Propagating Evolution Events in Data-Centric Software Artifacts

George Papastefanatos[1], Panos Vassiliadis[2], Alkis Simitsis[3]

[1]*Institute for the Management of Information Systems / RC "Athena"*
Athens, Hellas
gpapas@imis.athena-innovation.gr

[2] *Department of Computer Science, University of Ioannina*
Ioannina, Hellas
pvassil@cs.uoi.gr

[3]*HP Labs*
Palo Alto, CA, USA
alkis@hp.com

***Abstract*—The success and wellbeing of large organizations rely on the smooth functionality and operability of their software. Such qualities are largely affected by evolution events and changes, like software upgrades. In this paper, we are dealing with handling evolution events in data management systems. We consider a data-centric ecosystem that captures relational tables, views and queries (the latter are seeing as software modules that are either internal to the database, e.g., stored procedures, or external software applications that access the database). We also consider policies dictating the response of a software module to a possible event. We investigate the impact of such events to the database and present a graph-based mechanism to control event propagation. We show that our mechanism terminates and that every database construct is annotated with a single status, regardless of the sequence of messages that the node receives.**

Keywords- database evolution, confluence, event propagation

I. INTRODUCTION

The success and wellbeing of large organizations rely on the smooth functionality and operability of their software. Such qualities are largely affected by evolution events and changes, like software upgrade. The problem we are dealing with in this paper involves the identification and regulation of schema evolution impact in complex data-centric ecosystems.

We model a data-centric ecosystem as an *Architecture Graph*. This graph captures relational tables, along with their schemata and constraints, as well as, views defined on top of them and queries (being parts of software modules that are either internal to the database, e.g., stored procedures, or external software applications that access the database). Evolution changes affecting the database structure are mapped to graph operations on the graph nodes. The graph is also annotated with policies that dictate the response of a software module to a possible event. For example, when a database table acting as a provider of a view is about to be deleted, the view may veto the deletion if it is annotated by an appropriate policy.

For exploring the impact of a potential event to the graph, we study the message propagation. Every time a node receives an event, it (a) determines which policy rules apply for this event, (b) assumes the appropriate status based on these rules, and (c) notifies its neighbors for the event (if needed) via appropriate messages that act as events to their recipients. Thus, when a potential event is submitted, the graph is annotated with statuses that report on whether an event affects a node or not, and in the case that it does, what action should be taken for the affected node. Actions imposed on affected nodes may in turn generate evolution events propagated as new messages towards the rest of the dependent graph structures. Hence, this paper answers the following question: *Given an evolution event e over a node of the Architecture Graph v, how do we guarantee that (a) the event propagation terminates and (b) that every node is annotated with a single status, regardless of the sequence of messages that the node receives?*

Prior work focuses on a simpler data model, which did not prevent multiple messages arriving at the same node and, due to this shortcoming, it cannot guarantee confluence of the evolution process [7]. Here, we solve this issue by framing change messages within high level constructs before they are freely flooded over the whole ecosystem's graph. The benefits of this process are twofold: we achieve localization of decisions and guarantee satisfactory handling of event transactions; and we also achieve nice properties, like confluence.

II. BACKGROUND MODELING

We extend previous modeling of the Architecture graph (see [6]) in order to guarantee a safe, confluent mechanism for message propagation. Here, views and queries are considered as containers of nodes, which are encapsulated between the input and output schemata of a view or a query. Thus, we treat such schemata as first class citizens of the model. A full description of the model can be found in [11].

A. Architecture graph.

Database constructs are presented as a directed graph $G=(V,E)$, which we call *Architecture graph*. Its main components are as follows.

Modules. A *module* is a semantically high level construct of the ecosystem and stands for relations, views, and queries. Each module defines a scope and it is disjoint to the others.

Relations, R. Each relation $R(\Omega_1,\Omega_2,...,\Omega_n)$ is represented as a graph that comprises: (a) a *schema node*, R; (b) n *attribute nodes*, $\Omega_i \in \mathbf{\Omega}$, i=1..n; and (c) n *schema relationships*, $\mathbf{E_S}$, connecting relation and attribute nodes.

Conditions, C. Conditions refer to *selection conditions* of queries and views and *constraints* of the database schema and belong to three classes: (a) Ω op constant; (b) Ω op Ω′; and (c) Ω op Q, where op is a typical binary operator and Q is a query.

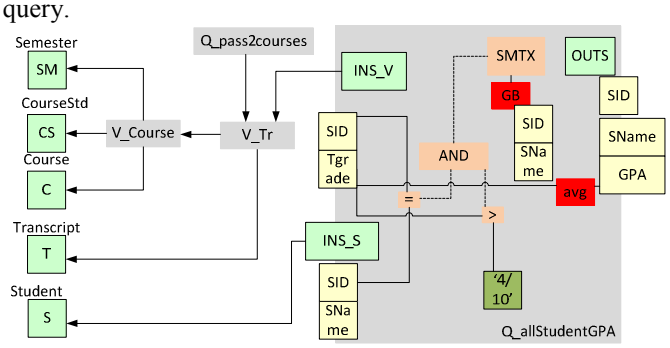

Figure 1. Architecture graph of an example database (our running example)

Queries, Q. The graph representation of a Select - Project - Join - Group By (SPJG) query involves: (a) a *query node*; (b) a set of *input schemata nodes* (one for every table in the FROM clause); (c) an *output schema node* comprising the set of attributes of the SELECT clause; (d) a *semantics* node, as the root node for the subgraph corresponding to the query semantics, including the WHERE/HAVING/GROUP BY clauses of the query; and (e) *attribute nodes* belonging to the various input and output schemata of the query.

Views, V. Views are treated as queries. The output schema of a view can be used as input by a subsequent view or query.

Summary. A *summary* of the Architecture graph is a zoomed-out variant of the graph that comprises only modules as nodes and edges denoting any possible form of provider relationship between modules. Formally, a summary is a directed acyclic graph $\mathbf{G_s}=(\mathbf{V_s},\mathbf{E_s})$, where $\mathbf{V_s} \subseteq \mathbf{R} \cup \mathbf{V} \cup \mathbf{Q}$ comprises the graph's module nodes and $\mathbf{E_s} \subseteq \mathbf{E_F}$ comprises pairs of providers and consumers as *from-relationship* edges, $\mathbf{E_F}$.

Example. Consider the example database of Figure 1. The database contains information on semesters, courses offered by a department, and students with their transcripts; i.e., what course they have enrolled to and with what grade. V_Course is a view that combines three relations, Semester, CourseStd, and Course, into a single view that contains the identifiers and descriptions of the involved entities. The V_Tr view joins V_Course with the Transcript relation, resulting in a view containing the information needed for students' enrollment. A query, Q_pass2courses, performs a self-join over view V_Tr and presents a report that compares the grades for two courses, DB_I and DB_II, for those students who enrolled in both courses. Another query, Q_allStudentGPA, reports the average grade (i.e., over successfully passed courses) for every student; the report requires students' names, so the relation Student is joined to the view V_Tr. For simplicity, we omit all constraints (e.g., primary and foreign keys) and some detailed edges, like selection edges. We intentionally do not expand the subgraph of Q_pass2courses, V_Course, and V_Tr for showing the benefits of handling evolution within modules.

B. Graph Annotation with Policies.

We evaluate the impact of a change over the system. In the past, we described how to map schema changes occurring in a database ecosystem to operations on graph nodes [7]; e.g., adding an attribute to a relation translates as adding a child node to a relation module. We also discussed how to enrich the graph with rules, called *policies*, which dictate how to act when specific events occur on the nodes of the graph. Policies can be applied at various granularity levels on the graph [8]; i.e., from the modules level down to the attribute and operand nodes levels. Two example rules are: (a) *propagate* the change, i.e., the graph must be reshaped to adjust to the new semantics incurred by the event; and (b) *block* the change, i.e., we retain the existing graph semantics either by blocking the event or by constraining it through a rewriting that preserves the old semantics. Default values and policy resolution rules guarantee that each node may determine the appropriate policy for any event it receives. For example, the policy "On add_attribute to Transcript Then propagate" defined on V_Tr.INS_T node, allows the propagation of the addition of a new attribute in the Transcript relation towards the schema of the view.

C. Message propagation and Status Resolution

When an event (e.g., attribute deletion) is submitted to the graph, a mechanism must ensure that this event is propagated to all nodes affected, either directly or transitively, and that each affected node acquires the correct status, according to its policies for this event. This mechanism has three main parts.

TABLE I.
EVENTS, POLICIES, AND STATUSES OF NODES

V_E	{SELF, CHILD, PROVIDER} x {ADD, DEL, UPD} x {STRUCTURE, SEMANTICS, S+S}
V_P	{BLOCK, PROPAGATE}
V_S	{SELF, CHILD, PROVIDER} x {ADD, DEL, UPD} x {STRUCTURE, SEMANTICS, S+S }

Policy determination. Clearly, it would be very hard for the user to have to define a policy per event for every module of the Architecture Graph. In [8], we have defined a language where the user can dictate "default" policies both at the graph level and for the children of individual nodes, in order to avoid this effort. In fact, the language allows the user to define policies at different levels of abstraction which can be overriding one another (so, for example, if the default policy for the deletion of input schema attributes is block, the user can override it for the input schema attributes of a particular view). Then, a late-binding mechanism determines the winner policy

for each specific node. For a most detailed description of the policy annotation and determination, we refer the interested user to [7].

Status determination. Given finite vocabularies of events, V_E, policies V_P, and statuses V_S, we consider a set of rules as a function *DS:* V_E x V_P → V_S (see Table I).

Event propagation. When a node acquires a status for an event, we need to broadcast messages to its neighbors. Each message corresponds to a unique event occurring on the sender node and describes the event type and the status assigned to it according to the prevailing policy. Each message is processed inside a module and may trigger one or more events for further propagation to the consumer modules. For example, the deletion of an attribute that participates in the SELECT and WHERE clauses of a view, generates a new message for the consumers of the view; this message encodes the modification of the view's structure and semantics.

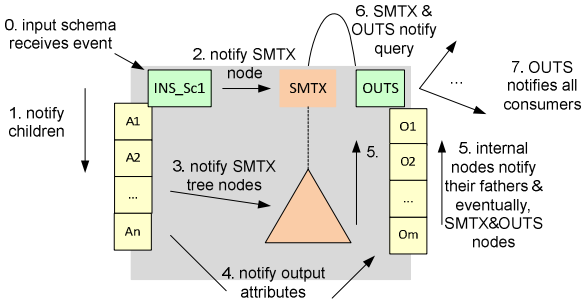

Figure 2. Protocol for the message propagation mechanism

III. MESSAGE PROPAGATION MECHANISM

The graph nodes form a directed acyclic graph of dependencies. Therefore, it is straightforward to get a topological sort of the summary of the architecture graph. We enforce the rule: *"modules communicate with each other via a single means: the output schema of a provider module notifies the input schema of a consumer module"*. Then, the following protocol is used:

(i) We topologically sort the graph at the module level.

(ii) We visit each module in its topological order and check whether there are incoming messages for it. If there are, the topological sort guarantees that all messages pending for the input schemata of the module are ready.

(iii) Every module processes the incoming events and decides the status for its semantics and output schema. Next, it propagates this information to all its consumers (if any).

Figure 2 shows the protocol for handling events within each module. The propagation mechanism involves four node types.

– *Input schemata nodes*. They receive notifications for changes from other modules.

– *Internal node*s. They are: (a) possibly affected by the changes to the input schema nodes, and (b) amenable to evolution events by the users (e.g., a user altering the selection condition or the grouper attributes of a view).

– *Output schema nodes*. They emit messages to their consumers for possible modification of their component.

– *Semantics nodes*. They determine whether the semantics of a component are the same or not and inform the output schema nodes for further propagation.

Message handling should ensure that within a module each event affects the appropriate nodes and that every node must be visited and processed (i.e., its status must be determined) once per message. The process mechanism is as follows.

Messages arriving at a node are propagated to all of its consumers (i.e., adjacent nodes connected with an incoming edge to this node) *according to the type of event* that they encode (e.g., the addition of an attribute is propagated only to semantics and output nodes, whereas the deletion of an attribute is propagated to attribute nodes). For each event initiated by the input schema or the user, we identify the affected subgraph of the module. For identifying the subgraph, we process each event by executing the protocol mechanism and assuming that no policies constrain the flooding process. The produced subgraph contains only nodes potentially affected by this event.

TABLE II.
MESSAGE PROPAGATION FOR NODES IN A MODULE

Messages arrive from	node type	Messages propagated to
{provider's output schema}	Input schema	{children, semantics, output schema}
{father, provider, children, user(self)}	Internal Nodes	{children (if any), consumers, father}
{input schema, children}	Semantics	{output schema}
{semantics, children, input schema}	Output schema	{consumers' input schema, module}

Each identified subgraph is acyclic (see Theorem 3). Then, a second execution of the protocol starts from the input schema (or the node affected explicitly by the user) and visits each node in a topological order of the subgraph. Based on the defined policy, the node is assigned with a status which is enqueued as a new event in the message list of all of its consumers. The process continues with the next node in the topological order of the identified subgraph. The method guarantees that each node is processed once, after all feasible messages have arrived at it. Next, we present the message handling mechanism for each class of nodes (see also Table II).

Input schema nodes. The input schema nodes receive messages from the output schema nodes of a provider relation/view. For example, the output schema of a relation module may report the following events to the input schema node of a succeeding query: (a) the relation is renamed or deleted; (b) attributes are added/deleted/updated; (c) constraints are added/deleted/updated. When an input schema node receives such a message, then, the following mechanism is triggered:

(i) The correct policy (based on the type of the received message) is determined for the receiving input schema node.

(ii) The rule dictating the policy is fired and the appropriate status is assumed. In the case of propagation, the node assumes a status for adjusting to the event, whereas in the case of block policy, the node takes a status for blocking the event. For example, in the case of an incoming message for the addition of a new attribute, for which the in-

put schema retains a propagate policy, the input schema node is assigned with a status for adding a child.

(iii) For changes referring to the input attributes (e.g., deletion of an attribute at the provider's schema, renaming, domain modification, etc.) the appropriate input attribute nodes of the input schema are notified.

(iv) The input schema node propagates a message containing changes on the semantics of the provider module directly to the semantic node of the current module – if such changes exist; otherwise no such action is taken.

(v) The input schema node propagates a message for adding children to the output schema node of the module and (if any) to the group by node via the semantic node.

Each input schema has exactly one provider (i.e., the output schema of the provider module) and it can receive exactly one message that triggers the evolution handling mechanism in every module. Thus, a module can receive, at most, as many event handling messages for the same original event as its input schemata. Alternatively, the mechanism starts when a user applies a change at a module, and this triggers exactly one 'input' message possibly at an internal node.

Internal nodes. These can be either attributes in the input/ output schemata of a module, or logical components of the semantics node of a module, like a function node, an operand or a constant node, group by node, etc. Intra-module nodes can receive messages either from (a) their father (e.g., an input schema node notifies that a specific input attribute must be deleted), (b) from their provider nodes (e.g., an output attribute node or an operand node is notified by its provider attribute in the input schema for its deletion), (c) one of its children or lastly (d) explicitly by the user who triggers the modification of the node itself. The message propagation for internal nodes mainly notifies all their consumers on what is happening to them and the semantics node on whether the semantics of the component change. The mechanism is as follows:

(i) – (ii) The first two steps are like those of the input schema nodes.

(iii) If the node has children and receives a notification from its father or if it initiates the event, then its children are notified too. This mainly applies to operand nodes in composite conditions at views and queries or relations' attributes having constraints (e.g., conditions) as children.

(iv) If the node is notified by one of its providers or one of its children, the father of the node is notified, too. This covers the case where a user triggers an event in the contents of a view (e.g., deletion of a condition) or a relation (e.g., modification of an attribute), so that the event would be also propagated upwards to the module node.

(v) The node consumers (if any) are notified too. This covers the case where an input attribute is changed, so that the event is propagated towards all nodes (e.g., output attributes, conditions, functions, group by attributes) that refer to this attribute.

Every node notifies its consumers, and since the event handling is performed in a topological order of the module graph, each node receives at most one message *per edge* for the same event. (For more details see the long version of the paper [11].)

Semantics nodes. A semantics node receives messages either from the input schema of the module, for messages containing changes on the semantics of the provider modules, or from its children. The mechanism triggered by the semantics node when receiving such messages is as follows:

(i) – (ii) The first two steps are like those of the input schema nodes.

(iii) The semantic node propagates a message for addition of children towards (if any) the group by node.

(iv) The semantics node propagates all other messages coming from either the input schema node (e.g., for changes in the semantics of a provider module) or its children (i.e., for changes in the semantics of the module itself) to the output schema node of the module.

Output schema nodes. The output schema is responsible for establishing the overall status of the module. An output schema node can receive messages from the semantics node regarding semantic changes in the module, from the input schema for additions of attributes or from one of its children for changes referring to the exposed structure of the module. The mechanism for handling received event signals is:

(i) – (ii) The first two steps are like those of the input schema nodes.

(iii) The father of the output schema node, i.e., the module's node, is notified too. Whenever the module's node gets a notification from the output schema it acquires the right status (i.e., *block* if a veto has been fired or the appropriate status in any other case).

(iv) Except for the case the assigned status is *block*, all consumers (input schemata) of the output schema node are notified with a message announcing the module's status.

TABLE III.
ADDING EXAMYEAR TO TRANSCRIPT

visited module	Visited Node	message arriving	status	Message Emitted	next node in queue
Transcript	Transcript	AC {EY}	AC	AC {EY}	V_TR.INS_T
V_TR	INS_T	AC {EY}	AC	AC {EY}	OUT_S
V_TR	OUT_S	AC {EY}	AC	AC {EY}	V_TR,Q_1.INS_V1, Q_1.INS_V2,Q_2.INS_V
V_TR	V_TR	AC {EY}	AC	None	none
Q_1	INS_V1	AC {EY}	AC	AC {EY}	OUT_S
Q_1	INS_V2	AC {EY}	AC	AC {EY}	OUT_S
Q_1	OUT_S	AC {EY}	AC	AC {EY}	Q_1
Q_1	Q_1	AC {EY}	AC	AC {EY}	none
Q_2	INS_V	AC {EY}	AC	AC {EY}	SMTX, OUT_S
Q_2	SMTX	AC {EY}	MS	AC{EY}, MS	GB, OUT_S
Q_2	GB	AC{EY},MS	AC	AC{EY}	none
Q_2	OUT_S	AC{EY},MS	AC,MS	AC{EY}, MS	Q_2
Q_2	Q_2	AC{EY},MS	AC,MS	None	none

Legend: AC:Add_Child, MS:Modify_Semantics

Example (cont'd). Assume that a user adds a new attribute, `ExamYear (EY)`, representing the year that the student has taken the exam on each course, to the `Transcript` relation

(see Figure 1). Assume also, that the *propagate* policy is assigned on all visited nodes. The message propagation for this event is presented in Table III. The message for adding EY to Transcript results in assigning the appropriate status for adding EY as a new child. Since the policy is *propagate*, an identical message is created and the input schema node V_TR.INS_T connected with the Transcript node is visited. The V_TR.INS_T node adapts the event and informs the output schema node for the addition. The affected subgraph for this event according to our mechanism comprises nodes {V_TR.INS_T, V_TR.OUT_S,V_TR} which are visited in this order. Next, the V_TR.OUT_S node propagates the event towards all input schema nodes referring to the V_Tr view. For the Q_pass2courses (Q_1) query, each input schema node (i.e., Q1.INS_V2 and Q_1.INS_V1) receives a distinct message for the attribute addition. These messages are propagated towards the query output schema Q_1.OUT_S as two separate events. The message propagation terminates on the output schema nodes of the two queries (see Figure 1 too), Q_1.OUT_S and Q_2.OUT_S, as no other consumer modules exists. The output schema of the Q_allStudentsGPA (Q_2) query, receives two messages for two separate events; one from the addition of the attribute in the input schema of the query and the other for the modification of the semantics as result of the incorporation of the new attribute to the group by clause of the query.

IV. THEORETICAL GUARANTEES

In this section, we present the theoretical guarantees for the correct execution, termination and confluence of the aforementioned protocol mechanism on the architecture graph. We examine and prove these properties both at the summary graph, i.e., at the intermodule level (theorems 1-3), as well as within each module (theorem 4).

A. Guarantees at the intermodule level

In this subsection, we prove that the mechanism for message propagation works correctly at the summary or, *intermodule* level. We assume that each module responds correctly to a given event; we prove this property in the subsequent subsection.

Theorem 1 (termination). The message propagation at the intermodule level terminates.

Proof: The summary of the Architecture Graph is a **directed acyclic cycle.** This is due to the fact that a query depends only on views and relations and relations do not depend on anything (in the context of this paper, we do not consider cyclic foreign key dependencies). Since the summary graph is a DAG, we can topologically sort it and propagate the messages according to this topological order. Thus, all that it takes for the message propagation mechanism to terminate is: (a) each module emits at most one message for each session to every one of its neighbors; (b) the graph is finite. Since both assumptions hold, the algorithm terminates. □

Theorem 2 (unique status). Each module in the graph will assume a unique status once the message propagation terminates.

Proof: At the summary level, each input schema of a consumer module receives the status and the output schema structure of its provider module. The **topological ordering** of the graph guarantees that whenever a module is considered, all its providers have already been processed. Then, Theorem 4 proves that once all notifications from the module's providers are in place, the module will uniquely acquire a status. □

Theorem 3 (correctness). Messages are correctly propagated to the modules of the graph.

*Proof***:** The modules that must be appropriately notified are these for which an event occurs at their providers. At the summary level, the Architecture graph is a connected graph, where one (or more) input schema node(s) of a consumer module is connected via **directed edges** to the output schema node(s) of its providers. The messaging mechanism dictates that each message is propagated from the output node of the provider module towards the input schema node of all consumer modules, unless a block policy explicitly halts the propagation. On the other hand, the modules that are not visited by the mechanism (a) either do not have any provider affected or (b) a block policy exists; therefore, they can safely ignore any notification. □

B. Guarantees at the intramodule level

In this subsection, we prove that once an event arrives at a module, the module responds to the event and annotates the output schema with the correct status.

Theorem 4 (termination and correctness). The message propagation at the intramodule level terminates and each node assumes a status.

Proof: At the intra-module level, for the termination of the mechanism, we must prove that each constructed subgraph per event type is a directed acyclic graph. For the correctness of the mechanism we require that every node is processed once (and thus assigned with a status) for all messages arriving at a module per session. The latter can be satisfied when the determined subgraph can be topologically sorted and traversed. Thus, for both requirements we must prove that the subgraph constructed per event type has no cycles. We cover the following events:

- *Change in semantics of provider*. The message arrives to the input schema node and is propagated to the semantics node. The affected subgraph comprises the following nodes and directed edges in topological order: {input schema→semantics→output schema→module}[1]. No cycles detected.
- *Internal change in the semantics of a module* (e.g., a user deletes a part of the condition expression of a view). the semantics node is eventually notified from the upwards flow of messages in the condition tree and the children are notified from the downwards flow of messages. For the case that a condition node is modified, the subgraph is: {internal node→(up)condition tree → semantics→output schema→module},

[1] For ease of graph serialization we denote an edge directing from input schema towards semantics as "input schema→semantics".

{internal node→(down)condition tree}. No cycles detected. For the case that a grouping attribute is modified, the subgraph comprises:
{GB Attributes → GB → semantics → output schema → module}.

- *Deletion in the structure of the input schema*. All affected nodes in the tree of the condition part are notified via the operand relationship edges; all group by and output schema are notified via the map-select edges. Subgraph potentially (if group by part exists) comprises:
{input schema → input attributes},
{input attributes → condition tree → semantics},
{input attributes → GB attributes →GB node → semantics},
{input attributes → output attributes → output schema},
{semantics → output schema},
{output schema → module}. No cycles detected.
- *Addition in the structure of the input schema*. A message is sent to the output schema and to the semantic node for informing the group by node (if any). Subgraph potentially comprises:
{input schema → semantics},
{input schema → output schema},
{semantics → GB node},
{semantics → output schema},
{output schema → module}. No cycles detected.
- *Deletion of in the input schema overall* (the provider dies overall too). The deletion is correctly propagated from the messages sent by all the child nodes of the schema.
- *Change in structure (deletion or addition) and semantics of a provider*. When messages arriving at an input schema node contain changes both at the structure and the semantics of the provider module, the subgraph is the union of the subgraphs corresponding to each case. Thus, for attribute addition and change in provider semantics, the subgraph is:
{input schema→semantics},
{input schema → output schema},
{semantics → GB},
{semantics → output schema→module}. No cycles detected. For attribute deletion and change in provider semantics, the subgraph is:
{input schema → semantics},
{input schema → input attributes},
{input attributes → condition tree → semantics},
{input attributes → GB attributes → GB node → semantics},
{input attributes → output attributes → output schema},
{semantics → output schema},{output schema → module}. No cycles detected. □

In all cases, at the end of the process, the output schema (and eventually the module itself) has knowledge (a) of what happens to their children and (b) what happens to module and can pass this information to the next consumer.

V. Related Work

Schema evolution has been studied in databases [9], [10]. Evolution related approaches have been proposed for the OO paradigm [13] and DW configurations [3]. A technique for publishing the history of a relational database in XML employs a set of schema modification operators (SMOs) to represent the mappings between successive schema versions and an XML query language to address queries expressed over different versions using the mappings established by the SMOs [4]. View adaptation after redefinition, where changes in views definition are invoked by the user and rewriting is used to keep the view consistent with the data sources, has been studied [1], [2]. A previous work considers the view synchronization problem, where views become invalid after schema changes in their definition [5]. Here, the policies act as regulators for the propagation of schema evolution on the graph, similarly to the evolution parameters introduced in [4]. Another effort presents a model for retaining the original semantics of the queries by preserving mappings consistent when changes occur [12]. Here, we allow restructuring of the database graph either for keeping the original semantics or for adapting to new ones and also, we employ a message propagation mechanism for detecting and regulating evolution impact in complex database ecosystems.

VI. Conclusions

We focused on the problem of change propagation in database ecosystems. Based on a graph representation of a database and considering that the graph is annotated with policies dictating the response of a software module to a possible event, we studied the impact of such events to the database and presented a graph-based mechanism to control event propagation.

References

[1] Z. Bellahsene, "Schema evolution in data warehouses". In Knowledge and Information Systems, 4(3), pp. 283-304, 2002.

[2] Gupta, I. S. Mumick, J. Rao, K. A. Ross, "Adapting materialized views after redefinitions: Techniques and a performance study". In Information Systems, 26(5), pp. 323-362, 2001.

[3] M. Golfarelli, J. Lechtenbörger, S. Rizzi, G. Vossen, "Schema Versioning in Data Warehouses". In ECDM, pp. 415–428, 2004.

[4] Moon, H.J., Curino, C.,Deutsch, A., Hou, C.Y., Zaniolo, C. Managing and querying transaction-time databases under schema evolution. In VLDB, pp. 882-895, 2008.

[5] Nica, A. J. Lee, E. A. Rundensteiner, "The CSV algorithm for view synchronization in evolvable large-scale information systems". In EDBT, pp. 359-373, 1998.

[6] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou, "What-If Analysis for Data Warehouse Evolution". In DAWAK, pp. 23-33, 2007.

[7] G. Papastefanatos, P. Vassiliadis, A. Simitsis, Y. Vassiliou. Policy-Regulated Management of ETL Evolution. In JoDS, vol. XIII, pp. 146–176, 2009.

[8] G. Papastefanatos, P.Vassiliadis, A.Simitsis, K.Aggistalis, F.Pechlivani, Y.Vassiliou, "Language Extensions for the Automation of Database Schema Evolution". In ICEIS, 2008.

[9] J.F. Roddick et al., "Evolution and Change in Data Management - Issues and Directions". In SIGMOD Record, 29(1), pp. 21-25, 2000.

[10] J.F. Roddick., "A survey of schema versioning issues for database systems". In Information Software Technology, 37(7), 1995.

[11] G. Papastefanatos, P. Vassiliadis, A. Simitsis, "Propagation of Evolution Events in Architecture Graphs" (long version of this paper), url: web.imis.athena-innovation.gr/~gpapas/Publications/TR2010.1.pdf

[12] Y. Velegrakis, R.J. Miller, L. Popa, "Preserving mapping consistency under schema changes". In VLDB J., 13(3), pp. 274-293, 2004.

[13] R. Zicari, "A framework for schema update in an object-oriented database system". In ICDE, pp. 2-13, 1991.