A GENERIC AND CUSTOMIZABLE FRAMEWORK FOR THE DESIGN OF ETL SCENARIOS

PANOS VASSILIADIS[1], ALKIS SIMITSIS[2], PANOS GEORGANTAS[2], MANOLIS TERROVITIS[2], SPIROS SKIADOPOULOS[2]

[1] University of Ioannina,
Dept. of Computer Science,
Ioannina, Greece
pvassil@cs.uoi.gr

[2] National Technical University of Athens,
Dept. of Electrical and Computer Eng.,
Athens, Greece
{asimi, pgeor, mter, spiros}@dbnet.ece.ntua.gr

Corresponding author:
Panos Vassiliadis
University of Ioannina,
Dept. of Computer Science,
Ioannina, 45110
Greece
Email: pvassil@cs.uoi.gr
Tel: +30-26510-98814
Fax: +30-26510-98890

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 delve into the logical design of ETL scenarios and provide a generic and customizable framework in order to support the DW designer in his task. First, we present a metamodel particularly customized for the definition of ETL activities. We follow a workflow-like approach, where the output of a certain activity can either be stored persistently or passed to a subsequent activity. Also, we employ a declarative database programming language, LDL, to define the semantics of each activity. The metamodel is generic enough to capture any possible ETL activity. Nevertheless, in the pursuit of higher reusability and flexibility, we specialize the set of our generic metamodel constructs with a palette of frequently-used ETL activities, which we call *templates*. Moreover, in order to achieve a uniform extensibility mechanism for this library of built-ins, we have to deal with specific language issues. Therefore, we also discuss the mechanics of template instantiation to concrete activities. The design concepts that we introduce have been implemented in a tool, ARKTOS II, which is also presented.

Keywords: Data warehousing, ETL

Table of Contents

A GENERIC AND CUSTOMIZABLE FRAMEWORK FOR THE DESIGN OF ETL SCENARIOS

PANOS VASSILIADIS[1], ALKIS SIMITSIS[2], PANOS GEORGANTAS[2], MANOLIS TERROVITIS[2], SPIROS SKIADOPOULOS[2]

[1] University of Ioannina,
Dept. of Computer Science,
Ioannina, Greece
pvassil@cs.uoi.gr

[2] National Technical University of Athens,
Dept. of Electrical and Computer Eng.,
Athens, Greece
{asimi, pgeor, mter, 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 delve into the logical design of ETL scenarios and provide a generic and customizable framework in order to support the DW designer in his task. First, we present a metamodel particularly customized for the definition of ETL activities. We follow a workflow-like approach, where the output of a certain activity can either be stored persistently or passed to a subsequent activity. Also, we employ a declarative database programming language, LDL, to define the semantics of each activity. The metamodel is generic enough to capture any possible ETL activity. Nevertheless, in the pursuit of higher reusability and flexibility, we specialize the set of our generic metamodel constructs with a palette of frequently-used ETL activities, which we call *templates*. Moreover, in order to achieve a uniform extensibility mechanism for this library of built-ins, we have to deal with specific language issues. Therefore, we also discuss the mechanics of template instantiation to concrete activities. The design concepts that we introduce have been implemented in a tool, ARKTOS II, which is also presented.

Keywords: Data warehousing, ETL

1. INTRODUCTION

Data warehouse operational processes normally compose a labor intensive workflow, involving data extraction, transformation, integration, cleaning and transport. To deal with this workflow, specialized tools are already available in the market [IBM03,Info03,Micr02,Orac03], 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.

If we treat an ETL scenario as a composite workflow, in a traditional way, its designer is obliged to define several of its parameters (Fig. 1.1). Here, we follow a multi-perspective approach that enables to separate these parameters and study them in a principled approach. We are mainly interested in the *design* and *administration* parts of the lifecycle of the overall ETL process, and we depict them at the upper and lower part of Fig. 1.1, respectively. At the top of Fig. 1.1, we are mainly concerned with the static design artifacts for a workflow environment. We will follow a traditional approach and group the design artifacts into *logical* and *physical*, with each category comprising its own *perspective*. We depict the logical perspective on the left hand side of Fig. 1.1, and the physical perspective on the right hand side. At the logical perspective, we classify the design artifacts that give an abstract description of the workflow environment. First, the designer is responsible for defining an *Execution Plan* for the scenario. The definition of an execution plan can be seen from various perspectives. The *Execution Sequence* involves the specification of which activity runs first, second, and so on, which activities run in parallel, or when a semaphore is defined so that several activities are synchronized at a rendezvous point. ETL activities normally run in batch, so the designer needs to specify an *Execution Schedule*, i.e., the time points or events that trigger the execution of the scenario as a whole. Finally, due to system crashes, it is imperative that there exists a *Recovery Plan*, specifying the sequence of steps to be taken in the case of failure for a certain activity (e.g., retry to execute the activity, or undo any intermediate results produced so far). On the right-hand side of Fig. 1.1, we can also see the physical perspective, involving the registration of the actual entities that exist in the real world. We will reuse the terminology of [AHKB00] for the physical perspective. The *Resource Layer* comprises the definition of roles (human or software) that are responsible for executing the activities of the workflow. The *Operational Layer*, at the same time, comprises the software modules that implement the design entities of the logical perspective in the real

world. In other words, the activities defined at the logical layer (in an abstract way) are materialized and executed through the specific software modules of the physical perspective.
At the lower part of Fig. 1.1, we are dealing with the tasks that concern the administration of the workflow environment and their dynamic behavior at runtime. First, an *Administration Plan* should be specified, involving the notification of the administrator either on-line (monitoring) or off-line (logging) for the status of an executed activity, as well as the security and authentication management for the ETL environment.

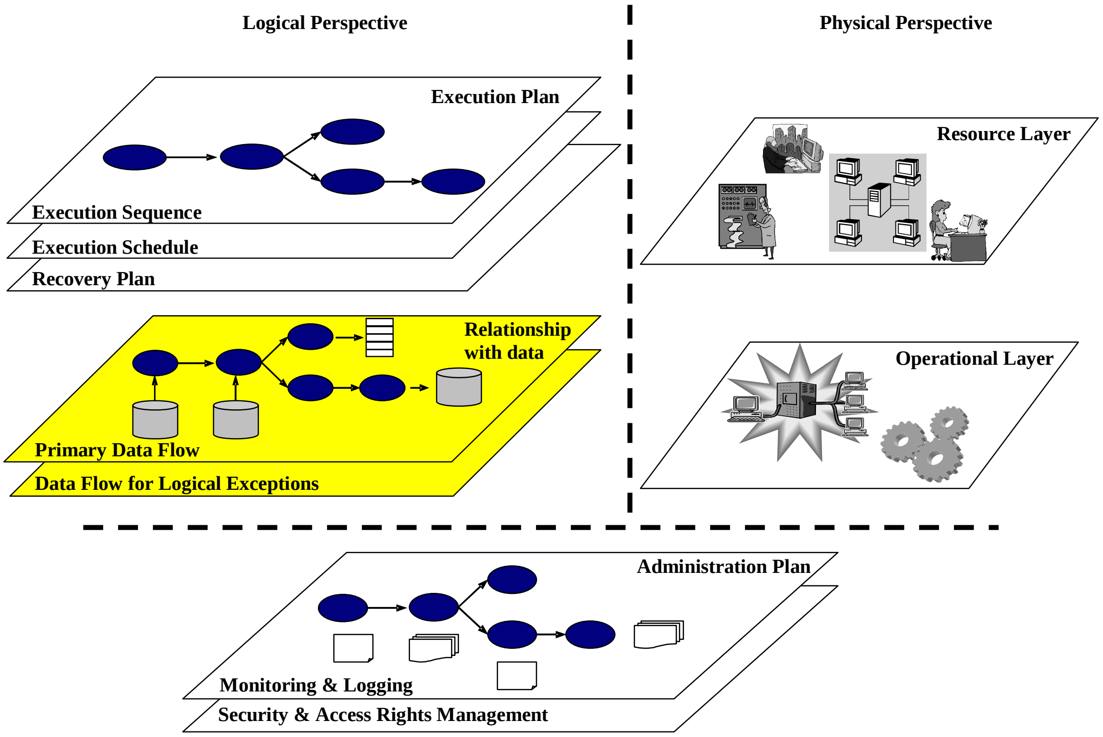


Fig. 1.1 Different perspectives for an ETL workflow

We find that research has not dealt with the definition of data-centric workflows to the entirety of its extent. In the ETL case, for example, due to the data centric nature of the process, the designer must deal with the *relationship of the involved activities with the underlying data*. This involves the definition of a *Primary Data Flow* that describes the route of data from the sources towards their final destination in the data warehouse, as they pass through the activities of the scenario. Also, due to possible quality problems of the processed data, the designer is obliged to define a *Data Flow for Logical Exceptions*, i.e., a flow for the problematic data, i.e., the rows that violate integrity or business rules. It is the combination of the execution sequence and the data flow that generates the semantics of the ETL workflow: the data flow defines what each activity does and the execution plan defines in which order and combination.

In this paper, we work in the internals of the data flow of ETL scenarios. First, we present a metamodel particularly customized for the definition of ETL activities. We follow a workflow-like approach, where the output of a certain activity can either be stored persistently or passed to a subsequent activity. Moreover, we employ a declarative database programming language, LDL, to define the semantics of each activity. The metamodel is generic enough to capture any possible ETL activity; nevertheless, reusability and ease-of-use dictate that we can do better in aiding the data warehouse designer in his task. In this pursuit of higher reusability and flexibility, we specialize the set of our generic metamodel constructs with a palette of frequently-used ETL activities, which we call *templates*. Moreover, in order to achieve a uniform extensibility mechanism for this library of built-ins, we have to deal with specific language issues: thus, we also discuss the mechanics of template instantiation to concrete activities. The design concepts that we introduce have been implemented in a tool, ARKTOS II, which is also presented.

Our contributions can be listed as follows:

- First, *we define a formal metamodel as an abstraction of ETL processes at the logical level*. The data stores, activities and their constituent parts are formally defined. An activity is defined as an entity with possibly more than one input schemata, an output schema 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. A serializable combination of ETL activities, provider relationships and data stores constitutes an ETL scenario.

- Second, *we provide a reusability framework* that complements the genericity of the metamodel. Practically, this is achieved from a set of "built-in" specializations of the entities of the Metamodel layer, specifically tailored for the most frequent elements of ETL scenarios. This palette of template activities will be referred to as *Template layer* and it is characterized by its extensibility; in fact, due to language considerations, we provide the details of the mechanism that instantiates templates to specific activities.
- Finally, we discuss *implementation* issues and we present a graphical tool, ARKTOS II that facilitates the design of ETL scenarios, based on our model.

This paper is organized as follows. In Section 2, we present a generic model of ETL activities. Section 3 describes the mechanism for specifying and materializing template definitions of frequently used ETL activities. Section 4 presents ARKTOS II, a prototype graphical tool. In Section 5, we present related work. In Section 6, we make a general discussion on the completeness and general applicability of our approach. Section 7 offers conclusions and presents topics for future research. Finally in the Appendix, we present a formal LDL description of the most frequently used ETL activities. Short versions of parts of this paper have been presented in [VaSS02, VSGT03].

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 and logging while it concentrates on the flow of data from the sources towards the data warehouse through the composition of activities and data stores. The full layout of an ETL scenario, involving activities, recordsets and functions can be modeled by a graph, which we call the *Architecture Graph*. We employ a uniform, graph-modeling framework for both the modeling of the internal structure of activities and for the modeling of the ETL scenario at large, which 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.

2.1 Graphical Notation and Motivating Example

Being a graph, the Architecture Graph of an ETL scenario comprises nodes and edges. The involved data types, function types, constants, attributes, activities, recordsets, parameters and functions constitute the nodes of the graph. The different kinds of relationships among these entities are modeled as the edges of the graph. In Fig. 2.1, we give the graphical notation for all the modeling constructs that will be presented in the sequel.

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 arrows (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. 2.1 Graphical notation for the Architecture Graph.

Motivating Example. To motivate our discussion we will present an example involving the propagation of data from a certain source S1, towards a data warehouse DW through intermediate recordsets. These recordsets belong to a Data Staging Area (DSA)[1] DS. The scenario involves the propagation of data from the table PARTSUPP of source S$_1$ to the data warehouse DW. Table DW.PARTSUPP(PKEY,SOURCE,DATE, QTY,COST) stores information for the available quantity (QTY) and cost (COST) of parts (PKEY) per source (SOURCE). The data source S$_1$.PARTSUPP(PKEY,DATE,QTY,COST) records the supplies from a

[1] In data warehousing terminology a DSA is an intermediate area of the data warehouse, specifically destined to enable the transformation, cleaning and integration of source data, before being loaded to the warehouse.

specific geographical region, e.g., Europe. All the attributes, except for the dates are instances of the Integer type. The scenario is graphically depicted in Fig. 2.2 and involves the following transformations.

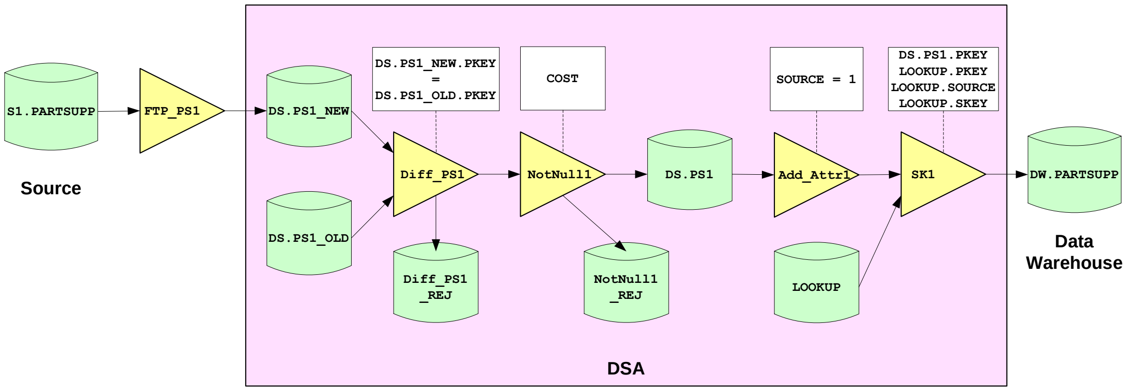


Fig. 2.2 Bird's-eye view of the motivating example

1. First, we transfer via FTP_PS1 the snapshot from the source S$_1$.PARTSUPP to the file DS.PS1_NEW of the DSA[2].
2. In the DSA we maintain locally a copy of the snapshot of the source as it was at the previous loading (we assume here the case of the incremental maintenance of the DW, instead of the case of the initial loading of the DW). The recordset DS.PS1_NEW(PKEY,DATE,QTY,COST) stands for the last transferred snapshot of S$_1$.PARTSUPP. By detecting the difference of this snapshot with the respective version of the previous loading, DS.PS1_OLD(PKEY,DATE,QTY,COST), we can derive the newly inserted rows in S$_1$.PARTSUPP. Note that the difference activity that we employ, namely Diff_PS1, checks for differences only on the primary key of the recordsets; thus, we ignore here any possible deletions or updates for the attributes COST, QTY of existing rows. Any not newly inserted row is rejected and so, it is propagated to Diff_PS1_REJ that stores all the rejected rows and its schema is identical to the input schema of the activity Diff_PS1.
3. The rows that pass the activity Diff_PS1 are checked for null values of the attribute COST through the activity NotNull1. We store the rows whose COST is not NULL in the recordset DS.PS$_1$(PKEY, DATE,QTY,COST). Rows having a NULL value for their COST are kept in the Diff_PS1_REJ recordset for further examination by the data warehouse administrator.
4. Although we consider the data flow for only one source, S1, the data warehouse can clearly have more than one sources for part supplies. In order to keep track of the source of each row that enters in the DW, we need to add a 'flag' attribute, namely SOURCE, indicating 1 for the respective source. This is achieved through the activity Add_Attr$_1$.
5. Next, we assign a surrogate key on 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 kind of 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 SK1. It uses the lookup table LOOKUP(PKEY,SOURCE,SKEY). Finally, we populate the data warehouse with the output of the previous activity.

[2] The technical points of the likes of FTP are mostly employed to show what kind of problems someone has to deal with in a practical situation, rather than to relate this kind of physical operations to a logical model. In terms of logical modelling this is a simple passing of data from one site to another.

The role of rejected rows depends on the peculiarities of each ETL scenario. If the designer needs to administrate these rows further, then he/she should use intermediate storage recordsets with the burden of an extra I/O cost. If the rejected rows should not have a special treatment, then the best solution is to be ignored; thus, in this case we avoid overload the scenario with any extra storage recordset. In our case, we annotate only two of the presented activities with a destination for rejected rows. Out of these, while `NotNull1_REJ` absolutely makes sense as a placeholder for problematic rows having non-acceptable `NULL` values, `Diff_PS1_REJ` is presented for demonstration reasons only.

Finally, before proceeding, we would like to stress that 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. To assist a more automatic construction of ETL scenarios, we have implemented the ARKTOS II tool that supports the designing process through a friendly GUI. We present ARKTOS II in Section 4.

2.2 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 `T` is characterized by a name and a domain, i.e., a countable set of values, called `dom(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*. 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. 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. 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 $[D_1,...,D_n,M_1,...,M_m]$ 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 $S=[A_1,\dots,A_k]\rightarrow dom(A_1)\times\ldots\times dom(A_k)$. Thus, the extension of the recordset is a finite subset of $dom(A_1)\times\ldots\times dom(A_k)$ and a record is the instance of a mapping $dom(A_1)\times\ldots\times dom(A_k)\rightarrow[x_1,\dots,x_k]$, $x_i\in dom(A_i)$.
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 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).

2.3 Activities

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 language (e.g., PL/SQL, Perl, C). Still, we want to deal with the general case of ETL activities. We employ an abstraction of the source code of an activity, in the form of an LDL statement. Using LDL we avoid dealing with the peculiarities of a particular programming language. Once again, we want to stress that the presented LDL description is intended to capture the semantics of each activity, instead of the way these activities are actually implemented.

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*: a 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 LDL statement describing the content passed to the output of the operation, with respect to its input. This LDL 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 LDL statement describing the rejected records, in a sense similar to the Output Operational Semantics. This statement is by default considered to be the complement of the Output Operational Semantics, except if explicitly defined differently.

There are two issues that we would like to elaborate on, here:

- ***NULL Schemata***. 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.
- ***Language Issues.*** Initially, we used to specify the semantics of activities with SQL statements. Still, although clear and easy to write and understand, SQL is rather hard to use if one is to perform rewriting and composition of statements. Thus, we have supplemented SQL with LDL [NaTs98], a logic-programming, declarative language as the basis of our scenario definition. LDL is a Datalog variant based on a Horn-clause logic that supports recursion, complex objects and negation. In the context of its implementation in an actual deductive database management system, LDL++ [Zani98], the language has been extended to support external functions, choice, aggregation (and even, user-defined aggregation), updates and several other features.

2.4 Relationships in the Architecture Graph

In this subsection, we will elaborate on the different kinds of relationships that the entities of an ETL scenario have. Whereas these entities are modeled as the nodes of the architecture graph, the relationships are modeled as its edges. Due to their diversity, before proceeding, we list these types of relationships along with the related terminology that we will employ for the rest of the paper. The graphical notation of entities (nodes) and relationships (edges) is presented in Fig. 2.1.

- *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.
- *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.

In the rest of this subsection, we will base our discussions on a part of the scenario of the motivating example (presented in Section 2.1), including the activities Add_Attr$_1$ and SK$_1$.

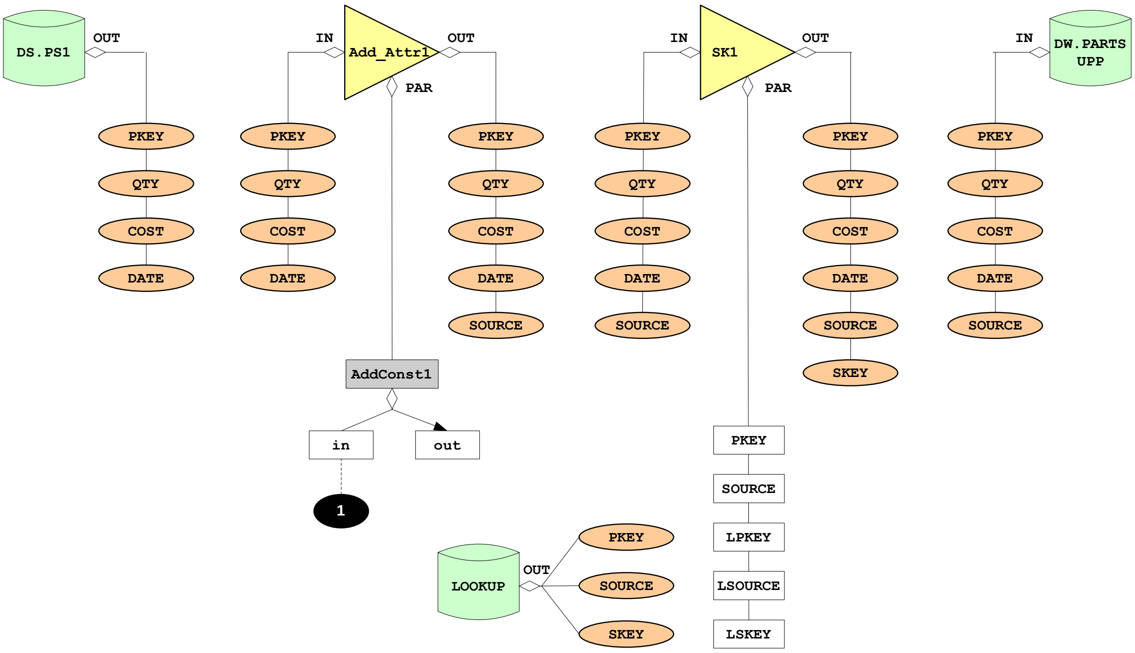


Fig. 2.3 Part-of relationships of the architecture graph

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 respectively). Naturally, if the activity involves more than one input schemata, the relationship is tagged with an IN$_i$ tag for the ith input schema. We also 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. 2.3 depicts a part of the motivating example, where we can see the decomposition of (a) the recordsets DS.PS$_1$, LOOKUP, DW.PARTSUPP; (b) the activities Add_Attr$_1$ 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 AddConst$_1$ 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 SOURCE 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_Attr$_1$ 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) LPKEY, which stands for the attribute of the lookup table which contains the production keys; (d) LSOURCE, which stands for the attribute of the lookup table which contains the source value (corresponding to the aforementioned SOURCE parameter); (e) LSKEY, which stands for the attribute of the lookup table which contains the surrogate keys.

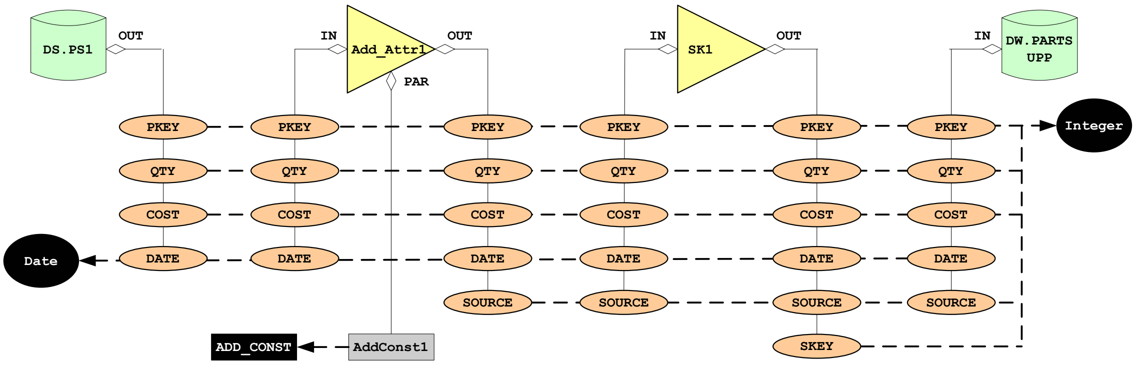


Fig. 2.4 Instance-of relationships of the architecture graph

Data types and instance-of relationships. To capture typing information on attributes and functions, the architecture graph comprises 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. 2.4, we observe the attributes of the two activities of our example and their correspondence to two data types, namely Integer and Date. For reasons of presentation, we merge several instantiation edges so that the figure does not become too crowded. At the bottom of Fig. 2.4, we can also see the fact that function $\texttt{AddConst}_1$ 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.

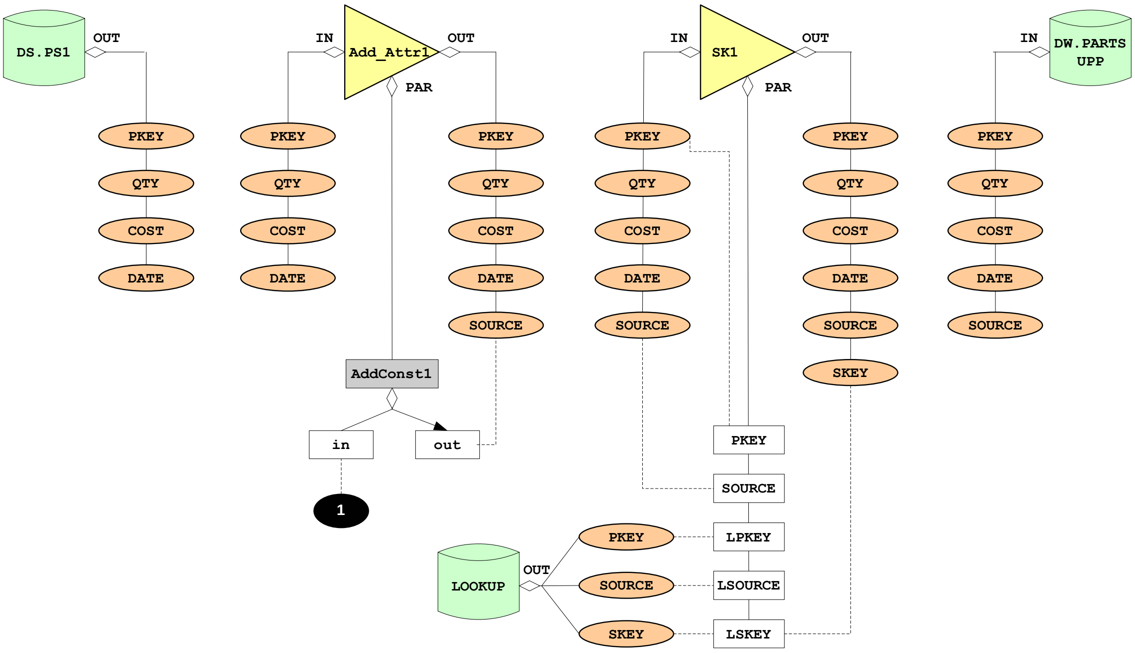


Fig. 2.5 Regulator relationships of the architecture graph

In the example of Fig. 2.5 we can observe how the parameters of the two activities are populated. First, we can see that activity $\texttt{Add\_Attr}_1$ receives an integer (1) as its input and uses the function $\texttt{AddConst}_1$

to populate its attribute SOURCE. 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, LPKEY, LSOURCE and LSKEY, are mapped to the respective attributes of either the activity's input schema or the employed lookup table LOOKUP.

The parameter LSKEY 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 LSKEY, 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 respective parameter.

Provider relationships. 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 schemata.

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, the opposite does not hold. 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 exists 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 or 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 LDL 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. Provider relationships are depicted with bold solid arrows that stem from the provider and end in the consumer attribute.

Observe Fig. 2.6. 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_Attr_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_Attr_1 towards the activity SK_1 in a similar manner. Note that, for the moment, we have not covered how the output of function $AddConst_1$ populates the output attribute SOURCE for the activity Add_Attr_1, or how the parameters of activity SK_1 populate the output attribute SKEY. Such information will be expressed using 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. 2.6 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.

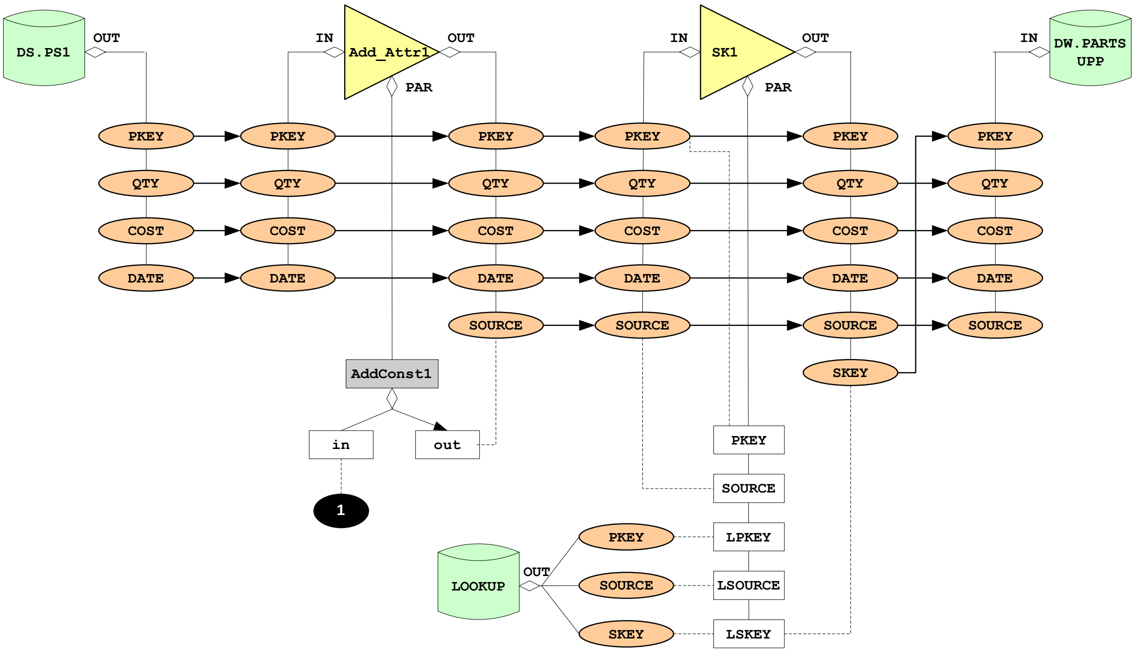


Fig. 2.6 Provider relationships of the architecture graph

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. It is not necessary that the parameters x and y 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).

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. 2.7, where we depict a small part of our running example. The legend in the left side of Fig. 2.7 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 SK_1.OUT.SKEY is not computed only from attribute LOOKUP.SKEY, but from the combination of all the attributes that populate the parameters.

As far as the parameters of activity Add_Attr_1 are concerned, we can also detect a derived provider relationship, between the constant 1 and the output attribute SOURCE. 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.SKEY and SK_1.OUT.SKEY).

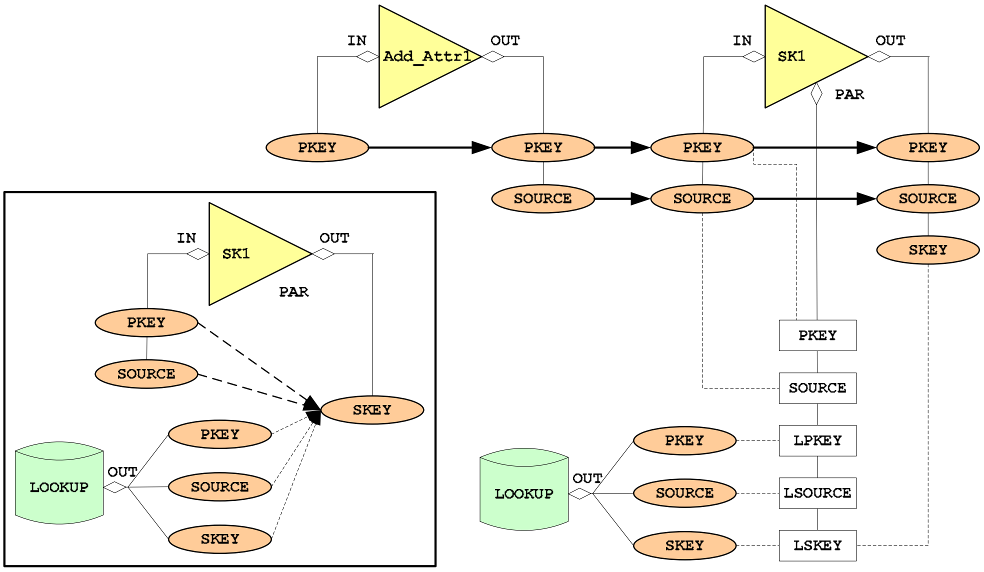


Fig. 2.7 Derived provider relationships of the architecture graph

2.5 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).

In general, there is a simple rule for constructing valid ETL scenarios in our setting. For each activity, the designer must provide three kinds of provider relationships: (a) a mapping of the activity's data provider(s) to the activity's input schema(ta); (b) a mapping of the activity's input schema(ta) to the activity's output, along with a specification of the semantics of the activity (i.e., the check / cleaning / transformation / value production that the activity performs), and (c) a mapping from the activity's output schema towards the data consumer of the activity.

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.

In terms of formal modeling of the architecture graph, 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. 2.8. As far as a specific scenario is concerned, we assume their respective finite subsets, depicted in column *Scenario-Specific* in Fig. 2.8. 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	$\mathbf{D}^I$	$\mathbf{D}$
	Function Types	$\mathbf{F}^I$	$\mathbf{F}$
	Constants	$\mathbf{C}^I$	$\mathbf{C}$
User-provided	Attributes	$\mathbf{\Omega}^I$	$\mathbf{\Omega}$
	Functions	$\mathbf{\Phi}^I$	$\mathbf{\Phi}$
	Schemata	$\mathbf{S}^I$	$\mathbf{S}$
	RecordSets	$\mathbf{RS}^I$	$\mathbf{RS}$
	Activities	$\mathbf{A}^I$	$\mathbf{A}$
	Provider Relationships	$\mathbf{Pr}^I$	$\mathbf{Pr}$
	Part-Of Relationships	$\mathbf{Po}^I$	$\mathbf{Po}$
	Instance-Of Relationships	$\mathbf{Io}^I$	$\mathbf{Io}$
	Regulator Relationships	$\mathbf{Rr}^I$	$\mathbf{Rr}$
	Derived Provider Relationships	$\mathbf{Dr}^I$	$\mathbf{Dr}$

Fig. 2.8 Formal definition of domains and notation

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

- $\mathbf{V} = \mathbf{D}\cup\mathbf{F}\cup\mathbf{C}\cup\mathbf{\Omega}\cup\mathbf{\Phi}\cup\mathbf{S}\cup\mathbf{RS}\cup\mathbf{A}$
- $\mathbf{E} = \mathbf{Pr}\cup\mathbf{Po}\cup\mathbf{Io}\cup\mathbf{Rr}\cup\mathbf{Dr}$

2.6 Motivating Example Revisited

In this subsection, we return to our motivating example, in order (a) to summarize how it is modeled in terms of our architecture graph, from different viewpoints and (b) to show how its declarative description in LDL looks like.

Fig. 2.9 depicts a screenshot of ARKTOS II that represents a zoom-in of the last two activities of the scenario. We can observe (from right to left): (i) the fact that the recordset DW.PARTSUPP comprises the attributes PKEY,SOURCE,DATE,QTY,COST (ii) the provider relationships (bold and solid arrows) between the output schema of the activity SK1 and the attributes of DW.PARTSUPP; (iii) the provider relationships between the input and the output schema of activity SK1; (iv) the provider relationships between the output schema of the activity Add_Attr$_1$ and the input schema of the activity SK1; (v) the population of the parameters of the surrogate key activity from regulator relationships (dotted bold arrows) by the attributes of table LOOKUP and some of the attribute of the input schema of SK1; (vi) the instance-of relationships (light dotted edges) between the attributes of the scenario and their data types (colored ovals at the bottom of the figure).

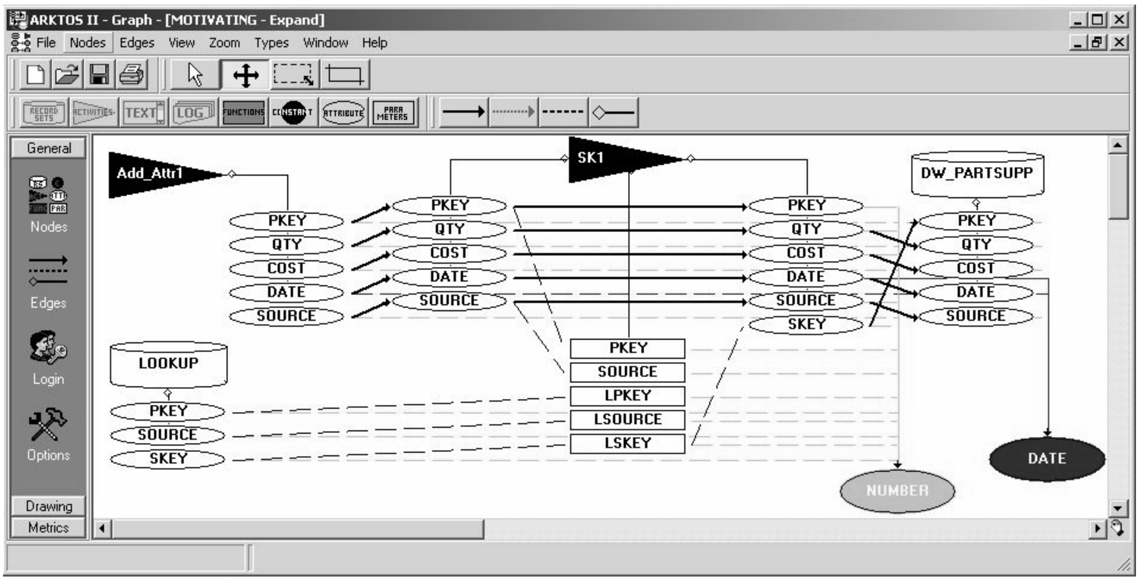


Fig. 2.9 Architecture graph of a part of the motivating example

In Fig. 2.10, we show the LDL program for our motivating example. In the next section, we will also elaborate in adequate detail on the mechanics of the usage of LDL for ETL scenarios.

```
diffPS1_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST) ←
 ds_ps1_new(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST).

diffPS1_in2(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST ←
 ds_ps1_old(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST).

semi_join(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST) ←
 diffPS1_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
 diffPS1_in2(A_IN2_PKEY,_,_,_),
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_PKEY=A_IN2_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST.

diffPS1_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST) ←
 diffPS1_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
 ~semi_join(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST.

diffPS1_rej(A_REJ_PKEY,A_REJ_DATE,A_REJ_QTY,A_REJ_COST) ←
 diffPS1_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
 semi_join(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
  A_REJ_PKEY=A_IN1_PKEY,
  A_REJ_DATE=A_IN1_DATE,
  A_REJ_QTY=A_IN1_QTY,
  A_REJ_COST=A_IN1_COST.

diff_PS1_REJ ← (A_REJ_PKEY,A_REJ_DATE,A_REJ_QTY,A_REJ_COST)
  diffPS1_rej (A_REJ_PKEY,A_REJ_DATE,A_REJ_QTY,A_REJ_COST)

notNull_in1(A_IN1_ PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST) ←
 diff_PS1_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST),
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST.

notNull_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST) ←
 notNull_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
  A_IN1_COST~=’null’,
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST.

notNull_rej(A_REJ_PKEY,A_REJ_DATE,A_REJ_QTY,A_REJ_COST) ←
 notNull_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
  A_IN1_COST=’null’,
  A_REJ_PKEY=A_IN1_PKEY,
  A_REJ_DATE=A_IN1_DATE,
  A_REJ_QTY=A_IN1_QTY,
  A_REJ_COST=A_IN1_COST.

not_Null_REJ(A_REJ_PKEY,A_REJ_DATE,A_REJ_QTY,A_REJ_COST) ←
  notNull_rej(A_REJ_PKEY,A_REJ_DATE,A_REJ_QTY,A_REJ_COST).

ds_ps1(PKEY,DATE,QTY,COST) ←
 a_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST),
  PKEY=A_OUT_PKEY,
  DATE=A_OUT_DATE,
  QTY=A_OUT_QTY,
  COST=A_OUT_COST.

addAttr_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST) ←
 ds_ps1(PKEY,DATE,QTY,COST),
  PKEY=A_IN1_PKEY,
  DATE=A_IN1_DATE,
  QTY=A_IN1_QTY,
  COST=A_IN1_COST.

addAttr_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST,A_OUT_SOURCE) ←
 addAttr_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST),
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST,
  A_OUT_SOURCE='SOURCE1'.

addSkey_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST,A_IN1_SOURCE) ←
 addAttr_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST,A_OUT_SOURCE),
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST,
  A_OUT_SOURCE=A_IN1_SOURCE.

addSkey_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST,A_OUT_SOURCE,A_OUT_SKEY) ←
 addSkey_in1(A_IN1_PKEY,A_IN1_DATE,A_IN1_QTY,A_IN1_COST,A_IN1_SOURCE),
  lookup(A_IN1_SOURCE,A_IN1_PKEY,A_OUT_SKEY),
  A_OUT_PKEY=A_IN1_PKEY,
  A_OUT_DATE=A_IN1_DATE,
  A_OUT_QTY=A_IN1_QTY,
  A_OUT_COST=A_IN1_COST,
  A_OUT_SOURCE=A_IN1_SOURCE.

dw_partsupp(PKEY,DATE,QTY,COST,SOURCE) ←
 addSkey_out(A_OUT_PKEY,A_OUT_DATE,A_OUT_QTY,A_OUT_COST,A_OUT_SOURCE,A_OUT_SKEY),
  DATE=A_IN1_DATE,
  QTY=A_IN1_QTY,
  COST=A_IN1_COST
  SOURCE=A_IN1_SOURCE,
  PKEY=A_IN1_SKEY.
```

NOTE: For reasons of readability we de not replace the ’A’ in attribute names with the activity name, i.e., A_OUT_PKEY should be diffPS1_OUT_PKEY.

Fig. 2.10 LDL specification of the motivating example

3. TEMPLATES FOR ETL ACTIVITIES

In this section, we present the mechanism for exploiting template definitions of frequently used ETL activities. The general framework for the exploitation of these templates is accompanied with the presentation of the language-related issues for template management and appropriate examples.

3.1 General Framework

Our philosophy during the construction of our metamodel was based on two pillars: (a) *genericity*, i.e., the derivation of a simple model, powerful to capture ideally all the cases of ETL activities and (b) *extensibility*, i.e., the possibility of extending the built-in functionality of the system with new, user-specific templates.

The genericity doctrine was pursued through the definition of a rather simple activity metamodel, as described in Section 2. Still, providing a single metaclass for all the possible activities of an ETL environment is not really enough for the designer of the overall process. A richer "language" should be available, in order to describe the structure of the process and facilitate its construction. To this end, we provide a palette of *template* activities, which are specializations of the generic metamodel class.

Observe Fig. 3.1 for a further explanation of our framework. The lower layer of Fig. 3.1, namely *Schema Layer*, involves a specific ETL scenario. All the entities of the Schema layer are *instances* of the classes `Data Type`, `Function Type`, `Elementary Activity`, `RecordSet` and `Relationship`. Thus, as one can see on the upper part of Fig. 3.1, we introduce a meta-class layer, namely *Metamodel Layer* involving the aforementioned classes. The linkage between the Metamodel and the Schema layers is achieved through instantiation (`InstanceOf`) relationships. The Metamodel layer implements the aforementioned genericity desideratum: the classes which are involved in the Metamodel layer are generic enough to model any ETL scenario, through the appropriate instantiation.

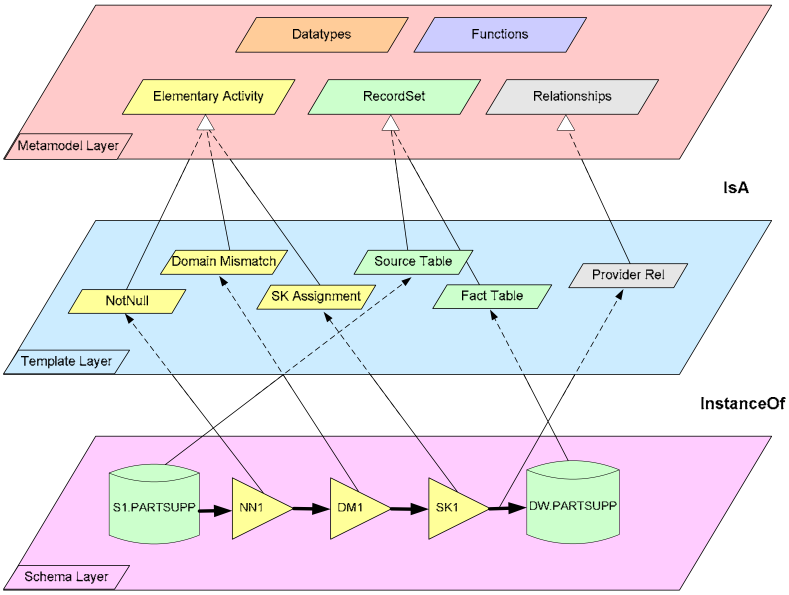


Fig. 3.1 The metamodel for the logical entities of the ETL environment

Still, we can do better than the simple provision of a meta- and an instance layer. In order to make our metamodel truly useful for practical cases of ETL activities, we enrich it with a set of ETL-specific constructs, which constitute a subset of the larger Metamodel layer, namely the *Template Layer*. The constructs in the Template layer are also meta-classes, but they are quite customized for the regular cases of ETL activities. Thus, the classes of the Template layer are specializations (i.e., subclasses) of the generic classes of the Metamodel layer (depicted as `IsA` relationships in Fig. 3.1). Through this customization mechanism, the designer can pick the instances of the Schema layer from a much richer palette of constructs; in this setting, the entities of the Schema layer are instantiations, not only of the respective classes of the Metamodel layer, but also of their subclasses in the Template layer.

In the example of Fig. 3.1 the concept DW.PARTSUPP must be populated from a certain source S_1.PARTSUPP. Several operations must intervene during the propagation: for example, checks for null values and domain violations, as well as a surrogate key assignment take place in the scenario. As one can observe, the recordsets that take part in this scenario are instances of class RecordSet (belonging to the Metamodel layer) and specifically of its subclasses Source Table and Fact Table. Instances and encompassing classes are related through links of type InstanceOf. The same mechanism applies to all the activities of the scenario, which are (a) instances of class Elementary Activity and (b) instances of one of its subclasses, depicted in Fig. 3.1. Relationships do not escape the rule either: observe how the provider links from the concept S1.PS towards the concept DW.PARTSUPP are related to class Provider Relationship through the appropriate InstanceOf links.

As far as the class Recordset is concerned, in the Template layer we can specialize it to several subclasses, based on orthogonal characteristics, such as whether it is a file or RDBMS table, or whether it is a source or target data store (as in Fig. 3.1). In the case of the class Relationship, there is a clear specialization in terms of the five classes of relationships which have already been mentioned in Section 2: Provider, Part-Of, Instance-Of, Regulator and Derived Provider.

Following the same framework, class Elementary Activity is further specialized to an extensible set of reoccurring patterns of ETL activities, depicted in Fig. 3.2. As one can see on the top side of Fig. 3.1, we group the template activities in five major logical groups. We do not depict the grouping of activities in subclasses in Fig. 3.1, in order to avoid overloading the figure; instead, we depict the specialization of class Elementary Activity to three of its subclasses whose instances appear in the employed scenario of the Schema layer. We now proceed to present each of the aforementioned groups in more detail.

Filters
- Selection (σ)
- Not null (NN)
- Primary key violation (PK)
- Foreign key violation (FK)
- Unique value (UN)
- Domain mismatch (DM)

Unary operations
- Push
- Aggregation (γ)
- Projection (π)
- Function application (f)
- Surrogate key assignment (SK)
- Tuple normalization (N)
- Tuple denormalization (DN)

File operations
- EBCDIC to ASCII conversion (EB2AS)
- Sort file (Sort)

Binary operations
- Union (U)
- Join (⋈)
- Diff (Δ)
- Update Detection (ΔUPD)

Transfer operations
- Ftp (FTP)
- Compress/Decompress (Z/dZ)
- Encrypt/Decrypt (Cr/dCr)

Fig. 3.1 Template activities, along with their graphical notation symbols, grouped by category

The first group, named *Filters*, provides checks for the satisfaction (or not) of a certain condition. The semantics of these filters are the obvious (starting from a generic selection condition and proceeding to the check for null values, primary or foreign key violation, etc.). The second group of template activities is called *Unary Operations* and except for the most generic push activity (which simply propagates data from the provider to the consumer), consists of the classical aggregation and function application operations along with three data warehouse specific transformations (surrogate key assignment, normalization and denormalization). The third group consists of classical *Binary Operations*, such as union, join and difference of recordsets/activities as well as with a special case of difference involving the detection of updates. Except for the aforementioned template activities, which mainly refer to logical transformations, we can also consider the case of physical operators that refer to the application of physical transformations to whole files/tables. In the ETL context, we are mainly interested in operations like *Transfer Operations* (ftp, compress/ decompress, encrypt/decrypt) and *File Operations* (EBCDIC to ASCII, sort file).

Summarizing, the Metamodel layer is a set of generic entities, able to represent any ETL scenario. At the same time, the genericity of the Metamodel layer is complemented with the extensibility of the Template layer, which is a set of "built-in" specializations of the entities of the Metamodel layer, specifically tailored for the most frequent elements of ETL scenarios. Moreover, apart from this "built-in", ETL-specific extension of the generic metamodel, if the designer decides that several 'patterns', not included in the palette of the Template layer, occur repeatedly in his data warehousing projects, he can easily fit them into the customizable Template layer through a specialization mechanism.

3.2 Formal Definition and Usage of Template Activities

Once the template layer has been introduced, the obvious issue that is raised is its linkage with the employed declarative language of our framework. In general, the broader issue is the usage of the template mechanism from the user; to this end, we will explain the substitution mechanism for templates in this subsection and refer the interested reader to Appendix A for a presentation of the specific templates that we have constructed.

A *Template Activity* is formally defined as follows:

- *Name*: a unique identifier for the template activity.
- *Parameter List*: a set of names which act as regulators in the expression of the semantics of the template activity. For example, the parameters are used to assign values to constants, create dynamic mapping at instantiation time, etc.
- *Expression*: a declarative statement describing the operation performed by the instances of the template activity. As with elementary activities, our model supports LDL as the formalism for the expression of this statement.
- *Mapping*: a set of bindings, mapping input to output attributes, possibly through intermediate placeholders. In general, mappings at the template level try to capture a default way of propagating incoming values from the input towards the output schema. These default bindings are easily refined and possibly rearranged at instantiation time.

The template mechanism we use is a substitution mechanism, based on macros, that facilitates the automatic creation of LDL code. This simple notation and instantiation mechanism permits the easy and fast registration of LDL templates. In the rest of this section, we will elaborate on the notation, instantiation mechanisms and template taxonomy particularities.

3.2.1 Notation

Our template notation is a simple language featuring five main mechanisms for dynamic production of LDL expressions: (a) *variables* that are replaced by their values at instantiation time; (b) a function that returns the *arity* of an input, output or parameter schema; (c) *loops*, where the loop body is repeated at instantiation time as many times as the iterator constraint defines; (d) *keywords* to simplify the creation of unique predicate and attribute names; and, finally, (e) *macros* which are used as syntactic sugar to simplify the way we handle complex expressions (especially in the case of variable size schemata).

Variables. We have two kinds of variables in the template mechanism: *parameter variables* and *loop iterators*. Parameter variables are marked with a `@` symbol at their beginning and they are replaced by user-defined values at instantiation time. A list of an arbitrary length of parameters is denoted by `@<parameter name>[]`. For such lists the user has to explicitly or implicitly provide their length at instantiation time. Loop iterators, on the other hand, are implicitly defined in the loop constraint. During each loop iteration, all the properly marked appearances of the iterator in the loop body are replaced by its current value (similarly to the way the C preprocessor treats `#DEFINE` statements). Iterators that appear marked in loop body are instantiated even when they are a part of another string or of a variable name. We mark such appearances by enclosing them with `$`. This functionality enables referencing all the values of a parameter list and facilitates the creation an arbitrary number of pre-formatted strings.

Functions. We employ a built-in function, `arityOf(<input/output/parameter schema>)`, which returns the arity of the respective schema, mainly in order to define upper bounds in loop iterators.

Loops. Loops are a powerful mechanism that enhances the genericity of the templates by allowing the designer to handle templates with unknown number of variables and with unknown arity for the input/output schemata. The general form of loops is

```
[<simple constraint>] { <loop body> }
```

where `simple constraint` has the form:

```
<lower bound> <comparison operator> <iterator> <comparison operator> <upper bound>
```

We consider only linear increase with step equal to 1, since this covers most possible cases. Upper bound and lower bound can be arithmetic expressions involving arityOf() function calls, variables and constants. Valid arithmetic operators are +, -, /, * and valid comparison operators are <, >, =, all with their usual semantics. If lower bound is omitted, 1 is assumed. During each iteration the loop body will be reproduced and the same time all the marked appearances of the loop iterator will be replaced by its current value, as described before. Loop nesting is permitted.

Keywords. Keywords are used in order to refer to input and output schemata. They provide two main functionalities: (a) they simplify the reference to the input output/schema by using standard names for the predicates and their attributes, and (b) they allow their renaming at instantiation time. This is done in such a way that no different predicates with the same name will appear in the same program, and no different attributes with the same name will appear in the same rule. Keywords are recognized even if they are parts of another string, without a special notation. This facilitates a homogenous renaming of multiple distinct input schemata at template level, to multiple distinct schemata at instantiation, with all of them having unique names in the LDL program scope. For example, if the template is expressed in terms of two different input schemata a_in1 and a_in2, at instantiation time they will be renamed to dm1_in1 and dm1_in2 so that the produced names will be unique throughout the scenario program. In Fig. 3.3, we depict the way the renaming is performed at instantiation time.

Keyword	Usage	Example
a_out a_in	A unique name for the output/input schema of the activity. The predicate that is produced when this template is instantiated has the form: <unique_pred_name>_out (or, _in respectively)	difference3_out difference3_in
A_OUT A_IN	A_OUT/A_IN is used for constructing the names of the a_out/a_in attributes. The names produced have the form: <predicate unique name in upper case>_OUT (or, _IN respectively)	DIFFERENCE3_OUT DIFFERENCE3_IN

Fig. 3.3 Keywords for Templates

Macros. To make the definition of templates easier and to improve their readability, we introduce a macro to facilitate attribute and variable name expansion. For example, one of the major problems in defining a language for templates is the difficulty of dealing with schemata of arbitrary arity. Clearly, at the template level, it is not possible to pin-down the number of attributes of the involved schemata to a specific value. For example, in order to create a series of name like the following

```
name_theme_1,name_theme_2,...,name_theme_k
```

we need to give the following expression:

```
[iterator<maxLimit]{name_theme$iterator$,}
[iterator=maxLimit]{name_theme$iterator$}
```

Obviously, this results in making the writing of templates hard and reduces their readability. To attack this problem, we resort to a simple reusable macro mechanism that enables the simplification of employed expressions. For example, observe the definition of a template for a simple relational selection:

```
a_out([i<arityOf(a_out)]{A_OUT_$i$,} [i=arityOf(a_out)]{A_OUT_$i$}) <-
      a_in1([i<arityOf(a_in1)]{A_IN1_$i$,} [i=arityOf(a_in1)] {A_IN1_$i$}),
      expr([i<arityOf(@PARAM)]{@PARAM[$i$],}[i=arityOf(@PARAM)]{@PARAM[$i$]}),
      [i<arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$,}
      [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}
```

As already mentioned at the syntax for loops, the expression

```
[i<arityOf(a_out)]{A_OUT_$i$,} [i=arityOf(a_out)]{A_OUT_$i$}
```

defining the attributes of the output schema `a_out` simply wants to list a variable number of attributes that will be fixed at instantiation time. Exactly the same tactics apply for the attributes of the predicate names `a_in1` and `expr`. Also, the final two lines state that each attribute of the output will be equal to the respective attribute of the input (so that the query is safe), e.g., `A_OUT_4 = A_IN1_4`. We can simplify the definition of the template by allowing the designer to define certain macros that simplify the management of temporary length attribute lists. We employ the following macros

```
DEFINE INPUT_SCHEMA AS [i<arityOf(a_in1)]{A_IN1_$i$,}
        [i=arityOf(a_in1)] {A_IN1_$i$}
DEFINE OUTPUT_SCHEMA AS [i<arityOf(a_in)]{A_OUT_$i$,}
        [i=arityOf(a_out)]{A_OUT_$i$}
DEFINE PARAM_SCHEMA AS [i<arityOf(@PARAM)]{@PARAM[$i$],}
        [i=arityOf(@PARAM)]{@PARAM[$i$]}
DEFINE DEFAULT_MAPPING AS [i<arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$,}
        [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}
```

Then, the template definition is as follows:

```
a_out(OUTPUT_SCHEMA) <- a_in1(INPUT_SCHEMA), expr(PARAM_SCHEMA), DEFAULT_MAPPING
```

3.2.2 Instantiation

Template instantiation is the process where the user decides to pick a certain template and create a concrete activity out of it. This procedure requires that the user specifies the schemata of the activity and gives concrete values to the template parameters. Then, the process of producing the respective LDL description of the activity is easily automated. Instantiation order is important in our template creation mechanism, since, as it can easily been seen from the notation definitions, different orders can lead to different results. The instantiation order is as follows:

1. Replacement of macro definitions with their expansions
2. `arityOf()` functions and parameter variables appearing in loop boundaries are calculated first.
3. Loop productions are done by instantiating the appearances of the iterators. This leads to intermediate results without any loops.
4. All the rest parameter variables are instantiated.
5. Keywords are recognized and renamed.

We will try to explain briefly the intuition behind this execution order. It is straightforward why macros are expanded first. The reasons why step (2) proceeds step (3) are as follows: loop boundaries have to be calculated before loop productions are done. Loops on the other hand, have to be expanded before parameter variables are instantiated, if we want to be able to reference lists of variables. The only exception to this is the parameter variables that appear in the loop boundaries, which have to be calculated first. Notice though, that variable list elements cannot appear in the loop constraint. Finally, we have to instantiate variables before keywords since variables are used to create a dynamic mapping between the input/output schemata and other attributes.

Fig. 3.4 shows a simple example of template instantiation for the *function application* activity. To understand the overall process better, first observe the outcome of it, i.e., the specific activity which is produced, as depicted in the final row of Fig. 3.4, labeled *Keyword renaming*. The output schema of the activity, `fa12_out`, is the head of the LDL rule that specifies the activity. The body of the rule says that the output records are specified by the conjunction of the following clauses: (a) the input schema `myFunc_in`, (b) the application of function `subtract` over the attributes `COST_IN`, `PRICE_IN` and the production of a value `PROFIT`, and (c) the mapping of the input to the respective output attributes as specified in the last three conjuncts of the rule.

Template

```
DEFINE INPUT_SCHEMA AS [i<arityOf(a_in1)]{A_IN1_$i$,}
      [i=arityOf(a_in1)] {A_IN1_$i$}
DEFINE OUTPUT_SCHEMA AS [i<arityOf(a_in)]{A_OUT_$i$,}
      [i=arityOf(a_out)]{A_OUT_$i$}
DEFINE FUNCTION_INPUT AS [i<arityOf(@PARAM)+1]{@PARAM[$i$],}
DEFINE DEFAULT_MAPPING AS [i<arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$,}
      [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}
      [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}

a_out(OUTPUT_SCHEMA) <- a_in1(INPUT_SCHEMA),
      @FUNCTION (FUNCTION_INPUT,
      @FunOutFIELD ),@OUTFIELD=@FunOutFIELD, DEFAULT_MAPPING.
```

Macro Expansion

```
a_out([i<arityOf(a_in)+1]{A_OUT_$i$,} OUTFIELD) <-
      a_in([i<arityOf(a_in)]{A_IN_$i$,}[i= arityOf(a_in)]{A_IN_$i$}),
      @FUNCTION([i< arityOf(@PARAM[$i$])+1]{@PARAM[$i$],} OUTFIELD),
      [i<arityOf(a_in)] {A_OUT_$i$=A_IN_$i$,}
      [i=arityOf(a_in)] {A_OUT_$i$=A_IN_$i$}.
```

Parameter instantiation

```
@FUNCTION=f1
@PARAM[1]=A_IN_2
@PARAM[2]=A_IN_3
```

Loop productions

```
a_out(A_OUT_1, A_OUT_2, A_OUT_3, OUTFIELD)<-
      a_in(A_IN_1, A_IN_2, A_IN_3),
      @FUNCTION(@PARAM[1],@PARAM[2],OUTFIELD),
      A_OUT_1=A_IN_1,A_OUT_2=A_IN_2,A_OUT_3=A_IN_3.
```

Variable Instantiation

```
a_out(A_OUT_1, A_OUT_2, A_OUT_3, OUTFIELD)<-
       a_in(A_IN_1, A_IN_2, A_IN_3),
       f1(A_IN_2, A_IN_3,OUTFIELD),
       A_OUT_1=A_IN_1, A_OUT_2=A_IN_2, A_OUT_3=A_IN_3.
```

Keyword Renaming

```
myFunc_out(PKEY_OUT, COST_OUT, PRICE_OUT, PROFIT)<-
       myFunc_in(PKEY_IN, COST_IN, PRICE_IN),
       subtract(COST_IN, PRICE_IN, PROFIT),
       PKEY_OUT=PKEY_IN, COST_OUT=COST_IN, PRICE_OUT=PRICE_IN.
```

Fig. 3.4 Instantiation procedure

The first row, *Template*, shows the initial template as it has been registered by the designer. `@FUNCTION` holds the name of the function to be used, `subtract` in our case, and the `@PARAM[]` holds the inputs of the function, which in our case are the two attributes of the input schema. The problem we have to face is that all input, output and function schemata have a variable number of parameters. To abstract from the complexity of this problem, we define four macro definitions, one for each schema (`INPUT_SCHEMA`, `OUTPUT_SCHEMA`, `FUNCTION_INPUT`) along with a macro for the mapping of input to output attributes (`DEFAULT_MAPPING`). The second row, *Macro Expansion*, shows how the template looks after the macros have been incorporated in the template definition. The mechanics of the expansion are straightforward: observe how the attributes of the output schema are specified by the expression `[i<arityOf(a_in)+1]{A_OUT_$i$,}OUTFIELD` as an expansion of the macro `OUTPUT_SCHEMA`. In a similar fashion, the attributes of the input schema and the parameters of the function are also specified; note that the expression for the last attribute in the list is different (to avoid repeating an erroneous comma). The mappings between the input and the output attributes are also shown in the last two lines of the template. In the third row, *Parameter instantiation*, we can see how the parameter variables were materialized at instantiation. In the fourth row, *Loop production*, we can see the intermediate results after the loop expansions are done. As it can easily be seen these expansions must be done before `@PARAM[]` variables are replaced by their values. In the fifth row, *Variable instantiation*, the parameter variables have been instantiated creating a default mapping between the input, the output and the function

attributes. Finally, in the last row, *Keyword renaming*, the output LDL code is presented after the keywords are renamed. Keyword instantiation is done on the basis of the schemata and the respective attributes of the activity that the user chooses.

3.2.3 Taxonomy: Simple and Program-Based Templates

Most commonly used activities can be easily expressed by a single predicate template; it is obvious, though, that it would be very inconvenient to restrict activity templates to single predicates. Thus, we separate template activities in two categories, *Simple Templates*, which cover single-predicate templates and *Program-Based Templates* where many predicates are used in the template definition.

In the case of *Simple Templates*, the output predicate is bound to the input through a mapping and an expression. Each of the rules for obtaining the output is expressed in terms of the input schemata and the parameters of the activity. In the case of *Program Templates*, the output of the activity is expressed in terms of its intermediate predicate schemata, as well as its input schemata and its parameters. Program-Based Templates are often used to define activities that employ constraints like *does-not-belong*, or *does-not-exist*, which need an intermediate negated predicate to be expressed intuitively. This predicate usually describes the conjunction of properties we want to avoid, and then it appears negated in the output predicate. Thus, in general, we allow the construction of a LDL program, with intermediate predicates, in order to enhance intuition. This classification is orthogonal to the logical one of Section 3.1.

Template	DEFINE INPUT_SCHEMA AS
	      [i<arityOf(a_in1)] {A_IN1_i,}
	      [i=arityOf(a_in1)] {A_IN1_i}
	DEFINE OUTPUT_SCHEMA AS
	      [i<arityOf(a_in)] {A_OUT_i,}
	      [i=arityOf(a_out)] {A_OUT_i}
	DEFINE DEFAULT_MAPPING AS
	      [i<arityOf(a_out)] {A_OUT_i= A_IN1_i,}
	      [i=arityOf(a_out)] {A_OUT_i= A_IN1_i}
	a_out(OUTPUT_SCHEMA) <-
	      a_in1(INPUT_SCHEMA),
	      @FIELD >=@Xlow,
	      @FIELD <= @Xhigh,
	      DEFAULT_MAPPING.

Full Definition of Template	a_out([i<arityOf(a_out)]{A_OUT_i,}[i=arityOf(a_out)]{A_OUT_i})<-
	      a_in1([i<arityOf(a_in1)]{A_IN1_i,}[i=arityOf(a_in1)]{A_IN1_i}),
	      @FIELD >=@Xlow,
	      @FIELD <= @Xhigh,
	      [i<arityOf(a_out)] {A_OUT_i= A_IN1_i,}
	      [i=arityOf(a_out)] {A_OUT_i= A_IN1_i}.

Parameter instantiation	@FIELD=A_IN_3 @Xlow =5 @Xhigh = 10

Example	dM1_out(DM1_OUT_1, DM1_OUT_2, DM1_OUT_3, DM1_OUT_4) <-
	     dM1_in(DM1_IN_1, DM1_IN_2, DM1_IN_3, DM1_IN_4),
	     DM1_IN_3 >=5, DM1_IN_3<=10,
	     DM1_OUT_1=DM1_IN_1,
	     DM1_OUT_2=DM1_IN_2,
	     DM1_OUT_3=DM1_IN_3,
	     DM1_OUT_4=DM1_IN_4.

Fig. 3.5 Simple Template Example: Domain Mismatch

Simple Templates. Formally, the expression of an activity which is based on a certain simple template is produced by a set of rules of the following form:

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

where `INPUT(),OUTPUT()` denote the full expression of the respective schemata; in the case of multiple input schemata, `INPUT()`expresses the conjunction of the input schemata. `MAPPING` denotes any mapping between the input, output, and expression attributes. A default mapping can be explicitly done at the template level, by specifying equalities between attributes, where the first attribute of the input schema is mapped to the first attribute of the output schema, the second to the respective second one and so on. At instantiation time, the user can change these mappings easily, especially in the presence of the graphical interface. Note also that despite the fact that LDL allows implicit mappings by giving identical names to attributes that must be equal our design choice was to give explicit equalities in order to support the preservation of the names of the attributes of the input and output schemata at instantiation time.

To make ourselves clear, we will demonstrate the usage of simple template activities through an example. Suppose, thus, the case of the `Domain Mismatch` template activity, checking whether the values for a certain attribute fall within a particular range. The rows that abide by the rule pass the check performed by the activity and they are propagated to the output.

Observe Fig. 3.5, where we present an example of the definition of a template activity and its instantiation in a concrete activity. The first row in Fig. 3.5 describes the definition of the template activity. There are three parameters; `@FIELD`, for the field that will be checked against the expression, `@Xlow` and `@Xhigh` for the lower and upper limit of acceptable values for attribute `@FIELD`. The expression of the template activity is a simple expression guaranteeing that `@FIELD` will be within the specified range. The second row of Fig. 3.5 shows the template after the macros are expanded. Let us suppose that the activity named `DM1` materializes the templates parameters that appear in the third row of Fig. 3.5, i.e., specifies the attribute over which the check will be performed (`A_IN_3`) and the actual ranges for this check (`5`, `10`). The fourth row of Fig. 3.5 shows the resulting instantiation after keyword renaming is done. The activity includes an input schema `dm1_in`, with attributes `DM1_IN_1`, `DM1_IN_2`, `DM1_IN_3`, `DM1_IN_4` and an output schema `dm1_out` with attributes `DM1_OUT_1`, `DM1_OUT_2`, `DM1_OUT_3`, `DM1_OUT_4`. In this case the parameter `@FIELD` implements a dynamic internal mapping in the template, whereas the `@Xlow`, `@Xigh` parameters provide values for constants. The mapping from the input to the output is hardcoded in the template.

Program-Based Templates. The case of *Program_Based Templates* is somewhat more complex, since the designer who records the template creates more than one predicate to describe the activity. This is usually the case of operations where we want to verify that some data do not have a conjunction of certain properties. Such constraints employ negation to assert that a tuple does not satisfy a predicate, which is defined in way that it requires that the data that satisfy it have the properties we want to avoid. Such negations can be expressed by *more than one rules*, for the same predicate, that each negates just one property according to the logical rule $\neg(q \wedge p) \equiv \neg q \vee \neg p$. Thus, in general, we allow the construction of a LDL program, with intermediate predicates, in order to enhance intuition. For example the *does-not-belong* relation, which is needed in the `Difference` activity template, needs a second predicate to be expressed intuitively.

Let us see in more detail the case of `Difference`. During the ETL process, one of the very first tasks that we perform is the detection of newly inserted and possibly updated records. Usually, this is physically performed by the comparison of two snapshots (one corresponding to the previous extraction and the other to the current one). To capture this process, we introduce a variation of the classical relational difference operator, which checks for equality only on a certain subset of attributes of the input records. Assume that during the extraction process we want to detect the newly inserted rows. Then, if **`PK`** is the set of attributes that uniquely identify rows (in the role of a primary key), the newly inserted rows can be found from the expression $\Delta_{<\mathbf{PK}>}(R_{new}, R)$. The formal semantics of the difference operator are given by the following calculus-like definition: $\Delta_{<A1 \ldots Ak>}(R, S) = \{x \in R \mid \neg\exists y \in S: x[A_1]=y[A_1] \wedge \ldots \wedge x[A_k]=y[A_k]\}$.

In Fig. 3.6, we can see the template of the `Difference` activity and a resulting instantiation for an activity named `dF1`. As we can see we need the `semijoin` predicate so we can exclude all tuples that satisfy it. Note also that we have two different inputs, which are denoted as distinct by adding a number at the end of the keyword `a_in`.

Template

```
 DEFINE INPUT_1_SCHEMA AS
       [i<arityOf(a_in1)] {A_IN1_$i$,}
       [i=arityOf(a_in1)] {A_IN1_$i$}
 DEFINE INPUTS_2_SCHEMA as
       [i<arityOf(a_in2)] {A_IN2_$i$,}
       [i=arityOf(a_in2)] {A_IN2_$i$}
 DEFINE OUTPUT_SCHEMA AS
       [i<arityOf(a_in)] {A_OUT_$i$,}
       [i=arityOf(a_out)] {A_OUT_$i$}
 DEFINE DEFAULT_MAPPING AS
       [i<arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$,}
       [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}
 DEFINE COMMON_MAPPING as
       [i<arityOf(@COMMON_IN1)] {@COMMON_IN1[$i$]= @COMMON_IN2[$i$],}
       [i=arityOf(@COMMON_IN1)] {@COMMON_IN1[$i$]= @COMMON_IN2[$i$]}.
a_out(OUTPUT_SCHEMA) <-
      a_in1(INPUTS_1_SCHEMA), a_in2(INPUTS_2_SCHEMA),
      semijoin(INPUTS_1_SCHEMA),
      DEFAULT_MAPPING.
semijoin(INPUTS_1_SCHEMA) <-
      a_in1(INPUTS_1_SCHEMA),
      a_in2(INPUTS_2_SCHEMA),
      COMMON_MAPPING.
```

Full Definition of Template

```
a_out([i<arityOf(a_out)]{A_OUT_$i$,}[i=arityOf(a_out)]{A_OUT_$i$})<-
      a_in1([i<arityOf(a_in1)]{A_IN1_$i$,}[i=arityOf(a_in1)]{A_IN1_$i$}),
      a_in2([i<arityOf(a_in2)]{A_IN2_$i$,}[i=arityOf(a_in2)]{A_IN2_$i$}),
      ~semijoin([i<arityOf(a_in1)]{A_IN1_$i$,}
      [i=arityOf(a_in1)]{A_IN1_$i$}),
      [i<arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$,}
      [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}
      .
semijoin([i<arityOf(a_in1)]{A_IN1_$i$,}[i=arityOf(a_in1)]{A_IN1_$i$})<-
      a_in1([i<arityOf(a_in1)]{A_IN1_$i$,}[i=arityOf(a_in1)]{A_IN1_$i$}),
      a_in2([i<arityOf(a_in2)]{A_IN2_$i$,}[i=arityOf(a_in2)]{A_IN2_$i$}),
      [i<arityOf ( @COMMON_IN1 )] {@COMMON_IN1[$i$]=@COMMON_IN2[$i$],}
      [i=arityOf ( @COMMON_IN1 )] {@COMMON_IN1[$i$]=@COMMON_IN2[$i$]}
      .
```

Parameter instantiation

```
@COMMON_IN1_1=A_IN1_1
@COMMON_IN1_2=A_IN1_3

@COMMON_IN2_1=A_IN2_3
@COMMON_IN2_2=A_IN2_2

@COMMON_NUM=2
```

Example

```
dF1_out(DF1_OUT_1,DFI1_OUT_2, DFI1_OUT_3) <-
      dF1_in1(DF1_IN1_1,DFI1_IN1_2, DFI1_IN1_3),
      dF1_in2(DF1_IN2_1,DFI1_IN2_2, DFI1_IN2_3),
      ~semijoin(DF1_OUT_1,DFI1_OUT_2, DFI1_OUT_3),
      DF1_OUT_1=DF1_IN1_1,
      DF1_OUT_2=DF1_IN1_2,
      DF1_OUT_3=DF1_IN1_3.

semijoin(DF1_IN1_1,DFI1_IN1_2, DFI1_IN1_3) <-
      dF1_in1(DF1_IN1_1,DFI1_IN1_2, DFI1_IN1_3),
      dF1_in2(DF1_IN2_1,DFI1_IN2_2, DFI1_IN2_3),
      DF1_IN1_1=DF1_IN2_1,
      DF1_IN1_2=DF1_IN2_2,
      DF1_IN1_3=DF1_IN2_3.
```

Fig. 3.6 Program-Based Template Example: `Difference` activity

4. IMPLEMENTATION

In the context of the aforementioned framework, we have implemented a graphical design tool, ARKTOS II, with the goal of facilitating the design of ETL scenarios, based on our model. In order to design a scenario, the user defines the source and target data stores, the participating activities and the flow of the data in the scenario. These tasks are greatly assisted (a) by a friendly GUI and (b) by a set of reusability templates.

All the details defining an activity can be captured through forms and/or simple point and click operations. More specifically, the user may explore the data sources and the activities already defined in the scenario, along with their schemata (input, output, and parameter). Attributes belonging to an output schema of an activity or a recordset can be “drag’n’dropped” in the input schema of a subsequent activity or recordset, in order to create the equivalent data flow in the scenario. In a similar design manner, one can also set the parameters of an activity. By default the output schema of the activity is instantiated as a copy of the input schema. Then, the user has the ability to modify this setting according to his demands, e.g., by deleting or renaming the proper attributes. The rejection schema of an activity is considered to be a copy of the input schema of the respective activity and the user may determine its physical location, e.g., the physical location of a log file that maintains the rejected rows of the specified activity. Apart from these features, the user can (a) draw the desirable attributes or parameters, (b) define their name and data type, (c) connect them to their schemata, (d) create provider and regulator relationships between them, and (e) drawing the proper edges from one node of the architecture graph to another. The system assures the consistency of a scenario, by allowing the user to draw only relationships respecting the restrictions imposed from the model. As far as the provider and instance-of relationships concerned, they are calculated automatically and their display can be turned on or off from an application’s menu. Moreover, the system allows the designer to define activities through a form-based interface, instead of defining them through the point-and-click interface. Naturally, the form automatically provides lists with the available recordsets, their attributes, etc. Fig. 4.1 shows the design canvas of our GUI, where our motivating example is depicted.

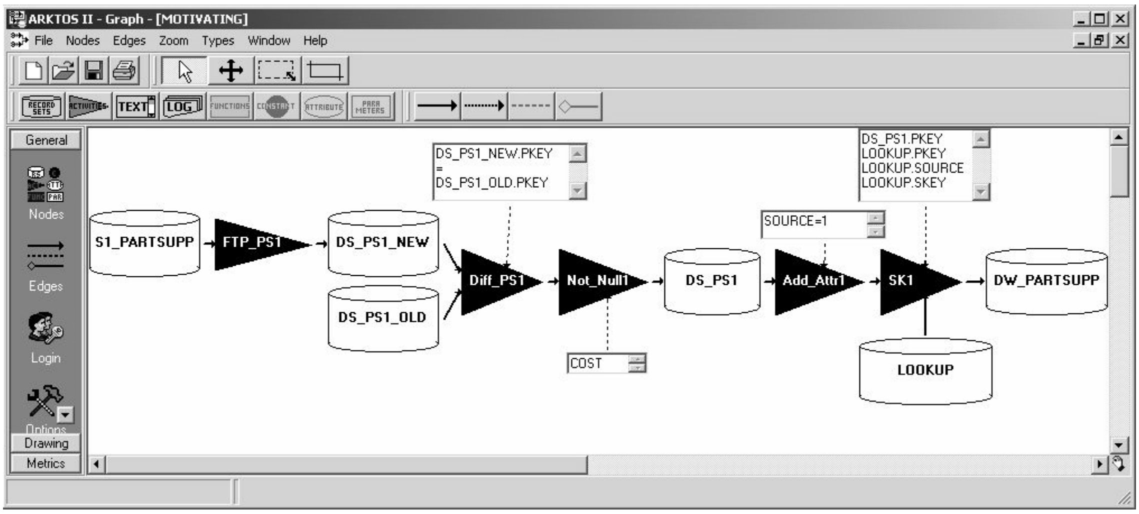


Fig. 4.1 The motivating example in ARKTOS II

ARKTOS II offers zoom-in/zoom-out capabilities, a particularly useful feature in the construction of the data flow of the scenario through inter-attribute “provider” mappings. The designer can deal with a scenario in two levels of granularity: (a) at the *entity* or *zoom-out level*, where only the participating recordsets and activities are visible and their provider relationships are abstracted as edges between the respective entities, or (b) at the *attribute* or *zoom-in level*, where the user can see and manipulate the constituent parts of an activity, along with their respective providers at the attribute level. In Fig. 4.2, we show a part of the scenario of Fig. 4.1. Observe (a) how part-of relationships are expanded to link attributes to their corresponding entities; (b) how provider relationships link attributes to each other (c) how regulator relationships populate activity parameters and (d) how instance-of relationships relate attributes with their respective data types that are depicted at the lower right part of the figure.

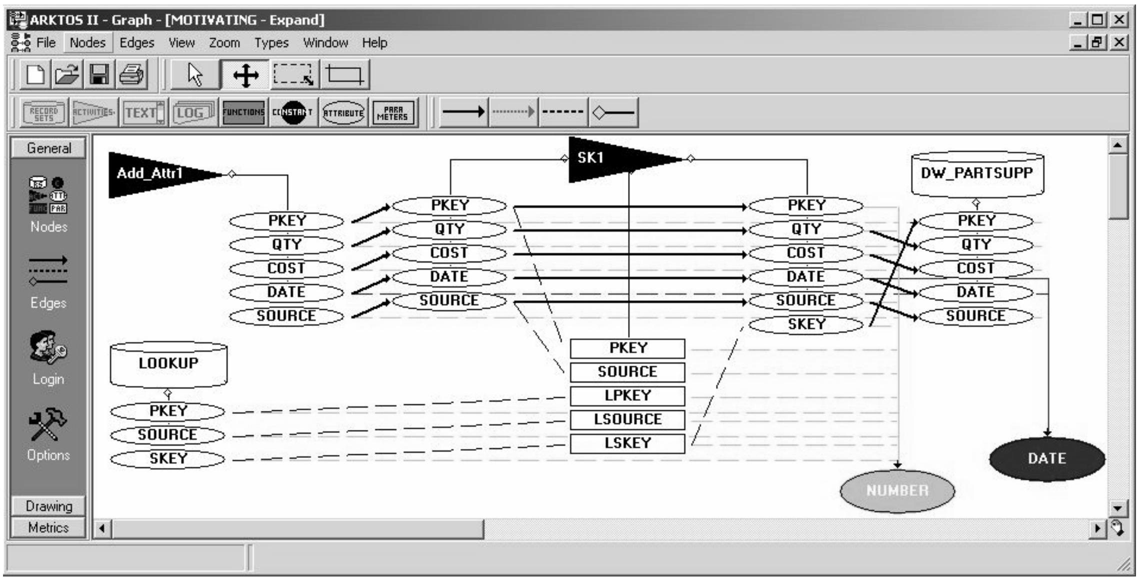
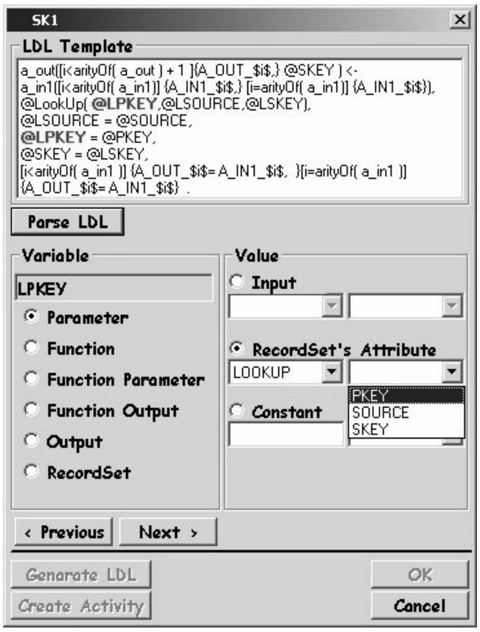


Fig. 4.2 A detailed zoom-in view of the motivating example

In ARKTOS II, the *customization* principle is supported by the reusability templates. The notion of template is in the heart of ARKTOS II, and there are templates for practically every aspect of the model: data types, functions and activities. Templates are extensible; thus, providing the user with the possibility of customizing the environment according to his/her own needs. Especially for activities, which form the core of our model, a specific menu with a set of frequently used ETL Activities is provided. The system has a built-in mechanism responsible for the instantiation of the LDL templates, supported by a graphical form (Fig. 4.3) that helps the user define the variables of the template by selecting its values among the appropriate scenario's objects.


SK1

LDL Template

```
a_out([i<arityOf( a_out ) + 1 ]{A_OUT_$i$,} @SKEY ) <-
a_in1([i<arityOf( a_in1)] {A_IN1_$i$,} [i=arityOf( a_in1)] {A_IN1_$i$}),
@LookUp( @LPKEY,@LSOURCE,@LSKEY),
@LSOURCE = @SOURCE,
@LPKEY = @PKEY,
@SKEY = @LSKEY,
[i<arityOf( a_in1 )] {A_OUT_$i$= A_IN1_$i$,  }[i=arityOf( a_in1 )]
{A_OUT_$i$= A_IN1_$i$}  .
```

Variable: LPKEY

● Parameter
○ Function
○ Function Parameter
○ Function Output
○ Output
○ RecordSet

Value

○ Input
● RecordSet's Attribute: LOOKUP
PKEY
SOURCE
SKEY
○ Constant

Fig. 4.3 The graphical form for the definition of the activity `SK1`.

Another distinctive feature of ARKTOS II is the computation of the scenario's design quality by employing a set of metrics that are presented in [VaSS02], either for the whole scenario or for each activity of it.

The scenarios are stored in ARKTOS II repository (implemented in a relational DBMS); the system allows the user to store, retrieve and reuse existing scenarios. All the metadata of the system involving the scenario configuration, the employed templates and their constituents are stored in the repository. The choice of a relational DBMS for our metadata repository allows its efficient querying as well as the smooth integration with external systems and/or future extensions of ARKTOS II. The connectivity to source and target data stores is achieved through ODBC connections and the tool offers an automatic reverse engineering of their schemata. We have implemented ARKTOS II with Oracle 8.1.7 as basis for our repository and Ms Visual Basic (Release 6) for developing our GUI.

5. RELATED WORK

In this section, we will report (a) on related commercial studies and tools in the field of ETL, (b) on related efforts in the academia in the issue, and (c) applications of workflow technology in the field of data warehousing.

5.1 Commercial studies and tools

In a recent study [Giga02], the authors report that due to the diversity and heterogeneity of data sources, ETL is unlikely to become an open commodity market. The ETL market has reached a size of $667 millions for year 2001; still the growth rate has reached a rather low 11% (as compared with a rate of 60% growth for year 2000). This is explained by the overall economic downturn environment. In terms of technological aspects, the main characteristic of the area is the involvement of traditional database vendors with ETL solutions built in the DBMS's. The three major database vendors that practically ship ETL solutions "at no extra charge" are pinpointed: Oracle with Oracle Warehouse Builder [Orac03], Microsoft with Data Transformation Services [Micr02] and IBM with the Data Warehouse Center [IBM03]. Still, the major vendors in the area are Informatica's Powercenter [Info03] and Ascential's DataStage suites [Asce03, Asce03a] (the latter being part of the IBM recommendations for ETL solutions). The study goes on to propose future technological challenges/forecasts that involve the integration of ETL with (a) XML adapters, (b) EAI (Enterprise Application Integration) tools (e.g., MQ-Series), (c) customized data quality tools, and (d) the move towards parallel processing of the ETL workflows.

The aforementioned discussion is supported from a second recent study [Gart03], where the authors note the decline in license revenue for pure ETL tools, mainly due to the crisis of IT spending and the appearance of ETL solutions from traditional database and business intelligence vendors. The Gartner study discusses the role of the three major database vendors (IBM, Microsoft, Oracle) and points that they slowly start to take a portion of the ETL market through their DBMS-built-in solutions.

In the sequel, we elaborate more on the major vendors in the area of the commercial ETL tools, and we discuss three tools that the major database vendors provide, as such two ETL tools that are considered as best sellers. But, we stress the fact that the former three have the benefit of the minimum cost, because they are shipped with the database, while the latter two have the benefit to aim at complex and deep solutions not envisioned by the generic products.

IBM. DB2 Universal Database offers the Data Warehouse Center [IBM03], a component that automates data warehouse processing, and the DB2 Warehouse Manager that extends the capabilities of the Data Warehouse Center with additional agents, transforms and metadata capabilities. Data Warehouse Center is used to define the processes that move and transform data for the warehouse. Warehouse Manager is used to schedule, maintain, and monitor these processes. Within the Data Warehouse Center, the *warehouse schema modeler* is a specialized tool for generating and storing schema associated with a data warehouse. Any schema resulting from this process can be passed as metadata to an OLAP tool. The *process modeler* allows user to graphically link the steps needed to build and maintain data warehouses and dependent data marts. DB2 Warehouse Manager includes enhanced ETL function over and above the base capabilities of DB2 Data Warehouse Center. Additionally, it provides metadata management, repository function, as such an integration point for third-party independent software vendors through the *information catalog*.

Microsoft. The tool that is offered by Microsoft to implement its proposal for the Open Information Model is presented under the name of *Data Transformation Services* [Micr00, BeBe99]. Data Transformation Services (DTS) are the data-manipulation utility services in SQL Server (from version 7.0) that provide import, export, and data-manipulating services between OLE DB [Micr00a], ODBC, and ASCII data stores. DTS are characterized by a basic object, called a package, that stores information on the aforementioned tasks and the order in which they need to be launched. A package can include one or more connections to different data sources, and different tasks and transformations that are executed as steps that define a workflow process [Gra+01]. The software modules that support DTS are shipped with MS SQL Server. These modules include:

- *DTS Designer* : A GUI used to interactively design and execute DTS packages

- *DTS Export and Import Wizards*: Wizards that ease the process of defining DTS packages for the import, export and transformation of data
- *DTS Programming Interfaces*: A set of OLE Automation and a set of COM interfaces to create customized transformation applications for any system supporting OLE automation or COM.

Oracle. Oracle Warehouse Builder [Orac02, Orac03] is a repository-based tool for ETL and data warehousing. The basic architecture comprises two components, the *design environment* and the *runtime environment*. Each of these components handles a different aspect of the system; the design environment handles metadata, the runtime environment handles physical data. The metadata component revolves around the metadata repository and the design tool. The repository is based on the Common Warehouse Model (CWM) standard and consists of a set of tables in an Oracle database that are accessed via a Java-based access layer. The front-end of the tool (entirely written in Java) features wizards and graphical editors for logging onto the repository. The data component revolves around the runtime environment and the warehouse database. The Warehouse Builder runtime is a set of tables, sequences, packages, and triggers that are installed in the target schema. The code generator that bases on the definitions stores in the repository, it creates the code necessary to implement the warehouse. Warehouse Builder generates extraction specific languages (SQL*Loader control files for flat files, ABAP for SAP/R3 extraction and PL/SQL for all other systems) for the ETL processes and SQL DDL statements for the database objects. The generated code is deployed, either to the file system or into the database.

Ascential Software. DataStage XE suite from Ascential Software [Asce03, Asce03a] (formerly Informix Business Solutions) is an integrated data warehouse development toolset that includes an ETL tool (DataStage), a data quality tool (Quality Manager), and a metadata management tool (MetaStage). The DataStage ETL component consists of four design and administration modules: *Manager*, *Designer*, *Director*, and *Administrator*, as such a *metadata repository*, and a *server*. The DataStage Manager is the basic metadata management tool. In the Designer module of DataStage, ETL tasks execute within individual “stage” objects (source, target, and transformation stages), in order to create ETL tasks. The Director is DataStage's job validation and scheduling module. The DataStage Administrator is primarily for controlling security functions. The DataStage Server is the engine that moves data from source to target.

Informatica. Informatica PowerCenter [Info03] is the industry-leading (according to recent studies [Giga02, Gart03]) data integration platform for building, deploying, and managing enterprise data warehouses, and other data integration projects. The workhorse of Informatica PowerCenter is a data integration engine that executes all data extraction, transformation, migration and loading functions in-memory, without generating code or requiring developers to hand-code these procedures. The PowerCenter data integration engine is metadata-driven, creating a repository-and-engine partnership that ensures data integration processes are optimally executed.

5.2 Research Efforts

Research focused specifically on ETL. The *AJAX* system [GFSS00] is a data cleaning tool developed at INRIA France. It deals with typical data quality problems, such as *the object identity problem* [Cohe99], *errors due to mistyping* and *data inconsistencies* between matching records. This tool can be used either for a single source or for integrating multiple data sources. AJAX provides a framework wherein the logic of a data cleaning program is modeled as a directed graph of data transformations that start from some input source data. Four types of data transformations are supported:

- *Mapping transformations* standardize data formats (e.g. date format) or simply merge or split columns in order to produce more suitable formats.
- *Matching transformations* find pairs of records that most probably refer to same object. These pairs are called *matching pairs* and each such pair is assigned a similarity value.
- *Clustering transformations* group together matching pairs with a high similarity value by applying a given grouping criteria (e.g. by transitive closure).
- *Merging transformations* are applied to each individual cluster in order to eliminate duplicates or produce new records for the resulting integrated data source.

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. Finally, a interactive environment is supplied to the user in order to resolve errors and inconsistencies that cannot be automatically handled and support a stepwise refinement design of data cleaning programs. The theoretic foundations of this tool can be found in [GFSS99], where apart from the presentation of a general framework for the data cleaning process, specific optimization techniques tailored for data cleaning applications are discussed.

[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. Users gradually build transformations to clean the data by adding or undoing transforms on a spreadsheet-like interface; the effect of a transform is shown at once on records visible on screen. These transforms are specified either through simple graphical operations, or by showing the desired effects on example data values. In the background, Potter's Wheel automatically infers structures for data values in terms of user-defined domains, and accordingly checks for constraint violations. Thus users can gradually build a transformation as discrepancies are found, and clean the data without writing complex programs or enduring long delays.

We believe that the AJAX tool is mostly oriented towards the integration of web data (which is also supported by the ontology of its algebraic transformations); at the same time, Potter's wheel is mostly oriented towards an interactive data cleaning tool, where the users interactively choose data. With respect to these approaches, we believe that our technique contributes (a) by offering an extensible framework though a uniform extensibility mechanism, and (b) by providing formal foundations to allow the reasoning over the constructed ETL scenarios. Clearly, ARKTOS II is a design tool for traditional data warehouse flows; therefore, we find the aforementioned approaches complementary (especially Potter's Wheel). At the same time, when contrasted with the industrial tools, it is evident that although ARKTOS II is only a design environment for the moment, the industrial tools lack the logical abstraction that our model, implemented in ARKTOS II, offers; on the contrary, industrial tools are concerned directly with the physical perspective (at least to the best of our knowledge).

Data quality and cleaning. An extensive review of data quality problems and related literature, along with quality management methodologies can be found in [JLVV00]. [Rund99] offers a discussion on various aspects on data transformations. [Sara00] is a similar collection of papers in the field of data including a survey [RaDo00] that provides an extensive overview of the field, along with research issues and a review of some commercial tools and solutions on specific problems, e.g., [Mong00, BoDS00]. In a related, still different, context, we would like to mention the IBIS tool [Cal+03]. IBIS is an integration tool following the global-as-view approach to answer queries in a mediated system. Departing from the traditional data integration literature though, IBIS brings the issue of data quality in the integration process. The system takes advantage of the definition of constraints at the intentional level (e.g., foreign key constraints) and tries to provide answers that resolve semantic conflicts (e.g., the violation of a foreign key constraint). The interesting aspect here is that consistency is traded for completeness. For example, whenever an offending row is detected over a foreign key constraint, instead of assuming the violation of consistency, the system assumes the absence of the appropriate lookup value and adjusts its answers to queries accordingly [CCDL02].

Workflows. To the best of our knowledge, research on workflows is focused around the following reoccurring themes: (a) modeling [WfMC98, AHKB00, EdGr02, SaOr00, KiHB00], where the authors are primarily concerned in providing a metamodel for workflows; (b) correctness issues [EdGr02, SaOr00, KiHB00], where criteria are established to determine whether a workflow is well formed, and (c) workflow transformations [EdGr02, SaOr00, KiHB00] where the authors are concerned on correctness issues in the evolution of the workflow from a certain plan to another.

[WfMC98] is a standard proposed by the Workflow Management Coalition (WfMC). The standard includes a metamodel for the description of a workflow process specification and a textual grammar for the interchange of process definitions. A *workflow process* comprises of a network of *activities*, their interrelationships, criteria for staring/ending a process and other information about *participants*, invoked *applications* and relevant *data*. Also, several other kinds of entities which are external to the workflow, such as system and environmental data or the organizational model are roughly described. In [DaRe99] several interesting research results on workflow management are presented in the field of electronic commerce, distributed execution and adaptive workflows. Still, there is no reference to data flow modeling efforts. In [AHKB00] the authors provide an overview of the most frequent control flow patterns in workflows. The patterns refer explicitly to control flow structures like activity sequence, AND/XOR/OR split/join and so on. Several commercial tools are evaluated against the 26 patterns presented. In [EdGr02, SaOr00, KiHB00] the authors, based on minimal metamodels, try to provide correctness criteria in order to derive equivalent plans for the same workflow scenario.

In more than one works [AHKB00, SaOr00] the authors mention the necessity for the perspectives already discussed in the introduction of the paper. Data flow or data dependencies are listed within the components of the definition of a workflow; still in all these works the authors quickly move on to assume that control flow is the primary aspect of workflow modeling and do not deal with data-centric issues any further. It is particularly interesting that the [WfMC] standard is not concerned with the role of business data at all. The primary focus of the WfMC standard is the interfaces that connect the different parts of a workflow engine and the transitions between the states of a workflow. No reference is made to business data (although the standard refers to data which are relevant for the transition from one state to another, under the name *workflow related data*).

5.3 Applications of ETL workflows in data warehouses.

Finally, we would like to mention that the literature reports several efforts (both research and industrial) for the management of processes and workflows that operate on data warehouse systems. In [JQB+99], the authors describe an industrial effort where the cleaning mechanisms of the data warehouse are employed in order to avoid the population of the sources with problematic data in the fist place. The described solution is based on a workflow which employs techniques from the field of view maintenance. [ScBJ00] describes an industrial effort at Deutche Bank, involving the import/export, transformation and cleaning and storage of data in a Terabyte-size data warehouse. The paper explains also the usage of metadata management techniques, which involves a broad spectrum of applications, from the import of data to the management of dimensional data and more importantly for the querying of the data warehouse. [JaLK00] presents a research effort (and its application in an industrial application) for the integration and central management of the processes that lie around an information system. A metadata management repository is employed to store the different activities of a large workflow, along with important data that these processes employ.

Finally, we should refer the interested reader to [VaSS02] for a detailed presentation of ARKTOS II model. The model is accompanied by a set of *importance metrics* where we exploit the graph structure to measure the degree to which activities/recordsets/attributes are bound to their data providers or consumers. In [VaSS02a] we propose a complementary conceptual model for ETL scenarios and in [SiVa03] a methodology for constructing it.

6. DISCUSSION

In this section we would like to briefly discuss some comments on the overall evaluation of our approach. Our proposal involves the data modeling part of ETL activities, which are modeled as workflows in our setting; nevertheless, it is not clear whether we covered all possible problems around the topic. Therefore, in this section, we will explore three issues as an overall assessment of our proposal. First, we will discuss its completeness, i.e., whether there are parts of the data modeling that we have missed. Second, we will discuss the possibility of further generalizing our approach to the general case of workflows. Finally, we will exit the domain of the logical design and deal with performance and stability concerns around ETL workflows.

Completeness. A first concern that arises, involves the completeness of our approach. We believe that the different layers of Fig. 1.1 fully cover the different aspects of workflow modeling. We would like to make clear that we focus on the data-oriented part of the modeling, since ETL activities are mostly concerned with a well established *automated* flow of cleanings and transformations, rather than an interactive session where *user decisions and actions* direct the flow (like for example in [CCPP95]).

Still, is this enough to capture all the aspects of the data-centric part of ETL activities? Clearly, we do not provide any "formal" proof for the completeness of our approach. Nevertheless, we can justify our basic assumptions based on the related literature in the field of software metrics, and in particular, on the method of *function points* [Albr79, Pres00]. Function points is a methodology trying to quantify the functionality (and thus the required development effort) of an application. Although based on assumptions that pertain to the technological environment of the late 70's, the methodology is still one of the most cited in the field of software measurement. In any case, function points compute the measurement values based on the five following characteristics: (i) user inputs, (ii) user outputs, (iii) user inquiries, (iv) employed files, and (v) external interfaces.

We believe that an activity in our setting covers all the above quite successfully, since (a) it employs input and output schemata to obtain and forward data (characteristics i, ii and iii), (b) communicates with files (characteristic iv) and other activities (practically characteristic v). Moreover, it is tuned by some user-provided parameters, which are not explicitly captured by the overall methodology but are quite related to characteristics (iii) and (v). As a more general view on the topic we could claim that it is sufficient to characterize activities with input and output schemata, in order to denote their linkage to data (and other activities, too), while treating parameters as part of the input and/or output of the activity, depending on their nature. We follow a more elaborate approach, treating parameters separately, mainly because they are instrumental in defining our template activities.

Generality of the results. A second issue that we would like to bring up is the general applicability of our approach. Is it possible that we apply this modeling for the general case of workflows, instead simply for the ETL ones? As already mentioned, to the best of our knowledge, typical research efforts in the context of workflow management are concerned with the management of the control flow in a workflow environment. This is clearly due to the complexity of the problem and its practical application to semi-automated, decision-based, interactive workflows where user choices play a crucial role. Therefore, our proposal for a structured management of the data flow, concerning both the interfaces and the internals of activities appears to be complementary to existing approaches for the case of workflows that need to access structured data in some kind of data store, or to exchange structured data between activities.

It is possibly however, that due to the complexity of the workflow, a more general approach should be followed, where activities have multiple inputs and outputs, covering all the cases of different interactions due to the control flow. We anticipate that a general model for business workflows will employ activities with inputs and outputs, internal processing, and communication with files and other activities (along with all the necessary information on control flow, resource management, etc); nevertheless, we find this to be outside the context of this paper.

Execution characteristics. A third concern involves performance. Is it possible to model ETL activities with workflow technology? Typically, the back-stage of the data warehouse operates under strict performance requirements, where a loading time-window dictates how much time is assigned to the overall ETL process to refresh the contents of the data warehouse. Therefore, performance is really a major concern in such an environment. Clearly, in our setting we do not have in mind EAI or other message-oriented technologies to bring the loading task to a successful end. On the contrary, we strongly believe that the volume of data is the major factor of the overall process (and not, for example, any user-oriented decisions). Nevertheless, to our point of view, the paradigm of activities that feed one another with data during the overall process is more than suitable.

Let us mention a recent experience report on the topic: in [AdFi03], the authors report on their data warehouse population system. The architecture of the system is discussed in the paper, with particular interest (a) in a “shared data area”, which is an in-memory area for data transformations, with a specialized area for rapid access to lookup tables and (b) the pipelining of the ETL processes. A case study for mobile network traffic data is also discussed, involving around 30 data flows, 10 sources, and around 2TB of data, with 3 billion rows for the major fact table. In order to achieve a throughput of 80M rows/hour and 100M rows/day, the designers of the system were practically obliged to exploit low level OCI calls, in order to avoid storing loading data to files and then loading them through loading tools. With 4 hours of loading window for all this workload, the main issues identified involve (a) performance, (b) recovery, (c) day by day maintenance of ETL activities, and (d) adaptable and flexible activities. Based on the above, we believe that the quest for a workflow, rather than a push-and-store paradigm, is quite often the only way to follow.

Of course, this kind of workflow approach possibly suffers in the issue of software stability, and mostly recovery. Having a big amount of transient data, processed through a large set of activities in main memory is clearly vulnerable to both software and hardware failures. Moreover, once a failure has occurred, rapid recovery, if possible within the loading time-window is also a strong desideratum. Techniques to handle the issue of recovery already exist. To our knowledge the most prominent one is the one by [LWGG00], where the ordering of data is taking into consideration. Checkpoint techniques guarantee that once the activity output is ordered, recovery can start right at the point where the activity did the last checkpoint, thus speeding up the whole process significantly.

7. CONCLUSIONS

In this paper, we have focused on the data-centric part of logical design of the ETL scenario of a data warehouse. First, we have defined a formal logical metamodel as a logical abstraction of ETL processes. The data stores, activities and their constituent parts, as well as the provider relationships that map data producers to data consumers have formally been defined. We have also employed a declarative database programming language, LDL, to define the semantics of each activity. Then, we have provided a reusability framework that complements the genericity of the aforementioned metamodel. Practically, this is achieved from an extensible set of specializations of the entities of the metamodel layer, specifically tailored for the most frequent elements of ETL scenarios, which we call template activities. In the context of template materialization, we have dealt with specific language issues, in terms of the mechanics of template instantiation to concrete activities. Finally, we have presented a graphical design tool, ARKTOS II, with the goal of facilitating the design of ETL scenarios, based on our model.

Still, there exist several research issues left open, on the grounds of this work. In the context of ETL, the most basic topic is the efficient and reliable execution of an ETL scenario. We are currently working on the optimization of ETL scenarios under time and throughput constraints. The topic appears interesting, since the frequent usage of functions in ETL scenarios, drives the problem outside the expressive power of relational algebra (and therefore the traditional optimization techniques, used in the context of relational query optimizers). To problem becomes even more complex if one considers issues of reliability and recovery in the presence of failures or even issues of software quality (e.g., resilience to changes in the underlying data stores). Of course, the issue of providing optimal algorithms for individual ETL tasks (e.g., duplicate detection, surrogate key assignment, or identification of differentials) is also very interesting. In a different line of research we are also working towards providing a general model for the data flow of data-centric business workflows, involving issues of transactions, alternative interfaces in the context of control flow decisions and contingency scenarios.

REFERENCES

[AdFi03] J. Adzic, V. Fiore. Data Warehouse Population Platform. In Proc. 5th Intl. Workshop on the Design and Management of Data Warehouses (DMDW'03), Berlin, Germany, September 2003.

[AHKB00] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, Eindhoven University of Technology, Eindhoven, 2000. Available at the Workflow Patterns web site, at http://tmitwww.tm.tue.nl/research/patterns/documentation.htm

[Albr79] Albrecht, A. J., Measuring Application Development Productivity, in IBM Applications Development Symposium, Monterey, CA, 1979, pp. 83-92.

[Asce03] Ascential Software Inc. Available at: http://www.ascentialsoftware.com

[Asce03a] Ascential Software Products - Data Warehousing Technology. Available at: http://www.ascentialsoftware.com/products/datastage.html

[BoDS00] V. Borkar, K. Deshmuk, S. Sarawagi. Automatically Extracting Structure form Free Text Addresses. Bulletin of the Technical Committee on Data Engineering, 23(4), (2000).

[Cal+03] A. Calì, D. Calvanese, G. De Giacomo, M. Lenzerini, P. Naggar, F. Vernacotola. IBIS: Semantic data integration at work. In Proc. of the 15th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2003), volume 2681 of Lecture Notes in Computer Science, pages 79-94. Springer, 2003

[CCDL02] A. Calì, D. Calvanese, G. De Giacomo, M. Lenzerini. Data integration under integrity constraints. In Proc. of the 14th Int. Conf. on Advanced Information Systems Engineering (CAiSE 2002), volume 2348 of Lecture Notes in Computer Science, pages 262-279. Springer, 2002.

[CCPP95] F. Casati, S. Ceri, B. Pernici, G. Pozzi. Conceptual Modeling of Workflows. In Proc. Of OO-ER Conference, Australia, 1995.

[Cohe99] W.Cohen. Some practical observations on integration of Web information. In WebDB'99 Workshop in conj. with ACM SIGMOD, 1999.

[DaRe99] P. Dadam, M. Reichert (eds.). Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications. GI Workshop Informatik'99, 1999. Available at http://www.informatik.uni-ulm.de/dbis/veranstaltungen/Workshop-Informatik99-Proceedings.pdf

[EdGr02] Johann Eder, Wolfgang Gruber: A Meta Model for Structured Workflows Supporting Workflow Transformations. In Proc. 6th East European Conference on Advances in Databases and Information Systems (ADBIS 2002), pp: 326-339, Bratislava, Slovakia, September 8-11, 2002.

[Gart03] Gartner. ETL Magic Quadrant Update: Market Pressure Increases. Gartner's Strategic Data Management Research Note, M-19-1108, January 2003.

[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).

[GFSS99] H. Galhardas, D. Florescu, D. Shasha and E. Simon: An Extensible Framework for Data Cleaning,Technical Report INRIA 1999 (RR-3742).

[Giga02] Giga Information Group. Market Overview Update: ETL. Technical Report RPA-032002-00021, March 2002.

[Gra+01] Chris Graves, Mark Scott, Mike Benkovich, Paul Turley, Robert Skoglund, Robin Dewson, Sakhr Youness, Denny Lee, Sam Ferguson, Tony Bain, Terrence Joubert. Professional SQL Server 2000 Data Warehousing with Analysis Services. Wrox Press Ltd. (1st edition), October 2001.

[IBM03] IBM. IBM Data Warehouse Manager. Available at http://www-3.ibm.com/software/data/db2/datawarehouse/

[Info03] Informatica. PowerCenter. Available at http://www.informatica.com/products/data+integration/powercenter/default.htm

[JaLK00] M. Jarke, T. List, J. Koller. The challenge of process warehousing. Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000.

[JLVV00] M. Jarke, M. Lenzerini, Y. Vassiliou, P. Vassiliadis (eds.). Fundamentals of Data Warehouses. Springer-Verlag, (2000).

[JQB+99] M. Jarke, C. Quix, G. Blees, D. Lehmann, G. Michalk, S. Stierl. Improving OLTP Data Quality Using Data Warehouse Mechanisms. Proceedings of 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, USA, June 1999, pp. 537-538.

[KiHB00] Bartek Kiepuszewski, Arthur H. M. ter Hofstede, Christoph Bussler: On Structured Workflow Modeling. In Proc. 12th International Conference on Advanced Information Systems Engineering (CAiSE 2000), pp: 431-445, Stockholm, Sweden, June 5-9, 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.

[LVJS03] H.J. Lenz, P. Vassiliadis, M. Jeusfled, M. Staudt. Report on the 5th Intl. Workshop on the Design and Management of Data Warehouses (DMDW'03). Submitted to SIGMOD Record.

[LWGG00] W. Labio, J.L. Wiener, H. Garcia-Molina, V. Gorelik. Efficient Resumption of Interrupted

Warehouse Loads. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 2000), pp. 46-57, Dallas, Texas, USA (2000).

[Micr02] Microsoft. Data Transformation Services. Available at www.microsoft.com

[Mong00] A. Monge. Matching Algorithms Within a Duplicate Detection System. Bulletin of the Technical Committee on Data Engineering, 23(4), (2000).

[NaTs98] S. Naqvi, S. Tsur. A Logical Language for Data and Knowledge Bases. Computer Science Press 1989.

[Orac02] Oracle. Oracle 9i Warehouse Builder, Architectural White paper. April 2002.

[Orac03] Oracle. Oracle Warehouse Builder Product Page. Available at http://otn.oracle.com/products/warehouse/content.html

[Pres00] R.S.Pressman, Software Engineering: A Practitioner' s Approach, 5th Edition, McGraw-Hill, New York, 2000.

[RaDo00] E. Rahm, H. Hai 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. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB)*, pp. 381-390, Roma, Italy, (2001).

[Rund99] E. Rundensteiner (editor). Special Issue on Data Transformations. Bulletin of the Technical Committee on Data Engineering, Vol. 22, No. 1, March 1999.

[SaOr00] Wasim Sadiq, Maria E. Orlowska: On Business Process Model Transformations. 19th International Conference on Conceptual Modeling (ER 2000), pp: 267-280, Salt Lake City, Utah, USA, October 9-12, 2000.

[Sara00] S. Sarawagi. Special Issue on Data Cleaning. Bulletin of the Technical Committee on Data Engineering, 23(4), (2000).

[ScBJ00] E. Schafer, J.-D. Becker, M. Jarke. DB-Prism: Integrated Data Warehouses and Knowledge Networks for Bank Controlling. Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000.

[SiVa03] A. Simitsis, P. Vassiliadis. A Methodology for the Conceptual Modeling of ETL Processes. In the Proceedings of the Decision Systems Engineering (DSE '03), Velden, Austria, June 17, 2003.

[VaSk00] P. Vassiliadis, S. Skiadopoulos. Modeling 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.

[VaSS02] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Modeling ETL Activities as Graphs. In Proc. 4th Intl. Workshop on Design and Management of Data Warehouses (DMDW), pp. 52–61, Toronto, Canada, (2002).

[VaSS02a] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Conceptual Modeling for ETL Processes. In Proc. 5th ACM Intl. Workshop on Data Warehousing and OLAP (DOLAP), pp. 14–21, McLean, Virginia, USA (2002).

[VQVJ01] P. Vassiliadis, C. Quix, Y. Vassiliou, M. Jarke. Data Warehouse Process Management. Information Systems, vol. 26, no.3, pp. 205-236, June 2001.

[VSGT03] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis. A Framework for the Design of ETL Scenarios. In Proc. 15th Conference on Advanced Information Systems Engineering (CAiSE '03), pp. 520- 535, Klagenfurt/Velden, Austria, 16 - 20 June, 2003.

[WfMC98] Workflow Management Coalition. Interface 1: Process Definition Interchange Process Model. Document number WfMC TC-1016-P (1998). Available at www.wfmc.org

[Zani98] C. Zaniolo. LDL++ Tutorial. UCLA. http://pike.cs.ucla.edu/ldl/, Dec. 1998.

APPENDIX

```
DEFINE INPUT_SCHEMA as
        [i<arityOf(a_in1)]{A_IN1_$i$,}
        [i=arityOf(a_in1)] {A_IN1_$i$}

DEFINE OUTPUT_SCHEMA as
        [i<arityOf(a_out)]{A_OUT_$i$,}
        [i=arityOf(a_out)]{A_OUT_$i$}

DEFINE PARAMS as
        [i<arityOf(@PARAM)]{@PARAM[$i$],}
        [i=arityOf(@PARAM)]{@PARAM[$i$]}

DEFINE DEFAULT_MAPPING as
        [i<arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$,}
        [i=arityOf( a_out )] {A_OUT_$i$= A_IN1_$i$}
```

Selection

```
a_out(OUTPUT_SCHEMA)←
        a_in1(INPUT_SCHEMA),
        expr(PARAMS),
        DEFAULT_MAPPING.
```

Domain Mismatch

```
a_out(OUTPUT_SCHEMA)←
        a_in1(INPUT_SCHEMA),
        @FIELD >=@Xlow,
        @FIELD <= @Xhigh,
        DEFAULT_MAPPING.
```

Projection

```
DEFINE PROJECT_MAPPING as
        [i<arityOf(@PROJECTED_FIELDS)] {A_OUT_$i$= @PROJECTED_FIELDS[$i$],}
        [i=arityOf(@PROJECTED_FIELDS)] {A_OUT_$i$= @PROJECTED_FIELDS[$i$]}

a_out(OUTPUT_SCHEMA)←
        a_in1(INPUT_SCHEMA),
        PROJECT_MAPPING.
```

Function Application

```
a_out(OUTPUT_SCHEMA,@OUTFIELD)←
        a_in1(INPUT_SCHEMA),
        @FUNCTION (PARAMS, @FunOutFIELD ),
        @OUTFIELD=@FunOutFIELD,
        DEFAULT_MAPPING.
```

Surrogate Key Assignement

```
a_out(OUTPUT_SCHEMA,@SKEY)←
        a_in1(INPUT_SCHEMA),
        @LookUp( @CompKey,@CompSource,@SurKey ),
        @SourceKey = @CompKey,
        @Source = @CompSource,
        @SKEY = @SurKey,
        DEFAULT_MAPPING.
```

Add Attribute

```
a_out(OUTPUT_SCHEMA,@OUTFIELD)←
        a_in1(INPUT_SCHEMA),
        @OUTFIELD= @VALUE,
        DEFAULT_MAPPING.
```

Aggregation

```
DEFINE ALL_GROUPERS AS
        [i<arityOf(@GROUPERs)] {@GROUPERS[$i$],}
        [i=arityOf(@GROUPERS)]{@GROUPERS[$i$]}

a_out(ALL_GROUPERS, @AggrFunction<@Field>) <-
        a_in1(INPUT_SCHEMA).
```

Unique Value

```
DEFINE INPUT_AUX as
        [i<arityOf(a_in1)]{A_IN1B_$i$,}
        [i=arityOf(a_in1)] {A_IN1B_$i$}

DEFINE ATTRS as
        [i<arityOf(a_in1)]{ATTR$i$,}
        [i=arityOf(a_in1)] {ATTR$i$}

DEFINE ATTR_MAPPING as
        [i<arityOf(a_in1)] {A_OUT_$i$= ATTR$i$,}
        [i=arityOf(a_in1)] {A_OUT_$i$= ATTR$i$}

a_out(OUTPUT_SCHEMA)←
        a_in1(INPUT_SCHEMA),
        ~dublicates(INPUT_SCHEMA),
        DEFAULT_MAPPING.

dublicates(ATTRS)←
        a_in1(INPUT_SCHEMA),
        a_in1(INPUT_SCHEMA)~=a_in1(INPUT_AUX),
        A_IN1B_@FIELD_POS=A_IN1_@FIELD_POS,
        ATTR_MAPPING.
```

Primary Key Violation

```
a_out(OUTPUT_SCHEMA)←
       a_in1(INPUT_SCHEMA),
       ~dublicates(INPUT_SCHEMA),
       A_IN1_@FIELD_POS~='null',
       DEFAULT_MAPPING.

dublicates(ATTRS)←
       a_in1(INPUT_SCHEMA),
       a_in1(INPUT_SCHEMA)~=a_in1(INPUT_AUX),
       A_IN1B_@FIELD_POS=A_IN1_@FIELD_POS,
       ATTR_MAPPING.
```

Difference

```
DEFINE COMMON_MAPPING as
       [i<arityOf(@COMMON_IN1)] {@COMMON_IN1[$i$]= @COMMON_IN2[$i$],}
       [i=arityOf(@COMMON_IN1)] {@COMMON_IN1[$i$]= @COMMON_IN2[$i$]}

DEFINE INPUT_SCHEMA2 as
       [i<arityOf(a_in2)]{A_IN2_$i$,}
       [i=arityOf(a_in2)] {A_IN2_$i$}

a_out(OUTPUT_SCHEMA)←
       a_in1(INPUT_SCHEMA),
       a_in2(INPUT_SCHEMA2),
       ~semijoin(INPUT_SCHEMA),
       DEFAULT_MAPPING.

semijoin(INPUT_SCHEMA)←
       a_in1(INPUT_SCHEMA),
       a_in2(INPUT_SCHEMA2),
       COMMON_MAPPING.
```

Foreign Key

```
a_out(OUTPUT_SCHEMA)←
       a_in1(INPUT_SCHEMA),
       @TARGET_TABLE([i<@TARGET_FIELD_POS]{_,}
            [i=@TARGET_FIELD_POS]{@TARGET_FIELD}
            [@TARGET_FIELD_POS<i<@TARGET_TABLE_ARITY+1]{,_}),
       @TARGET_FIELD=@FIELD,
       DEFAULT_MAPPING.
```

Normalization

```
[i<arityOf(@REP_FIELDS)+1]{
        a_out([j<arityOf(@COMMON_FIELDS)+1]{@COMMON_FIELDS[$j$],}CODE,VALUE) <-
             a_in1([j<arityOf(a_in1)] {A_IN1_$j$,} [j=arityOf( a_in1 )] {A_IN1_$j$}),
             @lookup_code(@FIELD_NAMES[$i$],CODE),
             VALUE=@REP_FIELDS[$i$]
        .

}
```

Denormalization

```
a_out(OUTPUT_SCHEMA)←
        [i<arityOf(@REP_FIELDS_NAME)]{
             a_in1_([j<@COMMON_FIELDS_NUM] {a_in1_$j$,}, CODE$i$, VALUE$i$),
             @lcode(@REP_FIELDS_NAME[$i$],CODE$i$),
             @OUT_FIELD_NAME[$i$]=VALUE$i$,

}

[i=arityOf(@REP_FIELDS_NAME)]{
        a_in1_([j<@COMMON_FIELDS_NUM] {a_in1_$j$,}, CODE$i$, VALUE$i$),
             @lcode(@REP_FIELDS_NAME[$i$],CODE$i$),
             @OUT_FIELD_NAME[$i$]=VALUE$i$

}.
```