Graph-Based Modeling of ETL Activities with Multi-level Transformations and Updates

Alkis Simitsis[1], Panos Vassiliadis[2], Manolis Terrovitis[1], and Spiros Skiadopoulos[1]

[1] National Technical University of Athens,
Dept. of Electrical and Computer Eng., Athens, Hellas
{asimi, mter, spiros}@dbnet.ece.ntua.gr
[2] University of Ioannina, Dept. of Computer Science, Ioannina, Hellas
pvassil@cs.uoi.gr

Abstract. Extract-Transform-Load (ETL) workflows are data centric workflows responsible for transferring, cleaning, and loading data from their respective sources to the warehouse. In this paper, we build upon existing graph-based modeling techniques that treat ETL workflows as graphs by (a) extending the activity semantics to incorporate negation, aggregation and self-joins, (b) complementing querying semantics with insertions, deletions and updates, and (c) transforming the graph to allow zoom-in/out at multiple levels of abstraction (i.e., passing from the detailed description of the graph at the attribute level to more compact variants involving programs, relations and queries and vice-versa).

1 Introduction

Conceptual and logical modeling of the design of data warehouse back-stage activities has been a relatively novel issue in the research community [3, 5, 6, 7]. The data warehouse back-stage activities are mainly implemented through tools, known as Extraction-Transformation-Loading (ETL) tools that employ data centric workflows to extract data from the sources, clean them from logical or syntactical inconsistencies, transform them into the format of the data warehouse, and eventually load these data into the warehouse.

The main issues concerning the modeling of these activities have to do (a) with the semantics of the involved activities and (b) with the exploitation of the deduced model to obtain a better understanding and a clearer evaluation of the quality of the produced design for a data warehouse scenario.

Several works in the area [2, 4] present systems tailored for ETL tasks (see also [9] for a broader discussion); nevertheless, the main focus of these works is on achieving functionality, rather than on modeling the internals or dealing with the software design or maintenance of these tasks. [3, 5] employ UML as a conceptual modeling language, whereas [7] introduces a generic graphical notation for the same task. Being defined at the conceptual level, these efforts lack a full model of the semantics of ETL workflows and a mechanism to allow the designer to navigate efficiently through large scale designs without being overwhelmed by their inherent complexity.

A Min Tjoa and J. Trujillo (Eds.): DaWaK 2005, LNCS 3589, pp. 43 – 52, 2005.

In our previous research, we have presented a first attempt towards a graph-based model for the definition of the ETL scenarios [6]. The model of [6, 8] treats an ETL scenario as a graph, which we call the *Architecture Graph*. Activities and data stores are modeled as the nodes of the graph; the attributes that constitute them are modeled as nodes too. Activities have input and output schemata and *provider relationships* relate inputs and outputs between data providers and data consumers. In this paper, we extend previous work in several ways. First, we complement the existing graph-based modeling of ETL activities by adding graph constructs to capture the semantics of insertions, deletions and updates. Second, we extend the previous results by adding negation, aggregation and self-joins in the expressive power of our graph-based approach. More importantly, we introduce a systematic way of transforming the Architecture Graph to allow zooming in and out at multiple levels of abstraction (i.e., passing from the detailed description of the graph at the attribute level to more compact variants involving programs, relations and queries and vice-versa). The visualization of the Architecture graph at multiple levels of granularity allows the easier understanding of the overall structure of the involved scenario, especially as the scale of the scenarios grows.

This paper is organized as follows. In Section 2, we discuss extensions to the graph model for ETL activities. Section 3 introduces a principled approach for zooming in and out the graph. In Section 4, we conclude our results and provide insights for future work.

2 Modeling of Side-Effects and Special Cases for ETL Activities

The purpose of this section is to present a formal logical model for the activities of an ETL environment and the extensions to existing work that we make. First, we start with the background constructs of the model, already introduced in [6, 8] and then, we move on to extend this modeling with update semantics, negations, aggregation and self-joins. We employ LDL++ [1, 10] in order to describe the semantics of an ETL scenario in a declarative nature and understandable way. LDL++ is a logic-programming, declarative language that supports recursion, complex objects and negation. Moreover, LDL++ supports external functions, choice, (user-defined) aggregation and updates.

2.1 Preliminaries

In this subsection, we introduce the formal model of data types, data stores and functions, before proceeding to the model of ETL activities. To this end, we reuse the modeling constructs of [6, 8] upon which we subsequently proceed to build our contribution. The basic components of this modeling framework are:

- *Data types*. Each data type `T` is characterized by a name and a domain, i.e., a countable set of values. The values of the domains are also referred to as *constants*.
- *Attributes*. Attributes are characterized by their name and data type. For single-valued attributes, the domain of an attribute is a subset of the domain of its data

type, whereas for set-valued, their domain is a subset of the powerset of the domain of their data type $2^{\mathrm{dom(T)}}$.

- A *Schema* is a finite list of attributes. Each entity that is characterized by one or more schemata will be called *Structured Entity*.
- *Records & 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. 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). In the rest of this paper, we will mainly deal with the two most popular types of recordsets, namely *relational tables* and *record files*.
- *Functions*. A *Function Type* comprises a name, a finite list of *parameter data types*, and a single *return data type*.
- *Elementary Activities*. In the framework of [8], activities are logical abstractions representing parts, or full modules of code. An *Elementary Activity* (simply referred to as *Activity* from now on) is formally described by the following elements:
  - *Name*: a unique identifier for the activity.
  - *Input Schemata*: a finite list of one or more input schemata that receive data from the data providers of the activity.
  - *Output Schemata*: a finite list of one or more output schemata that describe the placeholders for the rows that pass the checks and transformations performed by the elementary activity.
  - *Operational Semantics*: a program, in LDL++, describing the content passing from the input schemata towards the output schemata. For example, the operational semantics can describe the content that the activity reads from a data provider through an input schema, the operation performed on these rows before they arrive to an output schema and an implicit mapping between the attributes of the input schema(ta) and the respective attributes of the output schema(ta).
  - *Execution priority*. In the context of a scenario, an activity instance must have a priority of execution, determining when the activity will be initiated.
- *Provider* relationships. These are `1:N` relationships that involve attributes with a provider-consumer relationship. The flow of data from the data sources towards the data warehouse is performed through the composition of activities in a larger scenario. In this context, the input for an activity can be either a persistent data store, or another activity. Provider relationships capture the mapping between the attributes of the schemata of the involved entities. Note that a consumer attribute can also be populated by a constant, in certain cases.
- *Part_of* relationships. These relationships involve attributes and parameters and relate them to their respective activity, recordset or function to which they belong.

The previous constructs, can be complemented by incorporating the semantics of ETL workflow in our framework. Due to the lack of space, we do not elaborate in detail on the full mechanism of the mapping of LDL rules to the Architecture Graph (including details on intra-activity and inter-activity programs); we refer the interested reader to [9] for this task. Instead, in this paper, we focus on the parts concerning side-effect programs (which are most common in ETL environments), along with the modeling of aggregation and negation. To this end, we first need to introduce *programs* as another modeling construct.

– *Programs.* We assume that the semantics of each activity is given by a declarative program expressed in LDL++. Each program is a finite list of LDL++ rules. Each rule is identified by an (internal) rule identifier. We assume a normal form for the LDL++ rules that we employ. In our setting, there are three types of programs, and normal forms, respectively:
  (i) *intra-activity* programs that characterize the internals of activities (e.g., a program that declares that the activity reads data from the input schema, checks for NULL values and populates the output schema only with records having non-NULL values)
  (ii) *inter-activity* programs that link the input/output of an activity to a data provider/consumer
  (iii) *side-effect* programs that characterize whether the provision of data is an insert, update, or delete action.

We assume that each activity is defined in isolation. In other words, the inter-activity program for each activity is a stand-alone program, assuming the input schemata of the activity as its EDB predicates. Then, activities are plugged in the overall scenario that consists of inter-activity and side-effect rules and an overall *scenario program* can be obtained from this combination.

Side-effect programs. We employ side-effect rules to capture database updates. We will use the generic term *database updates* to refer to insertions, deletions and updates of the database content (in the regular relational sense). In LDL++, there is an easy way to define database updates. An update expression is of the form

```
head <- query part, update part
```

and has the following semantics: (a) we make a query to the database and specify the tuples that abide by the query part and (b) we update the predicate of the update part as specified in the rule.

```
raise1(Name, Sal, NewSal) <-
        employee(Name, Sal), Sal = 1100,              (a)
        NewSal = Sal * 1.1,                           (b)
        - employee(Name, Sal),                        (c)
        + employee(Name, NewSal).                     (d)
```

Fig. 1. Exemplary LDL++ rule for side-effect updates

For example, consider the rule depicted in Fig. 1. In Line (a) of the rule, we mark the employee tuples with salary equal to 1100 in the relation `employee(Name,Sal)`. For each the above marked tuples, Line (b) computes an updated salary with a 10% raise through the variable `NewSal`. In Line (c), we delete the originally marked tuples from the relation. Finally, Line (d) inserts the updated tuples, containing the new salaries in the relation. In LDL updates, the order of the individual atoms is important and the query part should always advance the update part, to avoid having undesired effects from a predicate failing after an update (more details for the syntax of LDL can be found in [10]).

2.2 Mapping Side-Effect Programs to the Architecture Graph of a Scenario

Concerning our modeling effort, the main part of our approach lies in mapping declarative rules, expressing the semantics of activities in LDL, to a graph, which we call the Architecture Graph. In our previous work, the focus of [8] is on the input-output role of the activities instead of their internal operation. It is quite straightforward to complement this modeling with the graph of intra- and inter-activity rules [9]. In principle, activities comprise input and output schemata. Intra-activity programs and their variables facilitate the mapping of inputs to outputs. All attributes, activities and relations are nodes of the graph, connected through the proper part-of relationships. Each LDL rule connecting inputs (body of the rule) to outputs (head of the rule) is practically mapped to a set of provider edges, connecting inputs to outputs. Special purpose regulatory edges, capturing filters or joins are also part of the graph.

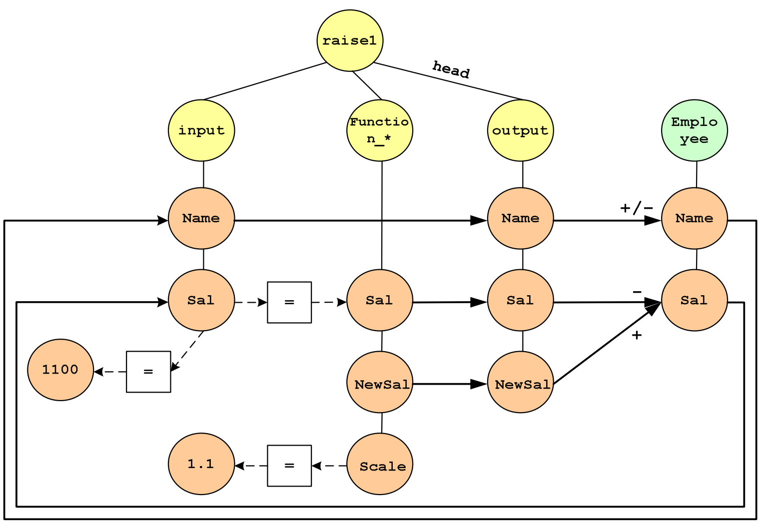

Fig. 2. Side-effects over the LDL++ rule of Fig. 1

While intra- and inter-activity rules are straightforwardly mapped to graph-based constructs, side-effects involve a rather complicated modeling, since there are both values to be inserted or deleted along with the rest of the values of a recordset. Still, there is a principled way to map LDL side-effects to the Architecture Graph.

1. A side-effect rule is treated as an activity, with the corresponding node. The output schema of the activity is derived from the structure of the predicate of the head of the rule.
2. For every predicate with a + or – in the body of the rule, a respective provider edge from the output schema of the side-effect activity is assumed. A basic syntactic restriction here is that the updated values appear in the output schema. All provider relations from the output schema to the recordset are tagged with a + or –.
3. For every predicate that appears in the rule without a + or – tag, we assume the respective input schema. Provider edges from this predicate towards these schemata are added as usual. The same applies for the attributes of the input and output schemata of the side effect activity. An obvious syntactic restriction is that all predicates appearing in the body of the rule involve recordsets or activity schemata (and not some intermediate rule).

Notice that it is permitted to have cycles in the graph, due to the existence of a recordset in the body of a rule both tagged and untagged (i.e., both with its old and new values). The old values are mapped to the input schema and the new to the output schema of the side-effect activity.

In Fig. 2, we depict an example for the usage of side-effects over the LDL++ rule of Fig. 1. Observe that `Name` is tagged both as `+` or `–`, due to its presence at two predicates, one removing the old value of `Sal` and another inserting `NewSal`, respectively. Observe, also, how the input is derived from the predicate `employee` at the body of the rule.

2.3 Special Cases for the Modeling of the Graph

In this subsection, we extend our basic modeling to cover special cases such as aliases, negation, aggregation and functions.

Alias relationships. An alias relationship is introduced whenever the same predicate appears in the same rule (e.g., in the case of a self-join). All the nodes representing these occurrences of the same predicate are connected through alias relationships to denote their semantic interrelationship. Note that due to the fact that intra-activity programs do not directly interact with external recordsets or activities, this practically involves the rare case of internal intermediate rules.

Negation. When a predicates appears negated in a rule body, then the respective part-of edge between the rule and the literal's node is tagged with '¬'. Note that negated predicates can appear only in the rule body.

Aggregation. Another interesting feature is the possibility of employing aggregation. In LDL, aggregation can be coded in two steps: (a) grouping of values to a bag and (b) application of an aggregate function over the values of the bag. Observe the example of Fig. 3, where data from the table `DW.PARTSUPP` are summarized, through activity $\texttt{Aggregate}_1$ to provide the minimum daily cost in view `V1`. In Fig. 3 we list the LDL program for this activity. Rules (`R16-R18`) explain how the data of table `DW.PARTSUPP` are aggregated to produce the minimum cost per supplier and day.

Observe how LDL models aggregation in rule R17. Then, rule R19 populates view V1 as an inter-activity program.

The graph of an LDL rule is created as usual with only 3 differences:

1. Relations which create a set from the values of a field employ a pair of regulator edges through an intermediate node ‘<>’.
2. Provider relations for attributes used as groupers are tagged with ‘g’.
3. One of the attributes of the aggr function node consumes data from a constant that indicates which aggregate function should be used (e.g., avg, min, max).

```
R16: aggregate1.a_in(skey,suppkey,date,qty,cost)<-
        dw.partsupp(skey,suppkey,date,qty,cost)
R17: temp(skey,day,<cost>) <-
        aggregate1.a_in(skey,suppkey,date,qty,cost).
R18: aggregate1.a_out(skey,day,min_cost) <-
        temp(skey,day,all_costs),
        aggr(min,all_costs,min_cost).
R19: v1(skey,day,min_cost) <-
        aggregate1.a_out(skey,day,min_cost).
```

Fig. 3. LDL Specification for an activity involving aggregation

Functions. Functions are treated as any other predicate in LDL, thus they appear as common nodes in the architecture graph. Nevertheless, there are certain special requirements for functions:

1. The function involves a list of parameters, the last of which is the return value of the function.
2. All function parameters referenced in the body of the rule either as homonyms with attributes, of other predicates or through equalities with such attributes, are linked through equality regulator relationships with these attributes.
3. The return value is possibly connected to the output through a provider relationship (or with some other predicate of the body, through a regulator relationship).

For example, observe Fig. 2 where a function involving the multiplication of attribute Sal with a constant is involved. Observe the part-of relationship of the function with its parameters and the regulator relationship with the first parameter and its populating attribute. The return value is linked to the output through a provider relationship.

3 Different Levels of Detail of the Architecture Graph

The Architecture Graph can become a complicated construct, involving the full detail of activities, recordsets, attributes and their interrelationships. Although it is important and necessary to track down this information at design time, in order to formally specify the scenario, it quite clear that this information overload might be cumbersome to manage at later stages of the workflow lifecycle. In other words, we need to provide the user with different versions of the scenario, each at a different level of detail.

We will frequently refer to these abstraction levels of detail simply, as *levels*. We have already defined the Architecture Graph at the *attribute level*. The attribute level is the most detailed level of abstraction of our framework. Yet, coarser levels of detail can also be defined. The *schema level*, abstracts the complexities of attribute interrelationships and presents only how the input and output schemata of activities interplay in the data flow of a scenario. In fact, due to the composite structure of the programs that characterize an activity, there are more than one variants that we can employ for this description. Finally, the coarser level of detail, the *activity level*, involves only activities and recordsets. In this case, the data flow is described only in terms of these entities.

Architecture Graph at the Schema Level. Let $G_S(V_S, E_S)$ be the architecture graph of an ETL scenario at the schema level. The scenario at the schema level has schemata, functions, recordsets and activities for nodes. The edges of the graph are part-of relationships among structured entities and their corresponding schemata and provider relationships among schemata. The direction of provider edges is again from the provider towards the consumer and the direction of the part-of edges is from the container entity towards its components (in this case just the involved schemata). Edges are tagged appropriately according to their type (part-of or provider).

Intuitively, at the schema level, instead of fully stating which attribute populates another attribute, we trace only how this is performed through the appropriate schemata of the activities. A program capturing the semantics of the transformations and cleanings that take place in the activity is the means through which the input and output schemata are interconnected. If we wish, instead of including all the schemata of the activity as they are determined by the intermediate rules of the activity's program, we can present only the program as a single node of the graph, to avoid the extra complexity.

There is a straightforward way to zoom out the Architecture Graph at the attribute level and derive its variant at the schema level. For each node x of the architecture graph $G(V, E)$ representing a schema:

1. for each provider edge (x_a, y) or (y, x_a), involving an attribute of x and an entity y, external to x, introduce the respective provider edge between x and y (unless it already exists, of course);
2. remove the provider edges (x_a, y) and (y, x_a) of the previous step;
3. remove the nodes of the attributes of x and the respective part-of edges.

We can iterate this simple algorithm over the different levels of part-of relationships, as depicted in Fig. 4.

Architecture Graph at the Activity Level. In this paragraph, we will deal with the model of ETL scenarios as graphs at the activity level. Only activities and recordsets are part of a scenario at this level. Let $G_A(V_A, E_A)$ be the architecture graph of an ETL scenario at the activity level. The scenario at the activity level has only recordsets and activities for nodes and a set of provider relationships among them for edges. The provider relationships are directed edges from the provider towards the consumer entity.

Intuitively, a scenario is a set of activities, deployed along a graph in an execution sequence that can be linearly serialized through topological ordering. There is a straightforward way to zoom out the Architecture Graph at the schema level and derive its variant at the activity level. For each node x of the architecture graph $G_A(V_A,E_A)$ representing a structured entity (i.e., activity or recordset):

1. for each provider edge (x_c,y) or (y,x_c), involving a schema of x and an entity y, external to x, introduce the respective provider edge between x and y (unless it already exists, of course);
2. remove the provider edges (x_c,y) and (y,x_c) of the previous step;
3. remove the nodes of the schema(ta) and program (if x is an activity) of x and the respective part-of edges.

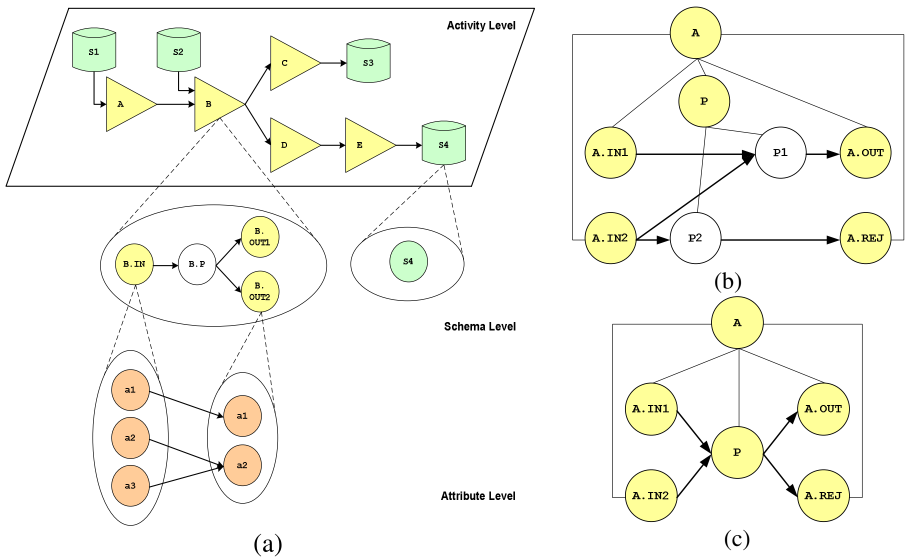

Fig. 4. Zooming in/out. (a) different levels of detail for ETL workflows; (b) an activity with two input schemata populating an output and a rejection schema as follows: a subprogram P1 is assigned the population of the output schema only and a subprogram P2 populates only the rejection schema using only one input schema; and (c) a single node abstracts the internal structure of the activity.

Discussion. Navigating through different levels of detail is a facility that primarily aims to make the life of the designer and the administrator easier throughout the full range of the lifecycle of the data warehouse. Through this mechanism, the designer can both avoid the complicated nature of parts that are not of interest at the time of the inspection and drill-down to the lowest level of detail for the parts of the design that he is interested in.

Moreover, apart from this simple observation, we can easily show how our graph-based modeling provides the fundamental platform for employing software engineering techniques for the measurement of the quality of the produced design [9]. Zooming in and out the graph in a principled way allows the evaluation of the overall design both at different depth of granularity and at any desired breadth of range (i.e., by isolating only the parts of the design that are currently of interest).

4 Conclusions

Previous research in the logical modeling of ETL workflows has identified graph-based techniques that capture the high-level structure of these workflows. In this paper, we have extended the semantics of the involved ETL activities to incorporate negation, aggregation and self-joins. Moreover, we have complemented this semantics in order to handle insertions, deletions and updates. Finally, we have provided a principled method for transforming the architecture graph of an ETL scenario to allow zoom-in/out at multiple levels of abstraction. This way, we can move from the detailed description of the graph at the attribute level to more compact variants involving programs, relations and queries and vice-versa.

Research can be continued in more than one direction, e.g., towards the derivation of precise algorithms for the evaluation of the impact of changes in the Architecture Graph. Finally, a field-study of the usage of the Architecture Graph in all the phases of a data warehouse project can also be pursued.

Acknowledgments

This work is co-funded by the European Social Fund (75%) and National Resources (25%) - Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS.

References

[1] S. Ceri, G. Gottlob, L. Tanca. Logic Programming and Databases. Springer-Verlag, 1990.

[2] H. Galhardas, D. Florescu, D. Shasha and E. Simon. Ajax: An Extensible Data Cleaning Tool. In Proc. of ACM SIGMOD'00, pp. 590, Dallas, Texas, 2000.

[3] S. Lujan-Mora, P. Vassiliadis, J. Trujillo. Data Mapping Diagrams for Data Warehouse Design with UML. In Proc. 23rd International Conference on Conceptual Modeling (ER 2004), pp. 191-204, Shanghai, China, 2004.

[4] V. Raman, J. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In Proc. of VLDB'01, pp. 381-390, Roma, Italy, 2001.

[5] J. Trujillo, S. Luján-Mora: A UML Based Approach for Modeling ETL Processes in Data Warehouses. In Proc. of ER'03, pp. 307-320, Chicago, USA, 2003.

[6] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Modeling ETL Activities as Graphs. In Proc. of DMDW'02, pp. 52–61, Toronto, Canada, 2002.

[7] P. Vassiliadis, A. Simitsis, S. Skiadopoulos. Conceptual Modeling for ETL Processes. In Proc. of DOLAP'02, pp. 14–21, McLean, Virginia, USA, 2002.

[8] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis. A Framework for the Design of ETL Scenarios. In Proc. of CAiSE'03, pp. 520-535, Austria, 2003.

[9] P. Vassiliadis, A. Simitsis, M. Terrovitis, S. Skiadopoulos. Blueprints for ETL workflows (long version). Available throughhttp://www.cs.uoi.gr/~pvassil/publications/2005_ER_AG/ETL_blueprints_long.pdf

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