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. Based on a formal logical model that includes the data stores, activities and their constituent parts, we model an ETL scenario as 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. In addition, 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

Related literature has characterized data warehouse processes as complex [BoFM99], costly and critical (covering thirty to eighty percent of effort and expenses of the overall data warehouse construction) [ShTy98, Vass00]. In order to facilitate and manage these data warehouse operational processes, specialized tools are already available in the market [Arde01, Data01, Micr01, ETI01], 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 (a) the identification of relevant information at the source side; (b) the extraction of this information; (c) the customization and integration of the information coming from multiple sources into a common format; (d) the cleaning of the resulting data set, on the basis of database and business rules, and (e) 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.

[KRRT98] gives an informal but detailed methodology for the management of ETL activities. Research has also provided preliminary results on the modeling and optimization of ETL activities. The AJAX data cleaning tool [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 mapping, matching, clustering and merging transformations over some input data. [RaHe01] present the Potter's Wheel data cleaning system, which offers the possibility of performing several algebraic operations over an underlying data set in an interactive/iterative way. [RaDo00] provides an extensive overview of the field of data cleaning, 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. Moreover, in previous lines of research [VQVJ01, VVS+01] there was a first effort to cover the design aspects of ETL 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.

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

We believe that these approaches have not considered the inner structure of the ETL activities in sufficient depth. In this paper, we start from a general framework for the logical design of an ETL scenario and focus on the internal structure of ETL activities (practically exploiting their data centric nature). We employ a uniform paradigm, namely graph modeling, for both the scenario at large and the internals of each activity. Our contributions can be listed as follows:

- First, *we briefly present a logical model* that includes the data stores, ETL activities and their constituent parts. 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.
- 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.

The reader is strongly encouraged to refer to the long version of the paper [VaSS02a] with further results and discussions that we cannot present here for lack of space. This paper is organized as follows. Section 2 presents how an ETL scenario can be modeled as a graph. Section 3 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 4 we conclude our results.

2 The Architecture Graph of an ETL Scenario

In this section, we first give a formal definition of activities, recordsets and other constituents of an ETL scenario. The full layout of such an ETL scenario can be modeled by a graph, which we call the *Architecture Graph*. Apart from this static description, the architecture graph also captures the data flow within the ETL environment. At the same time, the information on the typing of the involved entities and the regulation of the execution of a scenario, through specific parameters are also covered.

2.1 Preliminaries

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. In the sequel, we summarize the definition of these elements and refer the interested reader to [VaSS02a] for more details.

- *Data types*. Each data type `T` is characterized by a name and a domain, i.e., a countable set of values. The values of the domains are also referred to as *constants*.
- *Attributes*. Attributes are characterized by their name and 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*.
- *RecordSets.* 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). As mentioned in

[VQVJ01], we can treat any data structure as a "record set" provided that there are the means to logically restructure it into a flat, typed record schema. In the rest of this paper, we will mainly deal with the two most popular types of recordsets, namely *relational tables* and *record files*.

- *Functions*. 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.
- *Elementary Activities*. In our framework, activities are logical abstractions representing parts, or full modules of code. 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. 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.

To fully capture the characteristics and interactions of the static entities mentioned previously, 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 `1:N` relationships that involve attributes with a provider-consumer relationship. 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. Provider relationships capture the mapping between the attributes of the schemata of the involved entities. Note that a consumer attribute can also be populated by a constant, in certain cases.
- *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.

We assume the infinitely countable, mutually disjoint sets of names of column *Model-specific* in Fig. 2.1. As far as a specific scenario is concerned, we assume their respective finite subsets, depicted in column *Scenario-Specific* in Fig. 2.1. 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*). Formally, let

G(V,E) be the Architecture Graph of an ETL scenario. Then, **V** = **D**∪**F**∪**C**∪**Ω**∪**Φ**∪**S**∪**RS**∪**A** and **E** = **Pr**∪**Po**∪**Io**∪**Rr**∪**Dr**. The graphical notation for the Architecture Graph is depicted in Fig. 2.2.

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

Fig. 2.1 Formal definition of domains and notation

Data Types	Black ellipsis	Integer
Function Types	Black squares	$2€
Constants	Black cycles	1
Attributes	Hollow ellipsoid nodes	PKEY

RecordSets	Cylinders	R
Functions	Gray squares	my$2€
Parameters	White squares	rate
Activities	Triangles	SK

Part-Of Relationships	Simple edges annotated with diamond*	◇——
Instance-Of Relationships	Dotted arrows (from instance towards the type)	------▶
Regulator Relationships	Dotted edges	--------

Provider Relationships	Bold solid rows (from provider to consumer)	——▶
Derived Provider Relationships	Bold dotted arrows (from provider to consumer)	- - - -▶

* 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. 2.2 Graphical notation for the Architecture Graph.

2.2 Constructing the Architecture graph

In this subsection we will describe the precise structure of the architecture graph, based on the theoretical foundations and the graphical notation of the previous subsection. Clearly, we do not anticipate a manual construction of the entire graph by the designer, but rather, we anticipate that the designer will need to specify only the high level parts of the Architecture Graph during the construction of an ETL scenario. In general, this process can be facilitated by a graphical tool or a declarative language; we refer the interested reader to [VVS+01] for an example of both these alternatives.

To motivate our discussion we will present an example involving the existence of (a) two source databases S_1 and S_2; (b) a central data warehouse DW and (c) a Data Staging Area (DSA), where all the transformations take place. The scenario involves the propagation of data from the table PARTSUPP(PKEY, DATE, QTY, COST) of source S_1 as well as from the table PARTSUPP(PKEY, QTY,

COST) of source S_2 to the data warehouse. Table DW.PARTSUPP (PKEY, SUPPKEY, DATE, QTY, COST) stores information for the available quantity (QTY) and cost (COST) of parts (PKEY) per supplier (SUPPKEY). Practically, the two data sources S_1 and S_2 stand for the two suppliers of the data warehouse. The full-scale example can be found in the long version of the paper [VaSS02a]. Here we will employ only a small part of it, where the data from source S_1 have just been transferred to the DSA in an intermediate table $DS.PS_1$(PKEY,DATE,QTY,COST). We will particularly focus on two activities of the flow from table $DS.PS_1$ towards the target table DW.PARTSUPP.

1. In order to keep track of the supplier of each row (and ultimately populate attribute SUPPKEY of the table DW.PARTSUPP), we need to add a 'flag' attribute, namely SUPPKEY, indicating 1 or 2 for the respective supplier. In our case, this is achieved through the activity Add_SPK_1.
2. Then, we need to assign a surrogate key on attribute PKEY. 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* [KRRT98]. 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 activity that performs the surrogate key assignment for the attribute PKEY is SK_1, using the lookup table LOOKUP_PS(PKEY, SOURCE, SKEY).

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. 2.3a depicts the decomposition of the recordsets $DS.PS_1$, LOOKUP_PS and 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 propagates it towards the SUPPKEY attribute, in order to trace the fact that the 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.

Data types and instance-of relationships. Instantiation relationships are depicted as dotted arrows that stem from the instances and head towards their data/function types. In Fig. 2.3b, 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. 2.3b, we can also see the fact that function Add_const_1 is an instance of the function type ADD_CONST.

Parameters and regulator relationships. 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. In the example of Fig. 2.3a we can observe how the parameters of the two activities are populated. First, we can see that activity Add_SPK$_1$ receives an integer (1) as its input and uses the function Add_const$_1$ 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 SK$_1$. 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.

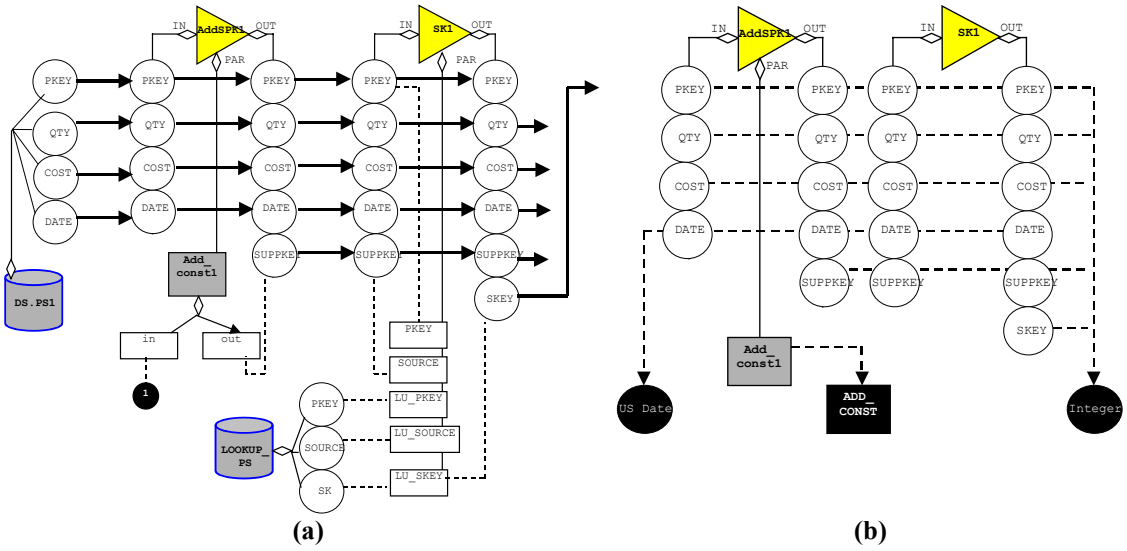


Fig. 2.3 Relationships of the architecture graph: (a) Part-of, provider and regulator relationships; (b) Instance-of relationships

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. As already mentioned, the provider relationships 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. Observe Fig. 2.3a. The flow starts from table DS.PS$_1$ of the data staging area. Each of the attributes of this table is mapped to an attribute of the input schema of activity Add_SPK$_1$. 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_SPK$_1$ towards the activity SK$_1$ 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. 2.3a 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. 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`(source,x)` and rr_2`(y, target)`.

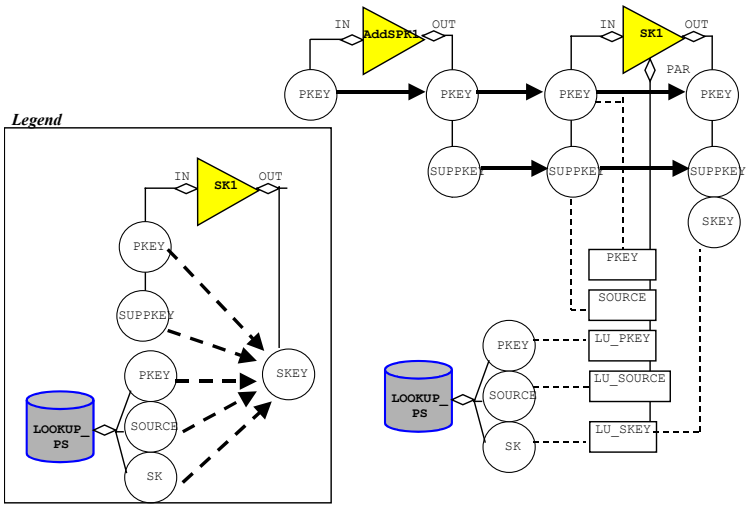


Fig. 2.4 Derived provider relationships of the architecture graph

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. 2.4, where we depict a small part of our running example. The legend in the left side of Fig. 2.4 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 SK_1`.OUT.SKEY` is not computed only from attribute `LOOKUP_PS.SKEY`, but from the combination of all the attributes that populate the parameters.

3. Exploitation of the Architecture Graph

In this section, we provide results on the exploitation of the Architecture Graph for several tasks. Specifically, in Section 3.1, we give a simple algorithm for zooming out the graph, a transformation that can be very useful for it 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 3.2, 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.

3.1 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. One would normally expect that the simplest view is produced 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.

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. 3.1 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. 3.1, 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 **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. 3.2.

Transformation ***Zoom_Out***
Input: the architecture graph **G(V,E)** and a structured entity A
Output: a new architecture graph **G′(V′,E′)**
Begin
G′ = **G;**
∀ node t∈**V′**, s.t. ¬∃ edge (A,t)∈**Po′** ∧ ∃ edge (A,x)∈**Po′** {
∀ edge (t,x)∈**Pr′**: **Pr′**= **Pr′**∪(t,A)-(t,x);
∀ edge (x,t)∈**Pr′**: **Pr′**= **Pr′**∪(A,t)-(x,t);
∀ edge (t,x)∈**Rr′**: **Rr′**= **Rr′**∪(t,A)-(t,x);
}
∀ node t∈**V′**, s.t. ∃ edges (A,t)∈**Po′**, (A,x)∈**Po′**{
∀ edge (t,x)∈**Pr′**: **Pr′**= **Pr′**-(t,x);
∀ edge (x,t)∈**Pr′**: **Pr′**= **Pr′**-(x,t);
∀ edge (t,x)∈**Rr′**: **Rr′**= **Rr′**-(t,x);
remove t;
}
End

Fig. 3.1 ***Zoom_Out*** **transformation**

Transformation ***Major_Flow***
Input: the architecture graph **G(V,E)** and the set of target recordsets **T**.
Output: a sub graph **G′(V′,E′)** containing information for the major flow of data from sources to targets.
Begin
Let **TΩ** be the set of attributes of all the recordsets of **T;**
V′=**T**∪**TΩ;**
do { **V"**=∅;
∀t∈**V"**,a∈**V**,e(a,t)∈**Pr:**{
V"=**V"**∪{t};**E′**=**E′**∪{e} };
∀t∈**V"**,a∈**V**,e(a,t)∈**Po:**{
V"=**V"**∪{t};**E′**=**E′**∪{e} };
∀t∈**V"**,a∈**V**,e(t,a)∈**Po:**{
V"=**V"**∪{t};**E′**=**E′**∪{e} };
∀t∈**V"**,a∈**V**,e(a,t)∈**Rr:**{
V"=**V"**∪{t};**E′**=**E′**∪{e} };
V′=**V′**∪**V"**;
} **while V"**≠∅;
End

Fig. 3.2 ***Major_Flow*** **transformation**

3.2 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 G′(V′,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:

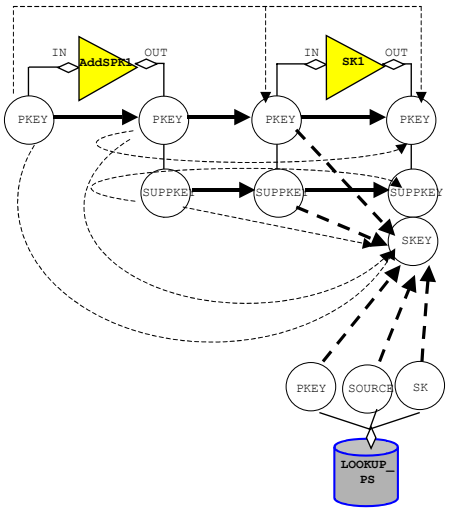
- *Local dependency:* the in-degree of the node with respect to the provider edges;
- *Local responsibility*: the out-degree of the node with respect to the provider edges;

- *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.

Except for the local interrelationships we can always consider what happens with the transitive closure of relationships. Assume the set $(\mathbf{Pr} \cup \mathbf{Dr})^+$, 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.



	LOCAL			TRANSITIVE			TOTAL		
	IN	OUT	DEG	IN	OUT	DEG	IN	OUT	DEG
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
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**	**3.57**
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**
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									
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	**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/attribute	**1.67**	**1.67**	**3.33**	**1.17**	**1.17**	**2.33**	**2.83**	**2.83**	**5.67**
AVG/entity	**3.33**	**3.33**	**6.67**	**2.33**	**2.33**	**4.67**	**5.67**	**5.67**	**11.33**

Fig. 3.3 Transitive closure for the entities of Fig. 2.4 and corresponding importance metrics

Consider the example of Fig. 3.3, where we have computed the transitive closure of provider edges for the example of Fig. 2.4. We use standard notation for provider and derived provider edges and depict the edges of transitive relationships with simple dotted arrows. Figure 3.3 depicts also the aforementioned metrics for the involved attributes. 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. 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. Once a zoom-out operation has been performed, data stores with in-degree greater than one, are hot-spots since they are related with more than one applications.
- *Detection of useless (source) attributes*. Any attribute having total responsibility equal to zero is useless for the cause of data propagation towards the sources.

4. Conclusions

In this paper, we have focused on the logical design of the ETL scenario of a data warehouse. Based on a formal logical model that includes the data stores, activities and their constituent parts, we model an ETL scenario as 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. We have provided several simple graph transformations that reduce the complexity of the graph. Finally, 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.

The results of this paper are part of a larger project on the design and management of ETL activities [VaSS02]. 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

[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).

[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.

[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).

[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).

[ShTy98] C. Shilakes, J. Tylman. Enterprise Information Portals. Enterprise Software Team. Available at http://www.sagemaker.com/company/downloads/eip/indepth.pdf (1998).

[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. The Arktos II Project. http://www.dblab.ece.ntua.gr/~pvassil/projects/arktos_II/index.html

[VaSS02a] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Modeling ETL Activities as Graphs (Extended version), available at http://www.dbnet.ece.ntua.gr/~pvassil/publications/VaSS02_graphs_long.pdf

[VQVJ01] P. Vassiliadis, C. Quix, Y. Vassiliou, M. Jarke. Data Warehouse Process Management. Information Systems, **26**(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.