual design $\mathbf{G}=(\mathbf{V},\mathbf{E})$; i.e., $\mathbf{V'}\subseteq\mathbf{V}$ and $\mathbf{E'}\subseteq\mathbf{E}$, that comprises a source concept C_S, a target concept C_T, their respective attributes, and all transformations between these two concepts. In the algorithm, we consider all transformations of $\mathbf{G_p}$ as a set **AT**, the creation of which is accounted as a trivial programming task. The FS outputs the set of stages of the 'one source-one target' flow concerning the concepts $\mathbf{C_S}$ and $\mathbf{C_T}$. We use an array of sets, named Stage[], to store all these stages. Each element id of Stage[] is a set that represents a stage, and it contains the transformations (or concepts) belonging to the stage with id equal to id.

The source concept is placed in stage 0 (Ln: 5). All the project-out cases are calculated first (Ln: 7–10). We are interested only in the attributes of the source concept (see subsection 4.6) that do not have a consumer to populate (i.e., they do not have a provider relationship). All project-out's are placed in the same stage (Ln: 9), because, obviously, they are order-equivalent transformations. Then, the FS calculates the rest of transformations (Ln: 11–17). We check all transformations in **AT**. If all the input attributes of a certain transformation have providers in a previously defined stage, then this transformation is removed from **AT** and placed to the current stage. In other words, if all providers of a certain transformation belong to previous stages, i.e., stages with smaller id than the current stage, then this transformation should be placed in the current stage (Ln: 14–17). If not, then we continue with the next transformation, until we check all transformations belonging to **AT**. Then, we proceed to the next stage and we check again the rest of transformations in **AT**, until **AT** becomes empty. Finally, we add the last stage that contains the target concept, in Stage[] (Ln: 18) and FS returns all stages of the 'one source-one target' flow concerning the concepts C_S and C_T in the array Stage[] (Ln: 19).

5.2.1. Correctness of the FS algorithm

Theorem 1 guarantees that the FS algorithm is correct, in the sense that it produces correct stages. Specifically, the theorem guarantees that all nodes, belonging to a simple 'one source — one target' flow, are placed to the stage that follows the maximal stage of their providers. In other words, if a node *n* has providers that belong to stages $s_1, s_2, \ldots, s_k$, with s_k being the latest stage, then node *n* is placed at stage s_{k+1}.

Theorem 1. *Every node of a simple 'one source — one target' flow is placed in a stage as soon as all its prerequisites are met; i.e., all its providers have already placed in previous stages.*

Proof. The validity of this theorem is assured from the if-statement of Ln: 14. Every transformation T_i is placed into the exact next stage than its provider(-s) which is(-are) placed in the stage with the greater id among all the other provides of T_i. Each loop of the while statement of Ln: 11 checks all the activities. Each time, the stage changes and the ones that are selected are the activities whose providers are all assigned to previous stages. No transformation T_i may be placed in an earlier or a later stage. Additionally, the other two nodes, the source and target concepts, that exist in this simple flow are placed in the first (Ln: 5) and in the last (Ln: 18) stage respectively. □

5.3. Stages in designs involving binary transformations

Up to now, we have dealt with simple 'one source–one target' ETL processes. We extend this assumption by taking into account the binary transformations that combine more than one flow. In this case, we follow a threefold procedure for finding the execution order of the activities in the overall workflow: (a) we compute the stages of each separate 'one source–one target' flow; (b) we construct the linear logical workflows; and (c) we unify them into one workflow. The union of two workflows is realized on their common node; i.e., either a recordset or a binary transformation, which is called the *joint–point* of the two workflows. As we have already discussed in subsection 4.6, if the joint–point is a data store, then for the convergence of two workflows we need an extra union activity (U) that should be placed exactly before the joint point; i.e., the common data store. If the joint–point is a binary transformation then we simply unify the two flows on that node.

At first, we give an intuitive description of how we deal with complex designs involving more than one source, and then present an algorithm for the formal finding of stages in such complex designs. Without loss of generality, assume the case of 'two sources–one target' flow. Obviously, for each binary transformation T_b there are two providers and one consumer. These three entities can be either concepts or transformations depending each time from the position of T_b in the workflow. Assume the case of the population of a target data store DW from two source data stores S_1 and S_2 (Fig. 11(a)) through several unary transformations and one binary transformation T_b. We use an incremental technique for the design of the logical workflow. At first, we compute $\mathrm{Stage}_{S1,DW}$ and $\mathrm{Stage}_{S2,DW}$ that contain the stages of the 'one source–one target' flows defined by the two pairs of concepts S_1–DW and S_2–DW. To find the appropriate order in the logical workflow when a binary transformation T_b is involved,

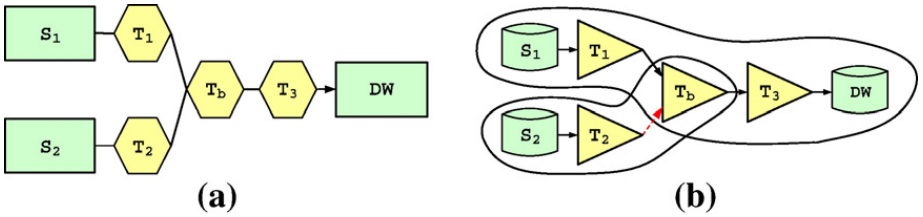

Fig. 11. The case of a binary transformation.

we select: a) all the transformations that take place in the flow from the first source S_1 to the target concept DW; and b) a subset of the transformations from the second flow that consists of all the transformations between the second source concept S_2 and the binary transformation T_b. Thus, from $Stage_{S1,DW}$ we use all its elements:

$$Stage_{S1,}DW[i], i = 0, \ldots, n \text{ where } Stage_{S1,}DW[n] = DW$$

while from $Stage_{S2,DW}$ we use only the elements:

$$Stage_{S2,DW}[i], i = 0, \ldots, k \text{ where } Stage_{S2,DW}[k] = T_b.$$

Finally, we unify the two flows on their joint–point, the activity T_b (dashed arrow in Fig. 11(b)).

For example, in Fig. 11 we have:

for S_1–DW:	for S_2–DW:
$Stage_{S1,DW}[0] = \{S_1\}$	$Stage_{S2,DW}[0] = \{S_2\}$
$Stage_{S1,DW}[1] = \{T_1\}$	$Stage_{S2,DW}[1] = \{T_2\}$
$Stage_{S1,DW}[2] = \{T_b\}$	$Stage_{S2,DW}[2] = \{T_b\}$
$Stage_{S1,DW}[3] = \{T_3\}$	$Stage_{S2,DW}[3] = \{T_3\}$
$Stage_{S1,DW}[4] = \{DW\}$	$Stage_{S2,DW}[4] = \{DW\}$

In this example $n=4$ and $k=2$, and thus, from the second flow we select only the first three stages (0..2). We do not compute all stages of the second flow, rather we compute only the k first stages ($0..k-1$). We construct the logical flow S_1–DW based on all five stages (0..4) and the logical flow S_2–DW based only on the first three stages (0..2). The two workflows are depicted in Fig. 11(b). Finally, we unify the two flows on their joint–point, the activity T_b (dashed arrow in Fig. 11(b)). The following proposition guarantees that we do not lose any transformation in this process.

Proposition. *If two flows,* S_1–S_T *and* S_2–S_T *that involve the population of a target data store* S_T *from two source data stores* S_1 *and* S_2*, respectively, have a binary transformation* T_b *as a joint point, then the sub-flow* T_b–S_T *is common in both initial flows.*

Proof. The proof is straightforward, since, by definition, each transformation has exactly one output schema. For instance, the situation depicted in Fig. 12 is unacceptable, because the only way to have different flows after a binary transformation is to allow it to have two output schemas; and this is not valid. □

Next, we present an algorithm for *Finding Stages in Complex Conceptual Designs*. The FSC algorithm (Fig. 13) formally determines the execution order of activities in workflows that involve more than one source. It checks all the flows from each source to the target. It applies the FS algorithm to each *simple* flow; i.e., to a flow that does not involve a binary transformation. If the flow is *complex*; i.e., a flow that involves at least one binary transformation, then each time FSC keeps track of binary transformations, and again, it applies the FS algorithm.

The FSC algorithm takes as input a directed graph **G**=(**V**,**E**) that represents a conceptual design involving probably more than one source along with binary transformations and produces an array that contains all the individual simple flows, each one in the form of a Stage[] array, that are needed for the construction of the logical workflow. At first, the algorithm checks every possible flow that may be created by any two concepts; a source C_S and a target C_T that belong to **G** (Ln: 6). It becomes obvious from the aforementioned intuitive analysis, that the main goal is to *find the*

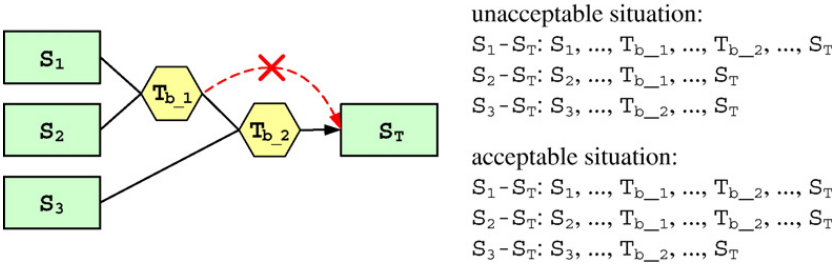

Fig. 12. Problematic conceptual design.

Algorithm Find Stages in Complex Conceptual Designs (FSC)

```
1.  Input: A graph G=(V,E) that represents the whole conceptual design
2.  Output: An array LW[id], that contains the stages of all the individual simple flows that may be produced in G
3.  Begin
4.    visited_Tb = {};
5.    id = 0;
6.    for each pair CS,CT {
7.      Target = CT;
8.      AT = find_all_transformations(CS,CT);
9.      for each binary transformation Tb∈AT {
10.       if ( Tb∈visited_Tb ) { Target = Tb; }
11.         else { visited_Tb = visited_Tb ∪ {Tb}; }
12.     }
13.     LW[++id]=FS(G'Cs-Target(V',E'), s.t. G'Cs-CT is the graph that represents the
        simple flow between CS and Target);
14.   }
15.   return LW[];
16. End.
```

Fig. 13. The FSC algorithm.

boundaries of each flow. Clearly, we search only for the target of a flow, given that the source is always known. Initially, in each flow the first concept C_S is the source and the second C_T is the target (Ln: 7). Then we find all transformations of the flow C_S–C_T as a set **AT**, the creation of which (function `find_all_transformations(⟨source⟩,⟨target⟩)`) is accounted as a trivial programming task (Ln: 8). If there does not exist any binary transformation in this flow then the target is still the second concept. If there exists a binary transformation T_b, then we confront this in a twofold way (Ln: 9–12). The very first time that we find T_b, we simply add it to a set, named `visited_`T_b, and the ordering is determined from the whole flow. In the next appearance (or appearances, if we consider that an n-ary transformation can have more than two input schemata) of T_b we use the binary transformation T_b as the target; in this case, the second concept is not considered anymore as the target of the current flow checked (Ln: 10). This is because the flow from T_b towards the second concept has been determined before, at the first appearance of T_b. The set `visited_`T_b is used for storing every binary transformation counted so far in its first appearance.

After having determined the target, we have fixed the boundaries of the flow that should be checked in order to determine the execution order of the logical activities. Once again, this is achieved through the application of the `FS` algorithm to the flow between the source concept and the target chosen (Ln: 13). The `FS` algorithm outputs the array `Stage[]` (see previous subsection) that contains the necessary information about the execution order of a certain flow. All arrays `Stage[]` of all the conceptual flows examined are stored in another array `LW[]`.

When `FSC` finishes, it returns the array `LW[]` that contains all the individual simple flows, each one in the form of a `Stage[]` array, that are needed for the construction of the logical workflow (Ln: 19). We should note here, that the term 'construction' refers to the determination of the placement (execution order) of all data stores and transformation involved in the whole ETL process, rather than the whole composition of all constituents of the logical model.

5.3.1. Correctness of the FSC algorithm

Theorem 2 guarantees that the `FSC` algorithm is correct; in the sense that it produces correct stages and that every transformation exists only in one stage. In other words, it guarantees that the whole conceptual graph is correctly divided into several stages, the ordering of stages is appropriate and that all its nodes are correctly placed in the appropriate stage.

Theorem 2. *The conceptual graph* $\boldsymbol{G=(V,E)}$ *may be divided into stages that: (a) uniquely comprise all its nodes; and (b) are ordered in such a way that the topological sort of the graph of stages is feasible.*

Proof. Assume the graph $\mathbf{G'=(V',E')}$ where each stage is represented as a node. It is easy to prove that there exists a topological sort of $\mathbf{G'=(V',E')}$ that produces the same ordering of the graph's stages. This is clear, since stages that: (a) are found after a binary transformation or recordset; and (b) belong to more than one flow are examined only once from algorithm `FSC` (Ln: 10–11). Clearly, the ordering of the non-common parts of the two flows, before their joint node can be ordered arbitrarily (remember also that if they had another common part, this would have been traced

Algorithm Execution Order in Logical Workflows (EOLW)

```
1.  Input: An array LW[id], id=0,...,n, where n is the number of all possible flows among concepts
2.  Output: A graph G=(V,E) that represents the logical design
3.  Begin
4.    for each LW[i] in LW {
5.      previous = LW[i].Stage[0];
6.      V = V ∪ {previous};
7.      for each Stage[j] in LW[i] {
8.        for each node n∈Stage[j] {
9.          if (n∉V) { // if this is the first appearance of the node
10.           V = V ∪ {n};
11.           E = E ∪ {previous,n};
12.           previous = n;
13.         }
14.         if (n∈V) {
15.           if (n∈A) { E = E ∪ (previous,n); }
16.           if (n∈RS) { // if it is a recordset
17.             E = E − {x,n}, s.t. x∈V ∧ ∃(x,n)∈E;
18.             V = V ∪ {U}; // add a Union activity
19.             E = E∪{previous,U}∪{x,U}∪{U,n};
20.   }}}}
21.   return G(V,E);
22. End.
```

Fig. 14. The EOLW algorithm.

in advance). The completeness of the algorithm is guaranteed by the Lines 6, 9, and 13: all paths are considered; all joint nodes; and all intermediate nodes, respectively. □

5.4. Execution order of activities

So far, we have introduced stages and we have presented how they can be automatically identified in the conceptual design. We are ready now to describe how the execution order of activities in the logical design can be determined.

Usually, ETL processes consist of more than one 'one source–one target' flow, probably involving more than one binary activity. Thus, in general we should follow the technique described in the previous subsection for finding the execution order of activities involved in the overall process. At first, we find the proper execution order for every simple flow in the conceptual design, in order to establish the proper placement of the activities in linear logical workflows. As we have discussed in the previous section, each element of the LW[] array contains all the nodes, separated into stages, for each one of these workflows.

The execution order of the activities in the logical workflow is determined by the EOLW algorithm. The EOLW algorithm (Fig. 14) takes as input the array LW[] computed by FSC algorithm. Again, its main task is to construct the logical design, in terms of finding the appropriate execution order, rather than build the whole ETL process with its semantics; the latter is a later task presented in the next section. Thus, it creates and outputs a graph **G**=(**V**,**E**) that contains all the necessary data stores and activities.

The procedure is realized as follows: For every simple flow, i.e., for every element of LW[], algorithm EOLW processes all transformation stages, i.e., all elements of Stage[] (Ln: 7–20). Each time it finds a new node, either data store or activity, it adds it to the graph along with an edge that connects this node with its prior node in the flow (Ln: 9–13). If a node has already been added in the graph then this is a case of convergence of two flows: the current one and the one already connected to this node (Ln: 14). We discriminate two possible cases: the node is either a binary activity or a data store, i.e., it belongs to **A** or **RS**. The case of an already visited binary activity is the simplest; then, we need only a connection from its prior node in the current flow processed to the binary activity (Ln: 15). The second case necessitates the usage of an extra Union activity (see subsection 4.6) placed exactly before the respective data store (Ln: 16–20). To achieve this, we first delete the edge between the data store and the last node of the flow already connected to it (Ln: 17). Then we add a new node representing the Union activity (Ln: 18) and connect it to both flows and the data store (Ln: 19). Finally, EOLW returns the graph that represents the logical design.

Theorem 3. *The* EOLW *algorithm is correct in the sense that: (a) all transformations are mapped to activities; and (b) the order of stages is preserved in the order of activities.*

Proof. The completeness part-of the algorithm is guaranteed by the completeness of algorithm FSC and the two for loops of Ln: 7–8. The order of the stages is preserved by the fact that the array LW is processed in order (Ln: 7). □

As we have already discussed, normally, a stage contains only order-equivalent transformations (those with their respective activities in the logical workflow that are swappable). The correct design assures that non-order-equivalent transformations shall be placed in different stages. However, since we have chosen to follow a more comprehensible approach for the non-expert users, mistakes are allowable. Even then, as we present in the next section, the last step of the mapping consists of a schema verification method that is able to indicate such errors, and in that case, the system administrator/designer shall deal with such problems.

6. Method

This section presents a sequence of steps that a designer should follow, during the transition from the conceptual to logical model, with the ultimate goal of the production of a mapping between the two models, along with any relevant auxiliary information.

Step 1: Preparation. We refine the conceptual model in terms that no ambiguity is allowed. After we decide the active candidate, a simplified 'working copy' of the scenario that eliminates all candidates, is produced [13]. Also, any additional information depicted as a note (e.g., comments, runtime constraints), which is not useful during the mapping, is discarded from the design and it is stored to a log file.

Step 2: Concepts and Attributes. Next, we add all necessary recordsets along with their attributes. All concepts of the conceptual design are mapped to recordsets in the logical design. Similarly, their attributes are mapped to the attributes of the respective recordsets. Obviously, the part-of relationships remain the same after the transition from one model to the other.

Step 3: Transformations. After that, we determine the appropriate activities along with their execution order. The determination of activities is captured by the conceptual design; all transformations involved in the conceptual design are mapped to logical activities. Also, we incorporate activities that are not shown in the conceptual design, to capture the rejection of an attribute or the convergence of two flows. Afterwards, we determine the execution order of the activities, and we exploit the information provided by the notes to fully capture the semantics of every activity.

Step 4: ETL Constraints. The next step takes into account the ETL constraints imposed on the data stores. Recall that when we transform an ETL constraint into an activity, we should place it exactly before the recordset involved. Thus, in this step, we enrich the logical design with extra activities that represent all ETL constraints of the conceptual design. The execution order of these newly inserted activities is defined according to the same criteria and techniques with those unfolded in the case of stage-equivalent transformations.

Step 5: Schemata Generation. As a final step, we should ensure that all the schemata involved in the overall process are valid. For this reason, we use the algorithm *Schema Generation* (SGen), introduced in [14]. SGen automatically creates all the schemata involved in the logical design. The main idea of SGen is that after the topological sorting of an ETL workflow, the input schema of an activity is the same with the (output) schema of its provider and the output schema of an activity is equal to (the union of) its input schema(ta), augmented by the generated schema, minus the projected-out attributes. Therefore, for two subsequent activities A_1 and A_2, the following equalities hold:

$$A_2.\mathtt{in} = A_1.\mathtt{out}$$

$$A_2.\mathtt{out} = A_2.\mathtt{in} \cup A_2.\mathtt{gen} - A_2.\mathtt{pro}.$$

Thus, given that the schemata of the source recordsets are known and the generated and projected-out schemata of each activity are provided by the template instantiation, the calculation of the schemata of the whole ETL process is feasible.

7. Related work

This section presents the state of the art concerning the correlation of two different levels of ETL design: conceptual and logical. Although, there exists several approaches [6,7,15,17] for the conceptual part-of the design of an ETL scenario, so far, we are not aware of any other research approach concerning a mapping from a conceptual to a logical model for ETL processes. However, in the literature there exist several approaches

concerning the automation or semi-automation of several tasks of logical DW design from conceptual models.

Ballard [1] proposes a method that starts applying transformations to the ER-enterprise model until a representation of the corporate dimensions is obtained. Then, the dimensional model is designed, mainly by taking into account the gathered user requirements.

Boehnlein and Ulbrich-vom Ende [3] present an approach to derive initial data warehouse structures from the conceptual schemes of operational sources. The authors introduce a detailed description of the derivation of initial data warehouse structures from the conceptual scheme, through an example of a flight reservation system. Their method consists of three stages: (a) identification of business measures; (b) identification of dimensions and dimension hierarchies; and (c) identification of integrity constraints along the dimension hierarchies.

Golfarelli and Rizzi [4] propose a general methodological framework for data warehouse design, based on Dimensional Fact Model (DFM). The authors present a method for the DW design that consists of six phases: (a) analysis of the existing information system; (b) collection of the user requirements; (c) automatic conceptual design based on the operational database scheme; (d) workload refinement and schema validation; (e) logical design; and (f) physical design. In general, the proposed method starts from an ER-enterprise model. Then, the model is restructured and transformed until a conceptual schema is obtained. Finally, the authors provide a method to pass from this model to a logical dimensional model (in particular the star schema in the relational model).

Hahn, Sapia and Blaschka [5] present a modelling framework, BabelFish, concerning the automatic generation of OLAP schemata from conceptual graphical models, and discuss the issues of this automatic generation process for both the OLAP database schema and the front-end configuration. The main idea of BabelFish is that the DW designer models the universe of discourse on a conceptual level using graphical notations, while he/she is being supported by a specialized Computer Aided Warehouse Engineering (CAWE) environment. This environment generates the implementation of the conceptual design models, but hides the implementation details. Moreover, the authors list typical mismatches between the data model of commercial OLAP tools and conceptual graphical modeling notations, and proposes methods to overcome these expressive differences during the generation process.

Moody and Kortink [8] describe a method for developing dimensional models from traditional ER-models. This method consists of several steps that occur after the completion of developing an enterprise data model (if one does not already exist) and the completion of the design for central DW: (a) the first step involves the classification of entities of the ER-enterprise model in a number of categories; (b) the second step concerns the identification of hierarchies that exist in the model; and (c) the final step involves the collapse of these hierarchies and the aggregation of the transaction data. The procedure described by this method is an iterative process that needs evaluation and refinement. Moreover, the authors present a range of options for developing data marts to support end user queries from an enterprise data model, including: snowflake; star cluster; star; terraced; and flat schemata.

Peralta [10] presents a framework for generating a DW logical schema from a conceptual schema. It presents a rule-based mechanism to automate the construction of DW logical schemata. This mechanism consists of a set of design rules that decide the application of the suitable transformations in order to solve different design problems. The proposed framework consists of: mappings between source and DW conceptual schemata; design guidelines that refine the conceptual schema; schema transformations which generate the target DW schema; and design rules that decide the application of the suitable transformations in order to solve different design problems.

Phipps and Davis [12] propose algorithms for the automatic design of DW conceptual schemata. Starting from an enterprise schema, candidate conceptual schemas are created using the ME/R model, extended to note where additional user input can be used to further refine a schema. Following a user-driven requirements approach, the authors propose a method towards the refinement of a conceptual schema. Additionally, Phipps [11] proposes an algorithm for the creation of a logical schema (dimensional star schema) from a conceptual one.

In a different line of research, Bekaert et al. [2] describe a semi-automatic transformation from object-oriented conceptual models in EROOS (an Entity Relationship based Object-Oriented Specification method that is an OO analysis method for building conceptual models) to logical theories in ID-Logic (Inductive Definition Logic that is an integration of first order logic and logic programming, interpreting the logic program as a non-monotone inductive definition).

8. Discussion

In this paper we have presented a semi-automatic transition from a conceptual model to the logical model. First, we have presented a rigorous definition for the constructs and constraints of the conceptual model. Then, we have formally described the mapping from conceptual to logical models. Also, we have provided a

method for the determination of a correct execution order of the activities in the logical model. Finally, we have offered a cohesive method that consists of a sequence of steps that a designer should follow, during the transition from the conceptual to logical model.

Once again, our method is not a fully automatic procedure. This is due to the fact that intentionally we do not provide any strict verification method for the conceptual model [17]. The goal of this mapping is to facilitate the integration of the results accumulated in the early phases of a data warehouse project into the logical model, such as the collection of requirements from the part-of the users, the analysis of the structure and content of the existing data sources along with their mapping to the common data warehouse model. Thus, the deliverable of this mapping could not necessary be a complete and accurate logical design. Hence, the designer during the mapping from the one model to the other or in the logical level, should examine, complement or change the outcome of this method.

As a final note, the design of Fig. 4 is not the only possible logical design that corresponds to the conceptual design of Fig. 1. The constraints in the determination of the execution order require only that the placement of activities should be semantically correct or equivalently, that these activities can interchange their position without any semantic conflict. Clearly, n order-equivalent activities can produce $n!$ different logical workflows. The final choice of one of them depends on other parameters beyond those examined in this paper; e.g., the total cost of a workflow. We have resolved this issue in [14], where we discuss optimization issues concerning ETL processes.

Acknowledgments

We would like to thank all the reviewers of an earlier draft of this paper for their detailed comments. This work is co-funded by the European Social Fund (75%) and National Resources (25%) — Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS.

References

[1] C. Ballard, Data Modeling Techniques for Data Warehousing. SG24-2238-00, IBM Red Book, 0738402451, 1998.
[2] P. Bekaert, B. Van Nuffelen, M. Bruynooghe, D. Gilis, M. Denecker, On the Transformation of Object-Oriented Conceptual Models to Logical Theories, in ER, 2002.
[3] M. Boehnlein, A. Ulbrich-vom Ende, Deriving the Initial Data Warehouse Structures from the Conceptual Data Models of the Underlying Operational Information Systems, in DOLAP, 1999.
[4] M. Golfarelli, S. Rizzi, Methodological Framework for Data Warehouse Design, in DOLAP, 1998.
[5] K. Hahn, C. Sapia, M. Blaschka, Automatically Generating OLAP Schemata from Conceptual Graphical Models, in DOLAP, 2000.
[6] R. Kimball, et al., The Data Warehouse Lifecycle Toolkit, John Wiley & Sons, 1998.
[7] S. Luján-Mora, P. Vassiliadis, J. Trujillo, Data Mapping Diagrams for Data Warehouse Design with UML, in ER, 2004.
[8] D.L. Moody, M.A.R. Kortink, From Enterprise Models to Dimensional Models: a Methodology for Data Warehouse and Data Mart Design, in DMDW, 2000.
[9] S. Naqvi, S. Tsur, A Logical Language for Data and Knowledge Bases, Computer Science Press, 1989.
[10] V. Peralta, Data Warehouse Logical Design from Multidimensional Conceptual Schemas, in CLEI, 2003.
[11] C. Phipps, Migrating an Operational Database Schema to a Data Warehouse Schema, PhD thesis (University of Cincinnati, USA, 2001).
[12] C. Phipps, K. Davis, Automating Data Warehouse Conceptual Schema Design and Evaluation, in DMDW, 2002.
[13] A. Simitsis, P. Vassiliadis, A Methodology for the Conceptual Modeling of ETL Processes, in DSE, 2003.
[14] A. Simitsis, P. Vassiliadis, T. Sellis, Optimizing ETL Processes in Data Warehouse Environments, in ICDE, 2005.
[15] J. Trujillo, S. Lujan-Mora, A UML Based Approach for Modeling ETL Processes in Data Warehouses, in ER, 2003.
[16] P. Vassiliadis, A. Simitsis, S. Skiadopoulos, Modeling ETL Activities as Graphs, in DMDW, 2002.
[17] P. Vassiliadis, A. Simitsis, S. Skiadopoulos, Conceptual Modeling for ETL Processes, in DOLAP, 2002.
[18] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, A Framework for the Design of ETL Scenarios, in CAiSE, 2003.
[19] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, S. Skiadopoulos, A generic and customizable framework for the design of ETL scenarios, Information Systems Journal 30 (7) (2005).
[20] C. Zaniolo, LDL++ Tutorial, UCLA, Dec. 1998 Available at: http://pike.cs.ucla.edu/ldl/.

Dr Alkis Simitsis is a visiting lecturer at the University of Peloponnese in Greece. He received his Diploma degree in 2000 and the PhD degree in 2004, both from the School of Electrical and Computer Engineering of the National Technical University of Athens (NTUA). His research interests include Extraction–Transformation–Loading (ETL) processes in data warehouses, modeling and designing of data warehouses, query processing/optimization, keyword search, and data cleaning. He has published more than 20 papers in refereed journals and international conferences in the above areas.

Dr Panos Vassiliadis is a lecturer at the University of Ioannina in Greece. He received his Diploma degree in 1995 and the PhD degree in 2000, both from the School of Electrical and Computer Engineering of the National Technical University of Athens (NTUA). His research interests include data warehousing, web services and database design and modeling. He has published more than 30 papers in refereed journals and international conferences in the above areas.