Blueprints for ETL workflows

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

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

[2] National Technical University of Athens,
Dept. of Electrical and Computer Eng.,
Athens, Hellas
{asimi,mter,spiros}@dbnet.ece.ntua.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. Previous research has identified graph-based techniques that construct the *blueprints* for the structure of such workflows. In this paper, we extend existing results by explicitly incorporating the internal semantics of each activity in the workflow graph. Apart from the value that blueprints have per se, we exploit our modeling to introduce rigorous techniques for the measurement of ETL workflows. To this end, we build upon an existing formal framework for software quality metrics and formally prove how our quality measures fit within this framework.

1. Introduction

All engineering disciplines employ blueprints during the design of their engineering artifacts. Modeling in this fashion is not a task with a value by itself; as [BoRJ99] mentions "we build models to communicate the desired structure and behavior of our system … to visualize and control the system's architecture …to better understand the system we are building … to manage risk".

In this paper, we discuss the constructing entities and the usage of blueprints for a particular category of database-centric software, namely, the Extract-Transform-Load (ETL) workflows. ETL workflows are an integral part of the back-stage of data warehouse architectures, where data are (a) collected from the operational sources, (b) cleansed (to remove any noise or inconsistencies), (c) transformed (so that they syntactically comply with the schema of the warehouse tables) and finally, (d) loaded to the target warehouse tables. Out of the aforementioned benefits of modeling, control of the system's architecture and risk management are of particular importance. For example, we would like to answer questions like:

- Which attributes/tables are involved in the population of a certain attribute?
- What part of the scenario is affected if we delete an attribute?
- How good is the design of my ETL scenario? Is variant `A` or variant `B` better?

Previous research has provided some results towards the aforementioned tasks. The work of [TrLu03, VaSS02a] provides conceptual modeling techniques for ETL. [VaSS02] presents a first attempt towards a graph-based model for the definition of the ETL scenarios. The model of [VaSS02] treats ETL scenarios as graphs. 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. Nevertheless, what is missing from previous efforts is a full model of the semantics of ETL workflows and a rigorous framework for the measurement of our design artifacts.

In this paper, we significantly extend previous works to capture the internals of the workflow activities in sufficient detail. We make use of a logical abstraction of ETL activity semantics in the form of LDL++ [Zani98] programs. The approach is not unrealistic: in fact, in [VSGT03] the authors discuss the possibility of providing extensible libraries of ETL tasks, logically described in LDL. On the basis of this result, it is reasonable to assume the reusability of these libraries. In this paper, we extend the graph of [VSGT03] by incorporating the internals of the activity semantics to the graph. To this end, we provide a principled way of transforming LDL programs to graphs, in a way that gracefully complements the model of [VaSS02,VSGT03]. The resulting graph, which is called *Architecture Graph* can provide sufficient answers to what-if and dependency analysis in the process of understanding or managing the risk of the environment.

Moreover, another question can also be answered: "How good is my design?". The community of software engineering has provided numerous metrics towards evaluating the quality of software designs [Dumk02]. Are these metrics sufficient? In this paper, we build upon the fundamental contribution of [BrMB96] that develop a rigorous and systematic framework that classifies usually encountered metrics into five families, each with its own characteristics. These five families are *size*, *length*, *complexity*, *cohesion* and *coupling* of software artifacts. In this paper, we develop specific measures for the Architecture Graph and formally prove their fitness for the rigorous framework of [BrMB96].

In a nutshell, our contributions can be listed as follows:

- an extension of [VaSS02] to incorporate internal semantics of activities in the architecture graph;
- a principled way of transforming LDL programs to the graph, so that the latter can be explored both at the granular (i.e., attribute) level of detail and at different levels of abstraction;
- a systematic definition of software measures for the Architecture Graph, based on the rigorous framework of [BrMB96].

This paper is organized as follows. In Section 2, we present the graph model for ETL activities. Section 3 discusses measures for the introduced model. In Section 4, we present related work. Finally, in Section 5 we conclude our results and provide insights for future work. The reader is encouraged to refer to the long version of this paper [VSTS05] for all the proofs and several technical issues, omitted from this paper for lack of space.

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. We start with the background constructs of the model, already introduced in [VaSS02,VSGT03]. Then, we move on to extend this modeling with formal semantics of the internals of the activities.

In order to formally define the semantics of ETL workflow, we can use any 3GL/4GL programming language (C++, PL/SQL etc.). We do not consider the actual implementation of the workflow in some programming language, but rather, we employ LDL++ [Zani98] in order to describe its semantics 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 modeling constructs of [VaSS02,VSGT03] upon which we will subsequently build our contribution. In brief, 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 $2^{\mathtt{dom(T)}}$.
– 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 [VSGT03] framework, 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.

Based upon the previous constructs, already available from [VSGT03], we proceed with their extension towards fully incorporating the semantics of ETL workflow in our framework. To this end we 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 operational semantics, i.e., 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. Due to lack of space, we discuss side-effect rules in detail in the long version of this paper [VSTS05].
- *Regulator Relationships.* A regulator relationship in a safe rule is an equality/inequality relationship between two terms, i.e., of the form $term_1$ θ $term_2$, such that neither of them appears in the head of a rule. In terms of the architecture graph, the regulator relationship is represented (a) by a node for each of the terms, (b) by a node representing the condition θ, and (c) by two edges among the node of the condition and the nodes of the term. The direction of the edges follows the way the expression is written in LDL (i.e., from the left to right). Regulator relationships are used in order to capture the selection conditions and joins that take place in an LDL program.

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.

Intra-activity programs. The intra-activity programs abide by the following rules:

1. All input schemata are EDB predicates.
2. All output schemata appear only as IDB predicates. Furthermore, output schemata are the only IDB predicates that appear in such a program.
3. Intermediate rules are possibly employed to help with intermediate results.
4. We assume non-recursive admissible programs. The safety of the program is guarantied by the requirement for *admissibility*, which is a generalization of stratifiability [CeGT90]. An admissible program does not contain any self-referential *set definitions* or any predicates defined in terms of their own *negations*.

Inter-activity programs. The inter-activity programs are very simple. There is exactly one rule per provider relationship, with the consumer in the head and the provider in the body. The consumer attributes are mapped to their corresponding providers either through the synonym mechanism or through explicit equalities. No other atoms or predicates are allowed in the body of an inter-activity program; all the consumer attributes should be populated from the provider.

Consumer_input(a_1,…,a_n) <- provider_output(a_1,…,a_m), $m \geq n$

2.2 Incorporating activity semantics in the Architecture Graph

The focus of [VaSS02, VSGT03] is on the input-output role of the activities instead of their internal operation. In this section, we extend the model of those works by translating the formal semantics of the internals of the activities to graph constructs, as part of the overall Architecture Graph. We organize this discussion as follows: first, we consider how individual rules are represented by graphs for intra-activity and inter-activity programs. The interested author can find a discussion about side-effect programs in the long version of this paper [VSTS05]. Then, we discuss how the programs of activities are constructed from the composition of different rules and finally, we discuss how a scenario program can be obtained from the composition of the graph representations of individual programs.

Intuitively, instead of simply stating which schema populates another schema, we trace how this is done through the internals of an activity. The programs that facilitate the input to output mappings take part in the graph, too. Any filters, joins or aggregations are part of the graph as first-class citizens. There is a straightforward way to determine the architecture graph with respect to the LDL program that defines the ETL scenario.

Intra-activity rules. Given the program of the activity as a stand-alone LDL++ program, we introduce the following constructs, by default:

- A node for the activity per se.
- A node for each of the schemata of the activity and a node for the activity program. Part-of edges connect the activity with these components.
- A node for each rule, connected through a part-of relationship to the program node of the activity.

If we treat each rule as a stand-alone program, we can construct its graph as follows:

- We introduce a node for each predicate of the rule. These nodes are connected to the rule node through a part-of relationship. The edge of the head predicate is tagged as '`head`' and the edges of the negated literals of the body are tagged as '¬'. Functions are treated as predicates. A different predicate node is introduced for each instance of the same predicate (e.g., in the case of a self-join). Such nodes are connected to each other through *alias* edges. In the long version [VSTS05], we detail the parts of the last cases that require extra attention.
- We introduce a node for each variable of a predicate. Part-of relationships connect these nodes with their corresponding predicates.
- For each condition of the form *Input attribute* = *Output attribute* (or its equivalent presence of *synonyms* in the output and input schemata), we add a provider edge. Here, we assume as input (output) attributes, attributes belonging to predicates of the rule body (head). A provider relationship is thus, an edge from the body towards the head of the rule.
- For relationships among input attributes (practically, involving functions and built-ins), a regulator edge is introduced.

```
R06: sk.a_in1(pkey,suppkey,date,qty,cost)<-
        dsa_ps(pkey,suppkey,date,qty,cost).

R07: sk.a_in2(pkey,source,skey)<-
       lookUp(l_pkey,source,l_skey),pkey=l_pkey,skey=l_skey,
       source=1.
R08: sk.a_out(pkey,suppkey,date,qty,cost,skey)<-
       add_sk1.a_in1(pkey,date,qty,cost),
       add_sk1.a_in2(pkey,source,l_skey).

R09: dollar2euro.a_in(skey,suppkey,date,qty,cost)<-
       sk.a_out(pkey,suppkey,date,qty,cost,skey).
```

Fig. 1 LDL++ for a small part of a scenario

Inter-activity rules. For each recordset of a scenario, we assume a node representing its schema. For simplicity, we do not discriminate a recordset from its schema using different nodes. For each intra-activity rule (between input-output schemata of different activities and/or recordsets) there is a simple way to construct its corresponding graph: we introduce a provider edge from the input towards the output attributes.

Observe Fig. 1. Activity `SK` (Surrogate Key assignment) takes as input the data from a recordset `DSA_PS(`<u>`PKey`</u>`,`<u>`SuppKey`</u>`,`<u>`Date`</u>`,Qty,Cost)`, and obtains a globally unique surrogate key `SKey` for the production key `PKey`, through a lookup table `LookUp(`<u>`PKey`</u>, <u>`Source`</u>, `SKey)`; in this example, we consider that data originate from source `1`. Then, the transformed data are propagated to another activity `dollar2euro` that converts the dollar values of attribute `cost`, to Euros (only the input schema of this activity is depicted). The rules `R06`, `R07` and `R09` are inter-activity rules: they describe how the input schemata of the activities are populated from their providers. Activities and recordsets can both play the role of provider, as one can see. Rule `R08` is an intra-activity rule. Fig. 2 depicts the architecture graph for this example. The grey area concerns the intra-activity program; the rest concern the inter-activity program rules. Solid arrows depict provider relationships, dotted arrows depict regulator relationships and part-of relationships are depicted with simple lines.

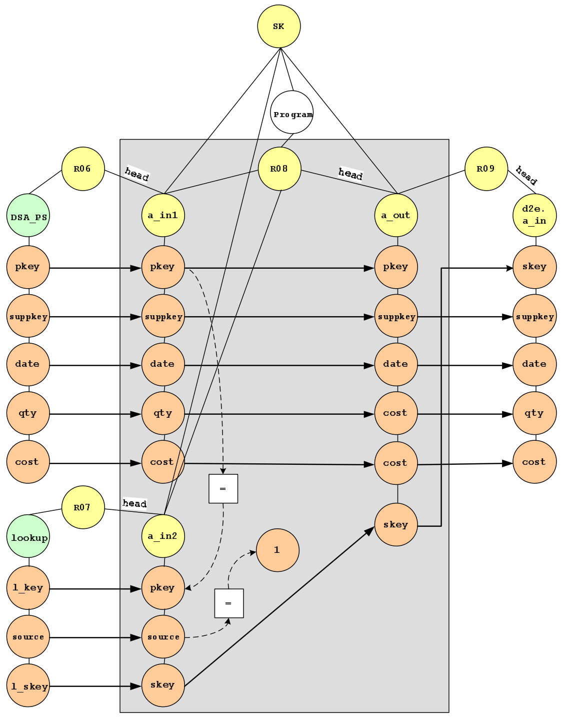

Fig. 2 Architecture graph for the example of Fig. 1

Deriving the graph of an activity program from the graphs of its rules. Combining the graphs of the rules of an activity is straightforward. Recall that so far, we have created the graph for each rule, considering each rule in isolation. Then, the graph for the overall activity is as follows:

- The nodes of the new graph comprise all the nodes of the rule graphs. If the same predicate appears to more than one rule, we merge all its corresponding nodes (i.e., the predicate node and all its variables). In the case where more than one instance of the same predicate exists in one rule, we randomly select one of these occurrences to be merged with the nodes of other rules.
- The edges of the new graph are all the edges, of the individual rules, after the merging takes place.
- All edges are tagged with the rule identifier of the rule they belong to. Through part of relationships and edge tagging, we can reconstruct the graphs of the individual rules, if necessary.

Deriving the graph of a scenario program from the graphs of its components. The construction of the graph for the scenario program is simple.

- First, we introduce all inter-activity and side-effect rules. We merge all multiple instances of the same recordset and its attributes. The same applies with the input and output schemata of an activity. We annotate all edges with the rule identifier of their corresponding rule.
- Then, intra-activity graphs are introduced too. Activity input and output schemata are merged with the nodes that already represent them in the combined intra-activity/side-effect graph. The same applies to activity nodes, too. No other action needs to be taken, since intra-activity programs are connected to the rest of the workflow only through their input and output schemata.

Once again, the reader is encouraged to refer to the long version of this paper [VSTS05], where we handle several issues omitted here due to lack of space. Specifically, these issues involve updates, aggregation, negation, functions and self-join queries. Moreover, the possibility of zooming in/out the Architecture Graph is also provided in [VSTS05]. The latter is a most useful interactive facility, necessary for avoiding the information overload due to the potentially high volume of detailed information at the attribute level, as described in this section.

3. Measuring the Architecture Graph: a principled approach

One of the main roles of blueprints is their usage as testbeds for the evaluation of the design of an engineer. In other words, blueprints serve as the modeling tool that provides answers the questions "*How good is my design?*" or "*Between these two designs, which one is better?*". In other words, one can define metrics or, more generally, measurement tests, to evaluate the quality of a design. In this section, we will address this issue, for our ETL workflows, in a principled manner.

There is a huge amount of literature devoted in the evaluation of software artifacts. Fenton proves that it is impossible to derive a unique measure of software quality [Fent94]. Rather, measurement theory should be employed in order to define meaningful measures of particular software attributes. A couple of years later, Briand et al., employ measurement theory to provide a set of five generic categories of measures for software artifacts [BrMB96]:

– *Size*, referring to the number of entities that constitute the software artifact.
– *Length*, referring to the longest path of relationships among these entities.
– *Complexity*, referring to the amount of inter-relationships of a component.
– *Cohesion*, measuring the extent to which each module performs exactly one job, by evaluating how closely related are its components.
– *Coupling*, capturing the amount of interrelationships between the different modules of a system.

Systems and their modules are considered to be graphs with the nodes representing their constituent entities and the edges representing different kinds of interrelationships among them (Fig. 3). The definition of these categories is generic, in the sense, that depending on the underlying context, one can define his own measures that fit within one of the aforementioned categories. In order to be able to claim fitness within one of the aforementioned categories, there is a specific list of properties that the proposed measure must fulfill. For example, the size of a system modeled as a graph `S(E,R)` is a function `Size(S)` that is characterized by the properties: (a) *nonnegativity*, i.e., $\texttt{Size(S)}\geq 0$; (b) *null value*; $E=\emptyset \Rightarrow \texttt{Size(S)}=0$; and (c) *module additivity*, i.e., if a system `S` has two modules m_1 and m_2, then $\texttt{Size(S)} = \texttt{Size}(m_1) + \texttt{Size}(m_2)$. The last property shows that adding elements to a system cannot decrease its size (*size monotonicity*). For instance, the size of the system in Fig. 3 is the sum of the sizes of its two modules m_1 and m_2. The intuition here is that if the size of a certain module is greater than the size of another, then we can safely argue that the former is comprised of more entities than the latter.

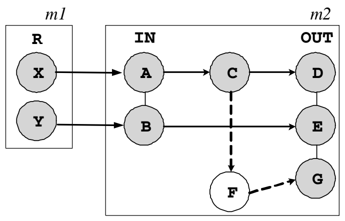

Fig. 3 A modular system graph containing two modules

Another important observation, found in both [Fent94] and [BrMB96], is that measurement theory imperatively demands that a measure describes an intuitively clear concept, i.e., there is a clear interpretation of what we measure. This should be coupled with clear procedures for determining the parameters of the model and interpreting the results.

In this paper, we propose a set of measures that evaluate our ETL blueprints and stay within the context of the measures proposed in [BrMB96]. *Our fundamental concern, for defining our measures is the effort required (a) to define and (b) to maintain the Architecture Graph, in the presence of changes*. Therefore, the statements that one can make, concerning our measures characterize the effort/impact of these two phases of the software lifecycle.

First, we identify the correspondence of the constructs of the Architecture Graph to the concepts of [BrMB96]. [BrMB96] defines a system, `S`, as a graph `S=(E,R)`, where `E` is the set of elements of the system and `R` is the set of relationships between the elements. A module `m` is a subset of the elements (i.e., the nodes) of the system (observe that a module is defined only in terms of nodes and not edges). In general, modules can overlap. However, when the modules partition the nodes in a system, then this system is called a modular system, `MS`. The authors distinguish two categories of edges: (a) the *intermodule* edges that have end points in different modules and (b) the *intramodule* edges that have end points in the same module. In terms of our modeling:

– The architecture graph `G(V,E)` is a modular *system*.
– Recordsets and activities are the *modules* of the graph. The nodes of the graph involve attributes, functions, constants, etc. All kinds of relationships are the *edges* of the graph.
– The system is indeed modular, i.e., there are no elements (nodes) that do not belong to exactly one module (activity or recordset).

– Inter-activity and side-effect rules result in intermodule edges. All the rest of the relationships result in intramodule edges.
– The union of two interacting activities can be defined: it requires merging the input/output nodes (attribute/schemata) connected by provider relationships.

3.1 Measures

Next, we define our measures. Due to lack of space, the proof of fitness within the set of properties of [BrMB96] for each measure can be found in [VSTS05]. We strongly encourage the reader to read the correctness proofs as they offer a deeper comprehension of the nature of the measures that we propose.

Size. Size is a measure of the amount of constituting elements of a system. Therefore, it can be considered as a reasonable indicator of the amount to define the system. In our framework, we adopt the *number of nodes* as the measuring rule for the size of the Architecture Graph; thus, the size of the architecture graph `G(V,E)` is given by the formula `Size(G) = card(V)`.

Length. Length is a measure that refers to the maximum length of "retransmission" of a certain attribute value. Length measures the longest path that we possibly need to maintain if we make an alteration in the structure of the Architecture Graph. For example, this could involve the deletion of an attribute at the source side. Then, the length characterizes how many nodes in the graph we need to modify as a result of this change (practically involving the nodes corresponding to this particular attribute, within the workflow).

In [VaSS02] the authors define the (transitive) dependency of a node as the cardinality of the (transitive closure of) provider relationships arriving at this node. To define the length of path from a module `m` backwards to the fountains of the graph, we use the maximum of its transitive dependency measure for the attributes of its output schemata. We avoid cycles in the graph by special treatment of side-effects [VSTS05]. Thereby, the *length of a module* `m` is given by the formula:

$$\mathtt{Length(m)} = \mathtt{max\{transitive\_dependency(i)\}},\ \mathtt{i} \in \mathtt{output\_schemata(m)}.$$

The *length of the graph* is defined as the maximum length over all its modules `m`:

$$\mathtt{Length(G)} = \mathtt{max(Length(m_j))}$$

Observe the reference example of Fig. 1. Although it does not depict a complete graph, the length of the depicted subgraph is 3, since the maximum length of its modules is 3 (input schema of activity `$2E`).

Complexity. Complexity is an inherent property of systems; in our case, complexity stands to the amount of interconnection of constituent entities of the Architecture Graph. This is an indicator of maintenance effort in the presence of changes. The more complex a system is the more amount of maintenance effort is expected to be required in the case of changes. [BrMB96] indicates that the properties of complexity focus on edges, thus, our function for complexity concerns the edges of the graph `G(V,E)` at the most detailed level. Again, we distinguish module from system complexity.

We define the *overall degree* of a module to be the overall number of edges of any kind (i.e., provider, part-of, etc) among its components, independently of direction. We count inter-module provider edges as half for each module. Then,

$$\texttt{Complexity(m)} = |E_{\texttt{intramodule}}| + 0.5 * |E_{\texttt{intermodule}}|$$

The complexity of the architecture graph G(V,E) is defined as the summary of the complexities of all the modules of the graph (i.e., recordsets and activities).

$$\texttt{Complexity(G)} = \texttt{overall\_degree(G)} = |E|$$

Cohesion. A commonly agreed upon property of modular software is that each module ideally performs exactly one job. Cohesion is the measure employed to assess the extent to which the modules of a system abide by this rule. In our case, we can exploit the peculiarities of our setting to assess the cohesion of our ETL workflows.

ETL operations can largely be classified in two categories. Each activity in our model performs one of two tasks: (a) filtering, meaning that a certain criterion is applied over the employed data in order to block those that do not pass the test and (b) transformation, meaning that a certain function is applied in order to generate some new value in the workflow. Both these tasks involve regulator relationships among the involved attributes and the functions/built-in selectors ($=$, $\leq$, etc.) of the activities. Therefore, the amount of regulator relationships should be a good indicator of the cohesion of a system. Moreover, we impose two extra requirements that we consider reasonable: (a) the more functions/built-ins employed, the less cohesive the module is (i.e., it is assumed/expected to perform more than one job) and (b) if more attributes are involved in regulator relationships, cohesion increases. In the sequel, we will refer to functions and built-ins as *functionality* nodes.

Before giving the formal definition, we will present the intuition of our proposed measure. Due to requirement (a), we need the inverse of the number of employed functionality nodes. Also due to the requirement (b) we need a measure analog to the number of attributes involved in a regulator relationship. Since [BrMB96] require cohesion to be normalized within a range [0…max] we need to normalize the number of attributes involved with the total number of attributes. To simplify things we measure only input and output attributes. Still, we count an input/output attribute as *functionality-related* even if it is not directly involved in a regulator relationship, but transitively dependent (or responsible) with an internal attribute that is involved.

In Fig. 3, we depict providers with solid lines and regulators with dotted lines. The input attribute A of module m_2 is involved in a regulator relationship transitively (through attribute C), whereas the attribute G is directly involved in a regulator relationship. Now, we are ready to define cohesion for our modules and system.

$$\texttt{Cohesion(m)} = \frac{\texttt{F\_IN} + \texttt{F\_OUT}}{\texttt{F} * (\texttt{IN} + \texttt{OUT})},$$

where F is the number of functions of the module, IN (OUT) is the number of input (output) attributes of the module, and F_IN (F_OUT) is the number of functionally-related input (output) attributes of module m.

$$\texttt{Cohesion(G)} = \texttt{avg(cohesion}(m_i)\texttt{)}, \text{ for all the modules } m_i \text{ of } G$$

Cohesion for the module m_2 of Fig. 3 takes the value of (1+1)/1*5=0.4

Coupling. In our framework, coupling captures the amount of relationship between the attributes belonging to different recordsets or activities (i.e., modules) of the graph. [BrMB96] indicates that the properties of complexity focus on intermodule edges, thus our function for complexity concerns the provider edges of the graph G(V,E) that start from an output node of a module and terminate to an input node of another module. Thus, we define the coupling of a graph G(V,E) as the sum of incoming and outgoing provider edges of each activity or recordset. This summary of edges for a certain module is called *local degree* according to the terminology we introduced in [VaSS02]. Thus, coupling is given by:

$$\text{Coupling(G)} = \sum_i \text{local\_degree}(m_i)\text{, for all the modules } m_i \text{ of G}$$

In the reference example of Section 2, the coupling of the activity SK is 13, i.e., the total number of its incoming and outgoing provider relationships.

3.2 Example

In order to demonstrate the usage of our proposed metrics, we present an exemplary scenario, implemented in three different ways. For each of these implementations we measure the different properties that we have proposed and discuss the observed phenomena.

The scenario involves the propagation of data from the product suppliers table DSA_PS(PKEY,SUPPKEY,DATE,QTY,COST) towards the table DW_V1(PKEY, SUM_COSTS) with the obvious semantics. Three operations need to take place between the two data stores: (a) a selection involving dates after 1/1/2004, (b) a second selection test involving only quantities greater than zero and (c) a summation of costs per product key. In the first scenario, we employ a different activity for each of the operations, with the activities connected serially. In the second scenario, we have merged the two filters in a single activity. In the third scenario, the selections are performed in parallel, the results are then joined and subsequently aggregated. The graph representation of the scenarios is partially depicted in Fig. 4, where the abstract representation of each scenario is shown in the upper part of each column and the part of the detailed representation is depicted in the lower part. We omit part-of relationships and details higher than the schema level for reasons of space and presentation. In the figure, we refer to attribute SUPPKEY as SUPP for lack of space. The metrics for each scenario are depicted in the tables 1- 3 and refer to the depicted graphs (with very small discrepancies from the overall graphs).

The observation of these measures reveals that the second scenario outperforms all the others in all categories. This is due to the fact that by merging the selections in a single activity, all provider relationships among modules are shortened. The same applies, of course, for the size of the graph. In terms of individual measures, we can observe the following:

- **Size** has obvious results, simply due to the number of attributes in the input and output schemata of the activities.

- The **length** is a clear indication of the maximum reproduction path of a datum and, obviously, no major differences are observed.
- The **complexity** and **cohesion** of the second scenario are quite impressing. Ideally, for reasons of maintainability, we would appreciate a scenario with low complexity and high cohesion. The complexity of the second scenario is significantly lower than any other alternative, since, obviously, fewer activities and fewer operations are performed. Although the combined selection activity has the same cohesion with the two individual ones (by a simple application of the formula), the overall cohesion drops due to the smaller number of involved activities. Thus, the cohesion of the second scenario is noticeably higher than the other two. Also, the fact that the cohesion of the combined activity remains the same is not surprising: although two selections are performed, the number of attributes involved increases, so the fraction remains stable.
- Finally, the **coupling** is clearly a subset of the complexity measure: by isolating only provider relationships, we clearly see that module interconnections are lowest when the second scenario is employed.

Table 1: Measures for Scenario 1, involving a linear composition of three activities

	Size	Length	Complexity	Cohesion	Coupling
DSA_PS	6	0	7.50	-	5
σ1	14	2	24.00	0.10	10
σ2	14	4	24.00	0.10	10
sum	14	7	20.75	0.29	7
DW_V1	3	8	3.00	-	2
Overall	**51**	**8**	**79.25**	**0.16**	**34**

Table 2: Measures for Scenario 2, where selections are merged

	Size	Length	Complexity	Cohesion	Coupling
DSA_PS	6	0	7.50	-	5
Σ	16	2	28.00	0.10	10
Sum	14	5	20.75	0.29	7
DW_V1	3	6	3.00	-	2
Overall	**39**	**6**	**59.25**	**0.19**	**24**

Table 3: Measures for Scenario 3, where selections are performed in parallel

	Size	Length	Complexity	Cohesion	Coupling
DSA_PS	6	0	7.50	-	5
σ1	14	2	24.00	0.10	10
σ2	14	2	24.00	0.10	10
join	20	4	36.50	0.07	15
sum	14	7	20.75	0.29	7
DW_V1	3	8	3.00	-	2
Overall	**71**	**8**	**115.75**	**0.14**	**49**

Finally, we can easily observe (both by visualization and measurement) that there exist attributes that should not participate in the workflow, either in the first place (e.g., attribute `SUPPKEY`) or after their corresponding selections have taken place (e.g., attributes `DATE` and `COST`).

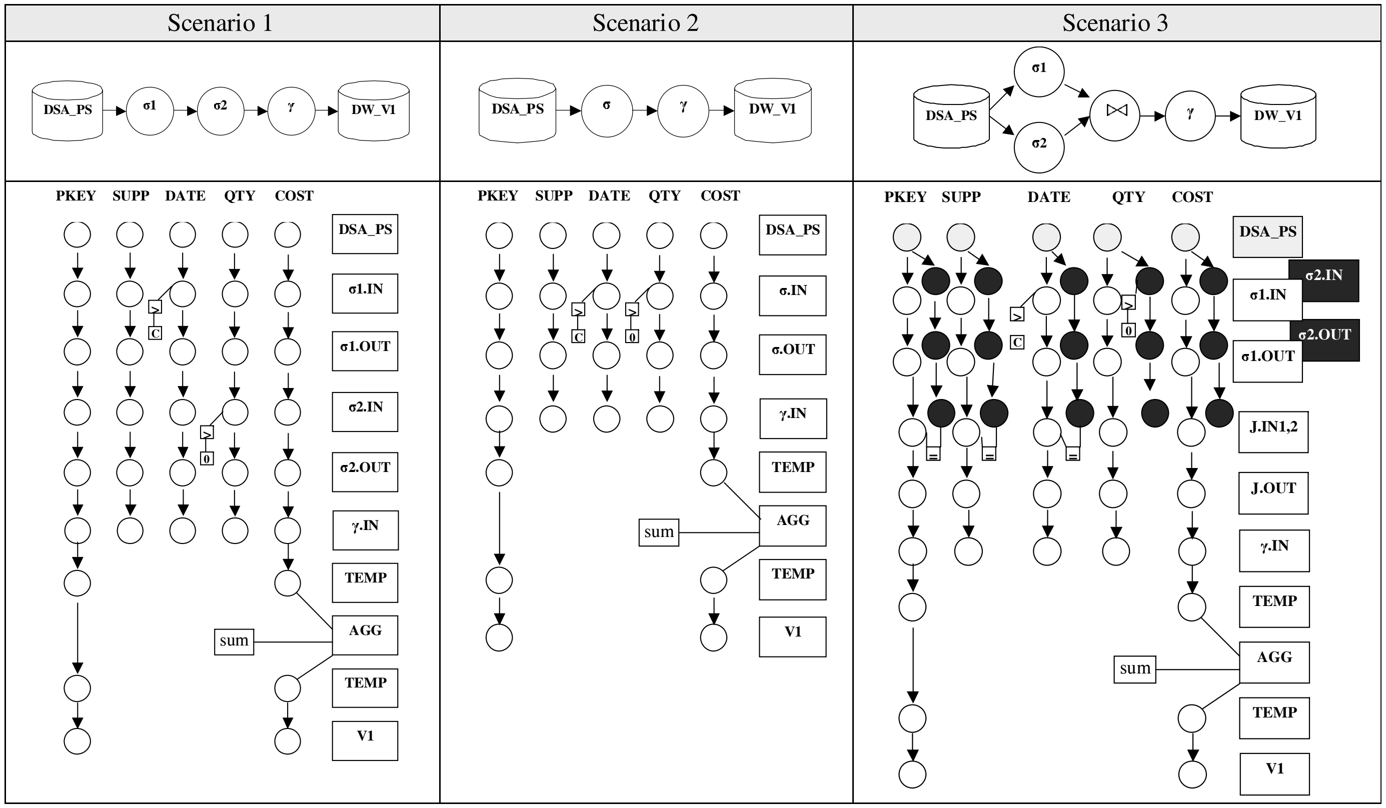

Fig 4. Equivalent scenarios for the propagation of data from a data source to the warehouse

4. Related Work

There are several efforts that present systems tailored for ETL tasks [GFSS00, RaHe01]. 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. Concerning the conceptual modeling of ETL, [TrLu03] and [VaSS02a] are the first attempts that we know of. The former approach employs UML as a modeling language whereas the latter introduces a generic graphical notation. Still, the focus is only on the conceptual modeling part. As far as the logical modeling of ETL is concerned, in [VaSS02, VSGT03] the authors present a graph-based model and an extensible template-based mechanism to define ETL workflows. As already mentioned, in this paper, we extend this model by incorporating the internals of the activity semantics to the graph (more extensions, e.g. updates, can be found in [VSTS05]).

Concerning related work on software measurement, we have already mentioned the fundamental works that have guided our approach. [Fent94] gives the fundamentals of measurement theory and the way they should applied in the case of measuring software artifacts. Briand et al. [BrMB96] present the overall framework for defining our particular measures. The particular contribution of this paper is that it gives the principles for defining large categories of software measures. In our case, we prove that the proposed measures fit within the context given by [BrMB96]. There is an extensive discussion of software metrics in [Dumk02] and an interesting discussion of this area in [FeNe02].

5. Conclusions

In this paper, we construct the blueprints for the structure ETL workflows by mapping both their inter-connection and their internal semantics to a graph, which we call the Architecture Graph. The Architecture Graph constitutes the blueprint over which we can perform further analysis for the structure of such a workflow. The first of our contributions involves extending existing results by capturing the internal semantics of each activity in the workflow. We employ the LDL language in order to capture the semantics of ETL activities and we have provided a principled way of transforming LDL programs to the graph. Apart from the value that blueprints have as modeling constructs, we can also exploit them in order to introduce rigorous techniques for the measurement of ETL workflows. To this end, we have built upon the formal framework [BrMB96] and provide software measures to quantify the size, length, complexity, cohesion and coupling of ETL workflows. Several issues omitted in this paper for lack of space, can be found in [VSTS05].

Research can be continued in more than one direction. We need an extra step, in order to link our results to the control flow of the graph. Precise algorithms for the evaluation of the impact of changes in the Architecture Graph can also be devised. New metrics can also be discovered, if they appear to reveal properties not covered here. Finally, the usage of the Architecture Graph in all phases of the software lifecycle (e.g., testing) can also be evaluated.

References

[BoRJ99] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1999

[BrMB96] L.C. Briand, S. Morasca, V.R. Basili. Property-Based Software Engineering Measurement. In IEEE Trans. on Software Engineering, 22(1), Jan 1996.

[BrMB97] L.C. Briand, S. Morasca, V.R. Basili. Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties". In IEEE Trans. on Software Engineering, 23(3), March 1997.

[CeGT90] S. Ceri, G. Gottlob, L. Tanca. Logic Programming and Databases. Springer-Verlag, 1990.

[Dumk02] R.R. Dumke. Software Metrics: a subdivided bibliography. Available at http://irb.cs.uni-magdeburg.de/sw-eng/us/bibliography/bib_main.shtml

[FeNe02] N.E. Fenton, M. Neil: Software metrics: roadmap. ICSE - Future of SE Track 2000: 357-370.

[Fent94] N. Fenton. Software Measurement: A Necessary Scientific Basis. In IEEE Trans. on Software Engineering, 20(3), March 1994.

[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.

[PoDe97] G. Poels, G. Dedene. Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties". In IEEE Trans. on Software Engineering, 23(3), March 1997.

[RaHe01] V. Raman, J. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. Proceedings of 27th International Conference on Very Large Data Bases (VLDB), pp. 381-390, Roma, Italy, 2001.

[TrLu03] J. Trujillo, S. Luján-Mora: A UML Based Approach for Modeling ETL Processes in Data Warehouses. In Proc. of the 22nd Intl. Conference on Conceptual Modeling (ER 2003), pp. 307-320, Chicago, IL, USA, October 13-16, 2003

[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.

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

[VSTS05] P. Vassiliadis, A. Simitsis, M. Terrovitis, S. Skiadopoulos. Blueprints for ETL workflows (long version). Available through http://www.cs.uoi.gr/~pvassil/publications/2005_ER_AG/ETL_blueprints_long.pdf

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