

Modeling ETL Activities as Graphs

Panos Vassiliadis, Alkis Simitsis, Spiros Skiadopoulos

National Technical University of Athens, Dept. of Electrical and Computer Eng.,
Computer Science Division, Iroon Polytechniou 9, 157 73, Athens, Greece
Email: {pvassil, asimi, spiros}@dbnet.ece.ntua.gr
Phone: +3010-772-1436, 1602
Fax: +3010-772-1442

1.	INTRODUCTION	2
2.	GENERIC MODEL OF ETL ACTIVITIES	6
2.1	PRELIMINARIES	6
2.2	ACTIVITIES AND RELATIONSHIPS	7
2.3	SCENARIOS	11
2.4	MOTIVATING EXAMPLE	11
3	THE ARCHITECTURE GRAPH OF AN ETL SCENARIO	15
3.1	PRELIMINARIES	15
3.2	CONSTRUCTING THE ARCHITECTURE GRAPH	17
4.	EXPLOITATION OF THE ARCHITECTURE GRAPH	22
4.1	GRAPH TRANSFORMATIONS	23
4.2	IMPORTANCE METRICS	24
5.	RELATED WORK	28
6.	CONCLUSIONS	30
	REFERENCES	30

Modeling ETL Activities as Graphs*

Panos Vassiliadis, Alkis Simitsis, Spiros Skiadopoulos

National Technical University of Athens, Dept. of Electrical and Computer Eng.,
Computer Science Division, Iroon Polytechniou 9, 157 73, Athens, Greece
{pvassil, asimi, spiros}@dbnet.ece.ntua.gr

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 this paper, we focus on the logical design of the ETL scenario of a data warehouse. We define a formal logical model for ETL processes. The data stores, activities and their constituent parts are formally introduced. An ETL scenario is defined as the combination of ETL activities and data stores. Then, we show how this model is reduced to a graph, which we call the *Architecture Graph*. We model all the aforementioned entities as nodes and four different kinds of relationships (instance-of, part-of, regulator and provider relationships) as edges. Also, we provide simple graph transformations that reduce the complexity of the graph. Finally, in order to support the engineering of the design and the evolution of the warehouse, we introduce specific *importance metrics*, namely *dependence* and *responsibility*, to measure the degree to which entities are bound to each other.

1. Introduction

For quite a long time in the past, research has treated data warehouses as collections of materialized views. Although this abstraction is elegant and possibly sufficient for the purpose of examining alternative strategies for view maintenance, it is simply insufficient to describe the structure and contents of a data warehouse. In [VQVJ01], the authors bring up the issue of *data warehouse operational processes* and deduce the definition of a table in the data warehouse as the outcome of the combination of the processes that populate it. This new kind of definition complements existing approaches, since it provides the operational semantics for the content of a data warehouse table, whereas the existing ones give an abstraction of its intentional semantics.

In order to facilitate and manage the data warehouse operational processes, specialized tools are already available in the market, under the general title *Extraction-Transformation-Loading* (ETL) tools. To give a general idea of the functionality of these tools we mention their most prominent tasks, which include:

- the identification of relevant information at the source side;

* This research has been partially funded by the European Union's Information Society Technologies Programme (IST) under project EDITH (IST-1999-20722).

- the extraction of this information;
- the customization and integration of the information coming from multiple sources into a common format;
- the cleaning of the resulting data set, on the basis of database and business rules, and
- the propagation of the data to the data warehouse and/or data marts.

In the sequel, we will not discriminate between the tasks of ETL and Data Cleaning and adopt the name ETL for both these kinds of activities.

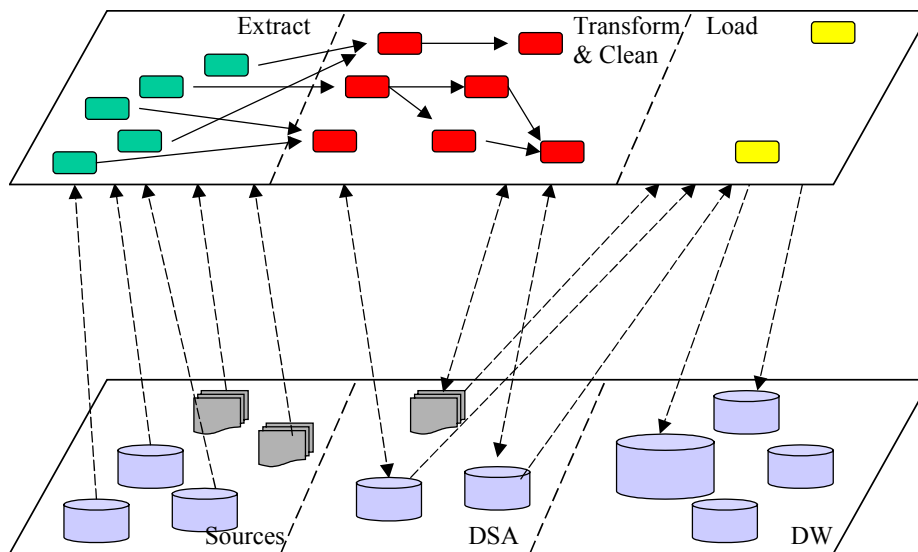


Fig. 1.1 The environment of Extract-Transform-Load processes

In Fig. 1.1 we abstractly describe the general framework for ETL processes. In the bottom layer we depict the data stores that are involved in the overall process. On the left side, we can observe the original data providers. Typically, data providers are relational databases and files. The data from these sources are extracted (as shown in the upper left part of Fig. 1.1) by extraction routines, which provide either complete snapshots or differentials of the data sources. Then, these data are propagated to the *Data Staging Area* (DSA) where they are transformed and cleaned before being loaded to the data warehouse. The data warehouse is depicted in the right part of the data store layer and comprises the target data stores, i.e., fact tables for the storage of information and dimension tables with the description and the multidimensional, roll-up hierarchies of the stored facts. The loading of the central warehouse is performed from the loading activities depicted on the upper right part of the figure.

Related literature has attributed several characteristics to ETL processes. First, we can clearly characterize them as *complex*. As mentioned in [BoFM99] the data warehouse refreshment process can consist of many different subprocesses, like data cleaning, archiving, transformations and aggregations, interconnected through a complex schedule. Second, data warehouse operational processes are *costly* and *critical* for the success of a data warehouse project. Actually, their design and

implementation has been characterized as a labor-intensive and lengthy procedure, covering thirty to eighty percent of effort and expenses of the overall data warehouse construction [Dema97, ShTy98, Vass00]. Third, these processes are *important* for the correctness, completeness and freshness of data warehouse contents, since not only do they facilitate the population of the warehouse with up-to-date data, but also they are responsible for homogenizing their structure and blocking the propagation of erroneous or inconsistent entries.

From the above, we believe that the research on ETL processes is a valid research goal. *The uttermost goal of our research is to facilitate, manage and optimize the design and implementation of the ETL processes both during the initial design and deployment stage and during the continuous evolution of the data warehouse.*

To probe into the aforementioned problem, we have to clarify how the ETL processes fit in the data warehouse lifecycle. As we can see in Fig. 1.2, the lifecycle of a data warehouse begins with an initial *Reverse Engineering and Requirements Collection* phase where the data sources are analyzed in order to comprehend their structure and contents. At the same time, any requirements from the part of the users (normally a few power users) are also collected. The deliverable of this stage is a *conceptual model* for the data stores and the activities. In a second stage, namely the *Logical Design* of the warehouse, the *logical schema* for the warehouse and the activities is constructed. Third, the logical design of the schema and processes is refined to the choice of specific physical structures in the warehouse (e.g., indexes) and environment-specific execution parameters for the operational processes. We call this stage *Tuning* and its deliverable, *the physical model* of the environment. In a fourth stage, *Software Construction*, the *software* is constructed, tested, evaluated and a first version of the warehouse is deployed. This process is guided through specific *software metrics*. Then, the cycle starts again, since data sources, user requirements and the data warehouse state are under continuous evolution. An extra feature that comes into the scene after the deployment of the warehouse is the *Administration* task, which also needs specific *metrics* for the maintenance and monitoring of the data warehouse.

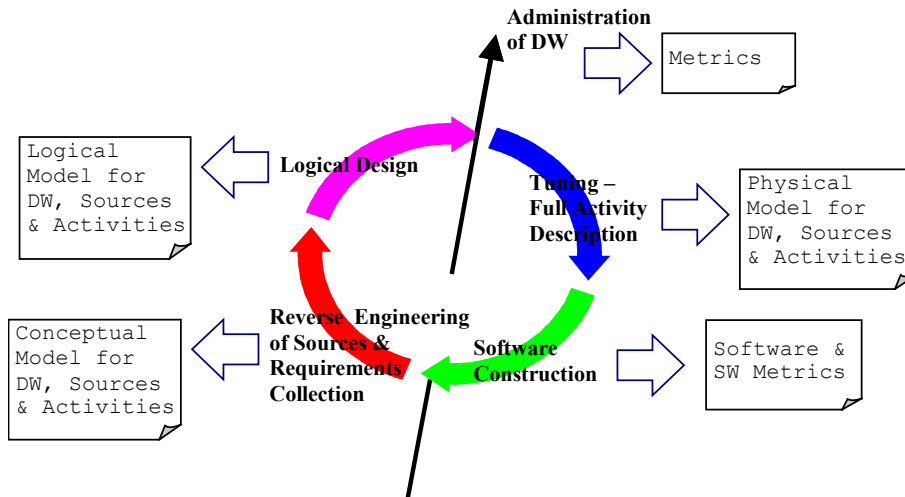


Fig. 1.2 The lifecycle of a Data Warehouse and its ETL processes

The conceptual model for ETL processes along with a methodology for its derivation is described in [VaSS02]. In this paper, we focus on the logical design of the ETL scenario of a data warehouse. Our contributions can be listed as follows:

- First, *we define a formal logical model as a logical abstraction of ETL processes*. The data stores, activities and their constituent parts are formally defined. An activity is defined as an entity with (possibly more than one) input schema(ta), an output schema, a rejection schema for the rows that do not pass the criteria of the activity and a parameter schema, so that the activity is populated each time with its proper parameter values. The flow of data from producers towards their consumers is achieved through the usage of *provider relationships* that map the attributes of the former to the respective attributes of the latter. The serializable combination of ETL activities, provider relationships and data stores constitutes an ETL scenario.
- Second, *we show how this model is reduced to a graph*, which we call the *Architecture Graph*. We model all the aforementioned entities as nodes and four different kinds of relationships as edges. These relationships involve (a) type checking information (i.e., which type an entity corresponds to), (b) part-of relationships (e.g., which activity does an attribute belong to), (c) regulator relationships, covering the population of the parameters of the activities from attributes or constant values and (d) provider relationships, covering the flow of data from providers to consumers.
- Finally, *we provide results on the exploitation of the Architecture Graph*. First we provide several simple graph transformations that reduce the complexity of the graph. For example, we give a simple algorithm for zooming out the graph, a transformation that can be very useful for the visualization of the graph. Second, we measure the importance and vulnerability of the nodes of the graph through specific *importance metrics*, namely *dependence* and *responsibility*. Dependence stands for the degree to which a node is bound to other entities that provide it with data and responsibility measures the degree

up to which other nodes of the graph depend on the node under consideration. Dependence and responsibility are crucial measures for the engineering of the evolution of the ETL environment.

This paper is organized as follows. In Section 2 we present a generic model of ETL activities. Section 3 presents how an ETL scenario can be modeled as a graph. Section 4 describes the exploitation of the architecture graph through several useful transformations and treats the design quality of an ETL scenario via this graph. In Section 5 we present related work. Finally, in Section 6 we conclude our results.

2. Generic Model of ETL Activities

The purpose of this section is to present a formal logical model for the activities of an ETL environment. This model abstracts from the technicalities of monitoring, scheduling, logging while it concentrates on the flow of data from the sources towards the data warehouse through the composition of activities and data stores.

2.1 Preliminaries

In this subsection, we will introduce the formal modeling of data types, data stores and functions, before proceeding to the modeling of ETL activities.

Elementary Entities. We assume the existence of a countable set of *data types*. Each data type \mathbb{T} is characterized by a name and a domain, i.e., a countable set of values, called $\text{dom}(\mathbb{T})$. The values of the domains are also referred to as *constants*.

We also assume the existence of a countable set of *attributes*, which constitute the most elementary granules of the infrastructure of the information system. Attributes are characterized by their name and data type. The domain of an attribute is a subset of the domain of its data type. Attributes and constants are uniformly referred to as *terms*.

A *Schema* is a finite *list* of attributes. Each entity that is characterized by one or more schemata will be called *Structured Entity*¹. Moreover, we assume the existence of a special family of schemata, all under the general name of *NULL Schema*, determined to act as placeholders for data which are not to be stored permanently in some data store. We refer to a family instead of a single NULL schema, due to a subtle technicality involving the number of attributes of such a schema (this will become clear in the sequel).

RecordSets. We define a *record* as the instantiation of a schema to a list of values belonging to the domains of the respective schema attributes. As mentioned in [VQVJ01], we can treat any data structure as a “record set” provided that there are the

¹ As opposed to unstructured or semi-structured data, which we do not consider in the context of this paper.

means to *logically* restructure it into a flat, typed record schema. Several *physical* storage structures abide by this rule, such as relational databases, COBOL or simple ASCII files, multidimensional cubes, etc. We will employ the general term *RecordSet* in order to refer to this kind of structures. For example, the schema of multidimensional cubes is of the form $[D_1, \dots, D_n, M_1, \dots, M_m]$ where the D_i represent dimensions (forming the primary key of the cube) and the M_j measures [VaSk00]. COBOL files, as another example, are records with fields having two peculiarities: nested records and alternative representations. One can easily unfold the nested records and choose one of the alternative representations. Relational databases are clearly recordsets, too. Formally, 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). If we consider a schema $S = [A_1, \dots, A_k]$, for a certain recordset, its extension is a mapping $S = [A_1, \dots, A_k] \rightarrow \text{dom}(A_1) \times \dots \times \text{dom}(A_k)$. Thus, the extension of the recordset is a finite subset of $\text{dom}(A_1) \times \dots \times \text{dom}(A_k)$ and a record is the instance of a mapping $\text{dom}(A_1) \times \dots \times \text{dom}(A_k) \rightarrow [x_1, \dots, x_k]$, $x_i \in \text{dom}(A_i)$.

In the rest of this paper we will mainly deal with the two most popular types of recordsets, namely *relational tables* and *record files*. A *database* is a finite set of relational tables.

Functions. We assume the existence of a countable set of built-in system *function types*. A function type comprises a name, a finite list of *parameter data types*, and a single *return data type*. A *function* is an instance of a function type; consequently it is also characterized by a name, a list of input parameters and a parameter for its return value. The data types of the parameters of the generating function type define also (a) the data types of the parameters of the function and (b) the legal candidates for the function parameters (i.e., attributes or constants of a suitable data type).

2.2 Activities and Relationships

Activities are the backbone of the structure of any information system. We adopt the WfMC terminology [WfMC98] for processes/programs and we will call them *activities* in the sequel. An activity is an amount of "work which is processed by a combination of resource and computer applications" [WfMC98]. In our framework, activities are logical abstractions representing parts, or full modules of code.

The execution of an activity is performed from a particular program. Normally, ETL activities will be either performed in a black-box manner by a dedicated tool, or they will be expressed in some scripting language (e.g., PL/SQL, Perl, C, etc). Still, we want to deal with the general case of ETL activities; thus we employ an abstraction of the source code of an activity, in the form of an SQL statement, in order to avoid dealing with the peculiarities of a particular programming language. Once again, we want to stress that the presented SQL description is not intended to capture the way these activities are actually implemented, but rather the semantics of each activity.

An *Elementary Activity* is formally described by the following elements:

- *Name*: a unique identifier for the activity.

- *Input Schemata*: a finite set of one or more input schemata that receive data from the data providers of the activity.
- *Output Schema*: the schema that describes the placeholder for the rows that pass the check performed by the elementary activity.
- *Rejections Schema*: a schema that describes the placeholder for the rows that do not pass the check performed by the activity, or their values are not appropriate for the performed transformation.
- *Parameter List*: a set of pairs which act as regulators for the functionality of the activity (the target attribute of a foreign key check, for example). The first component of the pair is a name and the second is a schema, an attribute, a function or a constant.
- *Output Operational Semantics*: an SQL statement describing the content passed to the output of the operation, with respect to its input. This SQL statement defines (a) the operation performed on the rows that pass through the activity and (b) an implicit mapping between the attributes of the input schema(ta) and the respective attributes of the output schema.
- *Rejection Operational Semantics*: an SQL statement describing the rejected records, in a sense similar to the *Output Operational Semantics*. This statement is by default considered to be the negation of the *Output Operational Semantics*, except if explicitly defined differently.
- *Data Provider/Consumer Semantics*: a modality of each schema which defines (a) whether the data are simply read (default) or read and subsequently deleted from the data provider and (b) whether the data are appended to the data consumer (default) or the contents of the data consumer are overwritten from the output of the operation. Thus, the domain of the *Data Provider Semantics* attribute is {SELECT,DELETE} and the domain of the *Data Consumer Semantics* attribute is {APPEND,OVERWRITE}.

Note that:

- The *Data Provider/Consumer Semantics* is a facility employed in order to capture the cases of activities that delete or update tables in the data warehouse. In the context of this paper, we will consider only cases where data are simply read from the sources and appended to the targets, without any further action; thus we will not deal with this attribute any more in our discussion. Still, we conjecture that our results are not particularly altered in the general case, where deletions and updates occur.
- Whenever we do not specify a data consumer for the output or rejection schemata, the respective NULL schema (involving the correct number of attributes) is implied. This practically means that the data targeted for this schema will neither be stored to some persistent data store, nor will they be propagated to another activity, but they will simply be ignored.

The flow of data from the data sources towards the data warehouse is performed through the composition of activities in a larger scenario. In this context, the input for an activity can be either a persistent data store, or another activity, i.e., any structured entity under a specific schema. Usually, this applies for the output of an activity, too.

We capture the passing of data from *providers* to *consumers* by a *Provider Relationship* among the attributes of the involved schemata².

Formally, a *Provider Relationship* is defined as follows:

- *Name*: a unique identifier for the provider relationship.
- *Mapping*: an ordered pair. The first part of the pair is a term (i.e., an attribute or constant), acting as a provider and the second part is an attribute acting as the consumer.

The mapping need not necessarily be 1:1 from provider to consumer attributes, since an input attribute can be mapped to more than one consumer attributes. Still, this does not hold the other way around. Note that a consumer attribute can also be populated by a constant, in certain cases.

In order to achieve the flow of data from the providers of an activity towards its consumers, we need the following three groups of provider relationships:

1. A mapping between the input schemata of the activity and the (output) schema of their data providers. In other words, for each attribute of an input schema of an activity, there must exist an attribute of the data provider, or a constant, which is mapped to the former attribute.
2. A mapping between the attributes of the activity input schemata and the activity output (or rejection, respectively) schema.
3. A mapping between the output (rejection) schema of the activity and the (input) schema of its data consumer.

The mappings of the second type are internal to the activity. Basically, they can be derived from the SQL statement for each of the output/rejection schemata. As far as the first and the third types of provider relationships are concerned, the mappings must be provided during the construction of the ETL scenario. This means that they are either (a) by default assumed by the order of the attributes of the involved schemata or (b) hard-coded by the user.

² By abuse of terminology, we will also apply the term both for the involved schemata and the respective structured entities.

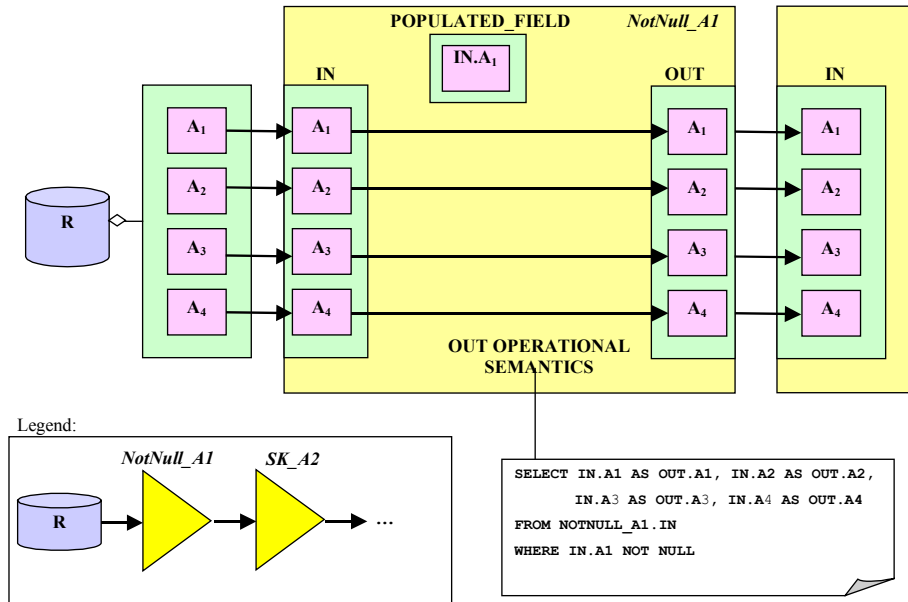


Fig. 2.1 Schemata and data flow for a small scenario

Observe the legend for the example of Fig. 2.1, where (a) we read data from the source table **R**, and pass them through a check for **NULL** values on attribute **A1**; (b) we propagate the qualifying data to a second transformation where a surrogate key is attributed to attribute **A2** (see Section 2.4 for more on surrogate keys) and (c) we pass the control to some other activity not depicted in Fig. 2.1. On the upper part of Fig. 2.1, where the (a) part of the scenario is depicted in more detail, we can observe:

- the fact that the recordset **R** comprises the attributes **A1, A2, A3, A4**;
- the provider relationships between the schema of **R** and the input schema **IN** of the activity **NotNull_A1**;
- the provider relationships between the input schema **IN** and the output schema **OUT** of the activity **NotNull_A1**;
- the provider relationships between the output schema **OUT** of the activity **NotNull_A1** and the input schema **IN** of the activity **SK_A2**;
- the *Output Operational Semantics* of the activity **NotNull_A1** in the form of an SQL statement;
- the population of the parameter **POPULATED_FIELD** that denotes the attribute over which the **NOT NULL** check is performed, by the attribute **IN.A1**.

All provider relationships are depicted as directed edges from the provider towards the consumer. For reasons of simplicity, we do not consider the rejection schemata for the aforementioned activities.

2.3 Scenarios

A *Scenario* is an enumeration of activities along with their source/target recordsets and the respective provider relationships for each activity. Formally, a Scenario consists of:

- *Name*: a unique identifier for the scenario.
- *Activities*: A finite *list* of activities. Note that by employing a list (instead of e.g., a set) of activities, we impose a total ordering on the execution of the scenario.
- *Recordsets*: A finite set of recordsets.
- *Targets*: A special-purpose subset of the recordsets of the scenario, which includes the final destinations of the overall process (i.e., the data warehouse tables that must be populated by the activities of the scenario).
- *Provider Relationships*: A finite list of provider relationships among activities and recordsets of the scenario.

Intuitively, a scenario is a set of activities, deployed along a graph in an execution sequence that can be linearly serialized. For the moment we do not consider the different alternatives for the ordering of the execution; we simply require that a total order for this execution is present (i.e., each activity has a discrete execution priority).

Moreover, we assume the following *Integrity Constraints* for a scenario:

Static Constraints:

- All the weak entities of a scenario (i.e., attributes or parameters) should be defined within a *part-of* relationship (i.e., they should have a container object).
- All the mappings in provider relationships should be defined among terms (i.e., attributes or constants) of the same data type.

Data Flow Constraints:

- All the attributes of the input schema(ta) of an activity should have a provider.
- Resulting from the previous requirement, if some attribute is a parameter in an activity A , the container of the attribute (i.e., recordset or activity) should precede A in the scenario.
- All the attributes of the schemata of the target recordsets should have a data provider.

2.4 Motivating example

To motivate our discussion we will present a motivating example based on the TPC-R/TPC-H standards [TPC00]. We assume the existence of two source databases S_1 and S_2 as well as a central data warehouse under the schemata of Fig. 2.2. Moreover, we assume the existence of a Data Staging Area (DSA), where all the transformations take place.

Source	Recordset Name	Recordset Schema
S ₁	S ₁ .PARTSUPP	<u>PKEY</u> , <u>DATE</u> , QTY, COST
S ₂	S ₂ .PARTSUPP	<u>PKEY</u> , QTY, COST
DSA	DS.PS_NEW ₁	<u>PKEY</u> , <u>DATE</u> , QTY, COST
	DS.PS_OLD ₁	<u>PKEY</u> , <u>DATE</u> , QTY, COST
	DS.PS ₁	<u>PKEY</u> , <u>DATE</u> , QTY, COST
	DS.PS_NEW ₂	<u>PKEY</u> , QTY, COST
	DS.PS_OLD ₂	<u>PKEY</u> , QTY, COST
	DS.PS ₂	<u>PKEY</u> , QTY, COST
DW	DW.PARTSUPP	<u>PKEY</u> , <u>SUPPKEY</u> , <u>DATE</u> , QTY, COST
	LOOKUP_PS	<u>PKEY</u> , <u>SOURCE</u> , <u>SKEY</u>
	V1	<u>PKEY</u> , <u>DAY</u> , <u>MIN_COST</u>
	V2	<u>PKEY</u> , <u>MONTH</u> , <u>AVG_COST</u>
	TIME	<u>DAY</u> , <u>MONTH</u> , <u>YEAR</u>

Fig. 2.2 The schemata of the source databases and of the data warehouse

The scenario involves the propagation of data from the table PARTSUPP of source S₁ as well as from the table PARTSUPP of source S₂ to the data warehouse. Table DW.PARTSUPP stores information for the available quantity (QTY) and cost (COST) of parts (PKEY) per supplier (SUPPKEY). Practically, the two data sources S₁ and S₂ stand for the two suppliers of the data warehouse. We assume that the first supplier is American and the second is European, thus the data coming from the first source need to be converted to European values and formats. Once the table PARTSUPP has been populated, data are further aggregated to populate two data marts V₁ and V₂ in the data warehouse. All the attributes, except for the dates are instances of the Integer type. The data mart V₁ stores information on the minimum cost of each part per day and the data mart V₂ stores information on the average cost of parts per month. The date attributes for tables S₁.PARTSUPP, DS.PS_NEW₁, DS.PS_OLD₁, DS.PS₁, are of type US_DATE and all the other date attributes are of type EU_DATE. The scenario is graphically depicted in Fig. 2.3 and involves the following transformations.

1. First, we transfer via ftp the snapshots from the sources S₁.PARTSUPP and S₂.PARTSUPP to the files DS.PS_NEW₁ and DS.PS_NEW₂ of the DSA.
2. In the DSA we maintain locally two copies of the snapshot for each source. The recordset DS.PS_NEW₁ stands for the last transferred snapshot of S₁.PARTSUPP and the recordset DS.PS_OLD₁ stands for the penultimate transferred snapshot. By detecting the difference of these snapshots we can derive the newly inserted rows in source S₁.PARTSUPP. We store these rows in the file DS.PS₁. Note that the difference activity that we employ, namely Diff₁, checks for differences only on the primary key of the recordsets; thus we ignore any possible deletions or updates for the attributes COST, QTY of existing rows. The data coming from source S₂ are treated similarly.
3. Next we start two flows, one for the rows of DS.PS₁ and another one for the rows of DS.PS₂. For both flows, in order to keep track of the supplier of each row, we

need to add a ‘flag’ attribute, namely `SUPPKEY`, indicating 1 or 2 for the respective supplier. This is achieved through the activities `Add_SPK1` and `Add_SPK2` respectively.

4. Again, in both rows, we assign a surrogate key on `PKEY`, for data of both sources. In the data warehouse context, it is common tactics to replace the keys of the production systems with a uniform key, which we call a *surrogate key* [KRR98]. The basic reasons for this replacement are performance and semantic homogeneity. Textual attributes are not the best candidates for indexed keys and thus need to be replaced by integer keys. At the same time, different production systems might use different keys for the same object, or the same key for different objects, resulting in the need for a global replacement of these values in the data warehouse. This replacement is performed through a lookup table of the form `L (PRODKEY, SOURCE, SKEY)`. The `SOURCE` column is due to the fact that there can be synonyms in the different sources, which are mapped to different objects in the data warehouse. In our case, the two activities that perform the surrogate key assignment for the attribute `PKEY` are `SK1` and `SK2` for each of the two flows. They both use the lookup table `LOOKUP_PS`.
5. In the flow for the data of S_1 , we need to we apply a function to the attribute `COST` that converts currency values from US Dollars to Euros. Also, we need to convert the American dates (attribute `DATE`) to the European format. These transformations are performed from activities `$2E1` and `A2Edate`, respectively.
6. Simultaneously, for the data coming from S_2 we need to perform (a) a test for `NULL` values for the attribute `COST` (activity `NotNULL`); (b) addition of a `DATE` attribute with the value of system’s date, since source S_2 does not contain date information (activity `AddDate`); and (c) select only the records for which an available quantity exists (`QTY>0`), using activity `CheckQTY`.
7. Then, the two flows are consolidated through a union activity, namely `U`, that also populates table `DW.PARTSUPP`.
8. Next, we need to populate the two data marts of the warehouse. For the data mart V_1 we simply group-by `DAY` and `PKEY` for the calculation of the minimum value per part through the activity `AGGREGATE1`.
9. As far as the data mart is concerned, we need to join the data from table `DW.PARTSUPP` with table `TIME` (activity `⋈`) and then aggregate the result by `MONTH` and `PKEY` for the calculation of the average value per part (activity `AGGREGATE2`).

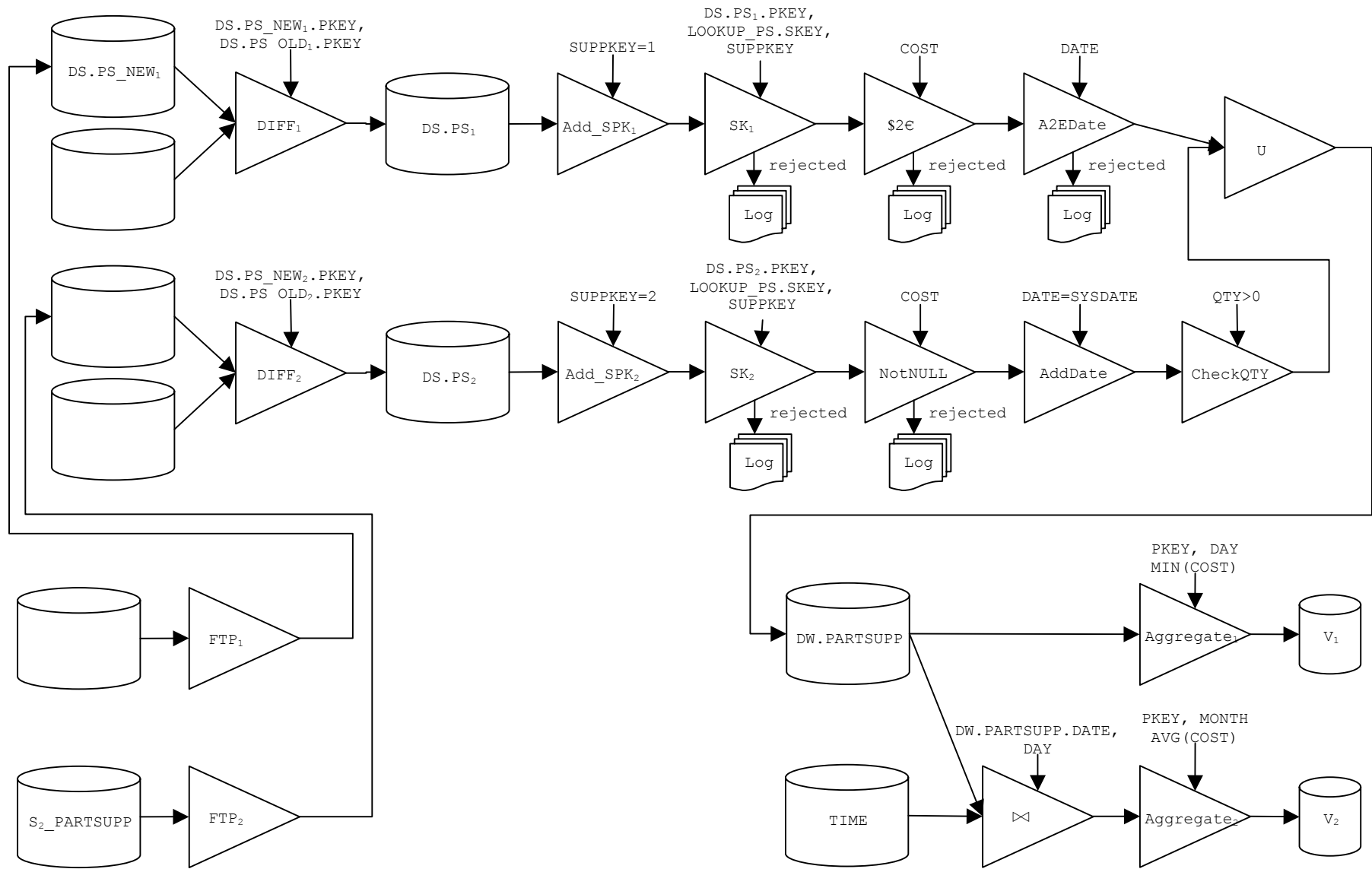


Figure 2.3: Motivating example

Note also, that for several of the aforementioned activities (i.e., $SK_{1,2}$, `NotNull`, $\$2E$, `A2EDate`), we need to trace the rows that do not pass the check or transformation performed. In this case, we employ the rejection schema of the activity to send these rows to their respective `Log` file.

3 The Architecture Graph of an ETL Scenario

In the previous sections, we have given a formal definition of activities, recordsets and other constituents of an ETL scenario. The full layout of an ETL scenario, involving activities, recordsets and functions can be modeled by a graph, which we call the *Architecture Graph*. The uniform, graph-modeling framework that we employ both for the modeling of the internal structure of activities and for the modeling of the ETL scenario at large, enables the treatment of the ETL environment from different viewpoints. First, the architecture graph comprises all the activities and data stores of a scenario, along with their components. Second, the architecture graph captures the data flow within the ETL environment. Finally, the information on the typing of the involved entities and the regulation of the execution of a scenario, through specific parameters are also covered.

3.1 Preliminaries

We assume the infinitely countable, mutually disjoint sets of names (i.e., the values of which respect the unique name assumption) of column *Model-specific* in Fig. 3.1. As far as a specific scenario is concerned, we assume their respective finite subsets, depicted in column *Scenario-Specific* in Fig. 3.1. Data types, function types and constants are considered *Built-in*'s of the system, whereas the rest of the entities are provided by the user (*User Provided*).

	Entity	Model-specific	Scenario-specific
Built-in	Data Types	D^I	D
	Function Types	F^I	F
	Constants	C^I	C
User-provided	Attributes	Ω^I	Ω
	Functions	Φ^I	Φ
	Schemata	S^I	S
	RecordSets	RS^I	RS
	Activities	A^I	A
	Provider Relationships	Pr^I	Pr
	Part-Of Relationships	Po^I	Po
	Instance-Of Relationships	Io^I	Io
	Regulator Relationships	Rr^I	Rr
Derived Provider Relationships	Dr^I	Dr	

Fig. 3.1 Formal definition of domains and notation

Being a graph, the Architecture Graph of an ETL scenario comprises nodes and edges. The involved data types, function types, constants, attributes, activities, recordsets and functions constitute the nodes of the graph. To fully capture the characteristics and interactions of the entities that have been mentioned in Section 2, we model the different kinds of their relationships as the edges of the graph. Here, we list these types of relationships along with the related terminology that we will employ for the rest of the paper.

- *Part-of* relationships. These relationships involve attributes and parameters and relate them to the respective activity, recordset or function to which they belong.
- *Instance-of* relationships. These relationships are defined among a data/function type and its instances.
- *Provider* relationships. These are relationships that involve attributes with a provider-consumer relationship, exactly in the sense of Section 2.2.
- *Regulator* relationships. These relationships are defined among the parameters of activities and the terms that populate these activities.
- *Derived provider relationships*. A special case of provider relationships that occurs whenever output attributes are computed through the composition of input attributes and parameters. Derived provider relationships can be deduced from a simple rule and do not originally constitute a part of the graph.


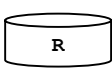

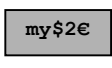

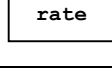
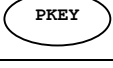
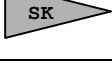
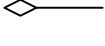
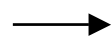
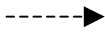
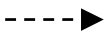
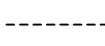
Data Types	Black ellipsis		RecordSets	Cylinders	
Function Types	Black squares		Functions	Gray squares	
Constants	Black cycles		Parameters	White squares	
Attributes	Hollow ellipsoid nodes		Activities	Triangles	
Part-Of Relationships	Simple edges annotated with diamond*		Provider Relationships	Bold solid rows (from provider to consumer)	
Instance-Of Relationships	Dotted arrows (from instance towards the type)		Derived Provider Relationships	Bold dotted arrows (from provider to consumer)	
Regulator Relationships	Dotted edges		* We annotate the part-of relationship among a function and its return type with a directed edge, to distinguish it from the rest of the parameters.		

Fig. 3.2 Graphical notation for the Architecture Graph.

Formally, let $G(V,E)$ be the Architecture Graph of an ETL scenario. Then,

- $V = D \cup F \cup C \cup \Omega \cup \Phi \cup S \cup R \cup A$
- $E = Pr \cup Po \cup Io \cup Rr \cup Dr$

The graphical notation for the Architecture Graph is depicted in Fig. 3.2.

3.2 Constructing the Architecture graph

In this subsection we will describe how we can construct the architecture graph, based on the theoretical foundations and the graphical notation of the previous sections. Clearly, we do not anticipate a manual construction of the graph by the designer; rather, we employ this section to clarify how the graph will look like once constructed. In general, the construction can be performed by a graphical tool, provided that the construction rules that we will present in this section are obeyed. In the sequel, we will employ a concrete example that involves a small part of the scenario of the motivating example, including the activities Add_SPK_1 and SK_1 .

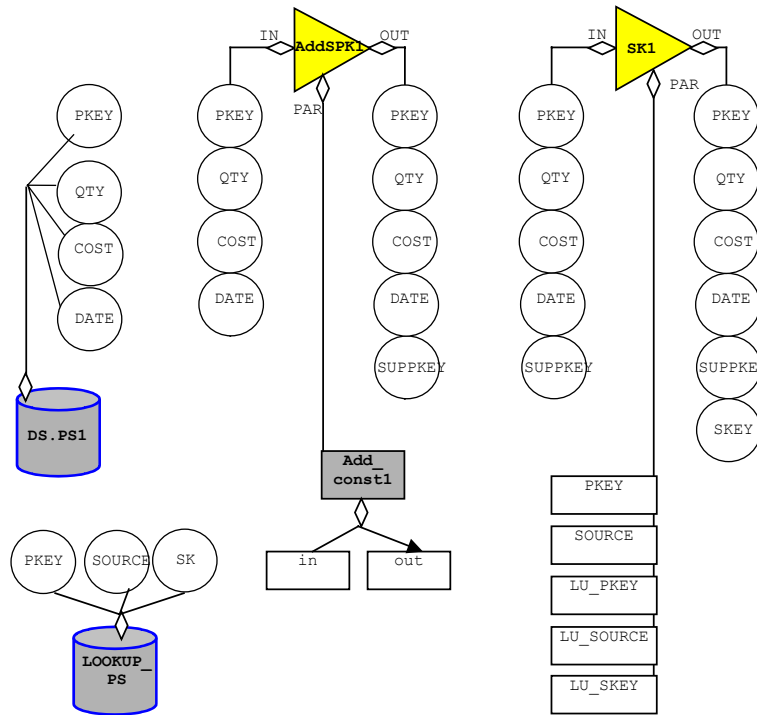


Fig. 3.3 Part-of relationships of the architecture graph

Attributes and part-of relationships. The first thing to incorporate in the architecture graph is the structured entities (activities and recordsets) along with all the attributes of their schemata. We choose to avoid overloading the notation by incorporating the schemata per se; instead we apply a direct part-of relationship between an activity node and the respective attributes. We annotate each such relationship with the name of the schema (by default, we assume a IN, OUT, PAR, REJ tag to denote whether the attribute belongs to the input, output, parameter or rejection

schema of the activity). Naturally, if the activity involves more than one input schemata, the relationship is tagged with an IN_i tag for the i -th input schema.

Then, we incorporate the functions along with their respective parameters and the part-of relationships among the former and the latter. We annotate the part-of relationship with the return type with a directed edge, to distinguish it from the rest of the parameters.

Fig. 3.3 depicts a part of the motivating example, where we can see the decomposition of (a) the recordsets $DS.PS_1, LOOKUP_PS$; (b) the activities Add_SPK_1 and SK_1 into the attributes of their input and output schemata. Note the tagging of the schemata of the involved activities. We do not consider the rejection schemata in order to avoid crowding the picture. At the same time, the function Add_const_1 is decomposed into its parameters. This function belongs to the function type ADD_CONST and comprises two parameters: in and out . The former receives an integer as input and the latter returns this integer. As we will see in the sequel, this value will be propagated towards the $SUPPKEY$ attribute, in order to trace the fact that the propagated rows come from source S_1 .

Note also, how the parameters of the two activities are also incorporated in the architecture graph. For the case of activity Add_SPK_1 the involved parameters are the parameters in and out of the employed function. For the case of activity SK_1 we have five parameters: (a) $PKEY$, which stands for the production key to be replaced; (b) $SOURCE$, which stands for an integer value that characterizes which source's data are processed; (c) LU_PKEY , which stands for the attribute of the lookup table which contains the production keys; (d) LU_SOURCE , which stands for the attribute of the lookup table which contains the source value (corresponding to the aforementioned $SOURCE$ parameter); (e) LU_SKEY , which stands for the attribute of the lookup table which contains the surrogate keys.

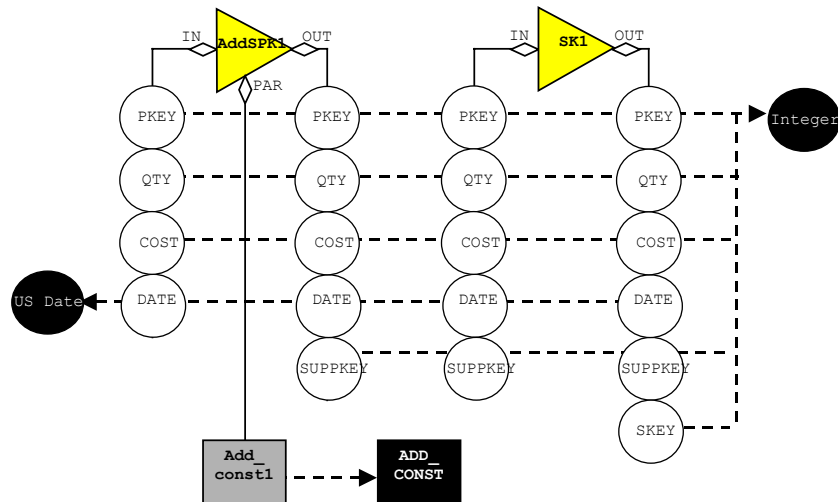


Fig. 3.4 Instance-of relationships of the architecture graph

Data types and instance-of relationships. Next, we incorporate data and function types. Instantiation relationships are depicted as dotted arrows that stem from the instances and head towards the data/function types. In Fig. 3.4, we observe the attributes of the two activities of our example and their correspondence to two data types, namely *Integer* and *US_Date*. For reasons of presentation, we merge several instantiation edges so that the figure does not become too crowded. At the bottom of Fig. 3.4, we can also see the fact that function *Add_const₁* is an instance of the function type *ADD_CONST*.

Parameters and regulator relationships. Once the part-of and instantiation relationships have been established, it is time to establish the regulator relationships of the scenario. In this case, we link the parameters of the activities to the terms (attributes or constants) that populate them. We depict regulator relationships with simple dotted edges.

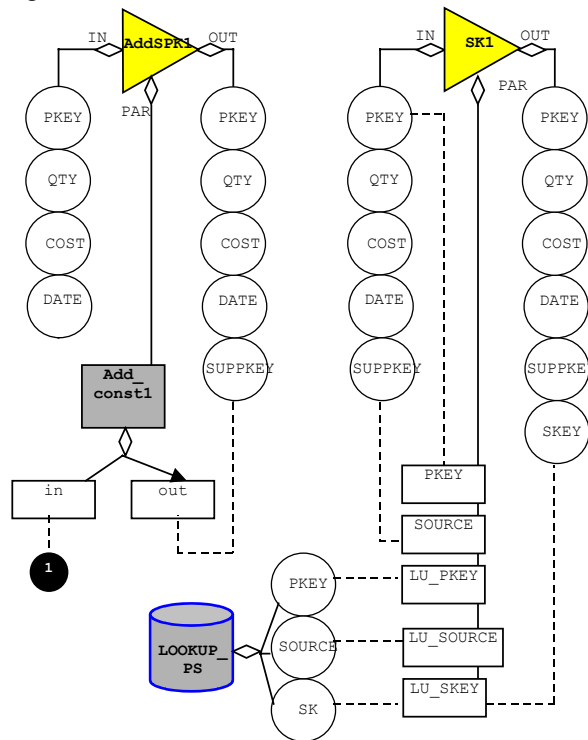


Fig. 3.5 Regulator relationships of the architecture graph

In the example of Fig. 3.5 we can observe how the parameters of the two activities are populated. First, we can see that activity *Add_SPK₁* receives an integer (1) as its input and uses the function *Add_const₁* to populate its attribute *SUPPKEY*. The parameters *in* and *out* are mapped to the respective terms through regulator relationships. The same applies also for activity *SK₁*. All its parameters, namely *PKEY*,

SOURCE, LU_PKEY, LU_SOURCE and LU_SKEY, are mapped to the respective attributes of either the activity's input schema or the employed lookup table LOOKUP_PS.

The parameter LU_SKEY deserves particular attention. This parameter is (a) populated from the attribute SKEY of the lookup table and (b) used to populate the attribute SKEY of the output schema of the activity. Thus, two regulator relationships are related with parameter LU_SKEY, one for each of the aforementioned attributes. The existence of a regulator relationship among a parameter and an output attribute of an activity, normally denotes that some external data provider is employed in order to derive a new attribute through the activity, through the respective parameter.

Provider relationships. The last thing to add in the architecture graph is the provider relationships that capture the data flow from the sources towards the target recordsets in the data warehouse. Provider relationships are depicted with bold solid arrows that stem from the provider and end in the consumer attribute.

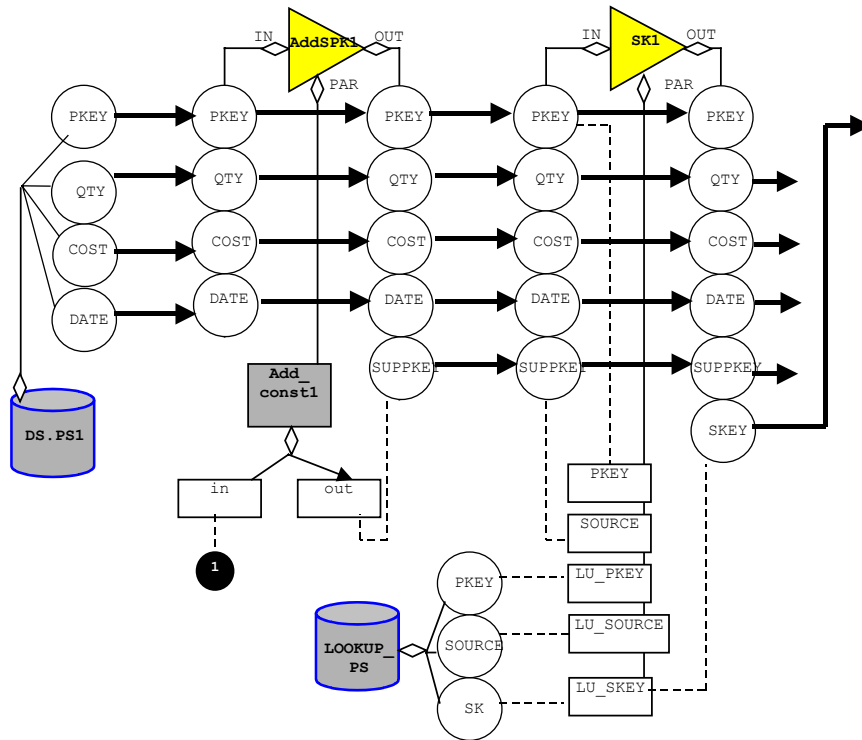


Fig. 3.6 Provider relationships of the architecture graph

Observe Fig. 3.6. The flow starts from table **DS.PS1** of the data staging area. Each of the attributes of this table is mapped to an attribute of the input schema of activity **Add_SPK1**. The attributes of the input schema of the latter are subsequently mapped to the attributes of the output schema of the activity. The flow continues from activity **Add_SPK1** towards the activity **SK1** in a similar manner. Note that, for the moment, we

have not covered how the output of function Add_Const_1 populates the output attribute $SUPPKEY$ for the activity $AddSPK_1$, or how the parameters of activity SK_1 populate the output attribute $SKEY$. This shortcoming is compensated through the usage of derived provider relationships, which we will introduce in the sequel.

Another interesting thing is that during the data flow, new attributes are generated, resulting on new ‘streams’ of data, whereas the flow seems to stop for other attributes. Observe the rightmost part of Fig. 3.6 where the values of attribute $PKEY$ are not further propagated (remember that the reason for the application of a surrogate key transformation is to replace the production keys of the source data to a homogeneous surrogate for the records of the data warehouse, which is independent of the source they have been collected from). Instead of the values of the production key, the values from the attribute $SKEY$ will be used to denote the unique identifier for a part in the rest of the flow.

Derived provider relationships. As we have already mentioned, there are certain output attributes that are computed through the composition of input attributes and parameters. A *derived provider relationship* is another form of provider relationship that captures the flow from the input to the respective output attributes.

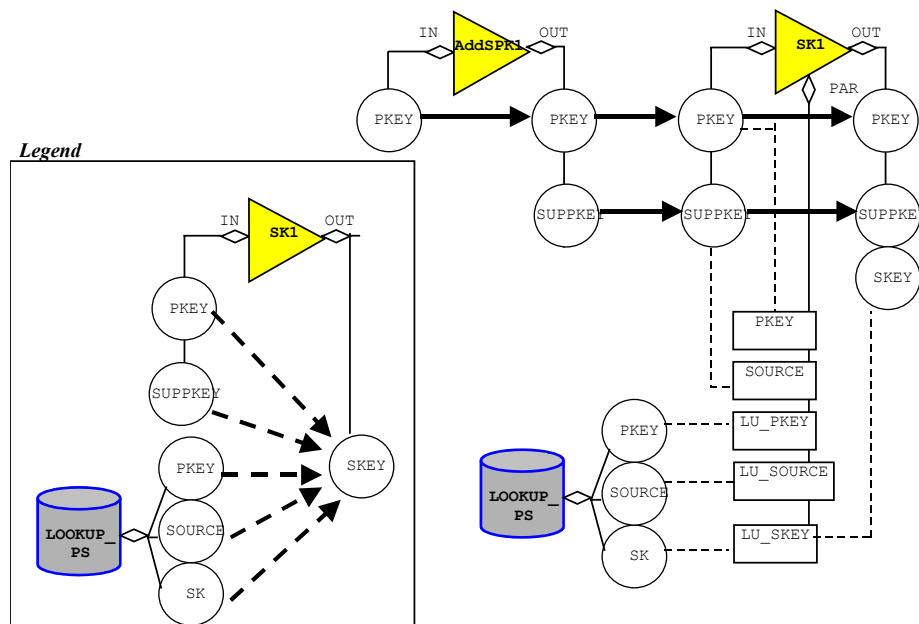


Fig. 3.7 Derived provider relationships of the architecture graph

Formally, assume that $source$ is a term in the architecture graph, $target$ is an attribute of the output schema of an activity A and x, y are parameters in the parameter list of A . The parameters x and y need not necessarily be different with each other. Then, a derived provider relationship $pr(source, target)$ exists iff the

following regulator relationships (i.e., edges) exist: $rr_1(\text{source}, x)$ and $rr_2(y, \text{target})$.

Intuitively, the case of derived relationships models the situation where the activity computes a new attribute in its output. In this case, the produced output depends on all the attributes that populate the parameters of the activity, resulting in the definition of the corresponding derived relationship.

Observe Fig. 3.7, where we depict a small part of our running example. The legend in the left side of Fig. 3.7 depicts how the attributes that populate the parameters of the activity are related through derived provider relationships with the computed output attribute `SKEY`. The meaning of these five relationships is that `SK1.OUT.SKEY` is not computed only from attribute `LOOKUP_PS.SKEY`, but from the combination of all the attributes that populate the parameters.

As far as the parameters of activity `AddSPK1` are concerned, we can also detect a derived provider relationship, between the constant `1` and the output attribute `SUPPKEY`. Again, in this case, the constant is the only term that applies for the parameters of the activity and the output attribute is linked to the parameter schema through a regulator relationship.

One can also assume different variations of derived provider relationships such as (a) relationships that do not involve constants (remember that we have defined `source` as a term); (b) relationships involving only attributes of the same/different activity (as a measure of internal complexity or external dependencies); (c) relationships relating attributes that populate only the same parameter (e.g., only the attributes `LOOKUP_PS.SKEY` and `SK1.OUT.SKEY`).

4. Exploitation of the Architecture Graph

In this section, we provide results on the exploitation of the Architecture Graph for several tasks. The architecture graph is a powerful, but complex, tool; thus, we discuss several simple transformations that reduce the complexity of the graph. Specifically, in Section 4.1, we give a simple algorithm for zooming out the graph, a transformation that can be very useful for its visualization. Also, we give a simple algorithm that returns a subgraph involving only the critical entities for the population of the target recordsets of the scenario. Then, in Section 4.2, we measure the importance and vulnerability of the nodes of the graph through specific *importance metrics*, namely *dependence* and *responsibility*. Dependence stands for the degree to which an entity is bound to other entities that provide it with data and responsibility measures the degree up to which other nodes of the graph depend on the node under consideration. Dependence and responsibility are crucial measures for the engineering of the evolution of the ETL environment.

4.1 Graph Transformations

In this subsection, we will show how we can employ trivial transformations in order to eliminate the detailed information on the attributes involved in an ETL scenario. Each transformation that we discuss is providing a ‘view’ on the graph. These views are simply subgraphs with a specific purpose. As we have shown in Section 3, the simplest view we can offer is by restricting the subgraph to only one of the four major types of relationships (*provider, part-of, instance-of, regulator*). We consider this as trivial, and we proceed to present two transformations, involving (a) how we can zoom out the architecture graph, in order to eliminate the information overflow, which can be caused by the vast number of involved attributes in a scenario and (b) how we can obtain a critical subgraph of the Architecture Graph that includes only the entities necessary for the population of the target recordsets of the scenario.

<p>Transformation <i>Zoom_Out</i> Input: the architecture graph $G(V, E)$ and a structured entity A Output: a new architecture graph $G'(V', E')$ Begin $G' = G;$ \forall node $t \in V',$ s.t. $\neg \exists$ edge $(A, t) \in Po' \wedge \exists$ edge $(A, x) \in Po'$ { \forall edge $(t, x) \in Pr': Pr' = Pr' \cup (t, A) - (t, x);$ \forall edge $(x, t) \in Pr': Pr' = Pr' \cup (A, t) - (x, t);$ \forall edge $(t, x) \in Rr': Rr' = Rr' \cup (t, A) - (t, x);$ } \forall node $t \in V',$ s.t. \exists edges $(A, t) \in Po', (A, x) \in Po'$ { \forall edge $(t, x) \in Pr': Pr' = Pr' - (t, x);$ \forall edge $(x, t) \in Pr': Pr' = Pr' - (x, t);$ \forall edge $(t, x) \in Rr': Rr' = Rr' - (t, x);$ remove $t;$ } End</p>

Fig. 4.1 *Zoom_Out* transformation

Zooming In and Out the Architecture Graph. We give a practical zoom out transformation that involves provider and regulator relationships. We constraint the algorithm of Fig. 4.1 to a local transformation, i.e., we consider zooming out only a single activity or recordset. This can easily be generalized for the whole scenario, too. Assume a given structured entity (activity or recordset) A . The transformation *Zoom_Out* of Fig. 4.1, detects all the edges of its attributes. Then all these edges are transferred to link the structured entity A (instead of its attributes) with the corresponding nodes. We consider only edges that link an attribute of A to some node external to A , in order avoid local cycles in the graph. Finally, we remove the attribute nodes of A and the remaining internal edges. Note that there is no loss of information due to this relationship, in terms of interdependencies between objects. Moreover, we

can apply this local transformation to the full extent of the graph, involving the attributes of all the recordsets and activities of the scenario.

Major Flow. In a different kind of zooming, we can follow the major flow of data from sources to the targets. We follow a backward technique. Assume the set of recordsets \mathbf{T} , containing a set of target recordsets. Then, by recursively following the provider and regulator edges we can deduce the critical subgraph that models the flow of data from sources towards the critical part of the data warehouse. We incorporate the part-of relationships too, but we choose to ignore instantiation information. The transformation *Major_Flow* is shown in Fig. 4.2.

<p>Transformation <i>Major_Flow</i> Input: the architecture graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ and the set of target recordsets \mathbf{T}. Output: a sub graph $\mathbf{G}'(\mathbf{V}', \mathbf{E}')$ containing information for the major flow of data from sources to targets. Begin Let $\mathbf{T}\Omega$ be the set of attributes of all the recordsets of \mathbf{T}; $\mathbf{V}' = \mathbf{T} \cup \mathbf{T}\Omega$; do { $\mathbf{V}'' = \emptyset$; $\forall t \in \mathbf{V}'', a \in \mathbf{V}, e(a, t) \in \mathbf{Pr}: \{ \mathbf{V}'' = \mathbf{V}'' \cup \{t\}; \mathbf{E}' = \mathbf{E}' \cup \{e\} \}$; $\forall t \in \mathbf{V}'', a \in \mathbf{V}, e(a, t) \in \mathbf{Po}: \{ \mathbf{V}'' = \mathbf{V}'' \cup \{t\}; \mathbf{E}' = \mathbf{E}' \cup \{e\} \}$; $\forall t \in \mathbf{V}'', a \in \mathbf{V}, e(t, a) \in \mathbf{Po}: \{ \mathbf{V}'' = \mathbf{V}'' \cup \{t\}; \mathbf{E}' = \mathbf{E}' \cup \{e\} \}$; $\forall t \in \mathbf{V}'', a \in \mathbf{V}, e(a, t) \in \mathbf{Rr}: \{ \mathbf{V}'' = \mathbf{V}'' \cup \{t\}; \mathbf{E}' = \mathbf{E}' \cup \{e\} \}$; $\mathbf{V}' = \mathbf{V}' \cup \mathbf{V}''$; } while $\mathbf{V}'' \neq \emptyset$; End</p>
--

Fig. 4.2 *Major_Flow* transformation

4.2 Importance Metrics

One of the major contributions that our graph-modeling approach offers is the ability to treat the scenario as the skeleton of the overall environment. If we treat the problem from its software engineering perspective, the interesting problem is how to design the scenario in order to achieve effectiveness, efficiency and tolerance of the impacts of evolution. In this subsection, we will assign simple *importance metrics* to the nodes of the graph, in order to measure how crucial their existence is for the successful execution of the scenario.

Consider the subgraph $\mathbf{G}'(\mathbf{V}', \mathbf{E}')$ that includes only the part-of, provider and derived provider relationships among attributes. In the rest of this subsection, we will not discriminate between provider and derived provider relationships and will use the term ‘provider’ for both. For each node A , we can define the following measures:

- *Local dependency*: the in-degree of the node with respect to the provider edges;
- *Local responsibility*: the out-degree of the node with respect to the provider edges;
- *Local degree*: the degree of the node with respect to the provider edges (i.e., the sum of the previous two entries).

Intuitively, the local dependency characterizes the number of nodes that have to be ‘activated’ in order to populate a certain node. The local responsibility has the reverse meaning, i.e., how many nodes wait for the node under consideration to be activated, in order to receive data. The sum of the aforementioned quantities characterizes the total involvement of the node in the scenario.

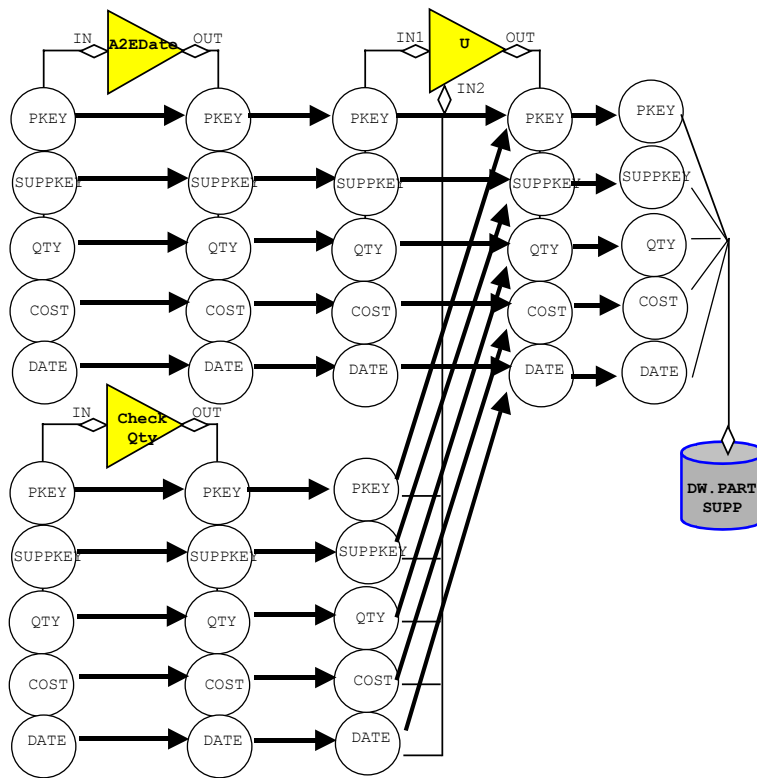


Fig. 4.3 Dependencies on the architecture graph

Consider the example of Fig. 4.3, where we depict three of the activities of our example. Specifically, we consider the activity *A2Edate* that converts the dates for the data of source S_1 to European format, the activity *CheckQty* that discards any rows with a non-positive quantity and the *Union* activity that consolidates the two flows towards the data warehouse table *DW.PARTSUPP*. Observe the node

UNION.OUT.PKEY of Fig. 4.3. Its local dependency is 2, its local responsibility is 1 and its local degree is 3.

Except for the local interrelationships we can always consider what happens with the transitive closure of relationships. Assume the set $(Pr \cup Dr)^+$, which contains the transitive closure of provider edges. We define the following measures:

- *Transitive dependency*: the in-degree of the node with respect to the provider edges;
- *Transitive responsibility*: the out-degree of the node with respect to the provider edges;
- *Transitive degree*: the degree of the node with respect to the provider edges.
- *Total dependency*: the sum of local and transitive dependency measures;
- *Total responsibility*: the sum of local and transitive responsibility measures;
- *Total degree*: the sum of local and transitive degrees.

Consider the example of Fig. 4.4, where we have computed the transitive closure of provider edges for the example of Fig. 3.7. We use standard notation for provider and derived provider edges and depict the edges of transitive relationships with simple dotted arrows. Figure 4.5 depicts the aforementioned metrics for the attributes of Fig. 3.7.

By comparing the individual values with the average ones, one can clearly see that attribute SK1.OUT.SKEY, for example, is the most ‘vulnerable’ attribute of all, since it depends directly on several provider attributes.

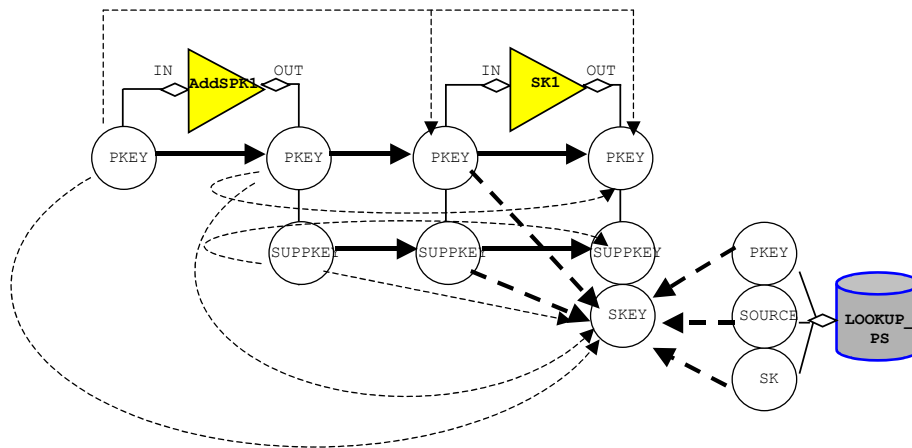


Fig. 4.4 Transitive closure for the entities of Fig. 3.7

Other interesting usages of the aforementioned measures include:

- *Detection of inconsistencies for attribute population*. For example, if some output attribute in a union activity is not populated by two input attributes, or, more importantly, if an integrity constraint is violated with regards to the

population of an attribute of a target data store (i.e., having in-degree equal to zero).

- *Detection of important data stores.* Clearly, data stores whose attributes have a positive out-degree are used for data propagation. Data stores with attributes having positive both in and out degrees, are transitive recordsets, which we use during the flow.
- *Detection of useless (source) attributes.* Any attribute having total responsibility equal to zero is useless for the cause of data propagation towards the sources.
- *Observations after zooming out the whole scenario.* Assume we have applied the zoom out transformation to the whole scenario. Again we can detect inconsistencies for activity/recordset population (as we did before for the case of attributes), or data stores with in-degree greater than one, which means that the impact of their evolution is critical since they are related with more than one applications.

	LOCAL			TRANSITIVE			TOTAL		
	IN	OUT	DEGREE	IN	OUT	DEGREE	IN	OUT	DEGREE
AddSPK1									
IN.PKEY	0	1	1	0	3	3	0	4	4
OUT.PKEY	1	1	2	0	2	2	1	3	4
OUT.SUPPKEY	0	1	1	0	2	2	0	3	3
SK1									
IN.PKEY	1	2	3	1	0	1	2	2	4
IN.SUPPKEY	1	2	3	0	0	0	1	2	3
OUT.PKEY	1	0	1	2	0	2	3	0	3
OUT.SUPPKEY	1	0	1	1	0	1	2	0	2
OUT.SKEY	5	0	5	3	0	3	8	0	8
LOOKUP PS									
PKEY	0	1	1	0	0	0	0	1	1
SOURCE	0	1	1	0	0	0	0	1	1
SK	0	1	1	0	0	0	0	1	1
SUM	10	10	20	7	7	14	17	17	34
AVG	1.67	1.67	3.33	1.17	1.17	2.33	2.83	2.83	5.67

Fig. 4.5 Importance metrics for the attributes of Fig. 4.4

We believe it is important to stress that the measures we have introduced in this section are not only applicable to attributes. The zoom-out transformation that we have defined in the previous subsection allows us to generalize these definitions to activities and recordsets, too. Moreover, we can define similar importance metrics for data/function types, to determine how useful and/or critical they are for the prompt execution of the ETL scenario. The only difference in this case, is that instead of

provider, we employ instantiation relationships. In Fig. 4.6 we can see the aggregated importance metrics of the structured entities that we have employed in our example.

Finally, we would like to comment on the usefulness of the introduced metrics. It is clear that the engineering of complex data flows is not a clearly resolved issue. The overloading of attributes, activities and recordsets can be a burden for the optimal execution of the scenario. The optimization of the overall process is an interesting problem, which our modeling facilitates gracefully (we already have some preliminary results on the issue). Most importantly, though, we find that the importance of these metrics lies in the evolution of the ETL scenario. As we mentioned before, we understand the Architecture Graph as the skeleton of the overall environment, around which the applications are built. By measuring the importance of each entity, we can predict the impact of modifying it.

	LOCAL			TRANSITIVE			TOTAL		
	IN	OUT	DEGREE	IN	OUT	DEGREE	IN	OUT	DEGREE
AddSPK1									
SUM	1	3	4	0	7	7	1	10	11
AVG/attribute	0.33	1.00	1.33	0.00	2.33	2.33	0.33	3.33	3.67
SK1									
SUM	9	4	13	7	0	7	16	4	20
AVG/attribute	1.8	0.8	2.6	1.4	0	1.4	3.2	0.8	4
LOOKUP_PS									
SUM	0	3	3	0	0	0	0	3	3
AVG/attribute	0	1	1	0	0	0	0	1	1
SUM	10	10	20	7	7	14	17	17	34
AVG/entity	3.33	3.33	6.67	2.33	2.33	4.67	5.67	5.67	11.33

Fig. 4.6 Importance metrics for the structured entities of Fig. 4.4

5. Related Work

In this section we discuss the state of art and practice for research efforts, commercial tools and standards in the field of ETL tools, along with any related technologies.

Commercial tools and Standards. Basically, commercial ETL tools are responsible for the implementation of the data flow in a data warehouse environment, which is only one (albeit important) of the data warehouse processes. Most ETL tools are of two flavors: *engine-based*, or *code-generation based*. The former assumes that all data have to go through an engine for transformation and processing. In code-generating tools all processing takes place only at the target or source systems. There is a variety of such tools in the market; we mention three engine-based tools, from Ardent [Arde01], DataMirror [Data01] and Microsoft [Mitr01], and one code-

generation based from ETI [ETI01]. The *Open Information Model (OIM)* [MeCo99] is a proposal (led by Microsoft) for the core metadata types found in the operational and data warehousing environment of enterprises.

Research focused specifically on ETL. The *AJAX* data cleaning tool developed at INRIA [GFSS00] deals with typical data quality problems, such as the object identity problem, errors due to mistyping and data inconsistencies between matching records. AJAX provides a framework wherein the logic of a data cleaning program is modeled as a directed graph of data transformations (mapping, matching, clustering and merging transformations) that start from some input source data. AJAX also provides a declarative language for specifying data cleaning programs, which consists of SQL statements enriched with a set of specific primitives to express mapping, matching, clustering and merging transformations. [RaHe00, RaHe01] present the *Potter's Wheel* system, which is targeted to provide interactive data cleaning to its users. The system offers the possibility of performing several algebraic operations over an underlying data set, including *format* (application of a function), *drop*, *copy*, *add* a column, *merge* delimited columns, *split* a column on the basis of a regular expression or a position in a string, *divide* a column on the basis of a predicate (resulting in two columns, the first involving the rows satisfying the condition of the predicate and the second involving the rest), *selection* of rows on the basis of a condition, *folding* columns (where a set of attributes of a record is split into several rows) and *unfolding*. Optimization algorithms are also provided for the CPU usage for certain classes of operators. The general idea behind Potter's Wheel is that users build data transformations in iterative and interactive way. In the background, Potter's Wheel automatically infers structures for data values in terms of user-defined domains, and accordingly checks for constraint violations. Moreover, in previous lines of research [VQVJ01, VVS+01] there was a first effort to cover the design aspects by trying (a) to show how data warehouse processes can be linked to a metadata repository; (b) to construct a running tool and (c) to cover some quality aspects of the data warehouse process. Still, these approaches were not considering the inner structure of the activities and were not tailored specifically for ETL processes.

Research on Data Cleaning. Data cleaning is another step in the ETL process, which unfortunately has not caught the attention of the research community. Still, [RaDo00] provide an extensive overview of the field, along with research issues and a review of some commercial tools. [Mong00] discusses a special case of the data cleaning process, namely the detection of duplicate records and extends previous algorithms on the issue. [BoDS00] focuses on another subproblem, namely the one of breaking address fields into different elements and suggest the training of a Hidden Markov Model to solve the problem.

Data Quality and Quality Management. There has been a lot of research on the definition and measurement of data quality dimensions [Wang98, WaKM93, WaSF95, WaWa96]. A very good review of research literature is found in [WaSF95]. [JJQV99] give an extensive list of quality dimensions for data warehouses, and in particular data warehouse relations and data.

Workflow and Process Modeling. Modeling ETL scenarios can be considered as a special case of the general problem of workflow and process modeling. There is a widely accepted standard proposed by the Workflow Management Coalition (WfMC) [WfMC98]. As far as process modeling is concerned, we reference the interested

reader to [Roll98] for a recent overview of the field. Many ideas about the organization of data warehouse activities in logical, conceptual and physical entities stem from [JaPo92], [JaJR90]. In another line of research, [BoFM99] is the first attempt to clearly separate the data warehouse refreshment process from its traditional treatment as a view maintenance or bulk loading process. The authors provide a conceptual model of the process, which is treated as a composite workflow.

6. Conclusions

In this paper, we have focused on the logical design of the ETL scenario of a data warehouse. We have defined a formal logical model for ETL processes. The data stores, activities and their constituent parts have been formally introduced. An ETL scenario has been defined as the combination of ETL activities and data stores. Then, we have shown how this model is reduced to a graph, which we call the *Architecture Graph*. We model all the aforementioned entities as nodes and four different kinds of relationships as edges. These relationships involve (a) type checking information (i.e., which type an entity corresponds to), (b) part-of relationships (e.g., which activity does an attribute belong to), (c) regulator relationships, covering the population of activity parameters from attributes or constant values and (d) provider relationships, covering the flow of data from providers to consumers. Finally, we have shown how we can exploit the Architecture Graph for several tasks. First we have provided several simple graph transformations that reduce the complexity of the graph. Second, we have introduced specific *importance metrics*, namely *dependence* and *responsibility*, to measure the degree to which an entity is bound to other entities that provide it with data and the degree up to which other nodes of the graph depend on the node under consideration.

As future work, we already have preliminary results for the optimization of ETL scenario under certain time and throughput constraints. A set of loosely coupled tools is also under construction for the purposes of visualization of the ETL scenarios and optimization of their execution.

References

- [Arde01] Ardent Software. DataStage Suite. Available at <http://www.ardentsoftware.com/>
- [BoDS00] V. Borkar, K. Deshmuk, S. Sarawagi. Automatically Extracting Structure from Free Text Addresses. Bulletin of the Technical Committee on Data Engineering, **23**(4), (2000).
- [BoFM99] M. Bouzeghoub, F. Fabret, M. Matulovic. Modeling Data Warehouse Refreshment Process as a Workflow Application. In Proc. Intl. Workshop on Design and Management of Data Warehouses (DMDW'99), Heidelberg, Germany, (1999).
- [Data01] DataMirror Corporation. Transformation Server. Available at <http://www.datamirror.com>
- [Dema97] M. Demarest. The politics of data warehousing.

- <http://www.hevanet.com/demarest/marc/dwpol.html> (1997).
- [ETI01] Evolutionary Technologies Intl. ETI*EXTRACT. Available at <http://www.eti.com/>
- [GFSS00] H. Galhardas, D. Florescu, D. Shasha and E. Simon. Ajax: An Extensible Data Cleaning Tool. In Proc. ACM SIGMOD Intl. Conf. On the Management of Data, pp. 590, Dallas, Texas, (2000).
- [JaJR90] M. Jarke, M.A. Jeusfeld, T. Rose: A software process data model for knowledge engineering in information systems. *Information Systems* **15**(1): 85-116 (1990).
- [JaPo92] M. Jarke, K. Pohl. Information systems quality and quality information systems. In Kendall/Lyytinen/DeGross (eds.): Proc. IFIP 8.2 Working Conf. The Impact of Computer-Supported Technologies on Information Systems Development, pp. 345-375, Minneapolis (1992).
- [JJQV99] M. Jarke, M.A. Jeusfeld, C. Quix, P. Vassiliadis: Architecture and quality in data warehouses: An extended repository approach. *Information Systems*, **24**(3): 229-253 (1999). A previous version appeared in Proc. 10th Conf. of Advanced Information Systems Engineering (CAiSE '98), Pisa, Italy (1998).
- [KRRT98] R. Kimbal, L. Reeves, M. Ross, W. Thornthwaite. The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses. John Wiley & Sons, February 1998.
- [MeCo99] MetaData Coalition. Open Information Model, version 1.0, 1999. Available at www.MDCinfo.com
- [Micr01] Microsoft Corp. MS Data Transformation Services. Available at <http://www.microsoft.com/sql/bizsol/comanddts.htm>
- [Mong00] A. Monge. Matching Algorithms Within a Duplicate Detection System. *Bulletin of the Technical Committee on Data Engineering*, **23**(4), (2000).
- [RaDo00] E. Rahm, H. Do. Data Cleaning: Problems and Current Approaches. *Bulletin of the Technical Committee on Data Engineering*, **23**(4), (2000).
- [RaHe00] V. Raman, J. Hellerstein. Potters Wheel: An Interactive Framework for Data Cleaning and Transformation. Technical Report University of California at Berkeley, Computer Science Division, 2000. Available at <http://www.cs.berkeley.edu/~rshankar/papers/pwheel.pdf>
- [RaHe01] V. Raman, J. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pp. 381-390, Roma, Italy (2001).
- [Roll98] C. Rolland: A comprehensive view of process engineering. In Proc. 10th Intl. Conf. Advanced Information Systems Engineering, (CAiSE'98), pp. 1-25, Pisa, Italy (1998).
- [ShTy98] C. Shilakes, J. Tylman. Enterprise Information Portals. Enterprise Software Team. Available at <http://www.sagemaker.com/company/downloads/eip/indepth.pdf> (1998).
- [TPC00] Transaction Processing Performance Council. TPC-H and TPC-R, 2000. Available at www.tcp.org
- [VaSk00] P. Vassiliadis, S. Skiadopoulos. Modelling and Optimization Issues for Multidimensional Databases. In Proc. 12th Conference on Advanced Information Systems Engineering (CAiSE '00), pp. 482-497, Stockholm, Sweden, 5-9 June 2000. *Lecture Notes in Computer Science*, Vol. 1789, Springer, 2000.
- [Vass00] P. Vassiliadis. Gulliver in the land of data warehousing: practical experiences and observations of a researcher. In Proc. 2nd Intl. Workshop on Design and Management of Data Warehouses (DMDW), pp. 12.1 –12.16, Stockholm, Sweden (2000).

- [VaSS02] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Conceptual Modeling for ETL Processes. Submitted to ER2002.
- [VQVJ01] P. Vassiliadis, C. Quix, Y. Vassiliou, M. Jarke. Data Warehouse Process Management. Information Systems, vol. 26, no.3, pp. 205-236, June 2001.
- [VVS+01] P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis, T. Sellis. ARKTOS: Towards the modeling, design, control and execution of ETL processes. Information Systems, **26**(8), pp. 537-561, December 2001, Elsevier Science Ltd.
- [WaKM93] R.Y. Wang, H.B. Kon, S.E. Madnick. Data Quality Requirements Analysis and Modeling. In Proc. of 9th Intl. Conf. On Data Engineering, pp. 670-677, IEEE Computer Society, Vienna, Austria (1993).
- [Wang98] R. Y. Wang. A product perspective on total data quality management. Comm. of the ACM, 41(2): 58-65 (1998).
- [WaSF95] R.Y. Wang, V.C. Storey, C.P. Firth. A Framework for Analysis of Data Quality Research. IEEE Transactions on Knowledge and Data Engineering, 7(4): 623-640 (1995).
- [WaWa96] Y. Wand, R.Y. Wang. Anchoring Data Quality Dimensions in Ontological Foundations. Communications of the ACM, 39(11): 86-95 (1996).
- [WfMC98] Workflow Management Coalition. Interface 1: Process Definition Interchange Process Model. Document number WfMC TC-1016-P (1998). Available at www.wfmc.org