The Formal Framework of the HyperView System

Lukas C. Faulstich
<http://www.inf.fu-berlin.de/~faulstic>

Technical Report B 3/99

March 11, 1999

Abstract

In this report, we introduce the graph rewriting formalism on which the HyperView System is based. We first present a data model for clustered graphs and our notion of graph schemata and graph layers. Then we formalize our concept of nondeleting typed graph rewriting rules with application conditions on attributes based on the Algebraic Single Push Out Approach to graph transformation and present the construction of the derived graph resulting from applying a rule.

The main contribution of this report is the formalization of an efficient strategy for materializing HyperViews based on demand-driven rule activation. We introduce the notion of an *oracle* against which *queries* in form of graph patterns can be posed. We show how to combine the rule set of a HyperView with an oracle to form a more powerful oracle that materializes this HyperView as a response to queries against it.

Finally we treat the problem of avoiding the introduction of redundancies in view graphs by reusing already materialized graph elements.

1 Clustered Graph Data Model (CGDM)

1.1 Motivation

The HyperView methodology perceives the WWW as a part of a large data graph, where each syntax tree of a HTML page forms a subgraph connected to other syntax trees by edges modeling hyper links. Traversing these edges causes the target pages to be loaded, parsed, and added to the graph on the fly. This structure motivates a modularization of the graph into so-called *clusters*.

Each HTML page is modeled by a cluster of the graph. On top of these HTML clusters a hierarchy of views is established. Each view extracts, combines, and restructures information to build a view-cluster at a higher level of abstraction. In particular, we introduce the *Abstract Content Representation (ACR) level* to organize the relevant information from each Web Site in a cluster of its own, the *ACR cluster*. In this cluster all irrelevant details of the layout are omitted while its overall structure is preserved. At the *database level* this information is integrated into the site independent and domain specific *database cluster*. Finally, the result level contains HTML clusters for result pages returned to the user's browser.

Each cluster in the data graph is structured according to a (sub)schema which forms a cluster of the global schema. There is a generic schema for HTML pages, an ACR schema for the ACR of each Web Site, and a database schema for the database cluster.

We introduce the *Clustered Graph Data Model* (CGDM) by extending the concept of directed labeled graphs. In a clustered graph, each vertex belongs to a *cluster*, and each edge between two clusters belongs to a *dependency* connecting these clusters. The term "dependency" stems from the fact that in our view mechanism, clusters containing derived data are connected to the clusters containing the original data by dependencies.

Edges between elements of the same cluster belong to a circular dependency from the cluster to itself. Vertices belonging to the same cluster form together with the edges connecting these vertices a subgraph of the global graph. In our data model *clusters* and *dependencies* are special vertices and edges which form a graph that specifies the module structure of a clustered graph. In Figure 1, an example of a clustered graph is shown.

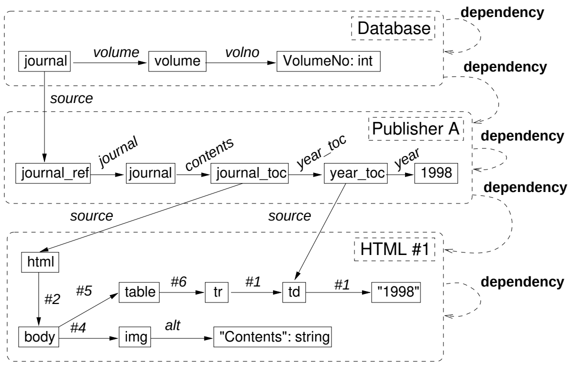

Figure 1: Example of a clustered graph

1.2 Basic definitions

As data model, we use a clustered graph data model called CGDM. The idea of clustered graphs is to modularize a large graph by introducing clusters and dependencies as first class objects which group vertices and edges, respectively. To each edge there exists a corresponding dependency which connects the cluster of its source with the cluster of its target. Hence, clusters and dependencies form a graph inside the clustered graph which summarizes the graph formed by vertices and edges.

Definition 1.1 (Plain Graph, Plain Graph Morphism)
An **(A-labeled) plain graph** $G = (V, E, A, s, t, a)$ consists of:

1. disjoint finite sets V of **vertices**, E of **edges**
2. an universal algebra A (the **attribute algebra**)
3. total functions $s, t : E \longrightarrow V$ indicating the **source** and **target** of an edge
4. a total **attribute** function $a : V \uplus E \longrightarrow A$

Let $G = (V_G, E_G, A_G, s_G, t_G, a_G)$ and $H = (V_H, E_H, A_H, s_H, t_H, a_H)$ be plain graphs. A plain graph morphism $f : G \longrightarrow H$ is a pair $f = (f_{vertex}, f_{edge})$, where $f_{vertex} : V_G \longrightarrow V_H$, $f_{edge} : E_G \longrightarrow E_H$ are total functions commuting with the source and target functions ($f_{vertex} \circ s_G = s_H \circ f_{edge}$ and $f_{vertex} \circ t_G = t_H \circ f_{edge}$). □

Remark 1.1 A plain graph morphism $f : G \longrightarrow H$ is a purely structural mapping which does not pose any constraints on the attribute function A_H. □

Definition 1.2 (Clustered Graph)
A **clustered (A-labeled) graph** $G = (G_{base}, G_{struct}, c)$ consists of two plain A-labeled graphs $G_{base} = (V, E, A, s_{edge}, t_{edge}, a_{base})$ (the **base graph**) and $G_{struct} = (C, D, A, s_{dep}, t_{dep}, a_{struct})$ (the **structure graph**) together with a plain graph morphism $c : G_{base} \longrightarrow G_{struct}$, the **clustering morphism**. We call the carrier sets C and D **clusters** and **dependencies**, respectively. □

Remark 1.2 Let $G = (G_{base}, G_{struct}, c)$ a clustered graph as defined above. Without loss of generality we require that all carrier sets V, E, C, D, and A are pairwise disjoint. We use the notation $G = (V, E, C, D, A, s, t, a, c)$ by using the definitions $s := s_{edge} \uplus s_{dep}$, $t := t_{edge} \uplus t_{dep}$, $a := a_{base} \uplus a_{struct}$, and $c := c_{vertex} \uplus c_{edge}$. To distinguish between different graphs, the graphs will be used as subscript, i.e., E_G, s_G etc.

Furthermore, we call edges, vertices, clusters, and dependencies **graph elements** and use the notation $x \in G$ to state that $x \in V_G \uplus E_G \uplus C_G \uplus D_G$ is an element of the clustered graph G.

In the following we call clustered graphs simply **graphs**. □

Definition 1.3 (Graph Morphism) A **graph morphism** $f : G \longrightarrow H$ between (clustered) graphs $G = (G_{base}, G_{struct}, c_G)$ and $H = (H_{base}, H_{struct}, c_H)$ is a triple $f = (f_{base}, f_{struct}, f_{attr})$ where $f_{base} : G_{base} \longrightarrow H_{base}$, $f_{struct} : G_{struct} \longrightarrow H_{struct}$ are plain graph morphisms commuting with the clustering morphisms ($c_H \circ f_{base} = f_{struct} \circ c_G$) and $f_{attr} : A_G \longrightarrow A_H$ is an algebra homomorphism compatible with f_{base} and f_{struct} ($f_{attr} \circ a_G = a_H \circ (f_{base} \uplus f_{struct})$).

The graph G is called the **domain** of f, denoted $dom(f)$. □

Remark 1.3 Using the flat notation for the graphs $G = (V_G, E_G, C_G, D_G, A_G, s_G, t_G, a_G, c_G)$ and $H = (V_H, E_H, C_H, D_H, A_H, s_H, t_H, a_H, c_H)$, a graph morphism $f = (f_{base}, f_{struct}, f_{attr}) : G \longrightarrow H$ can be represented by a family $(f_{vertex}, f_{edge}, f_{cluster}, f_{dep}, f_{attr})$ of *total* functions which map each of the sets V_G, E_G, C_G, D_G, A_G of G to the corresponding sets of H (i.e., $f_{vertex} : V_G \longrightarrow V_H$, $f_{edge} : E_G \longrightarrow E_H$, ...) and commute with the functions s, t, a and c in all possible compositions (e.g., $f_{vertex} \circ s_G = s_H \circ f_{edge}$, $f_{cluster} \circ c_G = c_H \circ f_{vertex}$,...).

This implies that the category of clustered graphs with graph morphisms forms an instance of an **attributed graph structure** with total morphisms as defined in [5, 4]. □

This definition implies in particular that the source (target) of an edge is always mapped to the source (target) of the image of this edge. Graph morphisms may not be injective in general. A path in a graph can be mapped to a circular edge, and several edges emanating from a vertex can be mapped to the same edge. Since we have graph morphisms defined to be total, all graph elements are included in the mapping. In the following, we denote the image of a graph element (i.e., vertex, edge, cluster, or dependency) x under a graph morphism m with $m(x)$.

Definition 1.4 (Compatible Functions or Morphisms) Let f, g be two functions or morphisms defined on overlapping domains $dom(f)$ and $dom(g)$.

Then f and g are **compatible** (denoted $f \nabla g$) if they coincide on the intersection of their domains, i.e., $f|_D = g|_D$ where $D = dom(f) \cap dom(g)$. $\square$

In the HyperView methodology we group the clusters of a graph into different layers with dependencies only within layers or between succeeding layers.

Definition 1.5 (Layered Graph) Let $G = (V, E, C, D, A, s, t, a, c)$ be a clustered graph and $l : C \longrightarrow \{1, \ldots, N\}$ a function which assigns each cluster $c \in C$ a level $l(c)$ such that for all dependencies $d \in D$ the level of the target is equal to or the predecessor of the level of its source, i.e., $\forall d \in D : l(s(d)) \in \{l(t(d)), l(t(d)) + 1\}$. $\square$

Since clustered graphs are a special case of *attributed graph structures* which in turn are many-sorted algebras [6], the notion of *subgraph* is inherited directly from the notion of *subalgebra*:

Definition 1.6 (Subgraph) Let $G = (V, E, C, D, A, s, t, a, c)$ be a clustered A-labeled graph. A clustered A-labeled graph $G' = (V', E', C', D', A', s', t', a', c')$ is a subgraph of G (denoted $G' \sqsubseteq G$) if the carrier sets V', E', C', D' of G' are subsets or equal to the respective carrier sets V, E, C, D of G and the operations s', t', a', c' are restrictions of the respective operations s, t, a, c. $\square$

Definition 1.7 (Reachability) Let $G = (V, E, A, s, t, a)$ be a plain graph. Then the **reachability relation** $\rightsquigarrow$ is defined by $(s^{-1} \cup t)^*$.

Let $G = (G_{base}, G_{struct}, c)$ a clustered graph. Then reachability between vertices and edges is defined as reachability on G_{base}, and reachability between clusters and dependencies is defined as reachability on G_{struct}. $\square$

Definition 1.7 implies that every edge is reachable from its source and the target of an edge is reachable from the edge.

Corrolary 1.1 If an element $y \in V \uplus E$ of the base graph is reachable from another element $x \in V \uplus E$ of the base graph ($x \rightsquigarrow y$), then the same holds for the corresponding elements $c(x), c(y) \in C \uplus D$ in the structure graph ($c(x) \rightsquigarrow c(y)$).

Proof: Since c is a graph morphism, it commutes with the source and target functions s, t. Hence $x = s(y) \implies c(x) = s(c(y))$ and $t(x) = y \implies t(c(x)) = c(y)$ proves this correspondence for the case $(s^{-1} \cup t)^i$, where $i = 1$. By induction, it follows for $(s^{-1} \cup t)^*$. $\square$

1.3 Schemata and instances

Definition 1.8 (Atomic Data) We fix a multi-sorted signature $\Sigma = (\mathbb{T}, \mathbb{O})$ for a set $\mathbb{T} = \{T_1, \ldots T_n\}$ of sorts and $\mathbb{O}$ of operation symbols on these sorts.

With $T_\Sigma(\mathbb{V})$ we denote the multi-sorted term algebra for signature Σ over a multi-sorted set $\mathbb{V}$ of variables.

We denote by $type : T_\Sigma(\mathbb{V}) \longrightarrow \mathbb{T}$ the algebra homomorphism which assigns each term its type in $\mathbb{T}$.

We define the universe $\mathbb{U}$ of atomic data to be the multi-sorted term algebra $T_\Sigma(\emptyset)$ of ground terms over signature Σ.

$\square$

Definition 1.9 (Pattern Graph, Data Graph, Schema Graph) A **pattern graph** (with variable set $\mathbb{V}$) is a $T_\Sigma(\mathbb{V})$-labeled clustered graph, i.e., its labels are (possibly non-ground) terms over Σ.

A **data graph** is a pattern graph whose labels are ground, i.e. it is a $\mathbb{U}$-labeled clustered graph.

A **schema graph** is a $\mathbb{T}$-labeled clustered graph, i.e., its labels are atomic data types $T_i \in \mathbb{T}$. $\square$

Remark 1.4 A data graph is a pattern graph carrying ground labels only. □

Example 1.1 (Schema for EJournal Publishers) Figure 2 shows a part of the schema developed for the integration of electronic journals in a Digital Library. This diagram shows only the ACR schema cluster for the Web Site of a particular publisher [1], together with the database schema cluster describing electronic journals in a publisher-independent way.

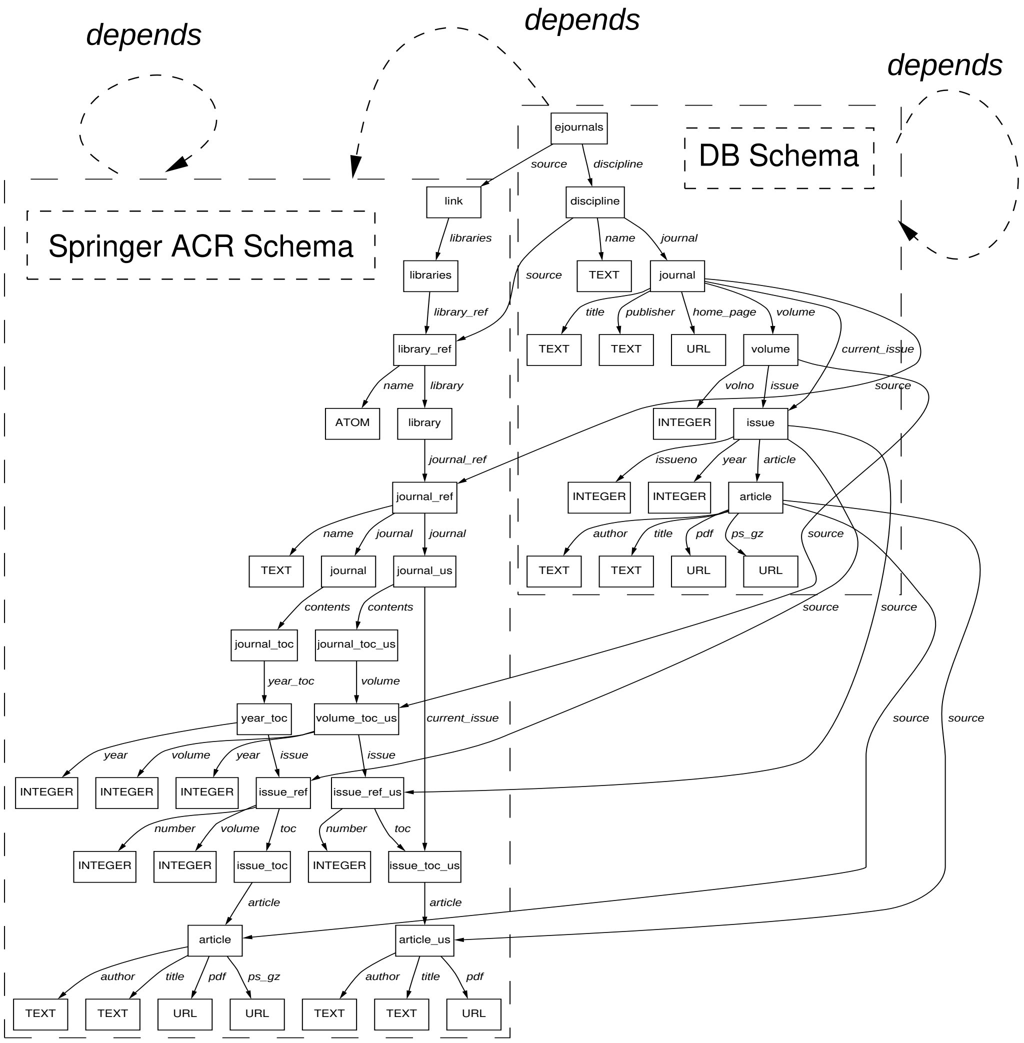

Figure 2: Clustered schema for Database and Springer ACR. Uppercase labels denote type predicates (e.g. INTEGER) which are fulfilled by all elements of that type, lowercase labels denote predicates matching only the constant indicated by their name (e.g. journal).

Correspondences between nodes of the ACR and database schema are indicated by source-edges. Note that certain nodes of the ACR schema appear in two versions (with and without extension _us, e.g., journal and journal_us). This is due to the fact that journals published by the US branch of Springer have a different layout, even though they are part of the same Web Site. □

To define the structural conformance of a data graph to a schema, we introduce the notion of an *interpretation* for the data graph in terms of a schema graph.

[1]Springer Berlin Heidelberg New York, <link.springer.de>

Definition 1.10 (Typing, Interpretation, Conformance, Instance) Let S be a schema and G be a pattern graph.

A graph morphism $\tau : G \longrightarrow S$ is a **typing** of G w.r.t. S, if its attribute component is the typing function, i.e., $\tau_{attr} = type : \mathbb{U} \longrightarrow \mathbb{T}$ such that τ assigns to each element x of G a schema element whose label $a_S(\iota(x))$ equals the type $type(a_G(x))$ of the label of x.

A pattern graph G **conforms** to a schema S, if there exists a typing $\tau : G \longrightarrow S$.

A typing $\iota : G \longrightarrow S$ is called an **interpretation** if G is a data graph (i.e., has ground labels only). In this case we call G an **instance** of S. □

This definition extends the typing concept of [3] to attributed graph structures. It implies following:

- there may be several interpretations for G w.r.t. S
- several parts of the instance graph may be interpreted by the same part of the schema
- an interpretation must cover all elements of the instance graph
- not all schema elements must have corresponding data elements. In particular, the empty data graph conforms to any schema.

In [1] a schema concept is presented which is based on schema graphs labeled with unary predicates. Conformance depends on the existance of a *simulation* relation between instance and schema graph. This schema concept is more general than ours. In particular, predicates may have overlapping solution sets whereas our atomic data types are disjoint. Predicates can model application specific data types like movie titles or names of months which are subtypes of more general types, e.g., string.

However, this limitation can be overcome by introducing application specific atomic data types and use conversion functions in rules (discussed in the next section) to convert instances of general data types into instances of application specific types.

Definition 1.11 (Type-Compatible Morphism) A morphism $f : G \longrightarrow H$ for pattern graphs with typings $\tau : G \longrightarrow S$ and $\rho : H \longrightarrow S$ is **type-compatible** if it satisfies $\rho \circ f = \tau$. □

Corrolary 1.2 Let S be a layered schema graph (according to definition 1.5 and $\iota : G \longrightarrow S$ an interpretation of a data graph G. Then G is a layered graph with the level function $l_G = l_S \circ \iota$ induced by ι.

2 Rules

A View defines the content of a new cluster of the global data graph (called *view cluster*) as the result of a mapping from one or more other clusters (called *source clusters*). This mapping is defined by a set of rules. In Section 3, we describe how views can be materialized on demand by invoking appropriate rules. When a rule is invoked, it matches some parts of the source clusters and produces new elements in the target cluster. Therefore we have chosen to use graph transformation rules for this purpose.

We base our definition of rules on the well-established algebraic *single pushout approach* to graph transformation as described in [5, 4]. This kind of graph transformation has the advantage that it does apply not only to conventional graphs, but to a wide range of graph data models including our notion of clustered graphs.

Since we do not need single pushout rules in their full generality, we can simplify the original definition. On the other hand, we need to add two new features, a typing morphism which ensures that a rule conforms to the schema, and a set of application constraints which is used to control rule application by posing additional restrictions on the matched labels in the data graph. Both additions restrict only the applicability of rules, but do not change the semantics of rule application. In summary, we use *typed attributed Single Push Out graph transformations with application conditions on attributes*.

In Sec. 3 we will enhance our rule concept further. Hence the following definition is preliminary.

Definition 2.1 (Rule) — *preliminary definition*

A **rule** $p = (L, R, \Gamma, \tau)$ for a schema S consists of:

1. a clustered pattern graph R (see definition 1.9), called the **right hand side** (RHS) graph.
2. a subgraph L of R, called the **left hand side** (LHS) graph.
3. a typing morphism $\tau : R \longrightarrow S$.
4. a boolean term Γ from $T_\Sigma(\mathbb{V})$ interpreted as an **application constraint** for p.

□

Example 2.1 An example of a rule is shown in Figure 9. The rule get_issue matches a journal vertex in the database cluster EJournal DB together with some elements of the Springer ACR cluster and adds the elements shown in boldface to the database cluster.

The application constraint is shown below the ACR cluster. It defines some integrity constraints for the occurring variables. The typing morphism τ is not shown explicitly, but is rather indicated by the graph labels. It maps the RHS of the rule into the schema graph shown in Figure 2. Variable labels are notated as "Variable: Type".

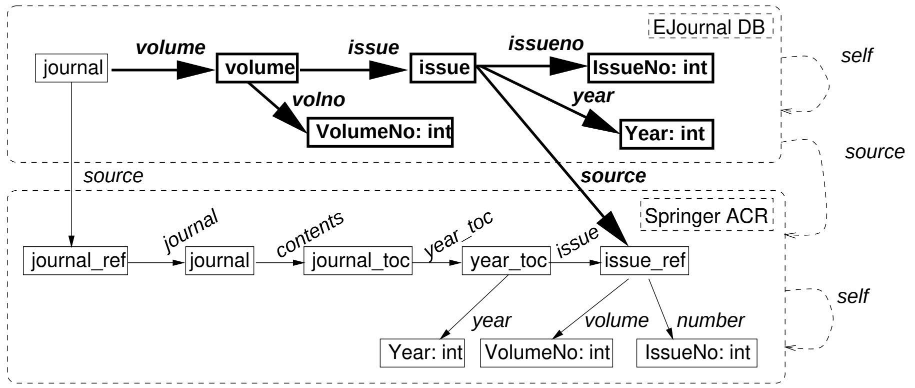

Figure 3: ACR Rule get_issue

□

Definition 2.2 (Variable Substitution) Let $\mathbb{V}$ and $\mathbb{V}'$ be disjoint sets of variables. A **variable substitution** (short: **substitution**) is a function $\sigma : \mathbb{V} \longrightarrow T_\Sigma(\mathbb{V}')$ such that $type(\sigma(X)) = type(X)$ for every $X \in \mathbb{V}$.

Let $t \in T_\Sigma(\mathbb{V}'')$ over some variable set $\mathbb{V}''$. Then the result of replacing all occurrences of a variable $v \in \mathbb{V}$ by $\sigma(v)$ in t is denoted by $t\sigma$. □

Remark 2.1 The disjointness of $\mathbb{V}$ and $\mathbb{V}'$ guarantees several nice properties:

1. a substitution is free of redundancies since the case $\sigma(v) = v$ is excluded.
2. a substitution does not allow cyclic variable definitions such as $\sigma(X) = f(Y), \sigma(Y) = f(X)$.
3. $t\sigma$ is well-defined since every variable occurrence in t has to be replaced at most once.

4. applying a substitution to a term does not change its type since variables are replaced only by terms of the same type
5. applying a substitution to a term is an idempotent operation.

□

Corrolary 2.1 Let $\mathbb{V}$ and $\mathbb{V}'$ be disjoint sets of variables. Let $f : T_\Sigma(\mathbb{V}) \longrightarrow T_\Sigma(\mathbb{V}')$ be a term algebra homomorphism. Let $\mathbb{V}_0 := \{v \in \mathbb{V} \mid f(v) \neq v\}$.

Then $\sigma_f := f|_{\mathbb{V}_0}$ is the **substitution induced by** f.

Conversely, every substitution $\sigma : \mathbb{V}_0 \longrightarrow T_\Sigma(\mathbb{V}')$ defines uniquely an algebra homomorphism $f_\sigma : T_\Sigma(\mathbb{V}) \longrightarrow T_\Sigma(\mathbb{V}')$.

Definition 2.3 (Variable Substitutions for Graphs) Let G be a pattern graph and $\sigma : \mathbb{V} \longrightarrow T_\Sigma(\mathbb{V}')$ a substitution.

Then $G\sigma$ is a copy of G where $a_{G\sigma}(x) = (a_G(x))\sigma$ for all elements x of $G\sigma$. □

Corrolary 2.2 (Preservation of typings under substitutions) Let G be a pattern graph having a typing $\tau : G \longrightarrow S$. Let σ be a substitution.

Then $\tau : G\sigma \longrightarrow S$ is a typing for $G\sigma$ since applying σ to the labels of G does not change their type.

Corrolary 2.3 (Induced Substitution) Let Q and G be $T_\Sigma(\mathbb{V})$-labeled pattern graphs.

Let σ be a substitution defined for a subset of $\mathbb{V}$ and $m : Q \longrightarrow G\sigma$ be a morphism such that σ extends the substitution $\sigma_{m_{attr}}$ induced by m_{attr}.

Then there is a unique minimal substitution σ_0 extending $\sigma_{m_{attr}}$ such that $m : Q \longrightarrow G\sigma_0$. This substitution σ_0 is a restriction of σ. It is called the **substitution induced by** m.

Furthermore, m is also a morphism $m : Q\sigma_0 \longrightarrow G\sigma_0$ since m_{attr} is the identity function for terms to which σ_0 has been applied already.

In the case that G is a data graph, $\sigma_0 = \sigma_{m_{attr}}$ and $m : Q \longrightarrow G$ since the labels of G are all ground and therefore $G\sigma_0 = G$.

Proof: Let σ_1 be a substitution which is a minimal extension of $\sigma_{m_{attr}}$ such that $m : Q \longrightarrow G\sigma_1$. To show that σ_1 is the substitution induced by m we have to show that σ_1 is unique and a restriction of σ.

By the definition of graph morphisms and variable substitution for graphs, $a_G(m(x))\sigma = a_Q(x) = a_G(m(x))\sigma_1$ holds for all elements $x \in Q$. By structural induction, one can show that for every variable X occuring in one of the terms $a_G(m(x))$ the equation $X\sigma = X\sigma_1$ holds. (Note, that $X\sigma = X$ if σ is not defined for X).

Both substitutions are extensions of $\sigma_{m_{attr}}$, hence they coincide also on all variables occurring in labels of Q.

Finally, since σ_1 is minimal, it cannot be defined for any further variables. Therefore σ_1 is a uniquely defined restriction of σ. □

Example 2.2 Let Q consist of a singleton vertex u labeled by $f(X, d, Z)$ and G consist of two vertices v, w labeled by $f(c, Y, Z)$ and $g(Y)$, respectively. Assume that all variables and terms have the same type.

Then the only possible morphism $m : Q \longrightarrow G\sigma$ maps u to v. The induced substitution σ_0 consists of the bindings $X = c$ and $Y = d$. Since σ_0 is minimal, it does not bind Z. Hence $G\sigma_0$ has the labels $f(c, d, Z)$ and $g(d)$. □

Definition 2.4 (Match, Match Set) Let S be a schema graph.

Let G and Q be pattern graphs over a variable set $\mathbb{V}$ with typing morphisms $\iota : G \longrightarrow S$ and $\tau : Q \longrightarrow S$, respectively.

A graph morphism $m : Q \longrightarrow G$ is called a **match** for Q in G.

We denote the set of all matches for Q in G by $Matches(Q, G)$. □

Remark 2.2 A match does not permit ground terms occurring in a label of Q to be mapped to variables occurring in labels of G. This can be remedied by applying an appropriate substitution σ to G first. If G is a data graph it does not carry variables and hence this case is excluded. $\square$

Definition 2.5 (Rule Match) Let S be a schema graph. Let G be a data graph conforming to S with interpretation $\iota : G \longrightarrow S$.

Let $p = (L, R, \Gamma, \tau)$ a rule. A **match** for p is a match $m : L \longrightarrow G$ for L in G.

Accordingly, a **partial match** is a partial match for L.

A **full match** is a match $\bar{m} : R \longrightarrow \bar{G}$ for R where $\bar{G} \sqsupseteq G$ such that the substitution induced by $\bar{m}$ satisfies Γ. $\square$

Remark 2.3 From the definition of full match it follows that application constraints in Γ can be used to compute bindings for variables occurring in R, but not in L. For instance, L could contain variables X and Y. Then the value of a variable Z occurring in R could be determined by a linear constraint $X + Y - Z = 0$. $\square$

A variable substitution (whether it is induced by a partial match or not) can be applied to a rule, resulting in an instantiation of this rule. The instantiation of a rule can be used just like the original rule, but is more specific.

Definition 2.6 (Rule Instantiation) Let $p = (L, R, \Gamma, \tau)$ be a rule and $\sigma : \mathbb{V} \longrightarrow T_\Sigma(\mathbb{V}')$ a variable substitution. Then $p\sigma$, the result of applying σ to p is the rule $(L\sigma, R\sigma, \Gamma\sigma, \tau\sigma)$ obtained by replacing any variable $v \in \mathbb{V}$ which occurs in a label of R or in Γ by $\sigma(v)$. The new typing $\tau\sigma : R\sigma \longrightarrow S$ is identical with τ. $\square$

2.1 Rule application

Before we give a formal construction for the result of applying a rule $p = (L, R, \Gamma, \tau)$ to a match m, we explain its intuitive meaning: the match $m : L \longrightarrow G$ specifies the subgraph $m(L)$ of G to which the rule is to be applied. This subgraph is extended with a new copy of $R - L$ whose labels have been fully instantiated with respect to the substitution σ induced by $m : L \longrightarrow G$ and the application constraint Γ. The resulting graph $\bar{G}$ is constructed in such a way that the match m can be extended to a full match $\bar{m} : R \longrightarrow \bar{G}$. figure 4 illustrates this description.

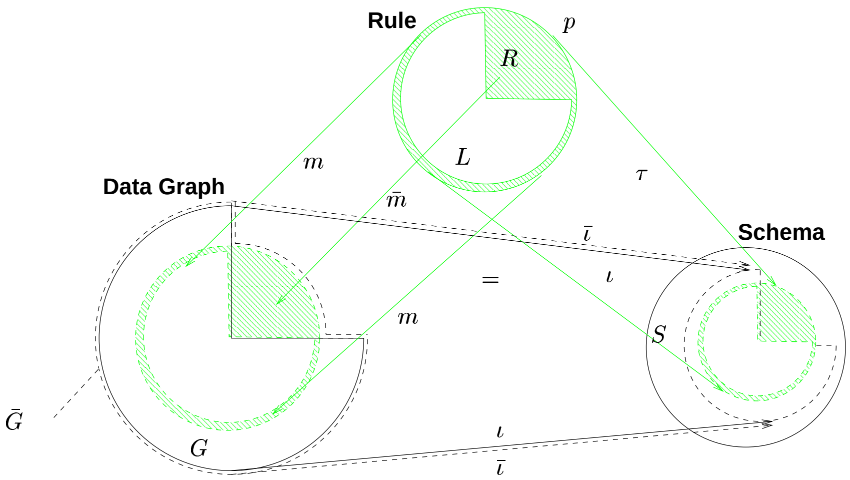

Figure 4: Application of a rule p to a data graph G.

Definition 2.7 (Rule Application) Let $p = (L, R, \Gamma, \tau)$ be a rule and $m : L \longrightarrow G$ a match for its LHS in the data graph G.

Let ρ' be a an extension of the substitution ρ induced by m which binds all variables occurring in R and satisfies Γ. We use the notation $R' := R\rho'$.

Let $r : L \hookrightarrow R'$ be the inclusion morphism for the left hand side L in the right hand side R'. Let $\bar{m}, \bar{r}$ be the pushout of m, r and the data graph $\bar{G}$ be the corresponding pushout object (see figure 2.1).

We call $\bar{G}$ a **result** of the **application** of rule p to match m.

We denote by $Apply(p|m)$ the set of all morphisms $\bar{m} : R' \longrightarrow \bar{G}$ resulting from the application of p to m with respect to different substitutions ρ'. □

Remark 2.4 The morphism $\bar{m}$ is a full match for the rule p. □

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "P.O."] & R' \arrow[d, "\bar{m}"] \\
G \arrow[r, "\bar{r}"] & \bar{G}
\end{tikzcd}$$

Figure 5: The push out defining the result of the application of rule $p = (L, R, \Gamma, \tau)$ (represented by the inclusion morphism $r : L \longrightarrow R'$ where $R' = R\sigma$) to a match m as specified in definition 2.7.

Theorem 2.1 The graph $\bar{G}$ resulting from applying a rule p to a data graph G at match m w.r.t. to a substitution ρ' of the right hand side is defined uniquely up to isomorphism and different attributes of the new elements.

Proof: The rule $p = (L, R, \Gamma, \tau)$ defines a single pushout rule $r : L \longrightarrow R$ which is the inclusion morphism $r : L \hookrightarrow R$ embedding L in R. In [4] it is shown that in the category of attributed graph structures the pushout object $\bar{G}$ of the graph morphisms r and m exists. Since the category of clustered graphs is an instance of this category, this result applies also to clustered graphs. By its definition, a pushout object (i.e., $\bar{G}$) is defined uniquely up to isomorphism. □

We now give a set-theoretic construction for the result of a rule application. Informally speaking, the pushout object of $m : L \longrightarrow G$ and $r : L \longrightarrow R$ in the domain of attributed graph structures is created by taking the disjoint union of G and R and gluing all elements with common preimages in L together. The attributes of old elements (in $\bar{r}(G)$) are kept, and the attributes of a new element $x \in \bar{m}(R)$ is the the attribute of its preimage in R under the substitution ρ' induced by $\bar{m}$. Since ρ' is an extension of the substitution induced by m no conflicts for the attributes of the elements in the intersection of $\bar{r}(G)$ and $\bar{m}(R)$ occur.

Construction 2.1 (Rule Application) Let $p = (L, R, \Gamma, \tau)$ a rule and $m : L \longrightarrow G$ with $m = (m_{vertex}, m_{edge}, m_{cluster}, m_{dep}, m_{attr})$ a match for p in G. Let ρ' a substitution assigning ground terms to all variables occurring in R such that ρ' extends the substitution induced by m and satisfies the application constraint Γ.

We call the $\mathbb{U}$-labeled result graph to be constructed $\bar{G}$. For each carrier set X_G of G (i.e., vertices, edges, clusters, and dependencies), the corresponding carrier set $X_{\bar{G}}$ of $\bar{G}$ is a disjoint union $X_G \uplus X_{R-L}$ of X_G and a new copy $X_{R-L} = \tilde{m}_X(X_R - X_L)$ of $X_R - X_L$ under an arbitrary bijection $\tilde{m}_X$ which satisfies $\tilde{m}_X(X_R - X_L) \cap X_G = \emptyset$. Therefore the respective component $m_X : X_K \longrightarrow X_G$ of the match m can be extended to a function $\bar{m}_X : X_R \longrightarrow X_{\bar{G}}$ by defining $\bar{m}_X := m_X \uplus \tilde{m}_X$.

The attribute component $\bar{m}_{attr}$ is the term algebra homomorphism induced by ρ'.

Together this yields an extension of the match m to a full match $\bar{m} : R \longrightarrow \bar{G}$ with $\bar{m} = (\bar{m}_{vertex}, \bar{m}_{edge}, \bar{m}_{cluster}, \bar{m}_{dep}, \bar{m}_{attr})$.

For each function $f_{\bar{G}} : X_{\bar{G}} \longrightarrow Y_{\bar{G}}$ which is one of the source, target, labeling, and clustering functions $(s_{\bar{G}}, t_{\bar{G}}, a_{\bar{G}}, c_{\bar{G}})$ of $\bar{G}$, $f_{\bar{G}}$ is defined as an extension of the respective function f_G of G which is compatible to the respective function f_R of R. This requires that $f_{\bar{G}}$ satisfies the equation $f_{\bar{G}} \circ \bar{m}_X = \bar{m}_Y \circ f_R$. Since $\bar{m}_X$ is injective on $X_R - X_L$, $f_{\bar{G}}$ is defined uniquely. □

Example 2.3 In order to demonstrate this construction, we show the effect of applying the rule get_issue (depicted in Figure 3) to a small fragment of a data graph. Since for this rule only the database cluster and the Springer ACR cluster are relevant, we show in Figure 6 only fragments of these clusters and disregard the HTML clusters on which the ACR cluster depends. □

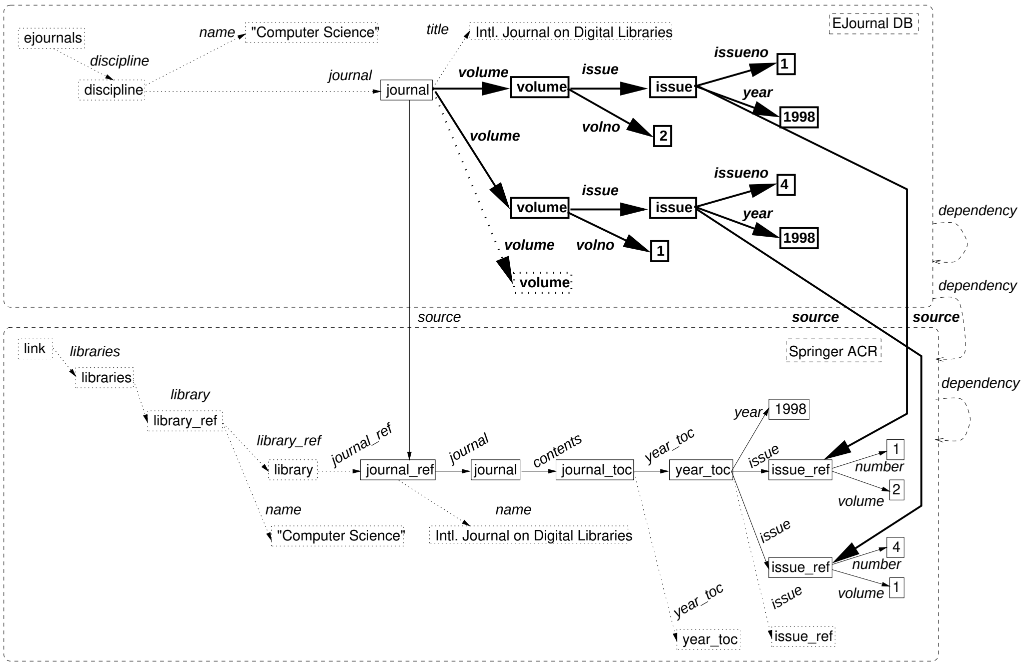

Figure 6: Two applications of the rule get_issue, corresponding to the issues 1/4 and 2/1 of the "International Journal on Digital Libraries". New elements are indicated by bold lines, elements matched by the left hand side by normal full lines, and irrelevant elements which are not matched and further matching elements which have been omitted are denoted with dotted lines.

3 Oracles and Queries

In the last section we have defined the application of single rules. In this section, we define how a set of rules is used for answering *queries*. Roughly speaking, a query is a pattern graph and the solutions of a query are matches from this graph into data graphs which result from extending a given data graph by applying rules to it. Later we will extend the definition of rules slightly to use left hand sides of rules as queries.

Definition 3.1 (Query) Let S be a schema.

A **query** on a data graph G_0 is a tuple (Q, Q_0, Γ, τ) consisting of:

1. a pattern graph Q over a variable set $\mathbb{V}$, the **query graph**
2. a subquery $Q_0 \sqsubseteq Q$, the **anchor**,
3. a constraint Γ being a boolean term from $T_\Sigma(\mathbb{V})$,
4. a typing $\tau : Q \longrightarrow S$ of Q

□

Definition 3.2 (Solution) Let G_0 be a data graph with interpretation $\iota_0 : G_0 \longrightarrow S$. Let q be a query as defined above.

A **solution** for q is a triple (G, ι, m) consisting of a supergraph G of G_0 having the extension $\iota : G \longrightarrow S$ of ι_0 as extension and the match $m : Q \longrightarrow G$ of Q in G such that the substitution induced by m on the variables of Q satisfies the query constraint Γ. □

Definition 3.3 (Oracle) An **oracle** Φ is a functor which takes a query $q = (Q, Q_0, \Gamma, \tau)$ and a data graph G_0 (with interpretation $\iota_0 : G_0 \longrightarrow S$) and returns a set $\Phi(q, G_0)$ of solutions (G, ι, m) for q w.r.t. G_0.

For notational ease we write $(m : Q \longrightarrow G) \in \Phi(q, G_0)$ instead of $(G, \iota, m) \in \Phi(q, G_0)$.

Furthermore we use the abbreviation $\Phi(q|m_0) := \{m \in \Phi(q, G_0) \mid m|_{G_0} = m_0\}$ to express a call of an oracle with a fixed initial match $m_0 : Q_0 \longrightarrow G_0$.

We say an oracle Φ is **competent** for schema clusters $c_1, \ldots, c_n \in C_S$ of a schema S if it answers only such queries where all elements of Q (except of those in Q_0) must be typed by elements from one of the schema clusters c_i, i.e., $\forall x \in Q : x \notin Q_0 \Rightarrow c_S(\tau(x)) \in \{c_1, \ldots, c_n \in C_S\}$. □

The solutions of a query are matches which extend existing matches for the anchor Q_0 of the query in the given data graph. Furthermore each solution must be compatible with the typing τ and satisfy all constraints in Γ. An *oracle* can be seen as a "black box" which computes for a given query and a data graph a solution set satisfying all these requirements. The result of such an oracle for a query q against a data graph G is shown schematically in figure 7.

The exception for the typing of the elements of Q_0 in the definition of an oracle competent for schema clusters $c_1, \ldots, c_n$ is motivated by the fact that Q_0 may have to match already existing inter-cluster edges leading to the clusters for which the oracle is competent.

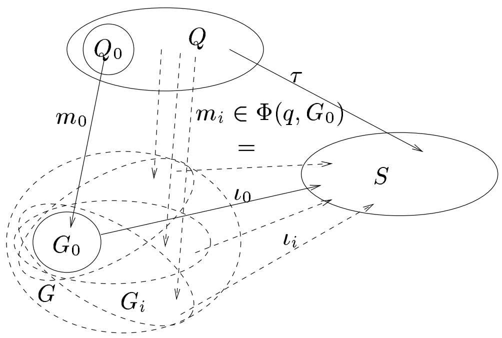

Figure 7: A query $q = (Q, Q_0, \Gamma, \tau)$ against an oracle Φ w.r.t. an initial data graph G_0. The solution set is indicated by dashed lines.

Example 3.1 The HyperView System provides one builtin oracle, the **WWW oracle**. As anchor Q_0 of a query it assumes a graph which matches a part of an existing HTML cluster. Every element of the rest of the query graph Q must be reachable from within Q_0 or from a node labeled by an URL.

Then the WWW oracle will try to find matches for Q_0 in the already materialized HTML clusters and for each of these matches it will try to complete this matches to matches for the whole query graph Q. To do so, it will load HTML pages from the WWW, triggered by the attempt to match edges representing hyperlinks, e.g., the href_target edge in the query depicted in figure 8. The WWW oracle supports sub edges to denote a transitive descendent relation among HTML page elements.

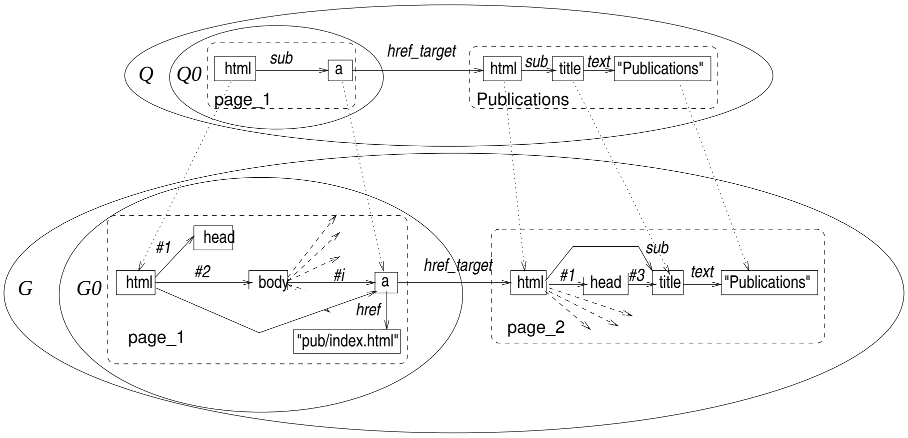

Figure 8: Query against the WWW oracle matching the hyperlink from the author's home page to his publications page. The home page is assumed to be already materialized in a cluster page_1.

To access a HTML page at a known URL directly, one can ask the WWW oracle for a root edge from a vertex labeled by the given URL to a html vertex. This query will then cause the page referenced by the URL to be loaded and its root node will become the target of root edge. □

3.1 Applying a rule to a virtual data graph

If there is an oracle Φ available for queries against a certain cluster of the data graph G_0, we can use this oracle to apply a rule $p = (L, R, \Gamma, \tau)$ to it even though we cannot find matches for p in G_0 itself.

To do so, we have to formulate a query q for Φ which will return matches for L. It follows immediately that L should be contained in the query graph; in fact, we choose $Q = L$. Furthermore, it is clear that the application constraint should be used as a constraint for q and the restriction $\tau|_L$ of the typing τ as typing of Q. The only open question is how to determine the anchor graph Q_0 of q. One solution would be to determine Q_0 by the form of L.

However, a more flexible approach is to add a graph $A \sqsubseteq L$ to the definition of p to indicate the portion of L for which a match in the already materialized data graph is required. Thus a rule gets the following form:

Definition 3.4 (Rule) — *final definition*

A **rule** $p = (A, L, R, \Gamma, \tau)$ for a schema S consists of:

1. a clustered pattern graph R (see definition 1.9), called the **right hand side** (RHS) graph.
2. a subgraph L of R, called the **left hand side** (LHS) graph.
3. a subgraph A of L, called the **anchor** graph.
4. a typing morphism $\tau : R \longrightarrow S$.
5. a boolean term Γ from $T_\Sigma(\mathbb{V})$ interpreted as an **application constraint** for p.

□

Hence, the query associated with this rule becomes $q = (L, A, \Gamma, \tau|_L)$. Using the oracle Φ we obtain a set $\Phi(q, G_0)$ of matches $m : L \longrightarrow G$ each of which is an extension of a match $m_0 : A \longrightarrow G_0$. In figure 9 the rule get_issue already introduced in figure 3 is depicted with anchor graph.

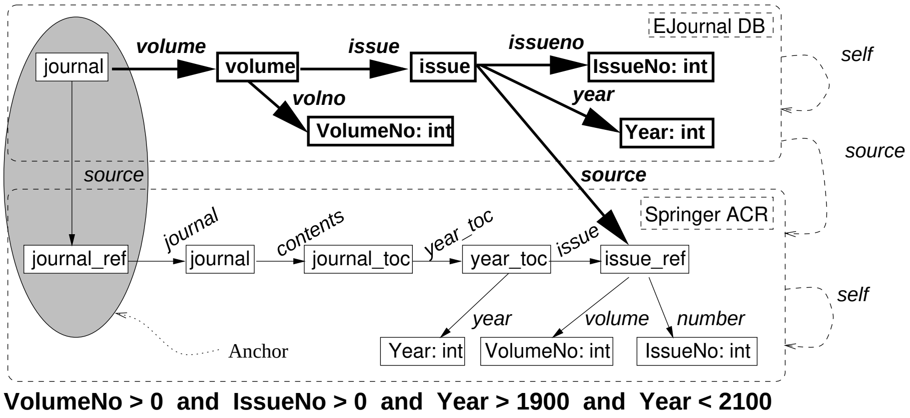

Figure 9: ACR Rule get_issue with anchor graph. Typically the source edge created by the right hand side would provide a match for the anchor graph of another rule to be called after get_issue.

To each $m \in \Phi(q, G_0)$, the rule can be applied in the usual way, producing full matches $\bar{m} : R \longrightarrow \bar{G}$. This application of p against the oracle Φ over the initial data graph G_0 is depicted in figure 10.

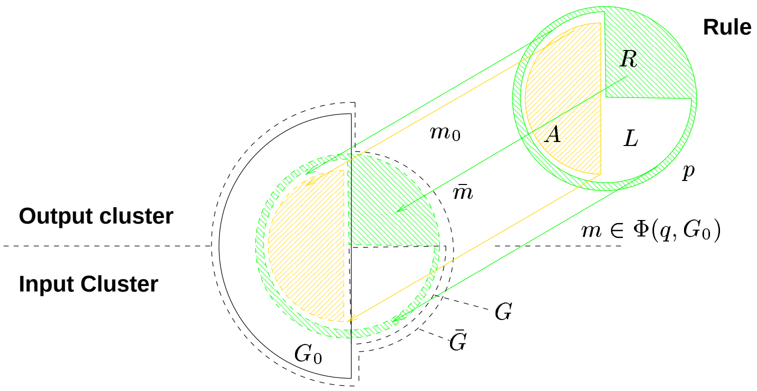

Figure 10: Applying a rule to a *virtual* data graph with oracle Φ. The match for A is $m_0 : A \longrightarrow G_0$, the match for the left hand side $m : L \longrightarrow G$, and the full match for the right hand side is $\bar{m} : R' \longrightarrow \bar{G}$.

Definition 3.5 (Rule Application against Oracle) Let $p = (A, L, R, \Gamma, \tau)$ be a production, Φ an oracle, and $m_0 : A \longrightarrow G_0$ a partial match for p.

Then we define $Apply^{\Phi}(p|m_0) := \{\bar{m} \in Apply(p|m) \mid m \in \Phi(q|m_0)\}$ where $q = (L, A, \Gamma, \tau|_L)$, called the **rule application functor** for oracle Φ. □

The rule application functor $Apply^{\Phi}(.|.)$ uses an oracle Φ to extend a partial match m_0 for a rule p to a total match m and then applies p to this match using the functor $Apply(.|.)$ for rule application without oracle as defined in definition 2.7.

3.2 Hyperviews

A hyperview defines a mapping which computes a cluster of a data graph as a function of several other clusters of this graph. This mapping is specified by a set of rules defined with respect to a subschema describing the input and output clusters of the hyperview.

Definition 3.6 (Hyperview) Let S be a schema. Let $Out, In_1, \ldots, In_n$ be disjoint clusters of S such that there exists in S a dependency from Out to each In_i, $i = 1, \ldots, n$. Let $S_0 \sqsubseteq S$ be the subschema constructed by ommitting all other clusters of S and their dependencies.

Let Π be a set of rules. We call Π a **hyperview** with input clusters $In_1, \ldots, In_n$ and output cluster Out iff each $p = (L, R, \Gamma, \tau) \in \Pi$ satisfies the following conditions:

1. p is typed w.r.t. S_0, i.e., $\tau : R \longrightarrow S_0$
2. each vertex $v \in V_R - V_L$ is typed by a schema vertex belonging to Out, i.e., $c(\tau(v)) = Out$
3. each edge $e \in E_R - E_L$ is typed by a schema edge belonging to a dependency emanating from Out, i.e., $s(c(\tau(e))) = Out$
4. the variable set of p does not overlap with the variable set of any other rule in Π

□

3.3 Using a rule to answer a subquery

Let Π be a hyperview (cf. Def 3.6) and Φ an oracle for data graph clusters described by the input clusters of Π. Let $p \in \Pi$ one of its rules.

Let $q = (Q, Q_0, \Gamma, \tau)$ be a query and $B \sqsubseteq Q$. We can use $p = (A_p, L_p, R_p, \Gamma_p, \tau_p)$ to find a match for B if we can come up with a suitable mapping between B and R. We call such a mapping a *binding morphism*. B can be compared to the call site of a procedure in an imperative program. It specifies which rule to activate, which parameters to supply, and where to use its result.

Definition 3.7 (Binding Morphism) Let $q = (Q, Q_0, \Gamma_q, \tau_q)$ a query and $B \sqsubseteq Q$.

Let $p = (A, L, R, \Gamma, \tau)$ be a rule.

A **binding morphism** $b : B \longrightarrow R$ is a type-compatible graph morphism which does not map B entirely into into L, i.e., $b(B) \not\sqsubseteq L$. We call B the **binding region** of b. □

Remark 3.1 In general, p and q will be the result of applying a variable substitution σ to a rule p_0 and a query q_0. This provides a means of communication by introducing common variables in p and q. In particular, this mechanism can be used to instantiate variables occurring in p with terms occurring as labels of B. □

Applying rule p to a match $m : L \longrightarrow G$ yields a full match $\bar{m} : R \longrightarrow \bar{G}$. This match can be lifted to a match $m_B : B \longrightarrow \bar{G}$ for B by defining $m_B = \bar{m} \circ b$. This is illustrated by figure 11.

3.4 Chaining rules to answer a query

To answer a whole query using a hyperview Π, we introduce the notion of a **query execution plan**. Such a plan consists essentially of a set of binding morphisms b_i for rules $p_i \in \Pi$ which cover (together with the anchor graph of the query) the whole query graph. If we can apply the rules p_i in such a way that the matches induced by the binding morphisms are compatible with each other and with a match for the anchor graph, we yield a match for the whole query graph being the union of all these matches.

Since a rule can be applied only if a match for its anchor graph can be found in the available data graph, care must be taken to activate rules in the right order. The query can be answered

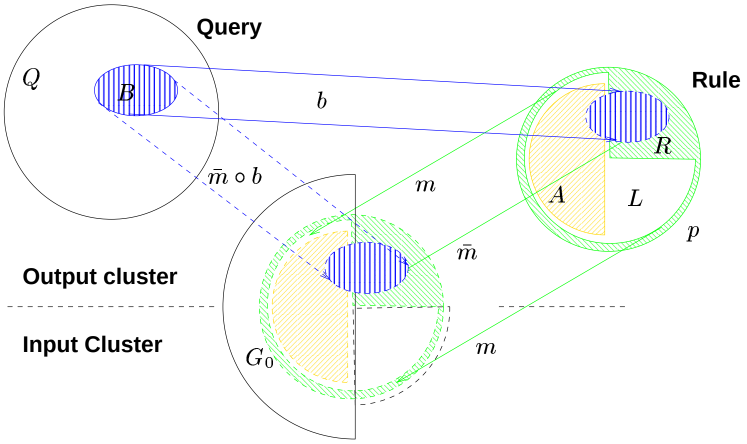

Figure 11: Using a binding morphism to obtain a match for a subquery B.

only if for each rule a subgraph matching its anchor graph already either exists in the initial data graph or is materialized by a preceding rule.

We have chosen a plan concept which ensures statically that rules are executed in the right order. This poses a slight restriction to the form of rule sets over which queries can be answered. The key idea is to require that the anchor graph of a rule is either to be matched against the initial data graph or there exists a so-called *rule dependency* morphism (see definition 3.8 below) which maps it to the right hand side of a rule which is to be executed before. This idea is illustrated schematically by figure 12 and can also be seen in the the example of a query execution plan shown in figure 13.

Definition 3.8 (Rule Dependency) Let Q be a query graph and $b_i : B_i \longrightarrow R_i$ be binding morphisms for rules $p_i = (A_i, L_i, R_i, \Gamma_i, \tau_i)$ for $i = 1, 2$, respectively. Let $F = B_1 \cap {b_2}^{-1}(A_2)$ be nonempty. (F contains the portion of the intersection of B_1 and B_2 which is mapped to the anchor graph A_2 by b_2.)

A **rule dependency** is a type-compatible morphism $d : A_2 \longrightarrow R_1$ mapping A_2 to the right hand side R_1 of p_1, that is compatible with the binding morphisms, i.e., $b_1|_F = d \circ b_2|_F$ holds. □

Definition 3.9 (Query Execution Plan) Let Π be a hyperview for a schema S.

Let $q = (Q, Q_0, \Gamma, \tau)$ be a query.

A **query execution plan (QEP)** for q is a tuple $P = (\sigma, \mathbb{B}, \mathbb{D}, \Gamma)$ consisting of:

1. a variable substitution σ
2. a sequence $\mathbb{B} = (b_1, \ldots, b_n)$ of binding morphisms $b_i : B_i \longrightarrow R_i$ where $B_i \sqsubseteq Q\sigma$, $p_i = (A_i, L_i, R_i, \Gamma_i, \tau_i) = p\sigma$ for some $p \in \Pi$
3. a sequence $\mathbb{D} = (d_1, \ldots, d_n)$ such that for each i either there is a $j < i$ such that d_i is a rule dependency $d_i : A_i \longrightarrow R_j$ compatible with b_i and b_j, or $d_i = \emptyset$ and $b_i(B_0) \sqsubseteq A_i$ for $B_0 := Q_0\sigma$.

Furthermore the query graph under the substitution σ has to be completely covered by binding regions and its anchor graph, i.e., $\bigcup_{i=0}^{n} B_i = Q\sigma$.

A plan $P = (\sigma, \mathbb{B}, \mathbb{D}, \Gamma)$ for q is called a **subplan** of plan $P' = (\sigma', \mathbb{B}', \mathbb{D}', \Gamma')$ for q if σ is a restriction of σ', and $\mathbb{B}$ and $\mathbb{D}$ are (possibly permuted) subsequences of $\mathbb{B}'$ and $\mathbb{D}'$, respectively.

A plan is **minimal** if it has no subplans other than itself. □

In figure 13 a simple QEP is shown. It involves only two productions, the rule p_1 =get_issue shown in figure 9 and the rule $p2$ =get_article which retrieves an article from an issue of a journal. The query selects all articles from issues of the "International Journal of Digital Libraries" having

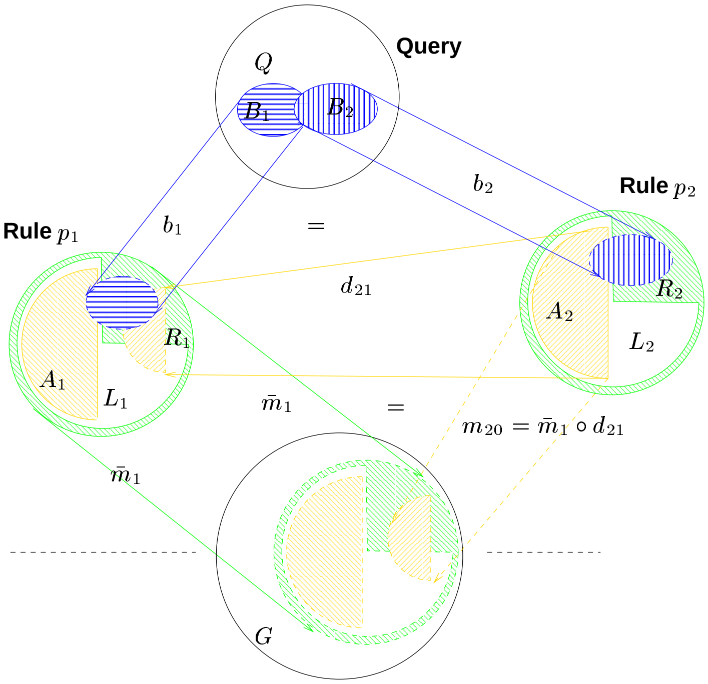

Figure 12: Chaining of rules p_1 and p_2. Rule p_1 has already been applied, yielding a full match $\bar{m}_1$. Morphism d_{21} maps the anchor graph of p_2 to the right hand side of p_1 and thus lifts the full match $\bar{m}_1$ for p_1 to a partial match m_{20} for p_2.

appeared in 1998. It is assumed that the vertex representing this journal is already present in the **EJournalDB** cluster, hence it is put into the anchor graph Q_0 of the query. The rule dependency morphism d_{21} maps the elements of the anchor graph of p_2 on the right hand side (excluding the left hand side) of p_1.

Definition 3.10 (Solution for a QEP, Plan Oracle)

Let $q = (Q, Q_0, \Gamma, \tau)$ be a query and $P = (\sigma, \mathbb{B}, \mathbb{D}, \Gamma)$ be a minimal QEP for q with binding morphisms $\mathbb{B} = (b_1, \ldots, b_n)$ and rule dependencies $\mathbb{D} = (d_1, \ldots, d_n)$.

Let Φ be an oracle.

Let G_0 be an initial data graph and $m_0 : B_0 \longrightarrow G_0 \in \mathit{Matches}(B_0, G_0)$ a match for $B_0 := Q_0\sigma$ in G_0. Let G be a supergraph of G_0.

Let $\bar{m}_i : R_i \longrightarrow G$ $(i = 1, \ldots, n)$ be full matches for the rules p_i such that for each rule dependency $d_i : A_i \longrightarrow R_j$ the match $\bar{m}_i$ is compatible to $\bar{m}_j$, i.e., $\bar{m}_i|_{A_i} = \bar{m}_j \circ d_i$ and for each $d_i = \emptyset$ the match $\bar{m}_i$ is compatible to m_0, i.e., $m_0 = \bar{m}_i \circ b_i|_{B_0}$. Let $m_i := \bar{m}_i \circ b_i$.

The union $m = \bigcup_{i=0}^{n} m_i$ is called a **solution** for plan P iff it forms a match $m : Q\sigma \longrightarrow G$ of $Q\sigma$ in G.

We denote the set of all solutions for P w.r.t. Φ by $\mathit{PlanOracle}^{\Phi}(P|m_0)$. □

Construction 3.1 (Solution for a QEP, Plan Oracle) Using the names of definition 3.10, we define recursively a set M_i of full matches for the first i rules in P. The initial set M_0 is defined as $\{(G_0)\}$.

Let $(G_{i-1}, \bar{m}_1, \ldots, \bar{m}_{i-1}) \in M_{i-1}$. Let $m_{i0} \in \mathit{Matches}(A_i, G_{i-1})$ in case that $d_i = \emptyset$ and $m_{i0} = \bar{m}_j \circ d_i$ if $d_i : A_i \longrightarrow R_j$.

Then for every $\bar{m}_i : R_i \longrightarrow G_i \in \mathit{Apply}(p_i|m_{i0})$ the set M_i contains the tuple $(G_i, \bar{m}_1, \ldots, \bar{m}_i)$.

For each tuple of M_n which consists of compatible matches $\bar{m}_i$, $\bar{m}_j$ fulfilling $m_i|_{B_i \cap B_j} = m_j|_{B_i \cap B_j}$ the corresponding union $m = \bigcup_{i=0}^{n} m_i$ is an element of $\mathit{PlanOracle}^{\Phi}(P|m_0)$. □

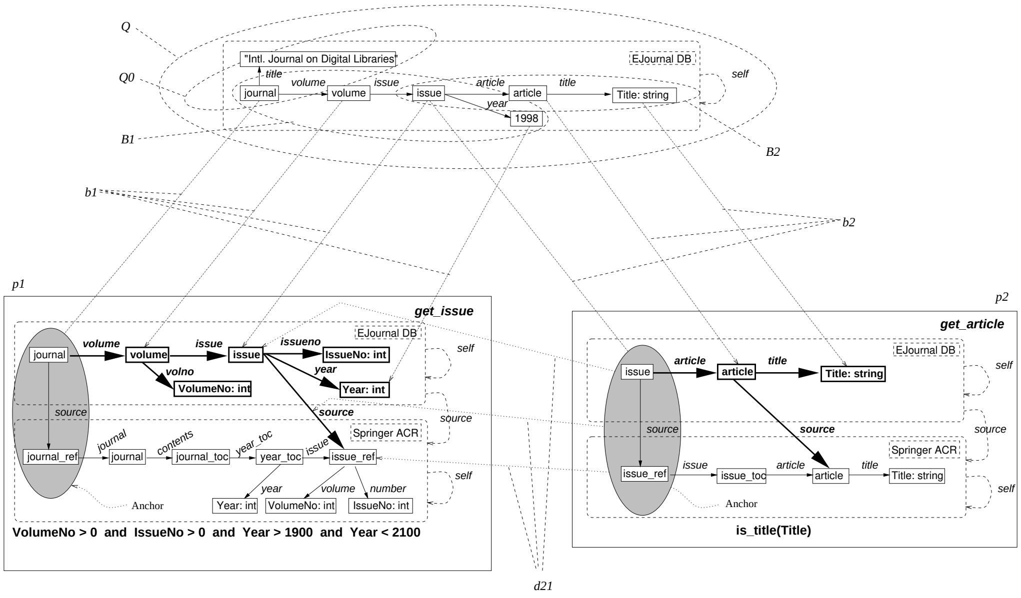

Figure 13: Example of a query execution plan

Remark 3.2 This construction shows that all solutions for a plan P can be found algorithmically by enumerating the matches for B_0, and for each match executing the rules whose binding regions B_i intersect with B_0, and then recursively firing rules having dependencies to already executed rules. By checking for each match for a new rule the compatibility with the matches for the rules executed before, branches not leading to solutions can be pruned out early. The rule dependencies ensure that matches for the anchor graphs are provided, thus avoiding the problem that the data graph is not sufficiently materialized to fire a rule. □

Remark 3.3 The term QEP is used here slightly differently than in the field of databases since a QEP defines an expression does not yield the complete result of a query, but rather a subset of it.

In order to get the complete result of a query, all minimal plans for this query have to be evaluated and the union of the returned partial results has to be built. □

Construction 3.2 (HyperView Oracle) Let $q = (Q, Q_0, \Gamma, \tau)$ be a query.

Let $Plans^{\Pi}$ be a functor which assigns to q the set $Plans^{\Pi}(q)$ of all minimal query execution plans for q with respect to hyperview Π.

Let Φ be an oracle for the input clusters of Π.

Then we define the query match functor $Oracle^{\Phi,\Pi}$ which returns all matches for q when starting from initial data graph G_0 with respect to the match functor Φ and rule set Π:

$$Oracle^{\Phi,\Pi}(q, G_0) := \{m \in PlanOracle^{\Phi}(P|m_0) \mid P \in Plans^{\Pi}(q), m_0 \in Matches(Q_0, G_0)\}$$ □

Remark 3.4 The construction of $Oracle^{\Phi,\Pi}$ for a hyperview Π with output schema cluster c yields an oracle competent for this cluster.

Different oracles competent for schema clusters $c_1, \ldots, c_n$ can be combined to one oracle competent for all these clusters, provided that there are no dependencies between these clusters in the schema.

A query against $c_1, \ldots, c_n$ can be decomposed into subqueries q_i against single clusters c_i. The anchor graphs Q_{0i} may intersect because by definition 3.3 need not conform to c_i. Unions of solutions m_i returned for the different q_i are solutions for q if they are compatible with each other and satisfy the constraint Γ of q.

This enables us to compose hyperviews in a way that allows information from different sources to be retrieved, restructured and combined on a higher level of abstraction. In a typical HyperView System several succeeding levels of abstraction exist, from the HTML layer up to the result layer presented to the user. □

4 Reuse of existing subgraphs

The problem of identifying entities uniquely by their properties applies not only to classical databases, but to data graphs as well. Hence, the concept of key attributes must be adapted appropriately. In particular, when applying rules, it must be avoided to create duplicates.

First we present a pragmatic solution based on *Reuse Specifications*, which has been implemented in the current HyperView prototype.

Definition 4.1 (Reuse Specification) Let $p = (A, L, R, \Gamma, \tau)$ a rule. A **reuse specification** for rule p is a list $K_1, \ldots, K_n \sqsubseteq R$ of subgraphs (called **reuse graphs**) of the RHS graph R. □

When applying rule p to a match m, we first check whether m can be extended to K_1. Only if this is not possible, new copies of the elements in K_1 are added to the data graph G. This process is repeated for the remaining graphs $K_2, \ldots, K_n$ (in that order). After that, new copies of the remaining elements in $R - L$ are added to G.

This approach can be formalized by introducing negative structural application conditions. Then, each rule with a reuse graph can be splitted into a pair of rules, one having the reuse graph as a negative application condition and the other including the reuse graph in the left hand side.

Reuse specifications have the advantage that they can be implemented efficiently. However, they depend on the assumption that all K_i are sufficiently selective to match at most one subgraph of the data graph. Otherwise the matching subgraphs have to be glued together or one of them has to be chosen nondeterministically. To achieve determinism, the rule set has to be carefully designed. The goal must be to specify key properties by schema annotations and to generate reuse specifications for all rules. Furthermore, the rule application mechanism must be enhanced in order to handle the case that a reuse graph is matched by several subgraphs of the data graph.

5 Open Problems

Further problems which are not essential, but for which solutions would have useful applications:

1. Can we support regular path expressions for graph matches? Currently we do not give a semantics for recursive rules. Hence we have to view transitive closure rules as built-in.

   One possible solution would be to assume that all graphs are implicitly equipped with all edges corresponding to regular path expressions. For instance, for every label a, there would be edges labeled by $a*$ representing the transitive closure of edges labeled by a.

2. How can keys be specified as schema annotations instead of by redundant and possibly inconsistent reuse specifications?

3. How can multi-valued primary keys for objects enforced? For instance, reports might be identified by their title and the set of authors.

4. How do we cope with the fact that many multiple-valued attributes are ordered? Examples are first names of a person, author lists of an article.

5. Are atomic data types in schema graphs sufficient, or are probabilistic or other less strict concepts like fuzzy sets more appropriate? Consider for instance the label sets describing first and last names: either they are too restrictive or too admissive, but there is no way to accumulate belief values from testing several labels.

6. How can inheritance of schema elements and rules be supported? A pragmatic solution taken in this work is to copy rules for all possible target types. Since this is needed only for a limited number of vertex types in HTML graphs, this approach is feasible. However, a a more formal concept would be interesting.

6 Conclusion

In this report the formalization of the HyperView concept for graph transformation based views has been presented. The introduced clustered data model CGDM uses term-attributed graphs to represent data. It supports the modularization of large graphs into loosely connected clusters. The schema concept of CGDM defines conformance by a graph morphism from an instance graph to a schema graph. Our notion of graph transformation uses typed attributed Single Push Our rules with application conditions on attributes. The main contribution of this work is a novel demand-driven rule activation mechanism by which the incremental materialization of HyperViews is achieved. This activation mechanism is based on the notion of *Oracles* against which *Queries* in form of graph patterns can be posed. In particular, the WWW can be modeled by such an oracle. HyperViews consist of rules which are evaluated against a number of existing oracles, thus combining them to a more powerful oracle on a higher level of abstraction. This ensures the composability of HyperViews which is essential for the layered architecture of the HyperView System.

The formal framework presented here forms the theoretic basis on top of which the HyperView System is implemented. The HyperView System is presented in [2]. Up-to-date information on the HyperView and the forthcoming prototype can be found on the Web Site <http://www.inf.fu-berlin.de/~faulstic/HyperView>.

References

[1] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350, Delphi, Greece, 8–10 January 1997. Springer.

[2] Lukas C. Faulstich and Myra Spiliopoulou. Building HyperNavigation wrappers for publisher web-sites. In *Second European Conference on Digital Libraries*, number 1513 in LNCS, pages 115–134, 1998. ©Springer-Verlag.

[3] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Mathematical Structures in Computer Science*, 6(6):613–648, 1996.

[4] Reiko Heckel, Jürgen Müller, Gabriele Taentzer, and Annika Wagner. Attributed graph transformations with controlled application of rules. In *Proc. Colloquium on Graph Transformation and its Application in Computer Science*, 1995.

[5] Michael Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109:181–224, 1993.

[6] Wolfgang Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer, Berlin, 1992.