Διάδοση Αλλαγών Σε Πληροφοριακά Οικο-συστήματα[1]

Γιώργος Παπαστεφανάτος, Πάνος Βασιλειάδης, Άλκης Σιμιτσής

Στο άρθρο αυτό ασχολούμαστε με τη διαχείριση γεγονότων εξέλιξης σε συστήματα διαχείρισης δεδομένων. Συγκεκριμένα, μας απασχολεί ένα προσανατολισμένο στα δεδομένα οικοσύστημα (data centric ecosystem) που στη μοντελοποίησή του περιλαμβάνει σχέσεις, όψεις και ερωτήσεις. Οι ερωτήσεις είναι μια καλή αρχική αφαίρεση για να μοντελοποιήσουμε πολύ πιο σύνθετες ενότητες (modules), οι οποίες είναι είτε εσωτερικές στη βάση δεδομένων (όπως για παράδειγμα αποθηκευμένες διαδικασίες – stored procedures) ή εξωτερικές (όπως για παράδειγμα, εφαρμογές που προσπελάζουν τη βάση δεδομένων, αναφορές, κλπ). Με άλλα λόγια, το οικοσύστημα επεκτείνεται και εκτός της βάσεως και περιλαμβάνει και τις εφαρμογές που προσπελάζουν τη βάση στη μοντελοποίησή του. Επιπλέον, επισημειώνουμε το πληροφοριακό οικοσύστημα με πολιτικές διαχείρισης εξελικτικών γεγονότων – δηλαδή, για κάθε πιθανή αλλαγή που μπορεί να φτάσει σε μια ενότητα λογισμικού (πίνακας, όψη, ή ερώτηση), η εν λόγω ενότητα είναι επισημειωμένη με πολιτικές που καθορίζουν την αντίδρασή της στο γεγονός της αλλαγής. Θεωρούμε ένα γράφημα που αναπαριστά το όλο οικοσύστημα (το οποίο ονομάζουμε Γράφημα Αρχιτεκτονικής του οικοσυστήματος) και διερευνούμε το πώς είναι εφικτή η διαχείριση της διάδοσης των αλλαγών στο γράφημα της αρχιτεκτονικής. Πρακτικά, κάθε αλλαγή που καταφτάνει σε μια ενότητα ενεργοποιεί και μια πολιτική, η οποία με τη σειρά της είτε τερματίζει τη διάδοση του γεγονότος, είτε καθορίζει την περαιτέρω μετάδοση σε επόμενες ενότητες. Δείχνουμε ότι ο μηχανισμός που προτείνουμε τερματίζει πάντα και στο τέλος κάθε διάδοσης, κάθε ενότητα είναι επισημειωμένη με μία μοναδική κατάσταση (status), ασχέτως της σειράς με την οποία καταφτάνουν σε αυτή οι αλλαγές που διαδίδονται στο γράφημα.

Στο εν λόγω άρθρο συνοψίζονται πρώιμα ερευνητικά αποτελέσματα (με τη μορφή σύντομου άρθρου), όπως παρουσιάστηκαν στο workshop HotSWUp:

George Papastefanatos, Panos Vassiliadis, Alkis Simitsis. **Propagating Evolution Events in Data-Centric Software Artifacts**. In 3rd Workshop on Hot Topics in Software Upgrades (*HotSWUp 2011*), in conjunction with ICDE 2011, pp. 162-167, April 16, 2011, Hannover, Germany

[1] Η παρούσα έρευνα έχει συγχρηματοδοτηθεί από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο - ΕΚΤ) και από εθνικούς πόρους μέσω του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» του Εθνικού Στρατηγικού Πλαισίου Αναφοράς (ΕΣΠΑ) - Ερευνητικό Χρηματοδοτούμενο Έργο: Θαλής. Επένδυση στην κοινωνία της γνώσης μέσω του Ευρωπαϊκού Κοινωνικού Ταμείου.

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(Ω_1,Ω_2,...,Ω_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.

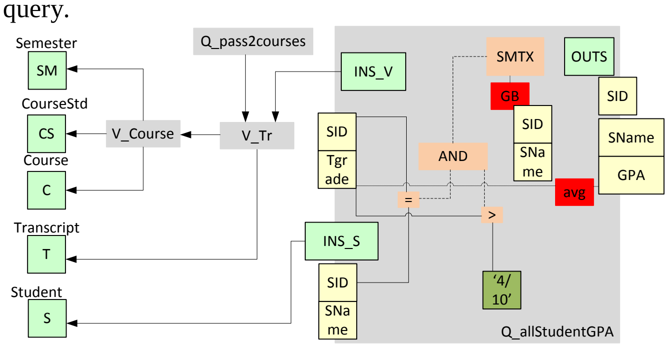


Figure 1. Architecture graph of an example database (our running example)

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:* $V_E \times V_P \rightarrow V_S$ (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.

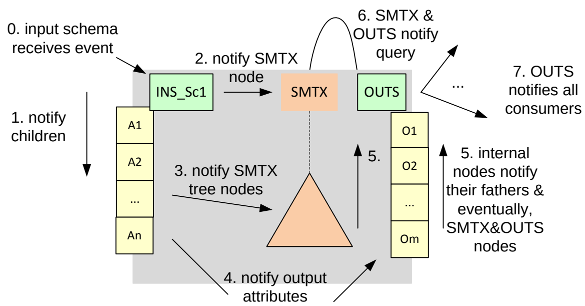


Figure 2. Protocol for the message propagation mechanism

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 nodes*. 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 input 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}[2]. 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},

[2] 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.

Automating the Adaptation of Evolving Data-Intensive Ecosystems

Petros Manousis[1], Panos Vassiliadis[1], and George Papastefanatos[2]

[1] Dept. of Computer Science University of Ioannina (Hellas)
{pmanousi, pvassil}@cs.uoi.gr
[2] Research Center "Athena" (Hellas)
gpapas@imis.athenainnovation.gr

Abstract. Data-intensive ecosystems are conglomerations of data repositories surrounded by applications that depend on them for their operation. To support the graceful evolution of the ecosystem's components we annotate them with policies for their response to evolutionary events. In this paper, we provide a method for the adaptation of ecosystems based on three algorithms that (i) assess the impact of a change, (ii) compute the need of different variants of an ecosystem's components, depending on policy conflicts, and (iii) rewrite the modules to adapt to the change.

Keywords: Evolution, data-intensive ecosystems, adaptation

1 Introduction

Data-intensive ecosystems are conglomerations of databases surrounded by applications that depend on them for their operation. Ecosystems differ from the typical information systems in the sense that the management of the database profoundly takes its surrounding applications into account. In this paper, we deal with the problem of facilitating the evolution of an ecosystem without impacting the smooth operation or the semantic consistency of its components.

Observe the ecosystem of Figure 1. On the left, we depict a small part of a university database with three relations and two views, one for the information around courses and another for the information concerning student transcripts. On the right, we isolate two queries that the developer has embedded in his applications, one concerning the statistics around the database course and the other reporting on the average grade of each student. If we were to delete attribute C_NAME, the ecosystem would be affected in two ways : (a) *syntactically*, as both the view V_TR and the query on the database course would crash, and, (b) *semantically*, as the latter query would no longer be able to work with the same selection condition on the course name. Similarly, if an attribute is added to a relation, we would like to inform dependent modules (views or queries) for the availability of this new information.

The means to facilitate the graceful evolution of the database without damaging the smooth operation of the ecosystem's applications is to allow all the

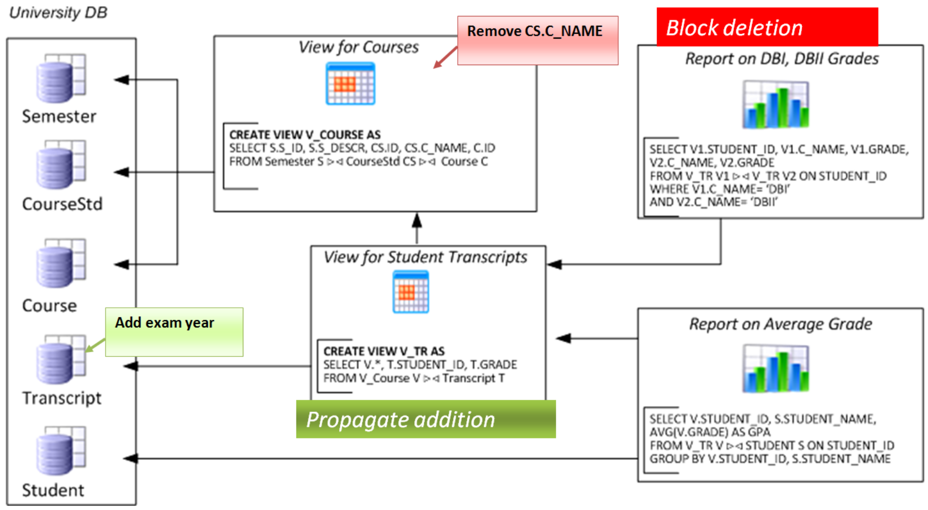


Fig. 1. Managing the adaptation of a University-DB Ecosystem.

involved stakeholders to *register veto's or preferences*: for example, we would like to allow a developer to state that she is really adamant on retaining the structure and semantics of a certain view. In our method, we can annotate a *module* (i.e., relation, view or query) with a *policy* for each possible event that it can withstand, in one of two possible modes: (a) *block*, to veto the event and demand that the module retains its previous structure and semantics, or, (b) *propagate*, to allow the event and adapt the module to a new internal structure.

In this paper, we model ecosystems as graphs annotated with policies for responding to evolutionary events (Sec. 2) and we address the problem of identifying (a) what parts of the ecosystem are affected whenever we test a potential change and (b) how will the ecosystem look like once the implications of conflicting policies are resolved and the graph is appropriately rewritten (Sec. 3). Related work in ecosystem adaptation has provided us with techniques for view adaptation [6], [3], [11] that do not allow the definition of the policies for the adaptation of the ecosystem modules. Our previous work [8] has proposed algorithms for impact assessment with explicit policy annotation; however, to the best of our knowledge, there is no method that allows both the impact assessment and the rewriting of the ecosystem's modules along with correctness guarantees.

We implemented our method in a *what-if analysis* tool, Hecataeus[3] where all stakeholders can pre-assess the impact of possible modifications before actually performing them, in a way that is loosely coupled to the ecosystem's components. Our experimentation with ecosystems of different policies and sizes (Sec. 4) indicates that our method offers significant effort gains for the maintenance team of the ecosystem and, at the same time, scales gracefully.

---

[3] http://www.cs.uoi.gr/ pvassil/projects/hecataeus/

2 Formal Background

Our modeling technique, extending [8], uniformly represents all the components of an ecosystem as a directed graph which we call the *Architecture Graph* of the ecosystem. Fig. 2 visually represents the internals of the modules of Fig. 1. To avoid overcrowding the figure, we omit different parts of the structure in different modules; the figure is self-explanatory on this.

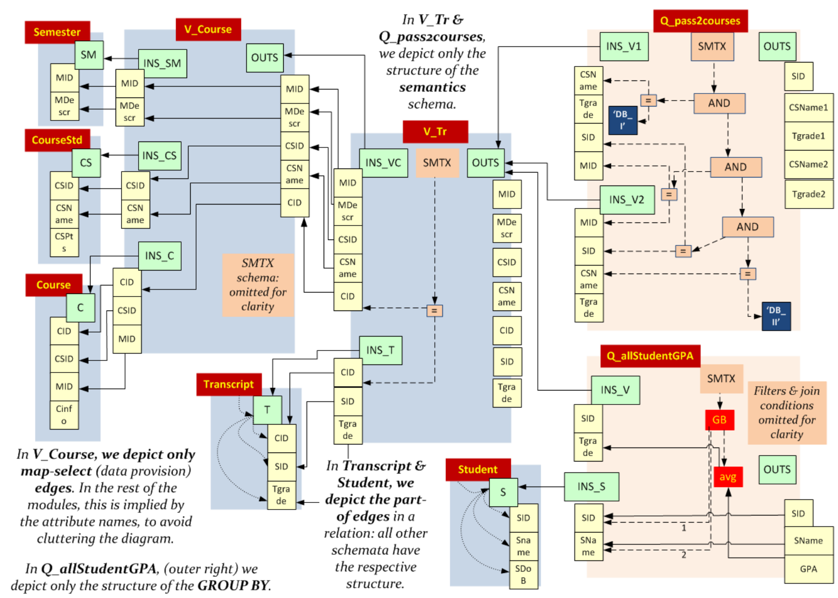


Fig. 2. A subset of the graph structure for the University-DB Ecosystem.

Modules. A module is a semantically high level construct of the ecosystem; specifically, the modules of the ecosystem are relations, views and queries. Every module defines a scope recursively: every module has one or more schemata in its scope (defined by part-of edges), with each schema including components (e.g., the attributes of a schema or the nodes of a semantics tree) linked to the schema also via part-of edges. In our model, all modules have a well defined scope, "fenced" by input and output schemata.

Relations. Each relation includes a node for the relation per se, a node for its (output) schema and a node for each for its attributes; all connected via the aforementioned part-of edges.

Queries. The graph representation of a Select - Project - Join - Group By (SPJG) query involves a new node representing the query, named *query node*, linked to the following schemata:

1. a set of *input schemata nodes* (one for every table appearing in the FROM clause). Each input schema includes the set of attributes that participate

in the syntax of the query (i.e., SELECT, WHERE and GROUP BY clauses, etc.). Each input attribute is linked via a provider, *map-select* edge to the appropriate attribute of the respective provider module.

2. an *output schema node* comprising the set of attributes present in the SELECT clause. The output attributes are linked to the appropriate input attributes that populate them through *map-select* edges, directing from the output towards the input attributes.
3. a *semantics* node as the root node for the sub-graph corresponding to the semantics of the query (specifically, the WHERE and GROUP-BY part).

We accommodate WHERE clauses in conjunctive normal form, where each atomic formula is expressed as: (i) Ω *op* constant, or (ii) Ω *op* Ω', or (iii) Ω *op* Q where Ω, Ω' are attributes of the underlying relations, Q is a nested query, and operator *op* belongs to the set $\{<, >, =, \leq, \geq, \neq, IN, EXISTS, ANY\}$). The entire WHERE clause is mapped to a tree, where (i) each atomic formula is mapped to a subtree with an operator node for *op* linked with *operand* edges pointing to the operand nodes of the formulae and (ii) nodes for the Boolean operators (AND, OR) connect with each other as well as with the operators of the atomic formulae via the respective operand edges. The GROUP BY part is mapped in the graph via (i) a node GB, to capture the set of attributes acting as the aggregators and (ii) one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN. For the aggregators, we use edges directing from the semantics node towards the GB node that are labeled *group-by.* The GB node is linked to the respective input attributes acting as aggregators with *group-by* edges, which are additionally tagged according to the order of the aggregators; we use an identifier i to represent the i-th aggregator. Moreover, for every aggregated attribute in the query's output schema, there exists a *map-select* edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective input attribute.

Views. Views are treated as queries; however the output schema of a view can be used as input by a subsequent view or query module.

Summary. A summary of the architecture graph is a zoomed-out variant of the graph at the schema level with provider edges only among schemata (instead of attributes too).

Events. We organize the events that can be tested via our method in the following groups.

- *Events at relations.* A relation can withstand deletion and renaming of itself as well as addition, deletion and renaming of its attributes.
- *Events at views and queries.* A view can withstand the deletion and renaming of itself, the addition, deletion or renaming of its output attributes and the update of the view's semantics (i.e., the modification of the WHERE clause of the respective SQL query that defines the view).

Policies. As already mentioned, the policy of a node for responding to an incoming event can be one of the following: (a) PROPAGATE, which means that the node is willing to adapt in order to be compatible with the new structure and semantics of the ecosystem, or, (b) BLOCK, which means that the node wants to retain the previous structure and semantics. We can *assign policies* to all the nodes of the ecosystem via a language [5] that provides guarantees for the complete coverage of *all* the graph's nodes along with syntax conciseness and customizability. The main idea is the usage of rules of the form <receiver node> : on <event> then <policy>, both at the default level –e.g.,

VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;

and at the node-specific level (overriding defaults) –e.g.,

V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;

3 Impact Assessment and Adaption of Ecosystems

The goal of our method is to assess the impact of a hypothetical event over an architecture graph annotated with policies and to adapt the graph to assume its new structure after the event has been propagated to all the affected modules. Before any event is tested, we topologically sort the modules of the architecture graph (always feasible as the summary graph is acyclic: relations have no cyclic dependencies and no query or view can have a cycle in their definition). This is performed once, in advance of any impact assessment. Then, in an on-line mode, we can perform what-if analysis for the impact of changes in two steps that involve: (i) the detection of the modules that are actually affected by the change and the identification of a status that characterizes their reaction to the event, and, (ii) the rewriting of the graph's modules to adapt to the applied change.

3.1 Detection of affected nodes and status determination

The assessment of the impact of an event to the ecosystem is a process that results in assigning every affected module with a status that characterizes its policy-driven response to the event. The task is reduced in (a) determining the affected modules in the correct order, and, (b) making them assume the appropriate status. Algorithm *Status Determination* (Fig. 3) details this process. In the following, we use the terms *node* and *module* interchangeably.

1. Whenever an event is assessed, we start from the module over which it is assessed and visit the rest of the nodes by following the topological sorting of the modules to ensure that a module is visited after *all* of its data providers have been visited. A visited node assesses the impact of the event internally (cf., "intra-module processing") and obtains a *status*, which can be one of the following: (a) BLOCK, meaning that the module is requesting that it remains structurally and semantically immune to the tested change and blocks the event (as its immunity obscures the event from its data consumers), (b) PROPAGATE, meaning that the modules concedes to adapt to the change and propagate the event to any subsequent data consumers, or, (c) retain a NO_STATUS status, already assigned by the topological sort, meaning that the module is not affected by the change.

Input: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a global queue Q that facilitates the exchange of messages between modules.
Output: A list of modules $Affected\ Modules \subseteq \mathbf{V_s}$ that were affected by the event and acquire a status other than NO_STATUS.

```
Q={original message}, Affected Modules = ∅;
For All node ∈ Gs(Vs, Es)
    node.status = NO_STATUS;
EndFor
While (size(Q) > 0)
    visit module (node) in head of Q;
    insert node in Affected Modules list;
    get all messages, Messages, that refer to node;
    SetStatus(node, Messages);
    If (node.status == PROPAGATE) Then
        insert node.Consumers Messages to the Q;
EndWhile
Return Affected Modules;

Procedure SetStatus(Module, Messages)
Consumers Messages = ∅;
For All Message ∈ Messages
    decide status of Module;
    put messages for Module's consumers in Consumers Messages;
EndFor
```

Fig. 3. Algorithm STATUS DETERMINATION

2. If the status of the module is PROPAGATE, the event must be propagated to the subsequent modules. To this end, the visited module prepares *messages* for its data consumers, notifying them about its own changes. These messages are pushed to a common *global message queue* (where messages are sorted by their target module's topological sorting identifier).
3. The process terminates whenever there are no more messages and no more modules to be visited.

Intra-module processing. Whenever visited, a module starts by retrieving from the common queue *all* the messages (i.e., events) that concern it. It is possible that more than one message exist in the global queue for a module: e.g., with the deletion of an attribute that was used both in the output schema of a module and in the semantics schema of a module, the module should inform its consumers that (a) the attribute was deleted and (b) its semantics has changed. The processing of the messages is performed as follows:

1. First, the module probes its schemata for their reaction to the incoming event, starting from the input schemata, next to the semantics and finally to the output schema. Naturally, relations deal only with the output schema.
2. Within each schema, the schema has to probe both itself and its contained nodes (attributes) for their reaction to the incoming event. At the end of this process, the schema assumes a status as previously discussed.

3. Once all schemata have assumed status, it is the output schema of the module that decides the reaction of the overall module; if any of the schemata raises a veto (BLOCK) the module assumes the BLOCK status too; otherwise, it assumes the PROPAGATE status.

Theoretical guarantees. Previous models of Architecture Graphs ([8]) allow queries and views to directly refer to the nodes representing the attributes of the involved relations. Due to the framing of modules within input and output schemata and the topological sorting, in [7] we have proved that the process (a) terminates and (b) correctly assigns statuses to modules.

3.2 Query and view rewriting to accommodate change

Once the first step of the method, *Status Determination*, has been completed and each module has obtained a status, the problem of adaptation would intuitively seem simple: each module gets rewritten if the status is PROPAGATE and remains the same if the status is BLOCK. This would require only the execution of the *Graph Rewrite* step – in fact, one could envision cases where *Status Determination* and *Graph Rewrite* could be combined in a single pass. Unfortunately, although the decision on *Status Determination* can be made locally in each module, taking into consideration only the events generated by previous modules and the local policies, the decision on rewriting has to take extra information into consideration. This information is not local, and even worse, it pertains to the subsequent, consumer modules of an affected module, making thus impossible to weave this information in the first step of the method, *Status Determination*. The example of Fig. 4 is illustrative of this case.

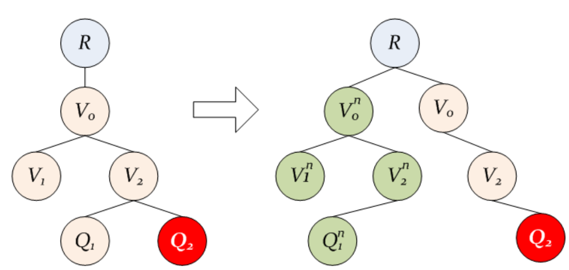


Fig. 4. Block rewriting example

In the example of Figure 4, we have a relation R and a view V_0 defined over the relation R. Two views (V_1 and V_2) use V_0 in order to get data. V_2 is further used by two queries (Q_1 and Q_2). The database administrator wants to change V_0, in a way that all modules depending on V_0 are going to be affected by that change (e.g., attribute deletion, for an attribute common to all the modules of the example). Assume now that all modules except Q_2 accept to adapt to the change, as they have a PROPAGATE policy annotation. Still, the vetoing Q_2

must be kept immune to the change; to achieve this we must retain the previous version of *all* the nodes in the path from the origin of the evolution (V_0) to the blocking Q_2. As one can see in the figure, we now have two variants of V_0 and V_2: the new ones (named V_0^n and V_2^n) that are adapted to the new structure of V_0 – now named V_0^n – and the old ones, that retain their name and are depicted in the rightmost part of the figure. The latter are immune to the change and their existence serves the purpose of correctly defining Q_2.

Input: An architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a list of modules $Affected\ modules$, affected by the event, and the $Initial\ Event$ of the user.
Output: Annotation of the modules of $Affected\ modules$ on the action needed to take, and specifically whether we have to make a new version of it, or, implement the change that the user asked on the current version

1. For All $Module \in\ Affected\ modules$
2. If($Module.status\ ==\ BLOCK$) Then
3. CheckModule($Module$, $Affected\ modules$, $Initial\ Event$);
4. mark $Module$ not to change; //Blockers do not change
5. EndFor

Procedure CheckModule($Module$, $Affected\ modules$, $Initial\ Event$)
If($Module$ has been marked) Then return; //Notified by previous block path
If ($Initial\ Event == ADD\_ATTRIBUTE$)
Then mark $Module$ to apply change on current version; //Blockers ignore provider addition
Else mark $Module$ to keep current version as is and apply the change on a clone;
For All $Module\ provider \in\ Affected\ modules$ feeding $Module$
CheckModule($Module\ provider$, $Affected\ modules$, $Initial\ Event$); //Notify path
EndFor

Fig. 5. Algorithm PATH CHECK

The crux of the problem is as follows: if a module has PROPAGATE status and none of its consumers (including both its immediate and its transitive consumers) raises a BLOCK veto, then both the module and all of these consumers are rewritten to a new version. However, if any of the immediate consumers, or any of the transitive consumers of a module raises a veto, then *the entire path towards this vetoing node must hold two versions of each module*: (a) the new version, as the module has accepted to adapt to the change by assuming a PROPAGATE status, and, (b) the old version in order to serve the correct definition of the vetoing module.

To correctly serve the above purpose, the adaptation process is split in two steps. The first of them, *Path Check*, works from the consumers towards the providers in order to determine the number of variants (old and new) for each module. Whenever the algorithm visits a module, if its status is BLOCK, it starts a reverse traversal of the nodes, starting from the blocker module towards the module that initialized the flow and marks each module in that path (a) to keep its present form and (b) prepare for a cloned version (identical copy) where the rewriting will take place. The only exception to this rewriting is when the module of the initial message is a relation module and the event is an attribute deletion, in which case a BLOCK signifies a veto for the adaptation of the relation.

Input: A list of modules *Affected modules*, knowing the number of versions they have to retain, initial messages of *Affected modules*
Output: Architecture graph after the implementation of the change the user asked

```
If(any of Affected modules has status BLOCK) Then
    If(initial message started from Relation module type AND event == DELETE_ATTRIBUTE) Then Return;
    Else
    For All (Module ∈ Affected modules)
        If(Module needs only new version) Then
            proceed with rewriting of Module;
            connect Module to new providers; //new version goes to new path
        Else
            clone Module; //clone module, to keep both versions
            connect cloned Module to new providers; //clone is the new version
            proceed with rewriting of cloned Module;
    EndFor
Else
For All Module ∈ Affected modules
    proceed with rewriting of Module //no blocker node
EndFor
```

Fig. 6. Algorithm Graph Rewrite

Finally, all nodes that have to be rewritten are getting their new definition according to their incoming events. Unfortunately, this step cannot be blended with *Path Check* straightforwardly: *Path Check* operates from the end of the graph backwards, to highlight cases of multiple variants; rewriting however, has to work from the beginning towards the end of the graph in order to correctly propagate information concerning the rewrite (e.g., the names of affected attributes, new semantics, etc.). So, the final part of the method, *Graph Rewrite*, visits each module and rewrites the module as follows:

- If the module must retain only the new version, once we have performed the needed change, we connect it correctly to the providers it should have.
- If the module needs both the old and the new versions, we make a clone of the module to our graph, perform the needed change over the cloned module and connect it correctly to the providers it should have.
- If the module retains only the old version, we do not perform any change.

4 Experiments

We assessed our method for its usefulness and scalability with varying graph configurations and policies; in this section, we report our findings.

Experimental setup. We have employed TPC-DS, version 1.1.0 [10] as our experimental testbed. TPC-DS is a benchmark that involves star schemata of a company that has the ability to *Sell* and receive *Returns* of its *Items* with the following ways: (a) the *Web*, or, (b) a *Catalog*, or, (c) directly at the *Store*. Since the Hecataeus' parser could not support all the advanced SQL constructs of TPC-DS, we employed several auxiliary views and slight query modifications.

Graphs and Events. To test the effect of graph size to our method's efficiency, we have created 3 graphs with gradually decreasing number of query modules: (a) a large ecosystem, *WCS*, with queries using all the available fact tables, (b) an ecosystem *CS*, where the queries to WEB_SALES have been removed, and (c) an ecosystem *S*, with queries using only the STORE_SALES fact table. The event workload consists of 51 events simulating a real case study of the Greek public sector. See Fig. 7 for an analysis of the module sizes within each scenario and the workload (listing the percentage of each event type as *pct*).

Policies. We have annotated the graphs with policies, in order to allow the management of evolution events. We have used two "profiles": (a) *MixtureDBA*, consisting of 20% of the relation modules annotated with BLOCK policy and (b) *MixtureAD*, consisting of 15% of the query modules annotated with BLOCK policy. The first profile corresponds to a developer-friendly DBA that agrees to prevent changes already within the database. The second profile tests an environment where the application developer is allowed to register veto's for the evolution of specific applications (here: specific queries). We have taken care to pick queries that span several relations of the database.

	Graph size		
	S	CS	WCS
Queries	27	68	89
Views	25	48	95
Relations	25	25	25
Sum	**77**	**141**	**218**

Event type	*pct*
Attribute Add	37.3%
Attribute Rename	43.2%
Attribute Del	13.7%
Relation Rename	1.9%
View alter semantics	3.9%

Fig. 7. Experimental configuration for the TPC-DS ecosystem

Experimental Protocol. We have used the following sequence of actions. First, we annotate the architecture graph with policies. Next, we sequentially apply the events over the graph – i.e., each event is applied over the graph that resulted from the application of the previous event. We have performed our experiments with hot cache. For each event we measure the elapsed time for each of the three algorithms, along with the number of affected, cloned and adapted modules. All the experiments have been performed in a typical PC with an Intel Quad core CPU at 2.66GHz and 1.9GB main memory.

Effectiveness. How useful is our method for the application developers and the DBA's? We can assess the effort gain of a developer using the highlighting of affected modules of Hecataeus compared to the situation where he would have to perform all checks by hand as the *fraction of Affected Modules of the ecosystem*. This gain, expressed via the $\%AM$ metric amounts to the percentage of useless checks the user would have made. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. Formally, $\%AM$ is given by the Equation 1.

$$\%AM = 1 - \frac{\#Affected\ Modules}{\#(Queries \cup Views)} \tag{1}$$

	%AM – Mixture AD			*%AM – Mixture DBA*		
	S	CS	WCS	S	CS	WCS
min	21%	35%	30%	60%	78%	84%
avg	**89%**	**91%**	**92%**	**97%**	**96%**	**97%**
max	100%	100%	100%	100%	100%	100%

Fig. 8. Effectiveness assessment as fraction of affected modules (%AM)

The results depicted in Fig. 8 demonstrate that the effort gains compared to the absence of our method are significant, as, on average, the effort is around 90% in the case of the AD mixture and 97% in the case of the DBA mixture. As the graph size increases, the benefits from the highlighting of affected modules that our method offers, increase too. Observe that in the case of the DBA case, where the flooding of events is restricted early enough at the database's relations, the minimum benefit in all 51 events ranges between 60% - 84%.

Effect of policy to the execution time. In the case of *Mixture DBA* we follow an aggressive blocking policy that stops the events early enough, at the relations, before they start being propagated in the ecosystem. On the other hand, in the case of *Mixture AD*, we follow a more conservative annotation approach, where the developer can assign blocker policies only to some module parts that he authors. In the latter case, it is clear that the events are propagated to larger parts of the ecosystem resulting in higher numbers of affected and rewritten nodes. If one compares the execution time of the three cases of the AD mixture in Fig. 9 with the execution time of the three cases of the DBA mixture the difference is in the area of one order of magnitude. It is however interesting to note the internal differences: the status determination time is scaled up with a factor of two; the rewriting time, however is scaled up by a factor of 10, 20 and 30 for the small, medium and large graph respectively!

Another interesting finding concerns the **internal breakdown of the execution time** in each case. A common pattern is that *path check is executed very efficiently*: in all cases it stays within 2% of the total time (thus practically invisible in the graphic). In the case of the AD mixture, the analogy between the status determination and the graph rewriting part starts from a 24% - 74% for the small graph and ends to a 7% - 93% for the large graph. In other words, *as the events are allowed to flow within the ecosystem, the amount of rewriting increases with the size of the graph*; in all cases, it dominates the overall execution time. This is due to the fact that rewriting involves memory management (module cloning, internal node additions, etc) that costs much more than the simple checks performed by *Status Determination*. In the case of the DBA mixture, however, where the events are quickly blocked, the times are not only significantly smaller, but also equi-balanced: 57% - 42% for the small graph (*Status Determination* costs more in this case) and 49% - 50% for the two other graphs. Again, this is due to the fact that the rewriting actions are the time consuming ones and therefore, their reduction significantly reduces the execution time too.

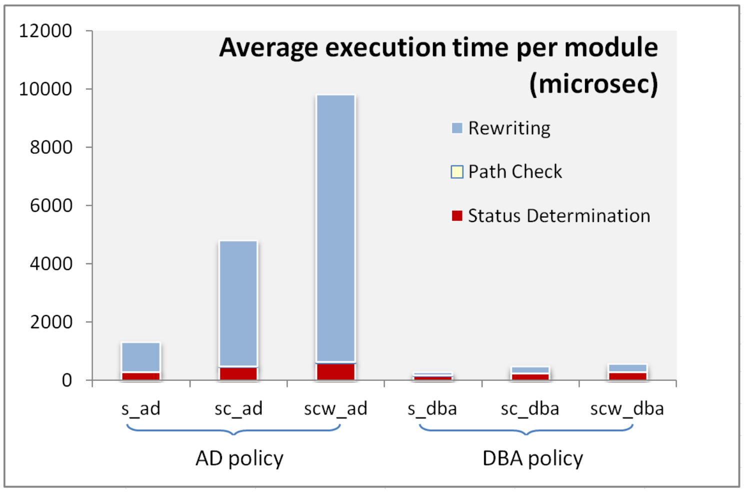


Fig. 9. Efficiency assessment for different policies, graph sizes and phases

Effect of graph size to the execution time. To assess the impact of graph size to the execution time one has to compare the three different graphs to one another within each policy. In the case of the AD mixture, where the rewriting dominates the execution time, there is a linear increase of both the rewriting and the execution time with the graph size. On the contrary, the rate of increase drops in the case of the DBA mixture: when the events are blocked early, the size of the graph plays less role to the execution time.

Overall, the main lesson learned from these observations is that the annotation of few database relations significantly restricts the rewriting time (and consequently the overall execution time) when compared to the case of annotating modules external to the database. In case the rewriting is not constrained early enough, then the execution cost grows linearly with the size of the ecosystem.

5 Related work

For an overview of the vast amount of work in the area of evolution, we refer the interested reader to an excellent, recent survey [4]. We also refer the interested reader to [9] for a survey of efforts towards bidirectional transformations. Here, we scope our discussion to works that pertain to the adaptation of data-intensive ecosystems.

Data-intensive ecosystems' evolution. Research activity around data-intensive ecosystems has been developed around two tools, Hecataeus and Prism. Hecataeus [8] models ecosystems as Architecture Graphs and allows the definition of policies, the impact assessment of potential changes and the computation of graph-theoretic properties as metrics for the vulnerability of the graph's nodes to change. The impact assessment mechanism was first introduced in [8] and subsequently modified in [7]. PRISM++ [2] lets the user define his policies about imminent changes. The authors use ICMOs (Integrity Constraints Modification Operators) and SMOs (Schema Modification Operators) in order to rewrite the queries/views in a way that the results of the query/view are the same as before.

View/schema mapping rewriting. Nica et al., [6] make legal rewritings of views affected by changes and they primarily deal with the case of relation deletion by finding valid replacements for the affected (deleted) components via a meta-knowledge base (MKB) that keeps meta-information about attributes and their join equivalence attributes on other tables in the form of a hyper-graph. Gupta et al., [3] redefine a *materialized* view as a sequence of primitive local changes in the view definition. On more complex adaptations, those local changes can be pipelined in order to compute the new view contents incrementally and avoid their full re-computation. Velegrakis, et al., [11], deal with the maintenance of a set of mappings in an environment where source and target schemata are integrated under schema mappings implemented via SPJ queries. Cleve et al., [1] introduce mappings among the applications and a conceptual representation of the database, again mapped to the database logical model; when the database changes, the mappings allow to highlight impacted areas in the source code.

Comparison to existing approaches.As already mentioned, the annotation of the ecosystem with policies imposes the new problem of maintaining different replicas of view definitions for different consumers; to the best of our knowledge, this is the first time that this problem is handled in a systematic way. Interestingly, although the existing approaches make no explicit treatment of policies, they differ in the implicit assumptions they make. Nica et al., operating mainly over virtual views [6], actually block the flooding of a deletion event by trying to compensate the deletion with equivalent expressions. At the same time, they do not handle additions or renamings. Velegrakis et al. [11] move towards the same goal but only for SPJ queries. On the other hand, Gupta et al., [3], working with materialized views, are focused to adapting the contents of the views, in a propagate-all fashion. A problem coming with a propagate-all policy is that the events might affect the semantical part of the views/queries (WHERE clause) without any notification to the involved users (observe that the problem scales up with multiple layers of views defined over other views).

Compared to previous editions of Hecataeus [8], this work reports on the first implementation of a status determination mechanism with correctness guarantees. The management of rewritings via the path checking to handle conflicting policies and the adaptation to accommodate change are completely novel.

6 Conclusions and Future Work

In this paper we have addressed the problem of adapting a data-intensive ecosystem in the presence of policies that regulate the flow of evolution events. Our method allows (a) the management of alternative variants of views and queries and (b) the rewriting of the ecosystem's affected modules in a way that respects the policy annotations and the correctness of the rewriting (even in the presence of policy conflicts). Our experiments confirm that the adaptation is performed efficiently as the size and complexity of the ecosystem grow. Future work can address the assessment of complicated events, the visualization of the ecosystem and the automatic suggestion of policies.

Acknowledgments. This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. Cleve, A., Brogneaux, A.F., Hainaut, J.L.: A conceptual approach to database applications evolution. In: 29th Intl. Conf. on Conceptual Modeling (ER), Vancouver, Canada. pp. 132–145 (2010)
2. Curino, C., Moon, H.J., Deutsch, A., Zaniolo, C.: Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. PVLDB 4(2), 117–128 (2010)
3. Gupta, A., Mumick, I.S., Rao, J., Ross, K.A.: Adapting materialized views after redefinitions: techniques and a performance study. Information Systems 26(5), 323–362 (2001)
4. Hartung, M., Terwilliger, J.F., Rahm, E.: Recent Advances in Schema and Ontology Evolution. In: Bellahsene, Z., Bonifati, A., Rahm, E. (eds.) Schema Matching and Mapping, pp. 149–190. Springer (2011)
5. Manousis, P.: Database evolution and maintenance of dependent applications via query rewriting. MSc. Thesis, Dept. of Computer Science, Univ. Ioannina. (February 2013), http://www.cs.uoi.gr/∼pmanousi/publications.html
6. Nica, A., Lee, A.J., Rundensteiner, E.A.: The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In: 6th Intl. Conf. on Extending Database Technology (EDBT 1998), Valencia, Spain. pp. 359–373 (1998)
7. Papastefanatos, G., Vassiliadis, P., Simitsis, A.: Propagating evolution events in data-centric software artifacts. In: ICDE Workshops. pp. 162–167 (2011)
8. Papastefanatos, G., Vassiliadis, P., Simitsis, A., Vassiliou, Y.: Policy-Regulated Management of ETL Evolution. J. Data Semantics 13, 147–177 (2009)
9. Terwilliger, J.F., Cleve, A., Curino, C.: How clean is your sandbox? - towards a unified theoretical framework for incremental bidirectional transformations. In: 5th Intl. Conf. Theory and Practice of Model Transformations (ICMT), Prague, Czech Rep. pp. 1–23 (2012)
10. Transaction Processing Performance Council: The New Decision Support Benchmark Standard. http://www.tpc.org/tpcds/default.asp (April 2012)
11. Velegrakis, Y., Miller, R.J., Popa, L.: Preserving mapping consistency under schema changes. VLDB Journal 13(3), 274–293 (2004)

Noname manuscript No.
(will be inserted by the editor)

Impact Analysis and Policy-Conforming Rewriting of Evolving Data-Intensive Ecosystems

Petros Manousis · Panos Vassiliadis · George Papastefanatos

the date of receipt and acceptance should be inserted later

Abstract Data-intensive ecosystems are conglomerations of data repositories surrounded by applications that depend on them for their operation. In this paper, we address the problem of performing what-if analysis for the evolution of the database part of a data-intensive ecosystem, in order to identify what other parts of an ecosystem are affected by a potential change in the database schema, and how will the ecosystem look like once the change has been performed, while, at the same time, retaining the ability to regulate the flow of events. We model the ecosystem as a graph, uniformly covering relations, views and queries as nodes and their internal structure and interdependencies as the edges of the graph. We provide a simple language to annotate the modules of the graph with policies for their response to evolutionary events in order to regulate the flow of events and their impact by (i) vetoing ("blocking") the change in parts that the developers want to retain unaffected and (ii) allowing ("propagating") the change in parts that we need to adapt to the new schema. Our method for the automatic adaptation of ecosystems is based on three algorithms that automatically (i) assess the impact of a change, (ii) compute the need of different variants of an ecosystem's components, depending on policy conflicts, and (iii) rewrite the modules to adapt to the change. We theoretically prove the coverage of the language, as well as the termination, consistency and confluence of our algorithms, and experimentally verify our methods effectiveness and efficiency.

Keywords Evolution, data-intensive ecosystems, adaptation

Petros Manousis
Dept. of Computer Science and Engineering, University of Ioannina (Hellas)
E-mail: pmanousi@cs.uoi.gr

Panos Vassiliadis
Dept. of Computer Science and Engineering, University of Ioannina (Hellas)
E-mail: pvassil@cs.uoi.gr

George Papastefanatos
Institute for the Management of Information Systems, Research Center "Athena" (Hellas)
E-mail: gpapas@imis.athena-innovation.gr

1 Introduction

A data-intensive ecosystem is a conglomeration of software modules that includes (a) at least one central database (typically, but not obligatorily, relational), and, (b) a set of software applications that require access to the central database via queries embedded in their code. The distinctive feature of data-intensive ecosystems is the cohesive management of databases and applications – plainly put, the management of the database has to profoundly takes its surrounding applications into account (and vice versa). In this paper, we deal with the problem of facilitating the evolution of an ecosystem without impacting the smooth operation or the semantic consistency of its components.

To operate smoothly, an ecosystem must withstand change gracefully. Software maintenance comprises 60% of the resources spent on building and operating a software system [20] and thus, it is of utmost importance for a system's life-cycle. In this context, the management of changes in a data-centric ecosystem is an important problem. In this paper, we extend the state of the art concerning several research questions in the area of managing the evolution of data-intensive ecosystems.

What does evolution of data-intensive ecosystems comprise? We start by example – here are a few examples of possible changes:

– A certain attribute of the schema of a view is about to be deleted, as the administrator wants to simplify the definition of the view
– A new attribute is added to a relation, because new content is available
– The WHERE clause of a view is modified with an extra condition, to account for a new definition of the view's contents

Taking the aforementioned examples at a more abstract level, we claim that *evolution is of significance if it affects the syntactic correctness, the semantic validity, the operational effectiveness, or the administrative overhead of a data-intensive ecosystem*. The most disordering (and also visible) type of impact is *syntactic impact*: in this case, a change might drive parts of the ecosystem to be syntactically inconsistent and thus fail. A deleted attribute might cause applications using it to crash. In this case, the applications' developers have to take care of the change: pinpoint its impact in their code, assess the necessity for the existence of this information in the applications and modify their applications accordingly. If things go wrong, this might even require negotiations with the DBAs to restore the deleted attribute. Second, applications can be affected *semantically*. If a new attribute is added to a relation it is possible that it contains important information that applications should be exploiting (and thus, have to be synchronized to the new contents of the relation). If the semantics of a view change, then the data delivered at the view's clients are different than the ones delivered before: in this case, the developers of the affected queries and applications would have to be notified and decide on whether the queries have to adapt to the new semantics of the view, or, they would have to retain the old semantics (again leading to the problem of compensating the change performed by the DBAs). A third type of impact (that falls outside the scope of this paper) involves the effect of a change to the performance and administrations of the ecosystem. Dropping an index may result in a large number of queries running unacceptably slow or moving a table may result in making less space for other tables to perform their insertions.

In all these occasions, we observe that a change performed by the DBA team can have several side-effects both for the team itself, the developers of applications of the ecosystem and the end-users. *The problem in all the aforementioned events is that the change is performed before assessing its impact over the ecosystem.* Therefore, addressing the impact assessment problem in advance of a potential change can be really valuable.

How can we assess the impact of a change in a data intensive ecosystem? Is it possible to regulate change in a data-intensive ecosystem? In this paper, we improve the state of the art with concrete results for the problem of impact assessment. We follow the model of *Architecture Graphs* [18,19] that capture all the interdependencies between the constructs of databases and the application queries via a graph. The graph models constraints, attributes, relations, views and queries along with their internal structure as the nodes of the graph. The edges of the graph denote dependency for data provision (e.g., between a view and a relation that populates it with data), part of relationships (e.g., between a relation and its attributes) and semantic relationships (e.g., the construction of the WHERE clause of a query as a tree of expressions). This way, the Architecture Graph models all the components of a data-intensive ecosystem in a uniform way. One of the main utilities of the Architecture Graph is that it facilitates impact assessment for potential changes in the ecosystem: whenever a potential change is tested over the Architecture Graph, the graph allows us to identify the area of impact by recursively following edges between affected nodes. Practically speaking, each node has to assume a status concerning its reaction to an event that we test; once a status is assumed, subsequent nodes of the graph have to be notified too.

At the same time, we are not helpless in managing potential changes in the core of the ecosystem. If an application developer is really adamant on retaining the structure and semantics of a database view, it is possible that this requirement is incorporated in the Architecture Graph, to prevent possible modifications? As previous research [16,18] has demonstrated, it is possible to regulate the flow of events by annotating the modules of the Architecture Graph with policies, i.e., rules for handling events. Specifically, we can annotate a module (i.e., relation, view or query) with a policy for each possible event that it can withstand, in one of two possible modes: (a) *block*, to veto the event and demand that the module retains its previous structure and semantics, or, (b) *propagate*, to allow the event and make the module adapt with an updated internal structure. Once the adaptation is complete, the module is also responsible for igniting the recursive notification of all the affected software modules in the graph.

To make the discussion a little more concrete, we present an evolving data-intensive ecosystem in Figure 1. On the left, we depict a small part of a university database with three relations and two views, one for the information around courses and another for the information concerning student transcripts. On the right, we isolate two queries that the developer has embedded in his applications, one concerning the statistics around the database course and the other reporting on

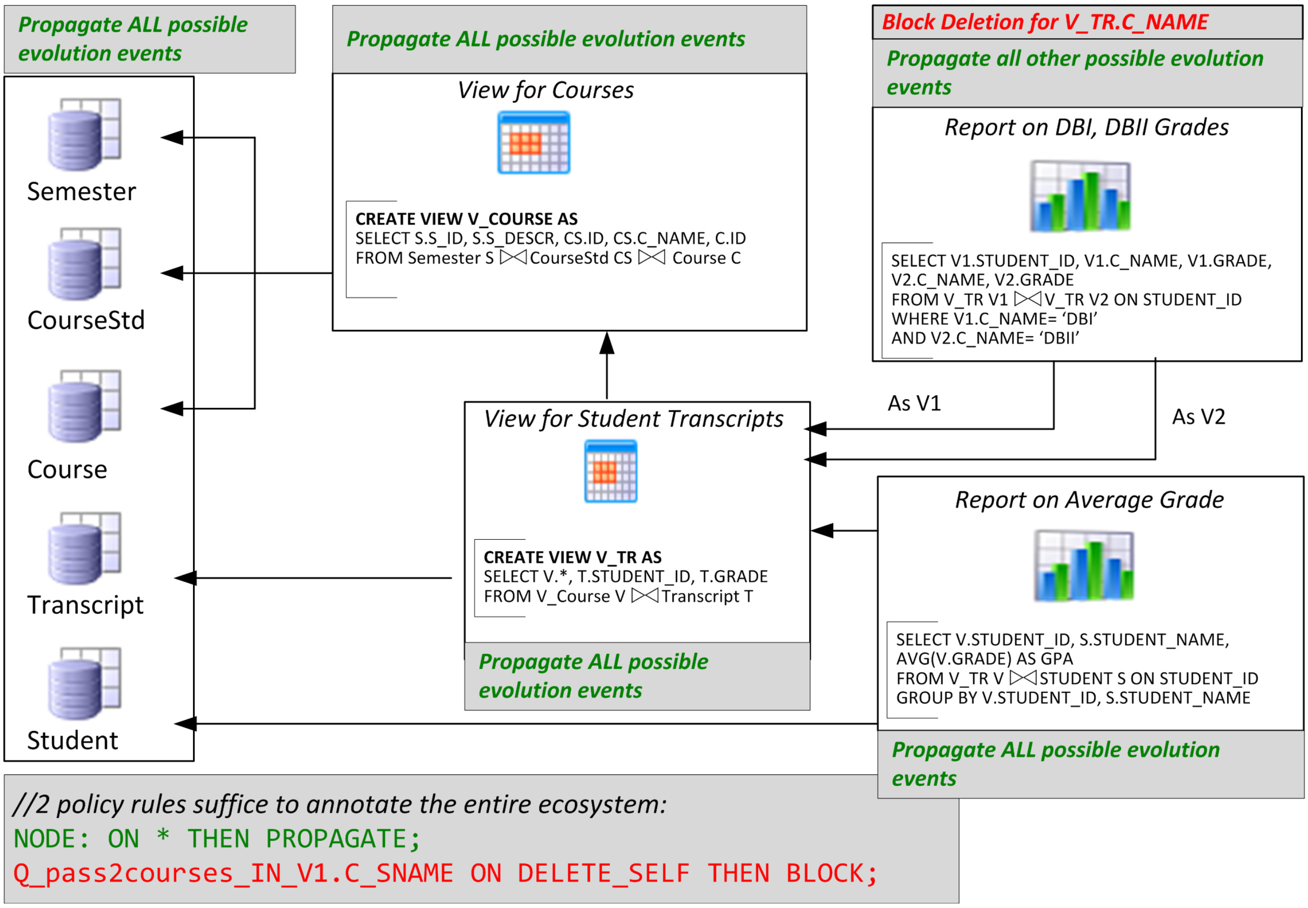


Fig. 1 An exemplary University-DB Ecosystem, annotated with policies.

the average grade of each student. If we were to delete attribute C_NAME, the ecosystem would be affected in two ways : (a) *syntactically*, as both the view V_TR and the query on the database course would crash, and, (b) *semantically*, as the latter query would no longer be able to work with the same selection condition on the course name. Similarly, if an attribute is added to a relation, we would like to inform dependent modules (views or queries) for the availability of this new information. Observe the two policy rules at the bottom of the figure. The first one dictates that every node of the graph adapts to any evolutionary event that appears in the future. The rule uses two shorthands: the term `NODE` refers to all the nodes of the graph and the term $*$ refers to any potential event that arrives. The second rule overrides the first global policy by stating that the report on the upper right has a veto over the deletion of one of the attributes exported by the view on student transcripts (V_TR). In Figure 1, we have used a lightly shaded box to show how these rules are assigned to each module.

Once the impact of a change has been assessed, is it possible to see how the ecosystem will look like if the change is eventually performed? Even with the presence of policies, it is possible that a potential modification in the database affects several queries and views that are willing to accept it and adapt to the new structure or semantics of the database. The problem becomes more complicated whenever a change ignites different reactions – e.g., some of the affected queries are willing to adapt whereas others assume a vetoing status. Then, the question that has to be answered is "what will the new structure and semantics of all the affected modules look like?". As we will show, the answer to the question is not straightforward and unfortunately, the state of the art in ecosystem adaptation is not sufficient to address it. Specifically, although previous work in ecosystem adaptation has provided us with techniques for view adaptation [13], [7], [26], the existing works do not allow the definition of policies for the adaptation of the ecosystem modules. At the same time, our own previous work [18] has proposed algorithms for impact assessment with explicit policy annotation but without the mechanisms to perform the rewriting of the ecosystem. Overall, to the best of our knowledge, there is no method that allows both the impact assessment and the rewriting of the ecosystem's modules along with correctness guarantees.

To address this shortcoming, the core result of this paper is the provision of algorithms that identify which parts of the ecosystem are affected by a potential change and perform the rewriting of affected modules to adapt to it, while fulfilling all the constraints imposed by the -possibly conflicting- policies of all affected modules. Specifically, our method works in the following three steps:

1. Impact assessment. Given a potential event, a status determination algorithm makes sure that the nodes of the ecosystem are assigned a status concerning (a) whether they are affected by the event or not and (b) what their reaction to the event is (block or propagate).
2. Conflict resolution and calculation of variants. Assume a view used by two queries is altered. Assume also that the first query vetoes the change and requires the structure and semantics of the old view to remain, whereas the second concedes to the change and states it will adapt to the new structure and semantics of the view. *The co-existence of blocker and adapter data consumers of an affected module signifies the need to retain both the old and the new version of the module, whenever this is possible.* To this end, we introduce an algorithm that checks the affected parts of the graph in order to highlight affected nodes with whether they will adapt to a new version or retain both their old and new variants.
3. Module Rewriting. Once the status and number of variants has been determined for the modules of the graph, we need to implement the rewritings. This is heavily dependent upon the nature of the event (obviously, a query adapts differently to the removal of an attribute and differently to the addition of an attribute, let alone changes in semantics). Our algorithm visits affected modules sequentially and performs the appropriate restructurings of nodes and edges.

Coming back to our motivating example, let's see what happens when the DBA of the ecosystem tries to delete attribute C_NAME from the intermediate view V_COURSE. As instructed by its policy, the view "agrees" to adapt to the event and adopts a *Propagate* status. Then, it notifies its consumer V_TR which also agrees and pushes the event to its consumers, specifically, Q_pass2courses which vetoes the event and assumes a *Block* status and Q_allStudentGPA which is actually unaffected by the event after it self-examines whether it is affected. The rest of the modules of the graph, and specifically, the source relations, have a status *NO_STATUS* as the propagation of the event has never reached them. The depiction of the status determination part is shown in the left part of Figure 2. Then, our method detects a conflict, as the view V_TR decides to adapt to the event in contrast to the veto from the application developer of the query Q_pass2courses. Once this conflict is detected, a cloning mechanism is initiated to satisfy both requirements. The result is depicted in the right part of Figure 2. The query Q_pass2courses retains the old definition of both views (i.e., the entire backwards path till the node initiating the event), whereas the two views are cloned and these clones (depicted in darker colors in the figure) are adapted to satisfy the requirement set by the DBA.

We have implemented our method in a *what-if analysis* tool, Hecataeus[1] where all stakeholders can pre-assess the impact of possible modifications before actually performing them, in a way that is loosely coupled to the ecosystem's components. We have assessed our method (Sec. 5) over ecosystems of increasing size and complexity and also varied the policy assignments in order to assess the method's scale up with size and the effect of the policy annotation to the method's usefulness. Our first experimental goal involved assessing the effectiveness of our method, i.e., the benefits introduced by our method concerning the effort performed by the application developers and administrators of the ecosystem. The results indicate that in the absence of our system, the typical developer would have to perform at least 21% of routine, useless checks to views and queries that are not related to the event at all; on average, the number of useless checks is located in the area of 90%-97%. A second observation has to do with the amount of rewriting: in all occasions, there have been several modules that had to be rewritten. Although the average numbers are not particularly high, ranging from 2 to 13 modules depending on the experimental setup, the maximum numbers are quite high and, in any case, the automation of the work, equips the involved stakeholders with correctness guarantees that would otherwise be non-existent. Another significant observation has to do with the conciseness of the policy annotation rules. The number of policy rules is practically equivalent to the number of the exceptions to the default policies (resulting in just a handful of rules in our experiments). In terms of efficiency, all the experiments show a completion of the tested changes as small fractions of a second. At the same time, the chosen policy significantly affects the spreading of the

[1] http://www.cs.uoi.gr/~pvassil/projects/hecataeus/

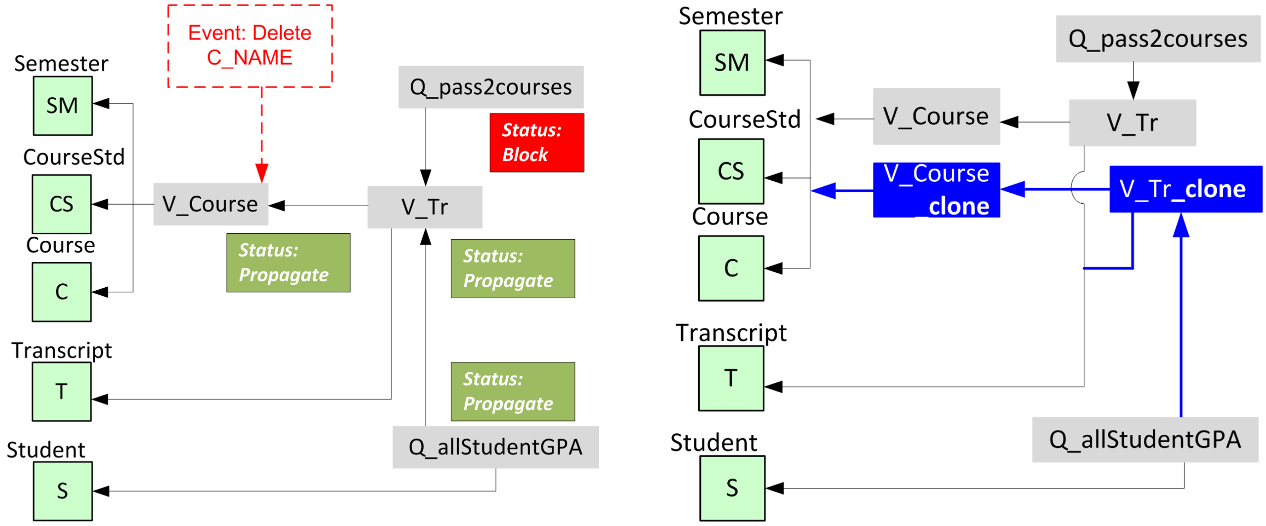


Fig. 2 Impact analysis (left) and ecosystem rewriting (right) for an event on our exemplary excosystem

impact of a change over the ecosystem: a policy with early containment of the event (by blocking it inside the database) can be an order of magnitude faster than a policy that blocks changes at the queries only. At the same time, the graph size is linearly related to the time needed to complete the impact analysis and rewriting task. Overall, our experimentation with ecosystems of different policies and sizes indicates that our method offers significant effort gains for the maintenance team of the ecosystem and, at the same time, is executed fast and scales gracefully.

Roadmap. The structure of this paper is as follows. In Section 2, we give the background modeling for the architecture graph, policies and events. In Section 3, we discuss impact assessment, conflict resolution and, module rewriting. In Section 4, we prove the theoretical guarantees of our approach. In Section 5, we present the experimental assessment of our method. In Section 6, we present related work. We conclude in Section 7, along with insights for future work.

2 Formal Background

To assess the impact of a potential change over the data centric ecosystem, we construct a graph of modules (relations, queries and views) where data consuming nodes are linked with edges to their providers. Whenever an event is applied over a module, the module has to assess the impact of the event and notify its consumers. This recursive process allows us to assess the impact of the event over the entire ecosystem. Naturally, to facilitate this process, we need to establish a formal model for the main constituents of the problem and its solution. So, before proceeding with the algorithmic parts of the adaptation process, in this Section, we present the formal background for the modeling of architecture graphs, along with the space of possible events and policy annotations. First, we present how relations, views and queries construct the architecture graph of the ecosystem. Then, we move on to present the space of possible events that can be applied to the nodes of the graph, either directly by the user (initiating the entire process of assessing the impact of an event) or transitively, as modules affected by the event notify other modules that depend on them for the change. Moreover, in order to regulate the propagation of events over the graph, we present the language for policy annotations, along with its semantics and the rules for policy overriding.

2.1 Architecture graph

Our modeling technique, following [15], represents all the aforementioned database constructs as a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, which we call *Architecture Graph* of the ecosystem. For the reader who is not interested in all the formalities, the following quick summary along with Figures 2 and 3 should be sufficient to allow the understanding of our graph modeling.

- *Relations, views and queries (or else modules) come with a subgraph, that includes (a) a node for the module itself, (b) a set of input schemata for views and queries, used for linking these modules with their data providers, (b) an output schema for the data exported by the module and (d) a semantics schema for any filtering or restructuring taking* *place inside a view or a query (WHERE, GROUP BY, etc).*

– *Input and output schemata include their respective attributes; semantics schemata include a tree representing the logical expression of the WHERE clause and a list of groupers for the GROUP-BY clause, in case these exist in a query or view.*
– *Edges of the graph signify dependency on data provision: at the schema level, input and output schemata are linked with data dependency edges from data consumers towards data providers; the respective holds for attributes of the schemata too. Note that this mechanism applies both between modules (intermodule edges) and within the same module (intramodule edges). Semantic-related edges are also used for the constructs related to the semantics schema within views and queries.*

The reader who wants to skip the detailed description of the graph can jump to Section 2.2. If this is not the case, our deliberations proceed with a presentation of the components of the architecture graph as follows. We start with the high level constructs, such as relations and queries, which we call modules of the Architecture Graph, and then we move on to discuss their main properties. Fig. 3 visually represents the internals of the modules of Fig. 1. To avoid overcrowding the figure, we omit different parts of the structure in different modules; the figure is self-explanatory on this.

Modules. A module is a semantically high level construct of the ecosystem; specifically, the modules of the ecosystem are relations, views and queries. Every module defines a scope recursively: every module has one or more schemata in its scope (defined by part-of edges), with each schema including components (e.g., the attributes of a schema or the nodes of a semantics tree) linked to the schema also via part-of edges. In our model, all modules have a well defined scope, "fenced" by input and output schemata.

Relations. Each relation R (A_1, A_2,. . . , A_n) in the database schema is represented as a directed graph, which comprises:

1. a node, R, representing the relation;
2. *an output schema node*, R_SCHEMA, representing the relation's output schema;
3. n attribute nodes $A_{i=1...n}$, one for each of the attributes and,
4. n+1 schema relationships $E_{i=1...(n+1)}$, directing from the schema node towards the attribute nodes, indicating that the attribute belongs to the relation's output schema and one directing from the relation node towards the output schema node indicating that the output schema belongs to the relation.

In our reference examples, we have the following relations, whose graphs are depicted in Fig. 3): relation $Semester(MID, MDescr)$ standing for information on semesters, relation $CourseStd$(*csid*,*csname*,*cspts*) standing for information on courses, relation $Course$(*cid*, *csid*, *mid*) standing for information on courses on semesters, relation $Student$(*sid*, *sname*) standing for information on students,and relation $Transcript$(*cid*, *sid*, *tgrade*) standing for information on the enrollment of students to courses and their grades.

Queries and Views. The graph representation of a Select - Project - Join - Group By (SPJG) query involves:

1. a new node representing the query, named *query node*,
2. a set of *input schemata nodes* (one for every table appearing in the FROM clause). Each input schema includes the set of attributes that participate in the syntax of the query (i.e., SELECT, WHERE clause, etc.)
3. an *output schema node* comprising the set of attributes present in the SELECT clause.
4. a *semantics* node as the root node for the sub-graph corresponding to the semantics of the query (specifically, the WHERE and GROUP-BY part), and,
5. *attribute nodes* belonging to the various input and output schemata of the query.

The query graph is therefore a directed graph connecting the query node with the high level schemata and semantics nodes. The query node contains an edge towards every schema it possesses. The schema nodes are connected to their attributes via part-of relationships. In order to explain the internal structure of a query, we structure our presentation of the query's graph in terms of its SQL parts: SELECT, FROM, WHERE, and GROUP BY, each of which is eventually mapped to a sub-graph.

Select part. Each query is assumed to own an output schema that comprises the attributes, either with their original or with alias names, appearing in the SELECT clause. In this context, the SELECT part of the query maps the respective attributes of the input schemata to the attributes of the query's output schema through *map-select* edges, directing from the output attributes towards the input schema attributes.

From part. The FROM clause of a query can be regarded as the relationship between the query and the relations (or views) involved in this query. Thus, the relations included in the FROM part are combined with the input schemata of the query node through *from* edges, directing from the nodes of the appropriate input schemata towards the output schema nodes of the

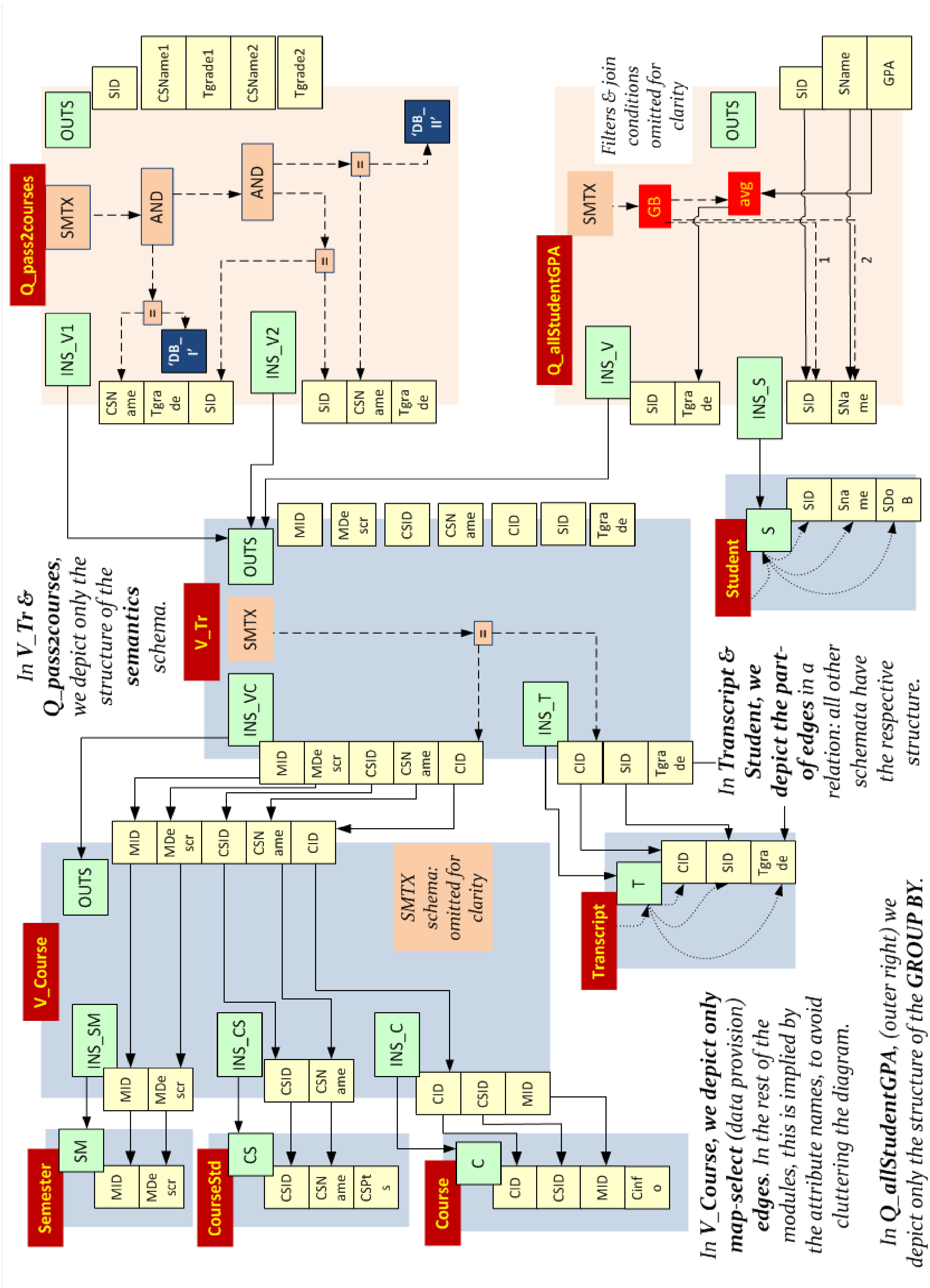


Fig. 3 A subset of the graph structure for the University-DB Ecosystem.

relation/view nodes. The input schemata of the query comprise only the attributes of the respective relations that participate in any way in the query; the attributes of the input schemata are connected to the respective attributes of the provider relations or views via map-select relationships.

Where part. We assume that the WHERE clause of a query is in conjunctive normal form. Thus, we introduce a directed edge, called *where* edge, starting from the semantics node of a query towards an operator node corresponding to the conjunction of the highest level. Then, there is a tree of nodes hanging from this conjunction involving condition nodes (to be defined right away). The edges are operand relationships as mentioned above among binary comparators, boolean operators, input attributes and constants. In Fig. 4 we depict the graph of $Q\_pass2courses$ query which performs a self-join over view V_TR and presents a report of the students and their grades, that enrolled in both DB_I and DB_II courses. A tree, starting from the $SMTX$ node, describes the conditions of the selected tuples. Initially, we take the tuples on which the name of the first course is equal to DB_I, then we filter them and take the ones that have the same SID for $V1$ and $V2$. Finally, we filter those results and take the ones on which the name of the second course is equal to DB_II.

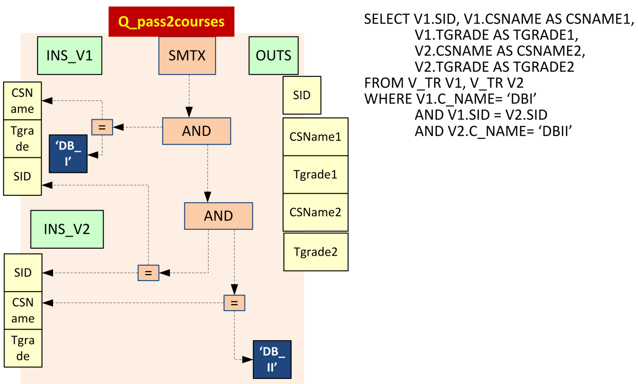


Fig. 4 The graph of the semantics schema for the Q_pass2courses query

We consider three classes of atomic conditions that are composed through the appropriate usage of an operator *op* belonging to the set of classic binary operators, *op* (e.g., $<$, $>$, $=$, $\leq$, $\geq$, $\neq$, IN, $EXISTS$, ANY): (i) Ω *op* constant, (ii) Ω *op* Ω', and (iii) Ω *op* Q where Ω, Ω' are attributes of the underlying relations and Q is a query. A condition node is used for the representation of the condition. Graphically, the node is tagged with the respective operator and it is connected to the operand nodes of the conjunct clause through the respective operand relationships, O. Composite conditions are easily constructed by tagging the condition node with a Boolean operator (e.g., AND or OR) and the respective edges to the conditions composing the composite condition.[2]

Group By part. The GROUP BY part is mapped in the graph via (i) a node GB, to capture the set of attributes acting as the aggregators and (ii) one node per aggregate function labeled with the name of the employed aggregate function; e.g., COUNT, SUM, MIN. For the aggregators, we use edges directing from the semantics node towards the GB node that are labeled *group-by*. The GB node is linked to the respective input attributes acting as aggregators with *group-by* edges, which are additionally tagged according to the order of the aggregators; we use an identifier i to represent the i-th aggregator. Moreover, for every aggregated attribute in the query's output schema, there exists a *map-select* edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective input attribute. In Fig. 5 we depict the graph of $Q\_allStudentGPA$ query. In the left part we have the edges that connect the output attributes with their providers in the input schemata. We have SID and $SName$ that are using as their providers the SID and $SName$ of $Semester$ relation, whilst the GPA is the $AVERAGE$ aggregate function of $TGrade$ coming from V_TR view. In the right part of the figure we have the GB node, which is used to describe the "group by" clause of the query. The numbers on the edges depict the order of the groupers, meaning that first we group by SID and then with $SName$ columns. Additionally, in $MSTX$ node, we have a node that describes that in the resulting tuples of the query, the SID that comes from V_TR view and the SID that comes from $Semester$ relation should be equal to each other.

Views. Views are treated as queries; however the output schema of a view can be used as input by a subsequent view or query module.

Summary. A summary of the architecture graph is a zoomed-out variant of the graph at the schema level with provider edges only among schemata (instead of attributes too). Formally, a summary is a directed acyclic graph $\mathbf{G}_s = (\mathbf{V}_s, \mathbf{E}_s)$, with $\mathbf{V}_s$ comprising the graph's module nodes (relations, views and queries) and

[2] Well-known constraints of database relations – i.e., primary/foreign key, unique, not null, and check constraints – can also be captured by this modeling technique. Foreign keys are subset relations of the source and the target attribute, check constraints are simple value-based conditions. Primary keys, which are unique-value constraints, are explicitly represented through a dedicated node tagged by their names and a single operand node.

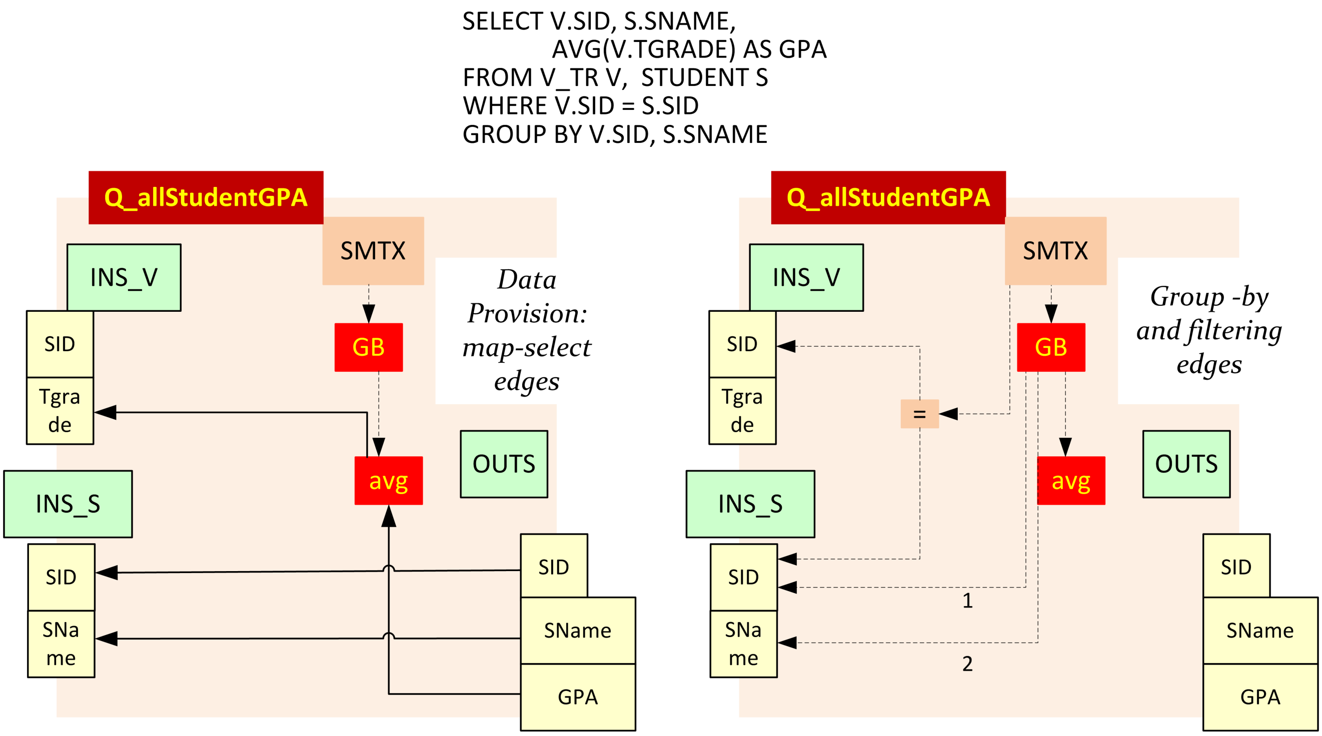


Fig. 5 The graph of a group-by query. To avoid confusion, we depict the edges in two snapshots of the graph: provider edges (left) and filtering and grouping edges (right).

$\mathbf{E}_s$ including an edge e(v,u) from a consumer module v to a provider module u if and only if there is an edge between an input schema of v and the output schema of u in the *Architecture Graph*. We can formally define different levels of zooming via summary graphs (i) at the schema level with input/output schemata, (ii) the module level as in Fig. 6.

2.2 Events

In this section we list the set of possible events that our method handles. We organize our discussion by classifying these events in three classes: (a) events pertaining to relations, (b) events pertaining to views or queries, and (c) events that occur as one module notifies another for the event it just received.

We can classify the impact of an event as *structural* whenever the exported schemata and their attributes are changed in terms of structure or naming. At the same time, the impact of an event is *semantic* whenever the internals of the semantics schema (i.e., the WHERE or the GROUP-BY clause of the respective SQL query) change.

Events that pertain to relations. The first class of events comprise changes on the schema of relations:

– *ADD_ATTRIBUTE*: in this case, a Relation should obtain another column
– *DELETE_ATTRIBUTE*: in this case, a Relation should drop a column
– *RENAME_ATTRIBUTE*: in this case, a Relation should rename a column
– *DELETE_SELF*: in this case, a Relation will be deleted
– *RENAME_SELF*: in this case, a Relation will be called with a new name from now on.

Events that pertain to views and queries. The second class of events involve changes on the definitions of Views/Queries:

– *ADD_ATTRIBUTE*: in this case, a Query/View should have another attribute (column, aggregate function or value) in its output
– *DELETE_ATTRIBUTE*: in this case, a Query/View should have less attributes in its output
– *RENAME_ATTRIBUTE*: in this case, an attribute is going to be called with a new name from now on
– *DELETE_SELF*: in this case, a View will be deleted (deleting queries is of no impact to the ecosystem anyway)
– *RENAME_SELF*: in this case, a View will be called with a new name from now on

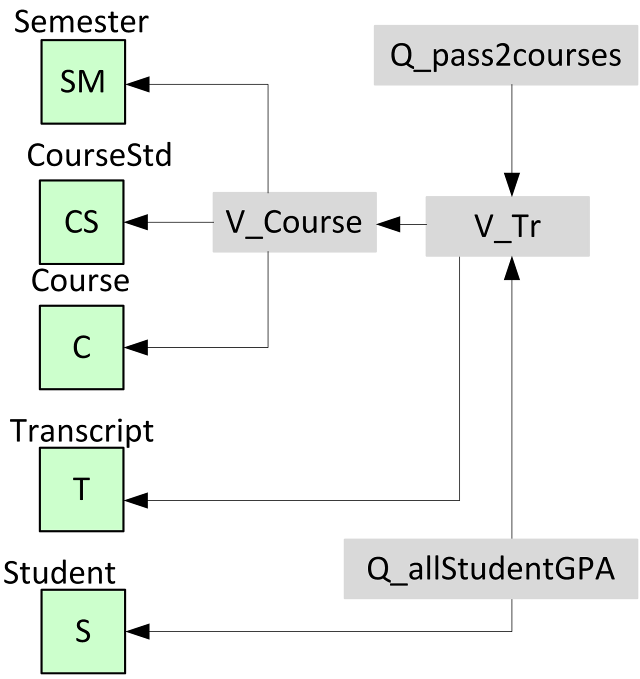


Fig. 6 The Summary Graph of the University-DB Ecosystem.

– *ALTER_SEMANTICS*: in this case, a View is going to have another WHERE clause or another GROUP BY clause.

Events that pertain to the notification of a change between modules. As we will see in Section 3, whenever a module has decided on its reaction against the incoming events, it assumes a status and notifies subsequent modules. Thus, besides the aforementioned events, we need to support the following list of events that accrue from the flow of an event to the graph.

– *ADD_ATTRIBUTE_PROVIDER*: this event is generated by a module in order to inform its consumers that the module has added an attribute to its output schema.
– *DELETE_PROVIDER*: this event is generated by a module in order to inform its consumers that this module has deleted one or all its attributes
– *RENAME_PROVIDER*: this event is generated by a module to inform its consumers that the module itself or one of the attributes that exist in output schema of the module want to change their name.
– *ALTER_SEMANTICS*: this event is generated by a module to inform its consumers that the semantics (as described previously: change of WHERE or/and GROUP BY clause) of a module have changed.

2.3 Policies

Our basic tool for the regulation of the propagation of an event's impact to the entire ecosystem is the ability to block further propagation at certain modules which veto the event. To achieve this, we employ *policies* that annotate the ecosystem's modules with predefined reactions to all possible incoming events they can receive. This way, whenever a node receives an event that concerns either itself or its constituents (e.g., the attributes of a schema), the node has already been instructed by the ecosystem's administrator on its reaction to the incoming event. The policy of a node for responding to an incoming event can be one of the following:

– PROPAGATE, which means that the node accepts the change and will adapt to the new reconfiguration of the ecosystem, or,
– BLOCK, which means that the node wants to retain the previous structure and semantics.

Requirements for policy annotation. We wish to provide a *language* that annotates nodes with policies and addresses the following usability requirements:

– *Completeness*: how can we be sure that we can define annotations for *all* the possible events that can arrive to a node, for all the nodes of the ecosystem?
– *Conciseness*: can we achieve this *easily* and *correctly* with respect to the user's intentions, without having the user going to great lengths of coding in order to annotate the ecosystem with policies?

Completeness. To achieve completeness, we need to be sure that we can provide an annotation for all the nodes of the graph and for all the events that each node can receive. To achieve this, we proceed in two steps: (a) we explicitly define the *node-event space*, i.e., the space of all valid combinations of nodes and incoming events, and (b) for each node-event combination, we define the respective *policy rule* that characterizes the reaction of the node to this event.

To implement the first of the aforementioned steps, we exhaustively enumerate all combinations of events and nodes (see Table 1). Observe, that Table 1 provides *a complete characterization of events that can arrive to a node organized per event type*. In Table 1, the rows (actually corresponding to the <receiver node> part of the above rule) are explained as follows:

1. [QUERY|VIEW].[OUT|IN].SELF standing for the node representing the output (input) schema of all queries (views)
2. [QUERY|VIEW].[OUT|IN].ATTRS standing for the nodes representing the attributes of the output (input) schema of all queries (views)

3. [QUERY|VIEW].SMTX.SELF standing for the root node of the semantics tree of all queries (views)
4. RELATION.OUT.[SELF|ATTRS] standing for the node representing the output schema of all relations (or its attributes)

Language for policies. Then, to implement the translation of the node-event space to policy rules, we need to provide a language that determines the policy for each event that appears to each node. The language that we introduce is used to *assign policies* to all the nodes of the ecosystem with guarantees for the complete coverage of *all* the graph's nodes along with syntax conciseness and customizability. In a nutshell, the main idea is the usage of rules of the form <receiver node [type]> : on <event> then <policy>, both at the default level –e.g.,

VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;

and at the node-specific level (overriding defaults) –e.g.,

V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;

Before formally specifying the syntax of the policy language, we first discuss the issues of language conciseness and rule overriding.

Conciseness. The observant reader might wonder on the reasoning behind providing rules both at the node type and the node level. The reason is conciseness: we want to avoid annotating the graph in a node per node, event per event basis. To this end, we provide a language that comes with the simple semantics that unless otherwise specified (see the next paragraph), each node-event pair implements the respective *node type - event type* policy. The default, fixed list, comprising 33 rules that can be derived from the entries of Table 1 is depicted in Fig. 7. In Section 4, we provide a proof for the language completeness in Theorem 1.

Still, even so, the number of rules needed for completeness could be considered too large by some users. To this end, we provide some additional rules that simplify our policy language. These rules come as syntactic sugar to our language. specifically, we introduce two syntactic sugar extensions as follows:

– the * notation for events allows the user to specify that a specific module type (i.e., all relations/views/queries of the ecosystem) of a specific node is annotated with the same policy for all the events that occur to it. In other words, the * notation signifies "for any incoming event"
– the NODE notation specifies that all nodes of the ecosystem, independently of their type, are annotated with the specified policy for the specified event (if, of course, the event pertains to the node).

Of course, the combination of the two syntactic shorthands is also allowed. Thus, we end up with the following list of syntactic sugar extensions:

<moduleType>: ON * THEN <policy>;
This rule groups the events that a module type (RELATION, VIEW, QUERY) can receive and sets the policy for all these events to <policy>.

<namedNode>: ON * THEN <policy>;
This rule finds the node that is specified by name <namedNode> and sets the policy for all these events to <policy>.

NODE: ON <event> THEN <policy>; This rule annotates all the nodes of the graph that can receive the specified event (named <event>) with the same policy, namely <policy>.

NODE: ON * THEN <policy>; This rule actually replaces the group of the 33 rules to one simple rule, saying that regardless of the event, the policy is uniformly set to <policy>.

Theorem 3 describes why these extra rules correctly cover up the needed events and correctly assign the policies to the nodes.

1. QUERY.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
2. QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
3. QUERY.OUT.SELF: on DELETE_SELF then <policy>;
4. QUERY.OUT.SELF: on RENAME_SELF then <policy>;
5. QUERY.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
6. QUERY.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;
7. QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
8. QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
9. QUERY.IN.SELF: on DELETE_PROVIDER then <policy>;
10. QUERY.IN.SELF: on RENAME_PROVIDER then <policy>;
11. QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
12. QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
13. QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
14. QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>;
15. VIEW.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
16. VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
17. VIEW.OUT.SELF: on DELETE_SELF then <policy>;
18. VIEW.OUT.SELF: on RENAME_SELF then <policy>;
19. VIEW.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
20. VIEW.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;
21. VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
22. VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
23. VIEW.IN.SELF: on DELETE_PROVIDER then <policy>;
24. VIEW.IN.SELF: on RENAME_PROVIDER then <policy>;
25. VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>;
26. VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>;
27. VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>;
28. VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>;
29. RELATION.OUT.SELF: on ADD_ATTRIBUTE then <policy>;
30. RELATION.OUT.SELF: on DELETE_SELF then <policy>;
31. RELATION.OUT.SELF: on RENAME_SELF then <policy>;
32. RELATION.OUT.ATTRIBUTES: on DELETE_SELF then <policy>;
33. RELATION.OUT.ATTRIBUTES: on RENAME_SELF then <policy>;

Fig. 7 The 33 combinations of events and node types that provide complete graph coverage; *policy* can be either BLOCK or PROPAGATE

Customizability and Rule Overriding. Whereas our small list of generic, default rules can cover all possible combinations of events and node

			ADD		DELETE		RENAME		ALTER
			ATTR	ATTR PROV	SELF	PROV	SELF	PROV	SMTX
QUERY	OUT	SELF	✓	✓	✓		✓		
		ATTRS			✓	✓	✓	✓	
	IN	SELF		✓		✓		✓	✓
		ATTRS				✓		✓	
	SMTX	SELF							✓
VIEW	OUT	SELF	✓	✓	✓		✓		
		ATTRS			✓	✓	✓	✓	
	IN	SELF		✓		✓		✓	✓
		ATTRS				✓		✓	
	SMTX	SELF							✓
RELATION	OUT	SELF	✓		✓		✓		
		ATTRS			✓		✓		

Table 1 The space of events that can be received by each node type

types, it is quite possible that we want to define a different reaction to the same event for different modules. For example, we might wish a certain view to block attribute addition, whereas we would allow another view to adapt to the same event. To facilitate this possibility we allow three *layers* of rules:

1. Layer 0: Rules that are applied to all the nodes of the *Architecture Graph* via the <NODE> notation.
2. Layer 1: General rules at the node type level, about *all* modules and their attributes.
3. Layer 2: Rules that apply to all the attributes of a *specific schema*.
4. Layer 3: Rules that apply to *specific attribute* nodes.

In our approach, the semantics of the layers of rules state that *each layer overrides the policy of its previous layers*. This way, if we have a default policy for all relations (layer 1) for a certain event (e.g., rename) we can customize the behavior of a specific relation to be different than the default by defining a specific rule for it (layer 2). Theorem 2 in Section 4 proves that our overriding mechanism assigns the correct policy to each node. Within each of the layers, the following ordering is imposed:

1. First, the * notation is transformed to the appropriate list of rules.
2. Second, any more specific rules override the * notation with their designated policies.

Language Syntax. The language's syntax comprises rules that abide to the following structure:

```
<receiver> : on <event> then <policy>
```

where:

1. <receiver> can be any of the ecosystem's node types
2. <event> can be any of the events that can arrive to an instance of this node type, either because the user initiated this as the starting event, or due to the propagation of the event in the ecosystem
3. <policy> can be either PROPAGATE or BLOCK

The above list of possible rules covers the node type layer (Layer 0), but not the two others. To this end, we introduce two extra kinds of potential values for the <receiver> part of the rules of our language.

1. <NAMED SCHEMA NODE>.ATTRIBUTES standing for the nodes representing the attributes of the <named schema node> of the graph.
2. <NAMED NODE> standing for the <named node> node of the graph.

The first of the two extra rules refers to all the attributes of a specific schema (layer 2), and, the second one refers to individual nodes of the graph (layer 3).

Reference Example. Returning to our reference example, the following text represents a set of rules of how policy rules should be written in order to have all nodes of the graph propagating all possible events for all modules, except for V_TR view, in which only the CID attribute will propagate any of its incoming events. Fig. 8 covers the first set of completeness-ensuring rules mentioned previously.

```
QUERY.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
QUERY.OUT.SELF: on DELETE_SELF then PROPAGATE;
QUERY.OUT.SELF: on RENAME_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
QUERY.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
QUERY.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
VIEW.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
VIEW.OUT.SELF: on DELETE_SELF then PROPAGATE;
VIEW.OUT.SELF: on RENAME_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on DELETE_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on RENAME_PROVIDER then PROPAGATE;
VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then PROPAGATE;
VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then PROPAGATE;
VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then PROPAGATE;
VIEW.SMTX.SELF: on ALTER_SEMANTICS then PROPAGATE;
RELATION.OUT.SELF: on ADD_ATTRIBUTE then PROPAGATE;
RELATION.OUT.SELF: on DELETE_SELF then PROPAGATE;
RELATION.OUT.SELF: on RENAME_SELF then PROPAGATE;
RELATION.OUT.ATTRIBUTES: on DELETE_SELF then PROPAGATE;
RELATION.OUT.ATTRIBUTES: on RENAME_SELF then PROPAGATE;
```

Fig. 8 Application of default rules for our reference example

Assuming now that the user wanted for the view V_TR to have a BLOCK policy for all possible events, Fig. 9 describes the set or rules needed to be issued after the general rules of Fig. 8.

Finally, the user decided that there is an exception to the rules of Fig. 9, and the attribute CID of the output schema of the V_TR module should have *again* a different policy than its siblings (switching again to PROPAGATE instead of BLOCK that was set in the previous set of rules), for its deletion. This is achieved by the set of policies depicted in Fig. 10.

Using the additional rules that simplify our policy language, the same example could be written as Fig. 11 describes.

```
V_TR_OUT.SELF: on ADD_ATTRIBUTE then BLOCK;
V_TR_OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_OUT.SELF: on DELETE_SELF then BLOCK;
V_TR_OUT.SELF: on RENAME_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on DELETE_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on RENAME_SELF then BLOCK;
V_TR_OUT.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_OUT.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on DELETE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on RENAME_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_IN_TRANSCRIPT.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.SELF: on DELETE_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.SELF: on RENAME_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.SELF: on ADD_ATTRIBUTE_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.ATTRIBUTES: on DELETE_PROVIDER then BLOCK;
V_TR_IN_V_COURSE.ATTRIBUTES: on RENAME_PROVIDER then BLOCK;
V_TR_SMTX.SELF: on ALTER_SEMANTICS then BLOCK;
```

Fig. 9 Overriding the default rules for a view in our reference example

```
V_TR_OUT.CID: on DELETE_SELF then PROPAGATE;
V_TR_OUT.CID: on DELETE_PROVIDER then PROPAGATE;
```

Fig. 10 Overriding the default rules for an attribute in our reference example

```
NODE: on * then PROPAGATE;
V_TR: on * then BLOCK;
V_TR_OUT.CID: on DELETE_SELF then PROPAGATE;
V_TR_OUT.CID: on DELETE_PROVIDER then PROPAGATE;
```

Fig. 11 Simplified policy language example

3 Impact Assessment and Adaptation of Ecosystems

The goal of our method is to assess the impact of a hypothetical event over an architecture graph annotated with policies and to adapt the graph to assume its new structure after the event has been propagated to all the affected modules. Before any event is tested, we topologically sort the modules of the architecture graph (always feasible as the summary graph is acyclic: relations have no cyclic dependencies and no query or view can have a cycle in their definition). This is performed once, in advance of any impact assessment. Then, in an on-line mode, we can perform what-if analysis for the impact of changes in two steps that involve: (i) the detection of the modules that are actually affected by the change and the identification of a status that characterizes their reaction to the event, and, (ii) the rewriting of the graph's modules to adapt to the applied change.

3.1 Topological sort

In order to make sure that the messages between modules are transferred in the right order from providers to consumers, we perform a topological sorting of the graph's modules prior to any other step. As Theorem 4 in Section 4 indicates, this is always feasible as the *Architecture Graph* does not contain cycles.

We follow a traditional approach to our topological sorting, which proceeds as follows: first we find the modules with zero incoming edges. These modules are removed from the examination set along with their outgoing edges, after being assigned a unique ID. This gives as a result a new set of modules with zero incoming edges. The algorithm stops when there are no more modules to visit. Relations have the smallest IDs, followed by Views and Queries.

Input: A summary of an architecture graph $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$ that comprises the modules of an architecture graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$.
Output: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, i.e. an annotation of the modules of $\mathbf{G_s}$ with a sequential id's, via a mapping $Y : \mathbf{V_s} \rightarrow \mathbf{N}$.

1. $notYetVisited \leftarrow V_s$;
2. $count = |V_s|$;
3. While ($notYetVisited$ not empty){
4.   ForEach ($v_i \in notYetVisited$){
5.   If (v_i has zero incoming edges){
6. $notYetVisited \leftarrow notYetVisited - v_i$;
7.   remove edges starting from v_i;
8. $v_i(id) \leftarrow count$;
9. $count \leftarrow count - 1$;
10.   }
11.   }
12. }

Fig. 12 Algorithm Topological sort

Observe that topological sorting of the graph is necessary, as opposed to a simple flooding of messages with events over the graph, due to the existence of multiple paths from data providers to their consumers (e.g., observe in Fig. 3 how the query Q_pass2Courses is fed by view V_TR via two paths, as it performs a self-join). Also, the existence of policies (which we detail in Section 3.3) require a strict order for visiting the nodes of the graph. Apart from the termination of our algorithms, we also want to guarantee the following properties:

– Confluence: each module in the graph will assume the same status, independently from the order of processing the incoming messages.
– Consistency: all the modules will be correctly rewritten.

In Theorem 6 and Theorem 11 in Section 4, we demonstrate why we need the principled visit of the nodes of the graph in a manner obeying the topological sort; had we not followed the topological sort it would be impossible to guarantee these correctness properties. Therefore, in the rest of our deliberations, unless explicitly mentioned, the propagation of the impact of events follows the topological sort.

Once the topological sort has been completed, we are ready to interactively work with the user towards highlighting the impact of a change and rewriting the graph accordingly. These two tasks are explained in the following two subsections.

3.2 Detection of affected nodes and status determination

The assessment of the impact of an event to the ecosystem is a process that results in assigning every affected module with a *status* that characterizes its policy-driven response to the event. In contrast to the policy, which is an annotation of each module with a directive on how to respond to a potential future event, a status is the decided reaction to an actual event, after it has reached the module. The status determination task is reduced in (a) determining the affected modules in the correct order, and, (b) making them assume the appropriate status. Algorithm *Status Determination* (Fig. 13) details this process. In the following, we use the terms *node* and *module* interchangeably.

1. Before assessing the event, all modules are set to status NO_STATUS. At the end of the algorithm's execution, the modules that will have retained this status will be the ones that have not been affected by the event.
2. Whenever an event is assessed,we start from the module over which it is assessed and visit the rest of the nodes by following the topological sorting of the modules to ensure that a module is visited after *all* of its data providers have been visited. A visited node assesses the impact of the event internally (cf., "intra-module processing") and, if there is reason to change its NO_STATUS status, due to incoming notifications from its providers, it obtains a new status, which can be one of the following: (a) BLOCK, meaning that the module is requesting that it remains structurally and semantically immune to the tested change and blocks the event (as its immunity obscures the event from its data consumers), (b) PROPAGATE, meaning that the module concedes to adapt and propagate the event to any subsequent data consumers.
3. If the status of the module is PROPAGATE, the event must be propagated to the subsequent modules. To this end, the visited module prepares *messages* for its data consumers, notifying them about its own changes. These messages are pushed to a common *global message queue* (where messages are sorted by their target module's topological sorting identifier).
4. The process terminates when there are no more messages and no more modules to be visited.

Input: A topologically sorted architecture graph summary $\mathbf{G_s}(\mathbf{V_s}, \mathbf{E_s})$, a global queue Q that facilitates the exchange of messages between modules.
Output: A list of modules $AffMdls \subseteq \mathbf{V_s}$ that were affected by the event and acquire a status other than NO_STATUS.

```
1.  Q={original message}, AffMdls = ∅;
2.  For All (node ∈ Gs(Vs, Es)){
3.      node.status = NO_STATUS;
4.  }
5.  While (size(Q) > 0){
6.      visit module (node) in head of Q;
7.      insert node in AffMdls list;
8.      get all messages, Messages, that refer to node;
9.      SetStatus(node, Messages);
10.     If (node.status == PROPAGATE) {
11.         insert node.ConsumersMsgs to the Q;
12.     }
13. }
14. Return AffMdls;
Procedure SetStatus(Module, Messages)
ConsumersMsgs = ∅;
For All (Message ∈ Messages){
   decide status of Module;
   put messages for Module's consumers in ConsumersMsgs;
}
```

Fig. 13 Algorithm Status determination

Intra-module processing. A module starts by retrieving from the global queue *all* the messages containing the events that concern it. For message processing within each module, a *local queue* is used. The processing of the messages is performed as follows:

1. First, the module probes its schemata for their reaction to the incoming event, starting from the input schemata, next to the semantics and finally to the output schema. Naturally, relations deal only with the output schema.
2. Within each schema, the schema probes both itself and its contained nodes (attributes) for their reaction to the incoming event. At the end of this process, the schema assumes a status as previously discussed.
3. Once all schemata have assumed status, the output schema decides the reaction of the overall module; if any of the schemata raises a veto (BLOCK) the module assumes the BLOCK status too; otherwise, it assumes the PROPAGATE status.
4. Finally, in case a PROPAGATE status is assumed, it prepares and inserts into the global queue appropriate messages for all its consumers.

Observe that a module may receive multiple messages. Typically this is due to the following two reasons: (a) cases of self-join, where a provider feeds (directly or indirectly) a consumer via multiple one paths (and thus, a change in the provider concerns more than one schemata of the consumer – observe here that it is not obligatory that these schemata have identical reaction towards the event) and (b) a deletion of an attribute in a view might affect both the semantics and the output schema of the view, producing thus, two messages to its consumers: one that notifies that output attributes have changed and another notifying that the semantics of the view has changed (e.g., a part of the SELECT clause has been dropped due to the attribute deletion).

Message structure and content. Each message msg is a quadruple $msg(n, s, e, p)$ with the following parts:

– n is the recipient module of the message.
– s is the specific schema of n, to which the message is sent (note that due to this information, we can also find who the sender of the message was, since an input schema has exactly one provider)
– e is the event that this message carries.
– p are message parameters containing additional information needed for some events (e.g., the new name of an attribute for attribute addition or renaming events).

All possible evolution events (as presented in Section 2.2) performed on relations, views and queries generate initial messages that fall into the following types:

– DELETE_ATTRIBUTE: the user deletes an attribute from the output schema
– RENAME_ATTRIBUTE: the user renames an attribute from the output schema.
– ADD_ATTRIBUTE: the user adds another attribute to the output schema of a module.
– DELETE_SELF: the user deletes a whole module.
– RENAME_SELF: when the user renames a whole module.
– ALTER_SEMANTICS: the user changes the semantics of a module.

Once the module has determined its reaction, it constructs messages for its data consumers. The contents of the messages depend on the type of event. Here, we list some examples of such cases.

– When a message is processed saying that an attribute is going to be deleted, the input schema of the consumers that are connected to that attribute is informed that the attribute will be deleted.
– If the whole module is going to be deleted then the consumers of this module will receive a message in their input schema saying that the provider of that input schema is going to be deleted.
– Likewise when an attribute is going to be renamed, the input schema of the consumers that are connected to that attribute is informed that the attribute will have a new name from now on.

- If the whole module is going to be renamed, then the consumers of this module will receive a message in their input schema saying that the provider of that input schema is going to be renamed.
- When a module processes a message saying that a new attribute is going to be added to its output schema, it informs all of its consumers in their input schema that a new attribute was added to their provider.
- Finally when a module processes a message saying that its semantics have changes, it informs all its consumers that it changed its semantics.

Maestros for the local processing. To facilitate the local, independent nature of message processing by the modules, each module awakes a maestro that handles the probing of schemata as well as the decision making on what status will the schema assume. A maestro is a simple piece of software (implemented as an abstract interface, later materialized on a case by case basis) that is specialized on the combination *type of event* $\times$ *module type*. For each type of module, there is a specialized maestro that takes care of status determination and rewriting for each possible event that can be received.

In terms of software architecture, the decision for this structuring of the code was done in order to decentralize event processing. It allows the reasonably smooth extension of the architecture with new types of events or modules at the price of some code reusability. In terms of algorithmic principles, we gain the benefits of module independence at the price of a common queue of messages.

In [9], we present how events are processed inside modules, organized by the type of the incoming message that the module is called to handle. For each event, we explain the structure of the incoming message and the list of steps that have to take place (organized per schema, if more than one schemata of the module are involved).

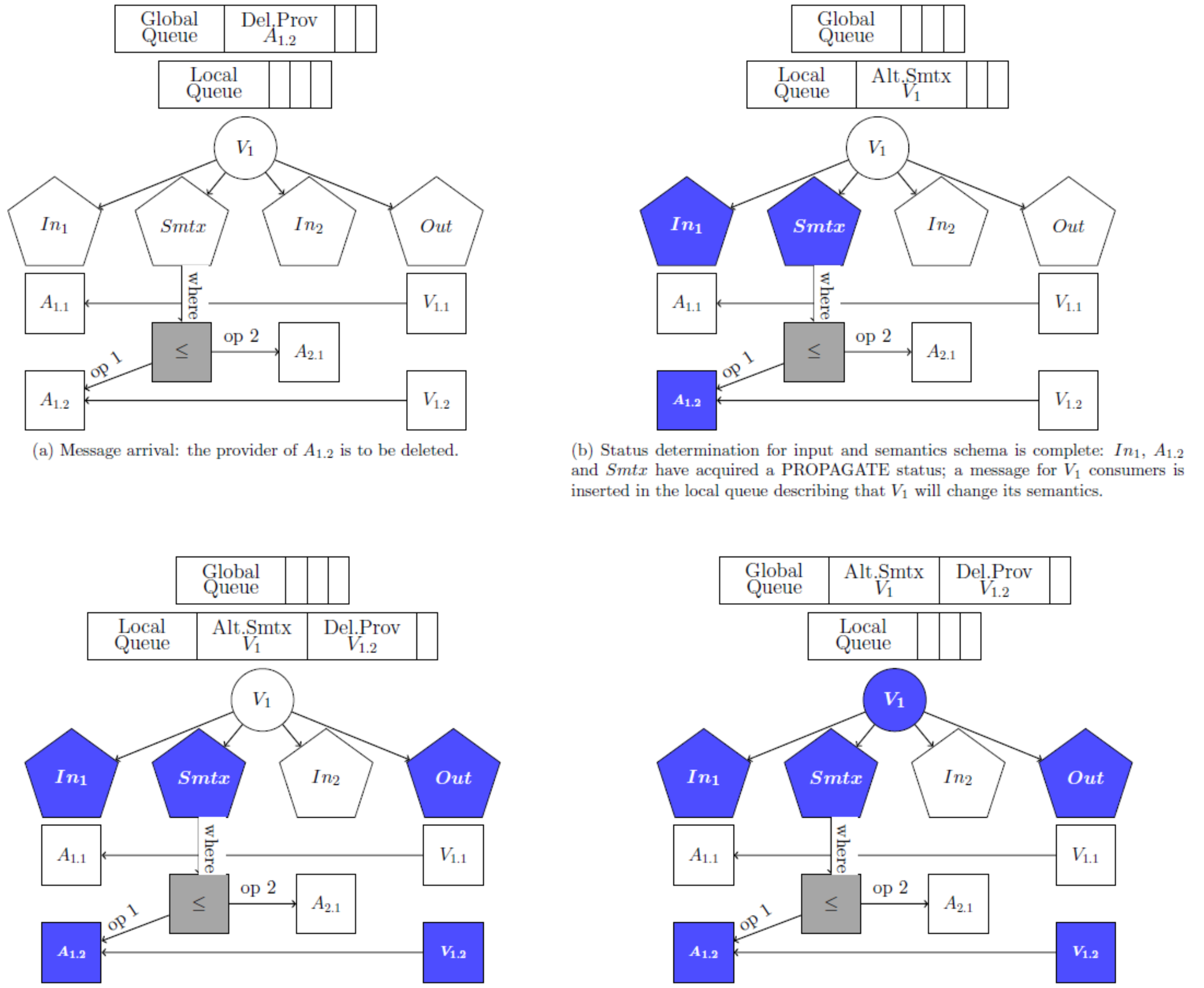
Assume a message with a provider attribute deletion event for the attribute named $A_{1.2}$, that a view module V_1 receives, as depicted in Figure 14 (a).

- Initially, the maestro of the V_1 module will find the attribute with name $A_{1.2}$ in the input schema that fetched the message to the module, denoted as $A_{1.2}$, too. Then, $A_{1.2}$ checks its policy for the event (*provider attribute deletion*) and acquires a status. The same status is assumed for the input schema node of the module V_1 as well. If there is any connection between $A_{1.2}$ and the semantics schema, then the semantics schema checks its policy for the alter semantics event, assumes a status, and creates messages for V_1's consumers, that describe that the semantics of V_1 will change. The newly created messages are kept in a local message queue of the maestro, as depicted in Figure 14 (b).
- Then, if there are attributes in the output schema of V_1 that are connected with $A_{1.2}$ (denoted as $V_{1.2}$), the maestro checks their policy for the event and acquires for each one a status. The output schema node of the module V_1 acquires a status as well. Finally, the maestro, for each of the $V_{1.2}$ attributes finds their consumers so as to notify them that their provider attributes are to be deleted. Those messages are also kept in the local message queue of the maestro, as depicted in Figure 14 (c).
- When all the above reach to an end, the V_1 module checks the statuses of the input, semantics, and output nodes. If none of them has acquired a BLOCK status, then the module acquires status PROPAGATE and notifies the consumers of V_1 about the change, by inserting all the messages of the local message queue in the global message queue, as depicted in Figure 14 (d).

Theoretical guarantees. Previous models of Architecture Graphs ([18]) allow queries and views to directly refer to the nodes representing the attributes of the involved relations. Due to the framing of modules within input and output schemata and the topological sorting, in Theorem 4, and Theorem 5 we prove that the process (a) terminates and (b) correctly assigns statuses to modules.

3.3 Query and view rewriting to accommodate change

Once the first step of the method, *Status Determination*, has been completed and each module has obtained a status, their rewriting would intuitively seem straightforward: each module gets rewritten if the status is PROPAGATE and remains the same if the status is BLOCK. This would require only the execution of the *Graph Rewrite* step – in fact, one could envision cases where *Status Determination* and *Graph Rewrite* could be combined in a single pass. Unfortunately, although the decision on *Status Determination* can be made locally in each module, taking into consideration only the events generated by previous modules and the local policies, the decision on rewriting has to take extra information into consideration. This information is not local, and even worse, it pertains to the subsequent, consumer modules of an affected module, making thus impossible to weave this information in the first step of the method, *Status Determination*. The example of Fig. 15 is illustrative of this case.



(a) Message arrival: the provider of $A_{1.2}$ is to be deleted.

(b) Status determination for input and semantics schema is complete: In_1, $A_{1.2}$ and $Smtx$ have acquired a PROPAGATE status; a message for V_1 consumers is inserted in the local queue describing that V_1 will change its semantics.

(c) Status determination for the output schema is complete: $V_{1.2}$ and Out have acquired a PROPAGATE status; a message for the consumers of $V_{1.2}$ is inserted in the local queue describing that $V_{1.2}$ will be deleted.

(d) Status determination for the entire module is complete (V_1 acquires a PROPAGATE status) and all messages are moved from the local to the global queue

Fig. 14 Status determination example

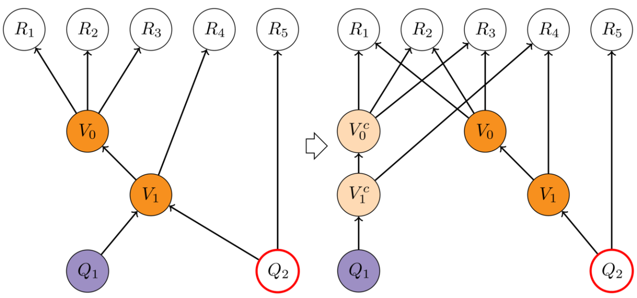


Fig. 15 Block rewriting example

Figure 15 depicts our reference example, which consists of 5 relations, 2 views and 2 queries. We have omitted the full names of the nodes, for illustration purposes. Assume now that the database administrator wants to change V_0, which is the *V_Course* view of our reference example, in a way that all modules depending on V_0 are going to be affected by that change (e.g., attribute addition, or attribute deletion/rename for an attribute common to all the modules of the example). Assume now that all modules except Q_2 accept to adapt to the change, as they have a PROPAGATE policy annotation. Still, the vetoing Q_2 must be kept immune to the change; to achieve this we must retain the previous version of *all* the nodes in the path from the origin of the evolution (V_0) to the blocking Q_2. As one can see in the figure, we now have two variants of V_0 and V_1: the new ones (named V_0^c and V_1^c) that are adapted to the new structure of V_0 – now named V_0^c –, are depicted in the leftmost part of the right figure, having lighter color, and the old ones, that retain their name, are depicted in the rightmost part of the figure. The latter are immune to the change and their existence serves the purpose of correctly defining Q_2.

Input: An architecture graph summary $\mathbf{G_s(V_s, E_s)}$, a list of modules *AffMdls*, affected by the event, and the *InitialEvt* of the user.
Output: Annotation of the modules of *AffMdls* on the action needed to take, and specifically whether we have to make a new version of it, or, implement the change that the user asked on the current version

```
For All (Module ∈ AffMdls){
    If(Module.status == BLOCK){
        CheckModule(Module, AffMdls, InitialEvt);
        mark Module not to change; //Blockers do not change
    }
}
Procedure CheckModule(Module, AffMdls, InitialEvt)
If (Module has been marked) {return;} //Already notified
If (InitialEvt == ADD_ATTRIBUTE){ //Allow additions
    mark Module to apply change on current version;
}
Else{
    mark Module to keep current version as is and apply the change on a clone;
}
For All(ModuleProv ∈ AffMdls feeding Module){ //Notify path
    CheckModule(ModuleProv, AffMdls, InitialEvt);
}
```

Fig. 16 Algorithm PATH CHECK

The crux of the problem is as follows: if a module has PROPAGATE status and none of its consumers (including both its immediate and its transitive consumers) raises a BLOCK veto, then both the module and all of these consumers are rewritten to a new version. However, if any of the immediate consumers, or any of the transitive consumers of a view module raises a veto, then *the entire path towards this vetoing node must hold two versions of each module*: (a) the new version, as the module has accepted to adapt to the change by assuming a PROPAGATE status, and, (b) the old version in order to serve the correct definition of the vetoing module. Exceptionally, if the event vetoed involves a relation, the veto freezes any other change and the event is blocked.

To correctly serve the versioning purpose, the adaptation process is split in two steps. The first of them, *Path Check*, works from the consumers towards the providers in order to determine the number of variants (old and new) for each module. Whenever the algorithm visits a module, if its status is BLOCK, it starts a reverse traversal of the nodes, starting from the blocker module towards the module that initialized the flow and marks each module in that path (a) to keep its present form and (b) prepare for a cloned version where the rewriting will take place. A cloned version is an identical copy of a module's subgraph, with the same providers but with different name. For example, if we already have a view in SQL as:

```
CREATE VIEW vn AS SELECT c FROM t;
```

then its clone would be

```
CREATE VIEW vn_Clone AS SELECT c FROM t;
```

The only exception to this rewriting is when the module of the initial message is a relation module and the event is an attribute deletion, in which case a BLOCK signifies a veto for the adaptation of the relation.

Input: A list of modules affected modules, *AffMdls*, knowing the number of versions they have to retain, initial messages of *AffMdls*, and initial evolution message, *IMsg*
Output: Architecture graph after the implementation of the change the user asked

```
If(any of AffMdls has status BLOCK){
    If(IMsg started from Relation module type AND
    event == DELETE_ATTRIBUTE) {Return};
    Else
    {
        module toConnect ← Module;
        For All (Module ∈ AffMdls){
            If(Module needs two versions){ //clone module to
                toConnect ← clone of Module; //keep both versions
            }
            connect toConnect to new providers;
            proceed with rewriting of toConnect;
        }
    }
}
Else
{
    For All(Module ∈ AffMdls){ //all nodes PROPAGATE
        proceed with rewriting of Module //edges fixed internally
    }
}
```

Fig. 17 Algorithm GRAPH REWRITE

Finally, all nodes that have to be rewritten are getting their new definition according to their incoming events. Unfortunately, this step cannot be blended with *Path Check* straightforwardly: *Path Check* operates from the end of the graph backwards, to highlight cases of multiple variants; rewriting however, has to work from the beginning towards the end of the graph in order to correctly propagate information concerning the rewrite (e.g., the names of affected attributes, new semantics, etc.). So, the final part of the method, *Graph Rewrite*, visits each module and rewrites the module as follows:

- If the module must retain only the new version, once we have performed the needed change, we connect it correctly to the providers it should have.
- If the module needs both the old and the new versions, we make a clone of the module to our graph, perform the needed change over the cloned module and connect it correctly to the providers it should have.
- If the module retains only the old version, we do not perform any change.

One could possibly argue that we could have used a principled way to mark the paths of the blocker modules, starting from the blocker module and visiting all the affected modules with ID smaller than the blocker's ID, marking them to have two versions in the new schema. Unfortunately, this method would have been insufficient as it would not be able to guarantee that the affected modules that are not in the path of a blocker module will not be marked to obtain two versions too. For example, in Figure 15, the $Q_1.ID$ could be either 8 or 9 after the topological sorting. If $Q_1.ID$ is 9, then the aforementioned ID based traversal could be used. If $Q_1.ID$ is 8, then $Q_2.ID$ is 9 and the aforementioned ID based traversal would mark the Q_1 module to obtain two versions, which is wrong.

How much cloning is required? Each execution of the *Path Check* and the *Graph Rewrite* algorithms involves one event only. For each such event, a cloning is required whenever (a) the event involves deletion or semantics update, (b) a view module initiates the propagation of an event due to its PROPAGATE policy and (c) some of its (possibly transitive) data consumers raises a veto. In this case, the entire path till the blocker (blocker excluded) must be cloned. If there are two blockers that have the same provider, then there is no extra duplication. For a given event that fulfils the aforementioned conditions, assuming n blockers in an event, and m paths, $m \leq n$, involving M nodes (excluding the blockers), we need M extra cloned modules. If the graph contains V views and Q queries, the maximum impact is when all of them are affected by an event. A worst case scenario can be conceived when there is a root view and everybody else is defined over this view either directly or transitively. Assume now that the root view is affected in a way that all views and queries are affected (e.g., change of semantics) and all queries are blockers, although all views are propagators (because if another view is a blocker, its queries are protected). Then, we need to clone V views, which is the maximum amount of cloning that can happen in an event. Our reference example is in fact such a worst case (see Fig. 15). Practically speaking of course, this possibility is rare (observe for example Fig. 19 on how a large subset of the Drupal ecosystem is constructed).

Returning to the rewriting process of modules with a PROPAGATE status, we can summarize this process as follows:

- Whenever the attributes of a modules output schema are deleted, renamed or inserted, the subsequent consumer schemata are adopted accordingly;
- Whenever entire modules are deleted or renamed, the respective schemata are deleted or renamed accordingly.

In the following paragraphs, we are going to discuss the way the rewriting process is performed within each module. Initially, we need to distinguish two categories, depending on the type of the module that is rewritten: (a) the rewriting processes that apply to Relation modules, and, (b) the processes that apply to Query/View modules. This differentiation is mainly due to the fact that, in contrast to the Relation modules that contain only an output schema, the Query/View modules additionally contain a semantics schema and a set of input schemata (one per provider). Therefore, queries and views require a different treatment.

In the following two subsections, we are going to briefly describe for each event, the steps that are followed in the module rewriting mechanism, when the module is accepting the change (its status is PROPAGATE). Naturally, if the status is BLOCK, no rewriting is required at the internals of the module.

3.3.1 Relation module rewriting

For each of the events applied to a relation (as presented in Table 1), we perform the following steps for rewriting the affected relation module and progating the event towards the rest of the graph:

Attribute addition When a new attribute is added to a relation module, the user is prompted for the name of the new attribute and the module checks if it is available or already in use by another attribute. If all conditions are met then the new attribute is added to the output schema of the module and a message with the addition along with the new attribute name is propagated to all dependent modules.

Attribute deletion When an attribute is deleted, the output schema searches for the specific attribute and deletes it. Similarly, a message for the deletion is propagated to all dependent modules.

Attribute rename When an attribute is renamed, the output schema searches for the specific attribute and renames it with the name provided by the user, unless there is conflict with any attribute having the same name in the output schema of the module; in this case the user is prompted to change it. Again, a message for the renamed attribute is generated.

Self (module) deletion When a relation module is deleted, its output schema with all its attributes are deleted and the module node itself is also deleted. A message for the deletion is propagated to all dependent modules.

Self (module) rename When a relation module is renamed, the user is prompted for the new name of the module. If it is unique, the module and its output schema are renamed accordingly. Moreover, a message with the renamed relation is propagated to all connected query/view modules, in order to update their input schemata with the new name.

3.3.2 Query/View module rewriting

Query and View have the same events, thus we do not separate their rewriting methods. The steps for rewriting (a PROPAGATE status is assumed) are as follows:

Attribute addition The user adds a new attribute by selecting it out of the list of attributes belonging in the output schemata of the query/view providers and sets a unique alias name for this attribute. In case there is a GROUP BY clause in the semantics schema, the user is prompted for adding the new attribute to the groupers or using any aggregate function. In any case, the new attribute is directly added to the output schema of the module. If the attribute was not used before in the query/view, it is added in the respective input schema and finally all the needed connections between the output node and the semantics (if applicable) and input node are set. Moreover, its name is propagated to the modules that are connected, in order to let them know the name of the new attribute.

Attribute provider addition When an attribute is added in a provider of the module, the input schema of the module adds a new attribute node with the specified name. If there is any connection to the semantics schema due to a GROUP BY clause, the user is again prompted for adding the new attribute to the groupers or using any aggregate function. Finally, when all conditions are met, the new attribute is added to the output schema of the module (checking to see if there are any conflicts with the name of the new attribute and if so, the user is prompted accordingly). Moreover, the name of the new attribute is propagated to the modules that are connected, in order to update their input schemata accordingly.

Attribute deletion For the case that an output attribute is deleted, it is first removed from the output schema. All connections of the output attribute with the semantics schema are removed and finally, if the attribute is not used in the semantics tree, it is removed from the respective input schema.

Attribute provider deletion When an attribute of a provider module is deleted, it is initially removed from the corresponding input schema of the module. If it is used in a condition in the semantics tree, then this condition is set to true or false, depending on the operator which connects this condition with the semantics tree (true if the condition was connected to an AND operator and false if it was connected to an OR operator). Finally, if this attribute is part of the SELECT clause of the query, it is removed from the output schema. See Fig. 18 for how the aforementioned example of Fig. 14 is rewritten.

Attribute rename When an attribute is renamed, the output schema searches for the specific attribute and renames it with then name provided by the user, unless there is conflict with any attribute already present in the output schema of the module, having the same name with the one chosen for renaming (in this case the user is prompted again). Moreover, its name is propagated to the modules that are connected, in order to let them know the new name of the attribute in order to update their schemata.

Attribute provider rename When an attribute is renamed in one of the providers of the module, the attribute is initially renamed in the corresponding input schema of the module. If there is a connection between any attribute of the output schema with the same name, this attribute is also renamed, unless the name is already used by any attribute of the output schema, in which case, the user is prompted for a new name. Finally, this new name is further propagated to the modules that are connected to this current one.

Module deletion When a query/view module is deleted, the schemata nodes of the module with all their attributes are deleted and the module node itself is also deleted.

Module rename When a query/view module is renamed, the user is prompted for the new name of the module and if it is unique in the graph, then the module and its output, semantics and input schema nodes are renamed accordingly. Moreover, the new name is propagated to the modules that are connected to this query/view, in order to know the new name of the module and update their input schemata.

Provider module deletion When a provider module is deleted, the respective input schema of the module that receives this notification is deleted. The steps applied to the deletion of a provider attribute are performed for all attributes of the the deleted provider module.

Provider module rename When a provider module is renamed, the module that receives this notification initially checks if its input schema that corresponds to the renamed provider had the same name with the old provider name (not always the case, since there could be an AS rename in the query/view definition). If this is the case, the input schema of the module is renamed accordingly (unless the new name conflicts with an AS rename of any other input schema of the module). If the new name due to conflicts cannot be set to the input schema, the user is prompted for a new one.

Alter of semantics When a query/view module changes it semantics, the user is prompted to alter the where and the group by clause of the module and the semantics tree is rewritten from this input. When an alter of semantics message arrives from any of the module's providers, and the module has PROPAGATE semantics, then, as we have already discussed in the previous subsection, there is no rewriting at all at the module.

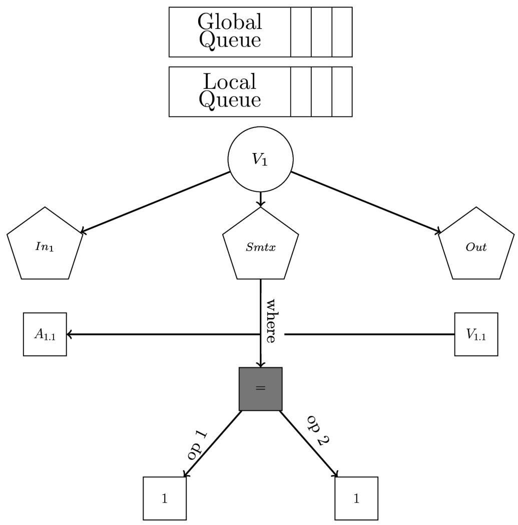


Fig. 18 Rewriting for the example of Fig. 14

4 Theoretical Guarantees

In this section, we provide a set of theoretical guarantees regarding the correct annotation of the graph with events and policies and the termination and confluence properties of our proposed algorithms.

4.1 Language Properties

Theorem 1 *The entries of Table 1 cover* completely *the space of node types with the events they can sustain.*

Proof 1 *The table that contains the events that each node type can receive was described earlier in Section 2 (Table 1). In Table 2 we have replaced the ✓ symbols of the table's cells with the numbers of the default policy rules, according to the numbering scheme of Figure 7. Two cells in the ALTER_SEMANTICS column are annotated with a ✓ and without a reference to a rule; we explain why in the following text.*

The events that are user generated and pertain to views and queries are:

UQV.1 ADD_ATTRIBUTE
UQV.2 DELETE_ATTRIBUTE
UQV.3 RENAME_ATTRIBUTE
UQV.4 DELETE_SELF
UQV.5 RENAME_SELF
UQV.6 ALTER_SEMANTICS

As mentioned previously in Section 2.2, the events that are user generated and pertain to relations are:

UR.1 ADD_ATTRIBUTE
UR.2 DELETE_SELF
UR.3 RENAME_SELF
UR.4 DELETE_ATTRIBUTE
UR.5 RENAME_ATTRIBUTE

Our policy language covers all these events that are related with the user interaction and are the marks of Table 1 that are in the lines that contain the OUT *and* SMTX *keywords on queries, views and relations.*

Due to the message propagation mechanism, additional events occur. These events (also described in Section 2.2) are received by the IN *nodes of the query and view modules. These events are:*

MP.1 ADD_ATTRIBUTE_PROVIDER
MP.2 DELETE_PROVIDER
MP.3 RENAME_PROVIDER
MP.4 ALTER_SEMANTICS

For our policy language to cover the three of the four previous events (MP.1, MP.2, and MP.3) that are related to the message propagation mechanism, additional policies are needed. The exception of MP.4 is due to the fact that the IN schema node who receives such an event forwards it to the respective SMTX node who is actually the one responsible for the handling of this event.

			ADD		DELETE		RENAME		ALTER
			ATTR	ATTR PROV	SELF	PROV	SELF	PROV	SMTX
QUERY	OUT	SELF	1	2	3		4		
		ATTRS			5	7	6	8	
	IN	SELF		11		9		10	✓
		ATTRS				12		13	
	SMTX	SELF							14
VIEW	OUT	SELF	15	16	17		18		
		ATTRS			19	21	20	22	
	IN	SELF		25		23		24	✓
		ATTRS				26		27	
	SMTX	SELF							28
RELATION	OUT	SELF	29		30		31		
		ATTRS			32		33		

Table 2 The space of events that can be received by each node type according to the line number in the rules of the policy file

Therefore, there is no need to define a policy at the IN schema node, as the event will be appropriately handled at the SMTX node.

In Table 2 we have all the above combinations of events and node types, thus, our default 33 rule have completely covered the space of node types with their incoming events.

Precisely, lines 1 to 14 concern queries.

As previously stated at the exception of MP.4, the 14 rule (QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>;) covers two events, since the IN node forwards this message to the SMTX node which is the only responsible for the policy over the ALTER_SEMANTICS event.

Likewise, lines 15 to 28 concern views.

The 28 rule (VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>;) covers two events just like 14 rule does, for the exact same reasons.

Finaly, lines 29 to 33 concern relations.

As one may notice the 33 rules cover all the 35 events that may appear in each one of the nodes. The inequality of the numbers is because of the exception of MP.4. Therefore, the fact that the 33 rules cover 33 events plus the two events that do not need any rule proves that all the events (UR., UQV.*, and MP.*) are covered by our policy rules.*

Moreover, if we override the 33 default rules, then, the most refined policy will be enforced for each node.■

1 QUERY.OUT.SELF: on ADD_ATTRIBUTE then <policy>; which is for UQV.1
2 QUERY.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in output schema node
3 QUERY.OUT.SELF: on DELETE_SELF then <policy>; which is for UQV.4
4 QUERY.OUT.SELF: on RENAME_SELF then <policy>; which is for UQV.5
5 QUERY.OUT.ATTRIBUTES: on DELETE_SELF then <policy>; which is for UQV.2
6 QUERY.OUT.ATTRIBUTES: on RENAME_SELF then <policy>; which is for UQV.3
7 QUERY.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
8 QUERY.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
9 QUERY.IN.SELF: on DELETE_PROVIDER then <policy>; which is for MP.2
10 QUERY.IN.SELF: on RENAME_PROVIDER then <policy>; which is for MP.3
11 QUERY.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in input schema node
12 QUERY.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
13 QUERY.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
14 QUERY.SMTX.SELF: on ALTER_SEMANTICS then <policy>; which is for both UQV.6 and MP.4

Table 3 Query policies with the addressed events

Theorem 2 *The policy overriding mechanism is correct (assigns the correct policy to each node).*

Proof 2 *One node may have more than one policies for a specific event. This occurs because a policy over an event may be set in any of the following three rules:*

15 VIEW.OUT.SELF: on ADD_ATTRIBUTE then <policy>; which is for UQV.1
16 VIEW.OUT.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in output schema node
17 VIEW.OUT.SELF: on DELETE_SELF then <policy>; which is for UQV.4
18 VIEW.OUT.SELF: on RENAME_SELF then <policy>; which is for UQV.5
19 VIEW.OUT.ATTRIBUTES: on DELETE_SELF then <policy>; which is for UQV.2
20 VIEW.OUT.ATTRIBUTES: on RENAME_SELF then <policy>; which is for UQV.3
21 VIEW.OUT.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
22 VIEW.OUT.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
23 VIEW.IN.SELF: on DELETE_PROVIDER then <policy>; which is for MP.2
24 VIEW.IN.SELF: on RENAME_PROVIDER then <policy>; which is for MP.3
25 VIEW.IN.SELF: on ADD_ATTRIBUTE_PROVIDER then <policy>; which is for MP.1 in input schema node
26 VIEW.IN.ATTRIBUTES: on DELETE_PROVIDER then <policy>; which is for MP.2
27 VIEW.IN.ATTRIBUTES: on RENAME_PROVIDER then <policy>; which is for MP.3
28 VIEW.SMTX.SELF: on ALTER_SEMANTICS then <policy>; which is for both UQV.6 and MP.4

Table 4 View policies with the addressed events

29 RELATION.OUT.SELF: on ADD_ATTRIBUTE then <policy>; which is for UR.1
30 RELATION.OUT.SELF: on DELETE_SELF then <policy>; which is for UR.2
31 RELATION.OUT.SELF: on RENAME_SELF then <policy>; which is for UR.3
32 RELATION.OUT.ATTRIBUTES: on DELETE_SELF then <policy>; which is for UR.4
33 RELATION.OUT.ATTRIBUTES: on RENAME_SELF then <policy>; which is for UR.5

Table 5 Relation policies with the addressed events

1. *Rules about all the nodes of the Architecture Graph.*
2. *Rules about all modules and their attributes.*
3. *Rules that apply to all the attributes of a* specific schema.
4. *Rules that apply to* specific attribute *nodes.*

The golden standard of correctness requires that whenever a node has more than one policies for the same event, the one that perseveres is the policy defined at the finest level of detail.

The overriding mechanism is correct because the following sequence of events is guaranteed: initially, it we apply the most general rules for all the nodes of the graph, then the rules per module type, then the rules referring to the attributes of specific schemata, and finally, the rules that apply to specific attributes.

Observe that this is independent on whether the policies are assigned a priori, during the construction of the graph, or, on demand, whenever a specific node needs to determine its policy.■

Theorem 3 *The extra rules*

- *<moduleType>: ON * THEN <policy>;*
- *<namedNode>: ON * THEN <policy>;*
- *NODE: ON <event> THEN <policy>;*
- *NODE: ON * THEN <policy>;*

can correctly cover up the events of the Table 2 and correctly override each other mechanism (assign the correct policy to each node).

Proof 3 *We need to prove that these rules will cover up all the events that a node might receive, as well as that these rules correctly override each other. The more general rules are the ones that contain the keyword* **NODE**. *These rules are applied first. Then the rules that apply to modules and finally the rules that are applied to specific nodes of the graph. Within each rule, the rules that contain the keyword* *** *are preceding over the others rules.*

The rule: NODE: ON * THEN <policy>; *is translated to all the 33 rules described in Figure 7 and prove their correctness in Theorem 1. So all the events are covered. This rule is also the first one to be applied in our graph, regardless of its position.*

The rule NODE: ON <event> THEN <policy>; *is translated to the rules that apply for the specified event type. We can follow the columns of the table 2 in order to see that for:*

- *ATTRIBUTE_ADDITION, the rules of Figure 7 that apply are: rule 1 for the queries, rule 15 for the views and rule 29 for the relations.*
- *ATTRIBUTE_PROVIDER_ADDITION, the rules of rules of Figure 7 that apply are: rule 2 and 11 for the queries and rule 16 and 25 for the views.*
- *DELETE_SELF, the rules of Figure 7 that apply are: rule 3 and rule 5 for queries, rule 17 and rule 19 for views, rule 30 and rule 32 for relations.*
- *DELETE_PROVIDER, the rules of Figure 7 that apply are: rule 7, rule 9 and rule 12 for the queries, rule 21, rule 23 and rule 26 for the views.*
- *RENAME_SELF, the rules of Figure 7 that apply are: rule 4 and rule 6 for the queries, rule 18 and rule 20 for the views, rule 31 and rule 33 for the relations.*
- *RENAME_PROVIDER, the rules of Figure 7 that apply are: rule 8, rule 10 and rule 13 for the queries, rule 22, rule 24 and rule 27 for the views.*
- *ALTER_SEMANTICS, the rules of Figure 7 that apply are: rule 14 for the queries and rule 28 for the views.*

*This rule is the second one that is applied in our graph, in order to correctly override the more general first rule (*NODE: ON * THEN <policy>;*).*

The rule <moduleType>: ON * THEN <policy>; *is translated to the set of rules that apply to the specific module type. For example,*

- *for the query module type, these rules are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, and, 14,*
- *for the view module type, these rules are: 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, and, 28,*
- *for the relation module type, these rules are: 29, 30, 31, 32, and, 33.*

This rule is the third one that is applied in our graph, in order to correctly override the two more general rules.

Finally, the rule <namedNode>: ON * THEN <policy>; *is translated to the rules that apply to the module type of the specific* <namedNode>. *This means that the rules that are generated have the* <namedNode>*'s name. For example, for a relation named TRANSCRIPT,*

1. *the rules will start with TRANSCRIPT_SCHEMA*
   - *for the ATTRIBUTE_ADDITION event, which is for the rule 29,*
   - *for the DELETE_SELF event, which is for the rule 30,*
   - *for the RENAME_SELF event, which is for the rule 31,*
2. *and two more rules that are for the attributes of the TRANSCRIPT relation, starting with TRANSCRIPT_SCHEMA.ATTRIBUTES*
   - *for the DELETE_SELF event of the attributes, which is for the rule 32,*
   - *for the RENAME_SELF event of the attributes, which is for the rule 33.*

This rule is applied after the rules per module type have been applied and before rules with specific events for specific nodes are applied.■

4.2 Theoretical Guarantees for the Status Determination Algorithm

First, we prove that the mechanism for message propagation works correctly at the inter-module level.

Theorem 4 *The message propagation at the inter-module level terminates.*

Proof 4 *The summary of the architecture graph is a directed acyclic cycle. This is due to the fact that (i) a query depends only on views and relations, and (ii) 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 a message only once for each session to its consumers that are related with the event/parameter; (b) the graph is finite. Since both assumptions hold, the algorithm terminates.■

Theorem 5 *Each module in the graph will assume a status once the message propagation terminates; this status is the same, independently from the order of processing the incoming messages.*

Proof 5 *Each module gathers from the common message queue* all *the messages that concern it. For each message, the module and its schemata, assume a status. A module's status can change only in the following order: NO_STATUS < PROPAGATE < BLOCK, meaning that if a module has assumed a PROPAGATE status, it can not change it to NO_STATUS but it may change it to BLOCK. Therefore, if a message that will ignite a BLOCK policy is found anywhere in the list of incoming messages, this BLOCK status will eventually be assumed and not overridden later. Otherwise, a PROPAGATE status will be assumed. At the end of the message processing, the module retains the final status it assumed.*■

Theorem 6 *Messages are correctly propagated to the modules of the graph.*

Proof 6 *For the node that receives the initial event we need to prove that:*

1. *the node either acquires BLOCK status, therefore the message propagation mechanism stops, or,*
2. *the node acquires PROPAGATE status and notifies its consumers about the change.*

   For all the other nodes we need to prove that:

3. *that a module will not be affected if none of its providers was affected by the imminent change,*
4. *there is no module that receives a message while its provider has a BLOCK status,*
5. *there is no module that should have received a message when it was its turn to acquire a status but the message was not in its input message list,*

The first two propositions stand because of the rewrite maestros mechanism. The modules communicate using a global list of messages. The rewrite maestro keeps a local list of all the outgoing messages of the module to its consumers. When the module finishes processing all its incoming messages, the maestro checks the module's status and if it is BLOCK, then returns, without *adding the outgoing messages to the global list, which proves the first proposition. If the status of the module is PROPAGATE then the output messages* are added *in the global list, so the consumers of the module are notified, which proves the second proposition.*

For the third proposition: 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. Due to its inherent construction, the modules which are eventually visited by the message propagation mechanism, have at least one of their providers affected. For the same reason, the modules that are not visited by the mechanism (a) either do not have any provider affected, or, (b) a block policy terminated the message propagation in provider modules, earlier.

For the fourth proposition: The messaging mechanism dictates that each message is propagated from the output node of the provider module towards the input schemata of all consumer modules, unless a BLOCK policy explicitly halts the propagation. Since a BLOCK policy terminates the message propagation from this provider module, we guarantee that there is no consumer module to receive any message from provider module.

For the fifth proposition: The messaging mechanism uses a list to transfer the messages between the modules. This list is sorted by the ID numbers that the modules have acquired by the topological sort algorithm (described in Figure 12). Since the list is topologically sorted too, we guarantee that there there is no module that should have received a message when it was its turn to acquire a status but the message was not in its input message list.■

Theorem 7 *The message propagation at the intra-module level: (i) terminates, with (ii) each node assuming a unique status according to its policy and the status precedence constraints.*

Proof 7 *We visit the schemata of a module in a fixed order: input schemata, semantics schema, output schema. For each of these schemata, we may visit its attributes too. All these constructs are finite and visited only once. This is a task that the maestros perform and the very reason for their existence, otherwise we could have allowed message propagation via the graph's edges within the modules too. Therefore, the algorithm terminates and (i) is proved.*

For requirement (ii) we need to prove the following:

1. for all messages, *vetoes override adaptation,*
2. per message, *for all the appropriate nodes (and only them) the status of the most detailed nodes overrides the decision of the status of the schema,*
3. if any of the schemata of a module has status BLOCK, the module assumes status BLOCK.

Regarding the first proposition: as already stated at the proof of Theorem 5, a node's status can change only in the following order: NO_STATUS < PROPAGATE < BLOCK, meaning that if a node has assumed a PROPAGATE status, it can not change it to NO_STATUS but it may change it to BLOCK.

Regarding the second proposition: every time a schema is probed on an event (a) the appropriate nodes within a schema are asked about their policy, or/and, (b) the schema itself is asked about its policy. Table 2 describes the relationship between events and nodes prompted, in the lines that say ATTRS the (a), (b) take place, while in the lines that say SELF only the (b) takes place. This is the correct and desired behavior. When an attribute acquires a status, the schema node is prompted to acquire the same status. The completeness of the language guarantees that all *nodes have a policy for any incoming event that can arrive to them. Therefore, in all occasions (i) the correct nodes are prompted for a response, (ii) the policy of the appropriate nodes prevails, (iii) it is impossible that such a policy does not exist. Therefore, for each message all nodes acquire the correct status.*

Regarding the third proposition: the proposition is inherently supported.■

4.3 Theoretical Guarantees for the Path Check Algorithm

We are going to prove that Path Check Algorithm terminates and all modules at the end have the correct number of versions they need to keep.

Theorem 8 *The module traversal terminates and the visited modules have the correct notification of how many versions they need.*

Proof 8 *Algorithm Path Check sequentially passes from each of the affected modules with BLOCK status and for each of them executes method* CheckModule. *If we can prove that CheckModule terminates, then the algorithm terminates too.*

The algorithm has as input: (i) a finite *set of modules (each module with BLOCK status starts the CheckModule method once), and (ii) the initial event placed by the user.*

In every step, the CheckModule method marks the module to keep two versions, and finds the providers of this module through which the module was marked about the change. These provider modules are listed in the set of the affected modules. If there are no more modules this means that the method reached the module from which the change started.

Since this is a recursive procedure, the providers of the providers of those modules are also marked and so *on. The condition that inspects whether the visited module was previously marked, is done by the following line:*

<u>If</u>(Module has been marked) <u>Then</u> return;

of the CheckModule method. This condition makes sure that the recursive traversals of the method terminate as soon as possible –since those modules have already been marked by a previous traversal– and every module that is part of the path that goes from a blocker module to the source of the changes has been marked to keep two versions.■

4.4 Theoretical Guarantees for the Graph Rewrite Algorithm

We are going to prove that (a) *Graph Rewrite* Algorithm terminates, (b) when *Graph Rewrite* terminates, all modules have the correct connections at the inter-module level, and (c) all modules are correctly rewritten at the intra-module level.

4.4.1 Termination and confluence at inter-module level

First, we prove that the mechanism for graph rewriting works correctly at the inter-module level.

Theorem 9 *The graph rewriting at the inter-module level terminates.*

Proof 9 *The Graph Rewriting Algorithm terminates when all the notified modules that accepted the change (meaning that those modules acquired a PROPAGATE status) are rewritten. The algorithm has as input the list of the affected modules, each one having its status and the number of versions it needs to keep, and the initial messages that each affected module received. In the special case of the DELETE_ATTRIBUTE event coming from a Relation module, the algorithm terminates right away. Otherwise, each one of the affected modules (which is a finite list of modules) is rewritten* once, *so the algorithm terminates.*

We need to prove:

1. *that each module is rewritten only once for each one of the messages it received,*
2. *that there is a finite list of messages, and*
3. *that there is a finite number of modules that are going to be rewritten.*

For 1 and 2 since the incoming messages of a module are finite (as proved earlier in theorem 4), and maestros are only once executed per message we are sure that each module is rewritten once per message received. For 3 the number of modules that acquired PROPAGATE status is finite, since the graph is finite. Therefore, since all assumptions hold, the algorithm terminates.■

Theorem 10 *The graph will be in the correct form after the rewrite.*

Proof 10 *We need to prove that:*

1. *All the modules that have no status will not be rewritten.*
2. *All the modules with BLOCK status will not be rewritten.*
3. *If there is no vetoer in the graph, then all the modules with PROPAGATE status will be rewritten.*
4. *If there is any vetoer, then the modules with PROPAGATE status will be rewritten (i) themselves – since there is only one version needed– if they are not part of a blocker path, or, (ii) as clones –since there are two versions needed– as parts of a blocker path. In both cases the modules will be connected to the appropriate path.*

A module is part of a blocker path when the module has PROPAGATE status but at least one of its descendants acquired status BLOCK.

We need two paths, the "new providers" in which all the nodes accepted the change, and the "old providers" in which we keep the old definitions of all the affected modules because a module declined the change. If a module needs to keep only one version then the path with the "new providers" is the one that this module belongs to. If a module needs to keep two versions then the path with the "old providers" that did not want to accept the change is the one that this module belongs to, while its clone belongs to the path with the "new providers" that accepted the change, thus providing right essence to the modules that need to keep only one version. The number of versions a module has to keep is given by the algorithm depicted in Figure 16.

If none of the modules vetoed, then the Graph Rewrite *Algorithm does a traversal visiting the affected nodes, in order to apply the change the user asked. The algorithm works only with the modules that have PROPAGATE status, thus 1, 2 and 3 are proved. For 4:*

1. *If the module needs to keep two versions we clone the module, we connect the cloned module to its new providers (if it is the module that started the event then we connect it to the providers that the original has), and we proceed with the rewrite of the cloned module. This way, the cloned modules are all in one path, and the modules that vetoed are all in the other path.*
2. *If the module belongs to a path without blocker modules, then it needs to keep only the new version we*

connect it to the path of the new providers and we proceed with the internal rewriting of the module.

■

4.4.2 Termination and confluence at intra-module level

Theorem 11 *The rewriting of modules at the intra-module level terminates and each module is rewritten correctly.*

Proof 11 *Sections 3.3.1 and 3.3.2 describe the intra module rewriting process, where we begin our module rewriting from the input schema, continue to the semantics schema and terminate to the output schema. There is only one exception at the aforementioned rule and that is on the attribute addition of query/view modules, where we start from the output schema of the module and move towards the semantics and input schemata.*

In both cases, we start rewriting from the one border of the module (input or output) and terminate to the other border of the module (output or input, respectively). Because of the previous statement, we guarantee that our method terminates because: (a) we perform a single visit per affected node, and (b) we work with a finite number of nodes. As for the validity of the intra-module rewriting, each event is rewritten as described in sections 3.3.1 and 3.3.2 and whenever information is needed, either the modules passes that information from the provider module to the consumer module, or prompt the user to provide the needed information.■

5 Experiments

We assessed our method for its usefulness and scalability with varying graph configurations and policies; in this section, we report our findings. As already mentioned, all the material for this work, including input ecosystems, links to the source code (publicly available at git) and results can be found in the paper's web page[3]. We have employed a real-world case, based on 7 major revisions of Drupal in the period 2003 - 2007 as the testbed for our experimentation. To further stress-test our method with more complicated scenarios, we have also performed a controlled experiment, based on a widely used benchmark, TPC-DS, to allow the evaluation of the effect of different problem parameters (like policy annotation and graph size) to the effectiveness, efficiency and required user effort of our method. Before proceeding, we describe the fundamental metrics that we employ for the assessment of our experiments.

[3] `http://www.cs.uoi.gr/~pmanousi/publications/2013_ER/index.html`

5.1 Effectiveness and Effort Metrics

In this subsection, we discuss the metrics used to assess the efficiency, the effort spent for the annotation of the graph and the effectiveness metrics that we employ in our experiments. Evaluating efficiency is straightforward, as we assess the breakdown of the time spent for each of the 3 steps of our method. For the rest of the metrics, we provide a more detailed discussion, right away. We conclude this subsection with a note on the experimental configuration of each experiment.

5.1.1 Effectiveness: do we gain from annotating the graph with policies and testing what-if scenarios this way?

How can we assess the effort gain of a developer using the highlighting of affected modules of Hecataeus? This gain should be contrasted to the effort spent in the case where he would have to perform all checks by hand. We employ the $\%AM$ metric, measuring the *fraction of Affected Modules of the ecosystem* as the gain that amounts to the percentage of useless checks the user would have made. We exclude the object that initiates the sequence of events from the computation, as it would be counted in both occasions. Formally, $\%AM$ is given by the Equation 1.

$$\%AM = 1 - \frac{\#Affected\ Modules}{\#(Queries \cup Views)} \quad (1)$$

Moreover, to assess the extent of rewritings that are automated by our method, for each event we measure the *number of Rewritten Modules* as the sum of the number of modules that were cloned (new versions of affected modules) with the number of existing adapted modules. We denote this measurement with the RM metric, given by the Equation 2.

$$RM = \#Adapted\ Modules\ +\ \#Cloned\ Modules \quad (2)$$

5.1.2 Policy annotation effort: how much time does it take to setup the policy rules in order to work with our what-if analysis tool?

How hard is it to annotate the graph with policies? How much time does the user have to spend for authoring the rules?

Our method comes with the possibility of using syntactic sugar rules that make life easy and fast. For the rare occasion when the user does not want to use these syntactic shortcuts, for every specific module that gets into the user's focus, the user has to provide as many rules as necessary to override the default policies. In the worst case, this requires $9 + 5 \times |input\ schemata|$ rules for a full re-specification of a query/view module. When the syntactic sugar is used, *one rule is sufficient to fully invert the policy of a module.* Of course, rules for more detailed subsets of the module can also override this default. In any case, in order to write these rules, the user has to locate the module in the graph and invoke the graphical policy editor; however, locating the module is actually the difficult part of the annotation. To address this task, Hecataeus comes with a layout containing the filesystem of the project that the user investigates. Initially, the user has to find the file that contains the query he wants to change its policy. When the user selects a file, the queries that are in this file, are highlighted in the visual representation of the *Architecture Graph* in our tool, providing a smaller set of modules that need to be searched. Finally, when the user finds the module he wants to differentiate from the global policy, he adds only *one* line of text to the policy file that says that this query has a specified policy. *We repeatedly monitored the annotation time, using a wristwatch, and this task takes at most 1 minute for each query that the user wants to set a specific policy.*

In each experiment, we also discuss the number of rules required for the execution of the experiment. We believe the annotation effort is practically negligible.

5.1.3 Experimental configuration

In all our experiments, we need to fix the following parameters for our experimental setup: (a) an ecosystem comprising a database schema surrounded by a set of queries and possibly a set of views, (b) a workload of events that are sequentially applied to the above configuration, and (c) a palette of "profiles" that determine the way the ecosystem's architecture graph is annotated with policies towards the management of hypothetical events; hence, these profiles simulate the intention of the administrating team for the management of the ecosystem.

5.2 Replaying the Evolution of Drupal

Ecosystem. In this experiment, we have employed Drupal, versions 4.1.0 to 4.7.11 [30] as our experimental testbed. Drupal is a Content Management System (CMS) that is written in PHP language, which contains SQL queries in its php script files. We used some of the major versions of this project that took place between 2003 and 2007. As one may notice in Table 6, there are no Views in this project; that is why we decided to split the experiment in two setups, described in Sections 5.2.1 and 5.2.2 respectively.

5.2.1 Original evolution scenario

In this setup, we replay the original evolution of the Drupal project, raising each one of the events that really occured, having no blocker modules.

Events. We have used the actual events that evolved the database schema of Drupal between the major versions we describe in Table 6. For example, in order to get to version 4.2.0 we had to perform 6 attribute additions and 2 attribute deletions.

Policies. The default reaction for the original scenario was to accept all changes between two subsequent versions. Thus, the policy for all modules was to propagate all events that occur on the system; this is expressed by having only one rule in the policy file (`NODE: on * THEN PROPAGATE;`).

5.2.2 Modified scenario with view cloning

In the second setup, we replay an alternative evolution case of the Drupal project, in order to examine the effect of cloning of views on the overall system. Specifically, we added a view named: "*UNView*", that is used to join the `USERS` and `NODE` relations. Then, we rewrote all queries joining the two tables to use the view instead. Moreover, we added one extra query that asks for all the attributes of *UNview* which would act as a blocker to all events that ultimately reach it. This setting allows us to see how view cloning operates "in the microscope".

Events. We have also used the actual events as in the previous setup. The only difference to the previous approach is that, when there was an attribute deletion in `USERS` or `NODE` relations, we performed the deletion to the *UNview* module, instead of the `USERS` or `NODE` relation modules.

Policies. The policy again was to propagate all the changes in all modules except for the additional query; this is expressed by the following two rules:

– `NODE: on * then PROPAGATE;` and
– `Qadditional: on * then BLOCK;`.

Experimental Protocol. We have used the following sequence of actions. First, we annotate the architecture graph with policies. Next, we sequentially apply the events over the graph – i.e., each event is applied over the graph that resulted from the application of the previous event. For each event we measure the elapsed time for each of the three algorithms, along with the number of affected, cloned and adapted modules. The experiment was performed in a typical PC with an Intel

Drupal Version	Published at	Relations	Queries	Attribute Addition	Attribute Deletion	Table Addition	Table Deletion
4.1.0	2003-02-01	38	234	6	2	0	0
4.2.0	2003-08-01	38	239	1	5	3	1
4.3.1	2003-12-01	40	251	8	4	1	1
4.4.3	2005-06-01	40	254	16	5	16	4
4.5.8	2006-03-14	52	277	12	11	4	1
4.6.11	2007-01-05	55	327	14	11	7	5

Table 6 Drupal dataset from ver. 4.1.0 to ver. 4.7.11

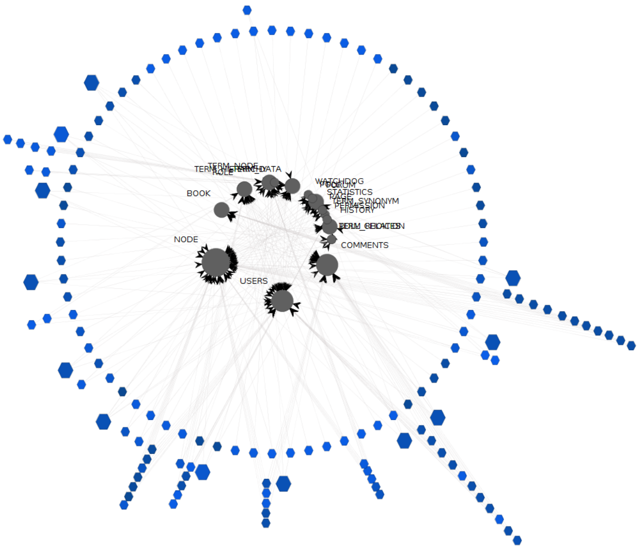


Fig. 19 Drupal 4.1.0 cluster with queries asking same tables as arcs.

i5 CPU at 2.90GHz and 32GB main memory and only one core being used.

5.2.3 *Annotation effort*

The "real world" experiment was conducted using the syntactic sugar policy annotation rules. When we used the setup that is described in Section 5.2.1 we did not have to write any rules (the default one is generated by our tool). When we used the setup that is described in Section 5.2.2, we had to write only one rule, in order to block the propagation of events to the extra query we deliberately inserted in the ecosystem.

5.2.4 *Effort gains*

In both variants of our experiments, we can see that the effectiveness is way too high for all events. This is because the average number of affected modules is small compared to the size of the graph. More precise results about this experimental setup you may see in Table 7, where we notice that the minimum benefit for

Metric		Version 4.*					
		1.0	2.0	3.1	4.3	5.8	6.11
%AM	min	97.1	88.2	94.2	71	93.1	79.7
	avg	**98.2**	**97.3**	**96.8**	**91**	**98.5**	**97.3**
	max	99.2	98.8	99.6	99.6	100	99.7
RM	min	2	4	2	2	1	2
	avg	**5.4**	**7.8**	**9.3**	**24.7**	**5**	**10**
	max	8	30	16	77	20	68

Table 7 Results of the original evolution scenario of Drupal

the developers is 71% while the average benefit ranges between 91% - 98.5%!

In the experimental setup that is described in Section 5.2.2, we can see that the metrics have not changed significantly. Also due to the blocker query and the *UNview* modules, we now have clones! This way, the query that was marked to block all the changes remains intact, while the rest of the ecosystem evolves.

The minimum number of clones per event is 0. Also, since the height of our tree is only one level, the maximum number of clones per event can not be greater than 1. Those metrics are displayed in Table 8.

Metric		Version 4.*					
		1.0	2.0	3.1	4.3	5.8	6.11
%AM	min	97.1	88.5	94.3	70.7	93.2	79.5
	avg	**98.2**	**96.8**	**96.9**	**91**	**98.5**	**97.2**
	max	99.2	98.8	99.6	99.6	100	99.7
RM	min	3	4	2	2	1	2
	avg	**5.4**	**8.4**	**9.2**	**24.7**	**5**	**10**
	max	8	30	16	79	20	70
$\sum$ *Cloned*		0	4	1	3	3	5

Table 8 Results of the modified evolution scenario of Drupal

5.2.5 *Efficiency assessment*

The time needed to perform the adaptation of the ecosystem is practically negligible. Table 9 displays the time needed for the original Drupal experimental setup, described in Section 5.2.1. Table 10 displays the time needed for the modified Drupal experimental setup, described in Section 5.2.2. The experiments of the Drupal project were conducted with cold cache (it is interesting

Step		Version 4.* 1.0	2.0	3.1	4.3	5.8	6.11
1	min	110	171	65	56	37	76
	avg	**1311**	**1732**	**1135**	**913**	**678**	**728**
	max	8048	8913	9035	8498	10426	11888
2	min	2	2	2	1	1	1
	avg	**4**	**4**	**7**	**7**	**5**	**8**
	max	11	15	25	28	24	64
3	min	105	154	76	48	29	59
	avg	**362**	**506**	**560**	**659**	**251**	**364**
	max	1282	1947	1773	2830	1812	1328

Table 9 Drupal project times (in microseconds) for "original" setup

Step		Version 4.* 1.0	2.0	3.1	4.3	5.8	6.11
1	min	120	323	81	46	40	124
	avg	**1357**	**2227**	**1241**	**929**	**632**	**686**
	max	8124	10782	9707	8957	87896	9706
2	min	3	3	3	1	1	1
	avg	**4**	**19**	**18**	**14**	**10**	**13**
	max	13	51	149	68	123	99
3	min	114	801	82	72	25	88
	avg	**395**	**8128**	**1443**	**2051**	**1316**	**2193**
	max	1364	15244	11917	13103	9177	11713

Table 10 Drupal project times (in microseconds) for "modified" setup

to note that in all occasions, the processing of the first event took an order of magnitude higher than the rest of the events; here we report the min, max and average of *all* events for each step).

5.3 Controlled experiment with TPC-DS

To better control and assess the behavior of our algorithms, we need a more complex environment than Drupal. In fact, our experience with several CMS's reveals that the internal structure of the database is intentionally kept as simple as possible, obviously in an attempt to maximize performance. Thus, we have employed a decision support benchmark, TPC-DS, as the testbed for our controlled experiment. We start with a description of the experimental setup.

5.3.1 Experimental setup for TPC-DS

Ecosystem. We have employed TPC-DS, version 1.1.0 [25] as our experimental testbed. TPC-DS is a benchmark that involves star schemata of a company that has the ability to *Sell* and receive *Returns* of its *Items* with the following ways: (a) the *Web*, or, (b) a *Catalog*, or, (c) directly at the *Store*. Moreover the company keeps data of *Customers*, regarding their *Income* band, or their *Demographics* data and additionally keep data about the *Promotion* of their *Items*. To handle advanced SQL constructs in the queries of TPC-DS, we had to add views for the handling of WITH clauses and to make modifications to queries containing keywords as LIMIT, HAVING in order to remove parser-offending parts that Hecataeus' parser cannot handle. To test the effect of graph size to our method's efficiency, we have created 3 graphs with gradually decreasing number of query modules: (a) a large ecosystem, *WCS*, with queries using all the available fact tables, (b) an ecosystem *CS*, where the queries to WEB_SALES have been removed, and (c) an ecosystem *S*, with queries using only the STORE_SALES fact table.

Events. The event workload consists of 51 events simulating a real case study of the Greek public sector. See Fig. 20, left, for an analysis of the module sizes within each scenario. In Fig. 20, right, we present the breakdown of the workload (listing the percentage of each event type as *pct*).

Policies. We have annotated the graphs with policies, in order to allow the management of evolution events. We have used two "profiles": (a) *MixtureDBA*, consisting of 20% of the relation modules annotated with BLOCK policy and (b) *MixtureAD*, consisting of 15% of the query modules annotated with BLOCK policy. The first profile corresponds to a developer-friendly DBA that agrees to prevent changes already within the database. The second profile tests an environment where the application developer is allowed to register veto's for the evolution of specific applications (here: specific queries). We have taken care to pick queries that span several relations of the database.

	Graph size S	CS	WCS
Queries	27	68	89
Views	25	48	95
Relations	25	25	25
Sum	**77**	**141**	**218**

Event type	*pct*
Attribute Add	37.3%
Attribute Rename	43.2%
Attribute Del	13.7%
Relation Rename	1.9%
View alter semantics	3.9%

Fig. 20 Experimental configuration for the TPC-DS ecosystem

Experimental Protocol. We have used the following sequence of actions. First, we annotate the architecture graph with policies. Next, we sequentially apply the events over the graph – i.e., each event is applied over the graph that resulted from the application of the previous event. The experiment was performed with hot cache in order to measure the time. For each event we measure the elapsed time for each of the three algorithms, along with the number of affected, cloned and adapted modules. The experiment was performed in a typical PC with an Intel Quad core CPU at 2.66GHz and 1.9GB main memory with only one core was being used.

5.3.2 Effectiveness assessment: How useful is our method for the application developers and the DBA's?

In this subsection, we discuss the evaluation of the effort gain metrics for our controlled experiment. We evaluated the $\%AM$ metric for each of the 51 events of the workload, performed over all three ecosystems (S, CS, WCS) and for both the policy annotation profiles ($MixtureDBA$ and $MixtureAD$). In the upper part of Fig. 21 we demonstrate the minimum, average and maximum value of the $\%AM$ metric for all these 51 runs, organized annotation policy and ecosystem. The results demonstrate that the effort gains compared to the absence of our method are significant, as, on average, the effort is around 90% in the case of the AD mixture and 97% in the case of the DBA mixture. As the graph size increases, the benefits from the highlighting of affected modules that our method offers, increase too. Observe that in the case of the DBA case, where the flooding of events is restricted early enough at the database's relations, the minimum benefit in all 51 events ranges between 60% - 84%.

	%AM – Mixture AD			*%AM – Mixture DBA*		
	S	CS	WCS	S	CS	WCS
min	21%	35%	30%	60%	78%	84%
avg	**89%**	**91%**	**92%**	**97%**	**96%**	**97%**
max	100%	100%	100%	100%	100%	100%

	RM – Mixture AD			*RM – Mixture DBA*		
	S	CS	WCS	S	CS	WCS
min	1	0	0	0	0	0
avg	**6.22**	**10.00**	**13.47**	**2.47**	**5.00**	**6.22**
max	38	66	117	22	27	30

Fig. 21 Effectiveness assessment as fraction of affected modules (%AM) and number of rewritten modules (RM) of the "controlled" experiment

Likewise, we evaluated the RM metric for each of the 51 events of the workload. The results demonstrate that the minimum number of modules needing a rewrite is 0 for almost all combinations of mixture and graph size for the event workload. This happened in both the $MixtureDBA$ and the $MixtureAD$ cases for different reasons – still both related to a veto. In the case of $MixtureDBA$, if a relation vetoes a possible change to it, then the event is immediately blocked and no rewriting or cloning takes place. Similarly, if a query vetoes a change in a relation (eg. attribute deletion), again, the event is frozen no rewriting or cloning takes place. At the same time, observe that the average and maximum number of modules needing a rewrite increases as the size of the graph increases. This is expected, as the increase in the graph size signifies that the new queries can possibly use some of the tables/views of the smaller graph (remember that the graphs are constructed by adding views and queries each time). Then, every event affects more modules as the graph increases. Another worth mentioning fact is that when the $MixtureDBA$ policy is used, the number of the modules needing rewrite drops, since the flooding of events is restricted early enough, inside the database.

5.3.3 Policy annotation effort: how many rules does one have to write in order to work with our what-if analysis tool?

In this subsection, we discuss the effort of the user for the annotation of the *Architecture Graph* ecosystem with policies, over the conducted controlled experiments. We have worked with both policy mixtures and observed the effort needed as the number of blockers increases. Remember that in the MixtureDBA policy of the "controlled" experiment we block events at relations; we have set 20% of our *relation* modules to block the events that they receive and kept the size of the *relation* modules is the same in all three experiments (*S*, *CS*, and *WCS*). In the MixtureAD policy –in the same experimental setup– we set about 15% of the *query* modules as blockers. Here, the number of the blockers depends on the numbers of the query modules, which is different for each one of the *S*, *CS*, and *WCS* experiments.

Table 11 displays our results. We have one row for the MixtureDBA and one row per size (*S*, *CS*, *WCS*) for the MixtureAD policy. The first columns explain the annotation policy, the nature and number of interesting modules (relations in the first case and queries in the latter) and the number of blockers within each configuration. The next three columns explain the effort spent to annotate the ecosystem without the syntactic sugar: we list the number of default rules that have to be defined for completeness reasons, the extra rules that pertain to the individual blocker modules and the sum of these two measures. Finally, the last group of columns, refers to the case where the syntactic sugar was available, in a manner similar to the previous.

For the case where the syntactic sugar was not used, we have to mention the following observations and clarifications. At first, the number of default rules (33) seems quite high. However, we should also mention that Hecataeus supports the user gracefully by offering the template list ready to the user. At the same time, the number of extra rules per blocking module is about 9 rules per module. Although the numbers for the entire ecosystem reach to a high level, in the regular case

					Without syntactic sugar			With syntactic sugar		
Mixture	Size	Modules of interest	Modules	Blockers	Default	Extra	Total	Default	Extra	Total
DBA	all	Relations	25	5	33	25	58	**1**	5	**6**
AD	S	Queries	27	4	33	36	69	**1**	4	**5**
AD	CS	Queries	68	10	33	90	123	**1**	10	**11**
AD	WCS	Queries	89	12	33	118	151	**1**	12	**13**

Table 11 Modules and rules for policy annotation effort.

where the annotation is performed in an incremental fashion, the ratio of 9 rules per module seems quite tolerable.

For the case where we have exploited the syntactic sugar, the set of rules needed decreases dramatically. This is due to the dramatic decrease in both the default rules (1 rule only) and the necessary rules per module (again one rule only). Specifically, observe that we can annotate with PROPAGATE policies the entire graph using only one rule (*NODE: ON * THEN PROPAGATE;*), and for each one of the blocker modules, we need to use, again, only one rule (*<namedNode>: ON * THEN BLOCK;*). Overall, the savings in effort and the speedup are too high both in batch and incremental ways of using our method.

5.3.4 Efficiency: how fast can we interact with our what-if analysis tool?

In this subsection, we evaluate the time needed to complete the process of what-if analysis with our tool. Efficiency plays an important role in the design and administration process of an ecosystem, if we wish to allow the involved stakeholders to interactively test alternative configurations and scenarios for policy annotations or restructuring of the ecosystem's architecture to accommodate forthcoming changes gracefully. To this end, we investigate the effect of policy annotation and graph size to the completion time and its breakdown in the three phases of our method.

Effect of policy to the execution time. In the case of *Mixture DBA* we follow an aggressive blocking policy that stops the events early enough, at the relations, before they start being propagated in the ecosystem. On the other hand, in the case of *Mixture AD*, we follow a more conservative annotation approach, where the developer can assign blocker policies only to some module parts that he authors. In the latter case, it is clear that the events are propagated to larger parts of the ecosystem resulting in higher numbers of affected and rewritten nodes. If one compares the execution time of the three cases of the AD mixture in Fig. 22 with the execution time of the three cases of the DBA mixture the difference is in the area of one order of magnitude. It is however interesting to note the internal differences: the status determination time is scaled up with a factor of two; the rewriting time, however is scaled up by a factor of 10, 20 and 30 for the small, medium and large graph respectively!

Another interesting finding concerns the **internal breakdown of the execution time** in each case. A common pattern is that *path check is executed very efficiently*: in all cases it stays within 2% of the total time (thus practically invisible in the graphic). In the case of the AD mixture, the analogy between the status determination and the graph rewriting part starts from a 24% - 74% for the small graph and ends to a 7% - 93% for the large graph. In other words, *as the events are allowed to flow within the ecosystem, the amount of rewriting increases with the size of the graph*; in all cases, it dominates the overall execution time. This is due to the fact that rewriting involves memory management (module cloning, internal node additions, etc) that costs much more than the simple checks performed by *Status Determination*. In the case of the DBA mixture, however, where the events are quickly blocked, the times are not only significantly smaller, but also equi-balanced: 57% - 42% for the small graph (*Status Determination* costs more in this case) and 49% - 50% for the two other graphs. Again, this is due to the fact that the rewriting actions are the time consuming ones and therefore, their reduction significantly reduces the execution time too.

Effect of graph size to the execution time. To assess the impact of graph size to the execution time one has to compare the three different graphs to one another within each policy. In the case of the AD mixture, where the rewriting dominates the execution time, there is a linear increase of both the rewriting and the execution time with the graph size. On the contrary, the rate of increase drops in the case of the DBA mixture: when the events are blocked early, the size of the graph plays less role to the execution time.

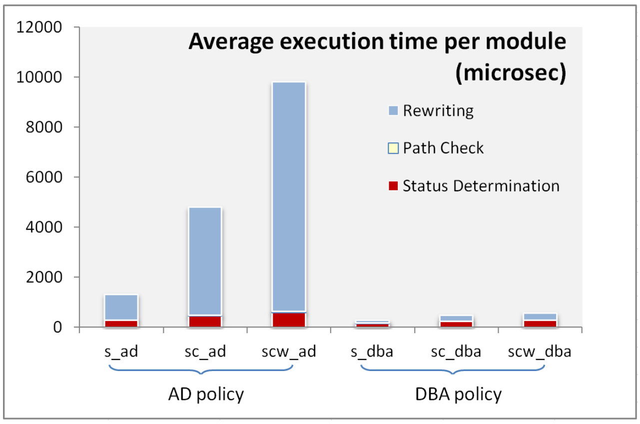


Fig. 22 Efficiency assessment for different policies, graph sizes and phases

Overall, the main lesson learned from these observations is that the annotation of few database relations significantly restricts the rewriting time (and consequently the overall execution time) when compared to the case of annotating modules external to the database. In case the rewriting is not constrained early enough, then the execution cost grows linearly with the size of the ecosystem.

6 Related work

Schema evolution is a long-studied problem in database and software research [21], [22]. For an overview of the related work, we refer the interested reader to a recent survey [8] and an up-to-date list of related publications[4]. In this section, we focus our discussion to works related to the adaptation of data-intensive ecosystems to schema evolution operations. We first highlight the main approaches that deal with evolution in relational data-intensive systems, mappings and view rewritings, bidirectional transformations and DW evolution. Then, we present and discuss the differences and contributions of our current work with them.

Evolution of Data-intensive ecosystems. Research activity related to ecosystems built on top of relational databases has been recently developed around two tools, Hecataeus and Prism.

Hecataeus is based on the notion of *Architecture Graphs*, as we have already seen. The first version of *Architecture Graphs* was introduced in [14] and comprehensively described in [18], along with the first version of an algorithm for the propagation of the changes of one entity to other related entities (see the end of this section for the improvements to the model and the algorithms that we introduce here). The annotation of tables, views and queries with policies came with an extension of SQL presented in [16]. In a different line of research, in [17], the authors proposed a set of graph metrics that assess the vulnerability and the maintenance effort of adapting a database ecosystem to evolution events. The assessment of these works has taken place over the evolution of real-world ETL scenarios.

Prism [2], [3] introduces a method for rewriting queries whenever their underlying database schema changes, with the aim of retaining the original query semantics. The authors introduce a set of Schema Modification Operators (SMOs) for categorizing schema changes. SMOs can be simple schema operations on tables, such as `CREATE TABLE`, `DROP TABLE`, `ADD COLUMN` or more complex operations between tables, such as `MERGE TABLES` and `JOIN TABLES`. Besides the SMOs, Prism considers changes on constraints and proposes a set of integrity constraints modification operators (ICMO) such as `ADD PRIMARY KEY` and `ADD FOREIGN KEY`. Policies (`CHECK`, `ENFORCE`, `IGNORE`) are also used in Prism, for enforcing data consistency in tables that evolve, rather than regulating the rewriting of queries and views. For example, when `ENFORCE` policy accompanies an `ADD PRIMARY KEY` change, then all violating tuples must be removed in order to help DBA carry out consistency validations. Regarding the rewriting process, the authors propose the Chase & Backchase algorithm which uses as input the SMOs and the query to be rewritten and applies the invert operation on its syntax such that the query retains its results unchanged. Finally, in [3], the authors extend their techniques to DML statements, as well.

Two noteworthy approaches from the areas of software engineering pertain to this work. First, an approach based on software slicing is presented in [11] where the authors propose techniques for the identification of the impact of relational database schema changes upon object-oriented applications. At first, the authors identify the database *queries* within the software and perform data-flow analysis for estimating the possible runtime values of the query. Then, they use dynamic slicing [5] for extracting the *lines of code of the program* related to the query. Second, [29] presents a method for analyzing the evolution of object-oriented software systems from the point of view of their logical design. The authors' method studies the lifetime of UML class models and performs several steps of analysis to determine phases, patterns, and similarities in the lifetime of the classes.

View/schema mappings rewriting and Bidirectional transformations. Another area related to our problem involves the works related to the adapta-

[4] `http://dbs.uni-leipzig.de/en/publications`

tion and rewriting of views and schema mappings as well as the works on bidirectional transformations. Regarding view rewriting, in [13] the authors propose legal rewritings of views affected by changes, focusing on the case of relation deletion by finding valid replacements for the affected (deleted) components via a meta-knowledge base (MKB) that keeps meta-information about attributes and their join equivalence attributes on other tables in the form of a hyper-graph. [7] redefines a *materialized* view as a sequence of primitive local changes in the view definition. On more complex adaptations, those local changes can be pipelined in order to compute the new view contents incrementally and avoid their full re-computation. In [26], the authors deal with the maintenance of a set of mappings in an environment where source and target schemata are integrated under schema mappings implemented via SPJ queries. Finally, [1] introduces mappings among the applications and a conceptual representation of the database, again mapped to the database logical model; when the database changes, the mappings allow to highlight impacted areas in the source code.

Regarding bidirectional transformations, [23] provides a survey on the techniques and tools related to the use of bidirectional transformations. The basic idea behind $Bidirectional\ Transformations$ is that "there are two models or schemata S and T and a mapping between them M that serves as a bridge to allow operations and data to flow between the two models. M must conform to the bidirectional properties that govern the quality of synchronization between S and T" [23]. The authors examine 5 different approaches of bidirectional transformations, namely: *Lenses* [4] which is a mathematical abstraction centered around a pair of functions called *get* and *put*, working on models S, which describes the source, and T, which is the target; $Schema\ Modification\ Operators$ [2] as previously described; $Channels$ [24], which is a discreet bidirectional transformation from S to T, described by a 4-tuple of functions (`S`,`I`,`Q`,`U`), that operate on Schema, data Instances, Queries and Updateds respectively;$DB-MAIN$ [1] which is a transformation toolkit, where the evolution transformations consist of a target schema T that derives from a source schema S, when a construct C (entities, relationships, attributes, constraints, etc.) is replaced by a new one, called C'; and finally, the $Both-As-View$ (BAV) approach [12], where the schemata of several databases are integrated to form a virtual database, with a global schema with a combination of LAV and GAV assertions. Bidirectional transformations come in a declarative way (as opposed to our graph-based representation), with the requirement of reversible transformations (so that both directions of the mapping work) and aim to support (with different degrees of effectiveness) the traditional data integration tasks (data migration, query rewriting and dispatching), cross-version transformations and inter-model mappings.

Data-warehouses evolution. Evolution in data warehouses is quite related to our problem as it involves the handling of interdependencies in a complex data ecosystem comprising data sources, ETL flows, warehouse tables and data marts. In this context, in [28] the authors deal with inconsistencies arising by schema changes on the external data sources of a data warehouse. They propose a method that uses wrappers connected to a monitor for detecting predefined events on the external sources and generating actions for the DBA. In [6], the authors employs a graph representation for data warehouse schemata and define an algebra for graph modifications that can be used to create new schema versions. They deal with multiple versions of the underlying database and discuss how cross-version queries can be answered with the help of augmented data warehouse schemata. Finally, [27] proposes a layered architecture for data warehouses, in order to achieve a much clearer, dedicated assignment of data transformation to each layer, providing –according to the authors– flexibility, consistency, re-usability and scalability of data.

Comparison to existing approaches. A first feature of our approach is that it enables the processing of multiple messages arriving at a module. The reader may wonder why simply flooding the *Architecture Graph* with messages on an event is not sufficient to solve the problem of impact analysis. Simply flooding compromises the confluence of the method as the same node might receive more than one messages (possibly contradicting in the presence of policies) for the same event. Our algorithm achieves confluence by properly processing all messages within a module before propagating its impact to next dependent consumers. Another distinctive feature is the presence of policies, exactly placed to avoid the "blind" flooding of messages and regulate the flow. As already mentioned, the annotation of the ecosystem with policies imposes the new problem of maintaining different replicas of view definitions for different consumers; to the best of our knowledge, this is the first time that this problem is handled in a systematic way. Interestingly, although the existing approaches make no explicit treatment of policies for the blocking or the propagation of evolution, they differ in the implicit assumptions they make. Nica et al., operating mainly over virtual views [13], actually block the flooding of a deletion event by trying to compensate the deletion with equivalent expressions. Velegrakis et al. [26] move towards the same goal but only for SPJ queries. On the other hand, Gupta et al., [7], working with materialized views, are focused to adapting the contents of the views, in a propagate-all fashion. A problem coming with a propagate-all policy is that the events might affect the semantical part of the views/queries (WHERE clause) without any notification to the involved users (observe that the problem scales up with multiple layers of views defined over other views). Bidirectional transformations, although very promising as a set of techniques, require the construction of assertions in a declarative language (which is not a straightforward task to automate) and they are typically tailored for different tasks than the ones addressed here (i.e., policy-based impact analysis and rewriting for data-intensive ecosystems): DB-MAIN for mappings among conceptual-logical-and physical models, Channels for logical-physical mappings among applications and data, BAV and lenses for data integration. Curino et al. [2] are not trying to adapt the queries to a new schema, but they are trying to rewrite queries in order to revert the schema modifications. Additionally, the proposed methods restrain rewritings only to changes in a relation; this way, Prism does not follow view redefinitions, which would cause problems in query defined over those views. Moreover, if a relation changes causing redefinitions on views used by queries then the queries are not automatically rewritten, causing again problems on the queries definitions. Emmerich et al. [11] on the other hand, stop their approach at the impact analysis step, without performing any rewritings of the entities that are affected by a change. Finally, Xing and Stroulia [29] provide a comprehensive method for class profiling that presents interesting opportunities for future work, if applied in the context of data-intensive ecosystems.

Concerning our previous works, as already mentioned, in ([18], [19]), we have presented our method for the background modeling, a first version of the mechanism of impact assessment in the presence of policies and the system architecture for our tool Hecataeus. In [10], we have extended the aforementioned works (a) by exploiting the scoping offered by the *new graph model* for the ecosystem's modules to provide a status determination mechanism with correctness guarantees, (b) by introducing path checking and multiple versions to resolve the adaptation of conflicting policies, and, (c) by presenting the management of rewritings to accommodate change. In this paper, we significantly extend [10] in several ways. First, we present the full-blown architecture graph model, along with the space of events. Second, we introduce a policy determination language to reduce programmer's effort in annotating the graph. Third, we discuss the message structure and an extensible software architecture for the propagation of the change. Fourth, we formally prove the correctness guarantees of our method. Finally, we complement the experimental findings with extra metrics that concern the extent of rewriting as well as the effort needed for annotating the graph with policies. Thus, we provide a thorough, rigorous and comprehensive record of our technique and software for managing the evolution of data-intensive ecosystems. All the material for this work, including input ecosystems, links to the source code (publicly available at git) and results can be found in the paper's web page: `http://www.cs.uoi.gr/~pmanousi/publications/2013_ER/index.html`.

7 Conclusions and Future Work

In this paper we have addressed the problem of adapting a data-intensive ecosystem in the presence of policies that regulate the flow of evolution events. Our method allows (a) the management of alternative variants of views and queries and (b) the rewriting of the ecosystem's affected modules in a way that respects the policy annotations and the correctness of the rewriting (even in the presence of policy conflicts). Our experiments confirm that the adaptation is performed efficiently as the size and complexity of the ecosystem grow.

Future work can continue in several directions. In this paper, we have performed what-if analysis where each time, only a single event is assessed. Future work can address the assessment of complicated events, involving a set of possible changes simultaneously applied over either the same or different modules. This would also involve some extra "garbage collection" of views that are redundant or useless. The possibility of adding more semantics to the Architecture Graph is also a possible path for future research. For example, constraints that are not necessarily extracted from the reverse engineering of the database, like functional or conditional functional dependencies, or logical constraints within the source code (e.g., pre- and post-conditions over the correctness of a stored procedure) can also become part of the graph. Adding more kinds of sources, like for example, web-services, or XML stores to the graph is also a possibility. Providing hints to the DBA's or the developers for policies in a semi-automatic way can also help with the annotation of the graph. Finally, algorithms for the visualization of the ecosystem can be an ongoing topic of research for long.

Acknowledgments. We would like to thank the reviewers of this paper for their constructive comments. This research has been co-financed by the European

Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

1. A. Cleve, A.-F. Brogneaux, and J.-L. Hainaut. A conceptual approach to database applications evolution. In *29th Intl. Conf. on Conceptual Modeling (ER), Vancouver, Canada*, pages 132–145, 2010.
2. C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *PVLDB*, 4(2):117–128, 2010.
3. C. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo. Automating the database schema evolution process. *VLDB J.*, 22(1):73–98, 2013.
4. J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 2007.
5. K. Gallagher and D. Binkley. Program slicing. In *Frontiers of Softw. Maint.* IEEE CS Press, 2008.
6. M. Golfarelli, J. Lechtenbörger, S. Rizzi, and G. Vossen. Schema versioning in data warehouses: Enabling cross-version querying via schema augmentation. *Data Knowl. Eng.*, 59(2):435–459, 2006.
7. A. Gupta, I. S. Mumick, J. Rao, and K. A. Ross. Adapting materialized views after redefinitions: techniques and a performance study. *Information Systems*, 26(5):323–362, 2001.
8. M. Hartung, J. F. Terwilliger, and E. Rahm. Recent Advances in Schema and Ontology Evolution. In Z. Bellahsene, A. Bonifati, and E. Rahm, editors, *Schema Matching and Mapping*, pages 149–190. Springer, 2011.
9. P. Manousis. Database evolution and maintenance of dependent applications via query rewriting. MSc. Thesis, Dept. of Computer Science, Univ. Ioannina., February 2013. http://www.cs.uoi.gr/~pmanousi/publications.html.
10. P. Manousis, P. Vassiliadis, and G. Papastefanatos. Automating the adaptation of evolving data-intensive ecosystems. In *32th International Conference on Conceptual Modeling (ER), Hong-Kong, China*, pages 182–196, 2013.
11. A. Maule, W. Emmerich, and D. S. Rosenblum. Impact analysis of database schema changes. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany*, pages 451–460, 2008.
12. P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 227–238, 2003.
13. A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *6th Intl. Conf. on Extending Database Technology (EDBT 1998), Valencia, Spain*, pages 359–373, 1998.
14. G. Papastefanatos, K. Kyzirakos, P. Vassiliadis, and Y. Vassiliou. Hecataeus: A Framework for Representing SQL Constructs as Graphs. In *Proceedings of 10th International Workshop on Exploring Modeling Methods for Systems Analysis and Design-EMMSAD*, Porto, Portugal, 2005.
15. G. Papastefanatos, P. Vassiliadis, and A. Simitsis. Propagating evolution events in data-centric software artifacts. In *ICDE Workshops*, pages 162–167, 2011.
16. G. Papastefanatos, P. Vassiliadis, A. Simitsis, K. Aggistalis, F. Pechlivani, and Y. Vassiliou. Language Extensions for the Automation of Database Schema Evolution. In *Proc. ICEIS (1), Barcelona, Spain*, pages 74–81, 2008.
17. G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. Design metrics for data warehouse evolution. In *27th International Conference on Conceptual Modeling (ER), Barcelona, Spain*, pages 440–454, 2008.
18. G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. Policy-Regulated Management of ETL Evolution. *J. Data Semantics*, 13:147–177, 2009.
19. G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou. HECATAEUS: Regulating schema evolution. In *Proceedings of the 26th International Conference on Data Engineering (ICDE), Long Beach, California, USA*, pages 1181–1184, 2010.
20. R. Pressman. *Software Engineering: A Practitioner's Approach: European Adaption.* McGraw-Hill, 5 edition, April 2000.
21. S. Ram and G. Shankaranarayanan. Research issues in database schema evolution: the road not taken. Working paper, Department of Information Systems, Boston University School of Management., 2003. http://smgapps.bu.edu/smgnet/Personal/Faculty/Publication/pubUploads/Shankar,_G_15.pdf?wid=1536.
22. J. F. Roddick. Schema evolution in database systems - an annotated bibliography. *SIGMOD Record*, 21(4):35–40, 1992.
23. J. F. Terwilliger, A. Cleve, and C. Curino. How clean is your sandbox? - towards a unified theoretical framework for incremental bidirectional transformations. In *5th International Conference on Theory and Practice of Model Transformations (ICMT), Prague, Czech Republic*, pages 1–23, 2012.
24. J. F. Terwilliger, L. M. L. Delcambre, D. Maier, J. Steinhauer, and S. Britell. Updatable and evolvable transforms for virtual databases. *PVLDB*, 3(1):309–319, 2010.
25. Transaction Processing Performance Council. The New Decision Support Benchmark Standard, April 2012. http://www.tpc.org/tpcds/default.asp.
26. Y. Velegrakis, R. J. Miller, and L. Popa. Preserving mapping consistency under schema changes. *VLDB Journal*, 13(3):274–293, 2004.
27. T. Winsemann, V. Köppen, and G. Saake. A layered architecture for enterprise data warehouse systems. In M. Bajec and J. Eder, editors, *CAiSE Workshops*, volume 112 of *Lecture Notes in Business Information Processing*, pages 192–199. Springer, 2012.
28. R. Wrembel and B. Bebel. Metadata management in a multiversion data warehouse. *J. Data Semantics*, 8:118–157, 2007.
29. Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Trans. Software Eng.*, 31(10):850–868, 2005.
30. Drupal Community. Drupal. http://ftp.drupal.org/files/projects/, 2014.