Chapter 3

Formal Framework

3.1 Clustered Graph Data Model (CGDM)

3.1.1 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* of closely related vertices and edges.

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 user interface 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 belongs to a *dependency* connecting the cluster of its source vertex with the cluster of its target vertex. 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.

Therefore, 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.

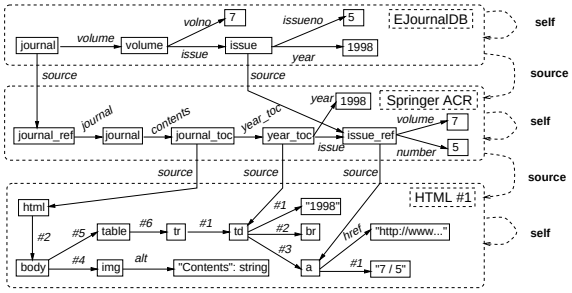

Figure 3.1: Example of a clustered graph: the cluster "HTML #1" models a (fragment of) a HTML page. Elements in the "Springer ACR" cluster are connected to the corresponding elements of the HTML page by source edges. A fragment of the "EJournal DB" database cluster is shown in the topmost box.

In Figure 3.1, an example of a clustered graph is depicted. It shows the same piece of information occurring on a web page of Springer Verlag at three levels of abstraction: as HTML graph, as ACR graph, and finally as database graph. Each of these representations are stored in clusters of their own. Thus the clustered data model allows to maintain separate representations of the same entities at different levels of abstraction.

Corresponding elements may be connected by source edges. However, not all elements at one level have corresponding elements at the other levels. For instance, in the two lower levels,

volumes are represented as attributes of issues, whereas at the database level volumes are entities of their own.

3.1.2 Basic definitions

As data model, we use a clustered graph data model called CGDM. We build our definition of clustered graphs on the standard definition of attributed (i.e., "labeled") directed graphs (cf. [Heckel *et al.*, 1995]). We call these graphs *plain* graphs to distinguish them from clustered graphs.

Plain graphs consist of vertices and edges. The source and target functions assign start and destination vertices to each edge. The attribute function assigns a label to each vertex and each edge. Labels are elements of an universal algebra A which means that almost arbitrary data structures can be used as labels.

Graph morphisms are functions which map the elements of one graph to the elements of another graph such that vertices are mapped to vertices, edges to edges, and the source (target) of an edge to the source (target) of its image. We first introduce *plain* graph morphisms first which are mappings between plain graphs.

Definition 3.1.1 (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 $a : V \uplus E \longrightarrow A$. [1]

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 $f_{vertex} : V_G \longrightarrow V_H$ and $f_{edge} : E_G \longrightarrow E_H$ are total functions commuting with the source and target functions (i.e., $f_{vertex} \circ s_G = s_H \circ f_{edge}$ and $f_{vertex} \circ t_G = t_H \circ f_{edge}$).

Two plain graph morphisms $f : G \longrightarrow H$ and $f' : G \longrightarrow H$ are **equal** iff $f_{vertex} = {f'}_{vertex}$ and $f_{edge} = {f'}_{edge}$.

The **result** of applying f to G is defined by $f(G) := (V_{f(G)}, E_{f(G)}, A_{f(G)}, s_{f(G)}, t_{f(G)}, a_{f(G)})$ where $V_{f(G)} := f_{vertex}(V_G)$, $E_{f(G)} := f_{edge}(E_G)$, $A_{f(G)} := A_H$, $s_{f(G)} := s_H|_{E_{f(G)}}$, $t_{f(G)} := t_H|_{E_{f(G)}}$, $a_{f(G)} := a_H|_{V_{f(G)} \uplus E_{f(G)}}$.

The **identity morphism** ι_G on G is defined by $\iota_G = (\iota_{V_G}, \iota_{E_G})$ where $\iota_{V_G} : V_G \longrightarrow V_G$ and $\iota_{E_G} : E_G \longrightarrow E_G$ are the identity functions on the respective carrier sets V_G, E_G of G.

Let $K = (V_K, E_K, A_K, s_K, t_K, a_K)$ be another plain graph and $g : G \longrightarrow H$ and $h : H \longrightarrow K$ be plain graph morphisms. The **composition** $h \circ g$ of g with h is defined component-wise by $(h \circ g)_{vertex} = h_{vertex} \circ g_{vertex}$ and $(h \circ g)_{edge} = h_{edge} \circ g_{edge}$. □

Remark 3.1.1 A plain graph morphism $f : G \longrightarrow H$ is a purely structural mapping which does not pose any constraints on the relation of the attribute functions a_G and a_H. □

Remark 3.1.2 It can be seen easily that: (i) the result $f(G)$ of applying $f : G \longrightarrow H$ to G is again a plain graph (ii) the result $\iota_G(G)$ of applying the identity morphism to G is G itself (iii) the $\circ$ operator defines indeed the sequential composition of morphisms, i.e., $(h \circ g)(G) = h(g((G))$. □

Theorem 3.1.1 (Plain Graphs form a Category) The class of plain graphs together with the class of plain graph morphisms forms a category $\mathbb{PG}$.

[1] $\uplus$ denotes the disjoint union of sets, cf. the table of mathematical symbols on page 135.

Proof: It has to be shown that the composition is associative and its neutral elements are identity morphisms.

Associativity: let $f : F \longrightarrow G, g : G \longrightarrow H, h : H \longrightarrow K$ be plain graph morphisms. Then $(h \circ g) \circ f = ((h_{vertex} \circ g_{vertex}) \circ f_{vertex} \ , \ (h_{edge} \circ g_{edge}) \circ f_{edge}) = (h_{vertex} \circ (g_{vertex} \circ f_{vertex}) \ , \ h_{edge} \circ (g_{edge} \circ f_{edge})) = h \circ (g \circ f)$.

Neutral elements: let $f : F \longrightarrow G$ be a plain graph morphism. Then $f \circ \iota_F = (f_{vertex} \circ \iota_{V_F}, f_{edge} \circ \iota_{E_F}) = (f_{vertex}, f_{edge}) = f$ and $f = (f_{vertex}, f_{edge}) = (\iota_{V_G} \circ f_{vertex}, \iota_{E_G} \circ f_{edge}) = \iota_G \circ f$.

□

As said before, 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 "*structure graph*" inside the clustered graph which summarizes the "*base graph*" formed by vertices and edges. Both graphs are plain graphs. We use a (plain) graph morphism mapping the base graph to the structure graph to assign clusters and dependencies to the vertices and edges of the clustered graph.

Definition 3.1.2 (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 3.1.3 Let $G = (G_{base}, G_{struct}, c)$ be 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**. □

The definition of graph morphisms for clustered graphs builds on the definition of graph morphisms for plain graphs. Essentially we need two plain graph morphisms one of which as mapping between the base graphs and the other as mapping between the structure graphs. It is natural to require that these two morphisms have to commute with the clustering morphisms. Finally, we introduce as a third component of a clustered graph morphism a mapping between the attribute algebras of the two involved clustered graphs.

Definition 3.1.3 (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 $dom(f)$.

Two graph morphisms are **equal** iff their respective components are pairwise equal, i.e., $f = f' \ \Leftrightarrow \ f_{base} = f'_{base} \wedge f_{struct} = f'_{struct} \wedge f_{attr} = f'_{attr}$.

The **result** $f(G)$ of applying f to G is defined by $f(G) := (H'_{base}, H'_{struct}, c_{H'})$ where $H'_{base} = f_{base}(G_{base})$, $H'_{struct} = f_{struct}(G_{struct})$, $c_{H'} = c_H|_{H'_{base}}$.

The **identity morphism** ι_G on G is defined by $\iota_G = (\iota_{G_{base}}, \iota_{G_{struct}}, \iota_{A_G})$ where $\iota_{G_{base}}$ is the identity morphism on the plain graph G_{base}, $\iota_{G_{struct}}$ the identity morphism on the plain graph G_{struct}, and ι_{A_G} the identity homomorphism on the attribute algebra A_G of G.

Let $K = (K_{base}, K_{struct}, c_K)$ another (clustered) graph and $g : G \longrightarrow H$ and $h : H \longrightarrow K$ graph morphisms. The **composition** $h \circ g$ of g with h is defined component-wise by $(h \circ g)_{base} = h_{base} \circ g_{base}$, $(h \circ g)_{struct} = h_{struct} \circ g_{struct}$, and $(h \circ g)_{attr} = h_{attr} \circ g_{attr}$. □

Theorem 3.1.2 (Clustered Graphs form a Category) The class of clustered graphs together with the graph morphisms between clustered graphs forms a category $\mathbb{CG}$.

Proof: It has to be shown that the composition is associative and its neutral elements are identity morphisms. This proof is analogous to the proof of Theorem 3.1.1.

Associativity: let $f : F \longrightarrow G$, $g : G \longrightarrow H$, $h : H \longrightarrow K$ be graph morphisms. Then $(h \circ g) \circ f = ((h_{base} \circ g_{base}) \circ f_{base}\ ,\ (h_{struct} \circ g_{struct}) \circ f_{struct}\ ,\ (h_{attr} \circ g_{attr}) \circ f_{attr}) = (h_{base} \circ (g_{base} \circ f_{base})\ ,\ h_{struct} \circ (g_{struct} \circ f_{struct})\ ,\ h_{attr} \circ (g_{attr} \circ f_{attr})) = h \circ (g \circ f)$.

Neutral elements: let $f : F \longrightarrow G$ be a graph morphism. Then $f \circ \iota_F = (f_{base} \circ \iota_{F_{base}}, f_{struct} \circ \iota_{F_{struct}}, f_{attr} \circ \iota_{A_F}) = (f_{base}, f_{struct}, f_{attr}) = f$ and $f = (f_{base}, f_{struct}, f_{attr}) = (\iota_{G_{base}} \circ f_{struct}, \iota_{G_{struct}} \circ f_{struct}, \iota_{A_G} \circ f_{attr}) = \iota_G \circ f$. □

Remark 3.1.4 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 carrier sets V_G, E_G, C_G, D_G, A_G of G to the corresponding carrier 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 $\mathbb{CG}$ of clustered graphs forms a specialization of the category of **attributed graph structures** with total morphisms as defined in [Löwe, 1993, Heckel *et al.*, 1995]. □

From this definition it follows 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 graph morphism may map a path in a graph to a circular edge, or several edges to a single 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 $m(x)$.

Since clustered graphs are a special case of *attributed graph structures* which in turn are many-sorted algebras [Wechler, 1992], the notion of *subgraph* is inherited directly from the notion of *subalgebra*:

Definition 3.1.4 (Subgraph) Let $G = (V, E, C, D, A, s, t, a, c)$ be an A-labeled graph. A 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, the attribute algebras are the same ($A' = A$), and the operations s', t', a', c' are restrictions of the respective operations s, t, a, c.

Consequently, we define $G \sqsupseteq G' :\Longleftrightarrow G' \sqsubseteq G$ to be the supergraph relationship.

A graph morphism f that is the identity morphism from a subgraph $G' \sqsubseteq G$ into the supergraph G is denoted by $f : G' \hookrightarrow G$. □

Since clustered graphs are algebras, the definitions for the union and intersection of graphs are inherited.

Definition 3.1.5 (Union, Intersection) We use $G \cup H$ to denote the smallest graph containing both G and H as subgraphs and $G \cap H$ to denote the largest subgraph of both G and H.

We denote by $G \uplus H$ the union $G \cup H$ iff the intersection of G and H is the empty graph. □

Definition 3.1.6 (Restriction) Let $f : G \longrightarrow H$ a graph morphism and K be a graph. Then we denote by $f|_K$ the graph morphism $f' : K \cap G$ that is identical with f on $K \cap G$. □

We use the same syntax $f|_X$ for the restriction operation on other functions as well.

Definition 3.1.7 (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 called **compatible** (denoted $f \nabla g$) if they coincide on the intersection of their domains, i.e., $f|_D = g|_D$ where $D = dom(f) \cap dom(g)$. □

Definition 3.1.8 (Reachability) Let $G = (V, E, A, s, t, a)$ be a plain graph. Then the **reachability relation** $\rightsquigarrow$ is defined by $(s^{-1} \cup t)^*$ [2].

Let $G = (G_{base}, G_{struct}, c)$ a 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}. □

Definition 3.1.8 implies that every edge is reachable from its source and the target of an edge is reachable from the edge.

Corollary 3.1.1 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 $(s^{-1} \cup t)^*$. □

3.1.3 Schemata and instances

Definition 3.1.9 (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.

□

Definition 3.1.10 (Pattern Graph, Data Graph, Schema Graph) A **pattern graph** (with variable set $\mathbb{V}$) is a $T_\Sigma(\mathbb{V})$-labeled 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 graph.

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

Remark 3.1.5 A data graph is a pattern graph carrying ground labels only. □

Example 3.1.1 (Schema for Electronic Journals) Figure 3.2 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 [3], together with the database schema cluster describing electronic journals in a publisher-independent way.

[2] The functions s and t are treated as relations here.

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

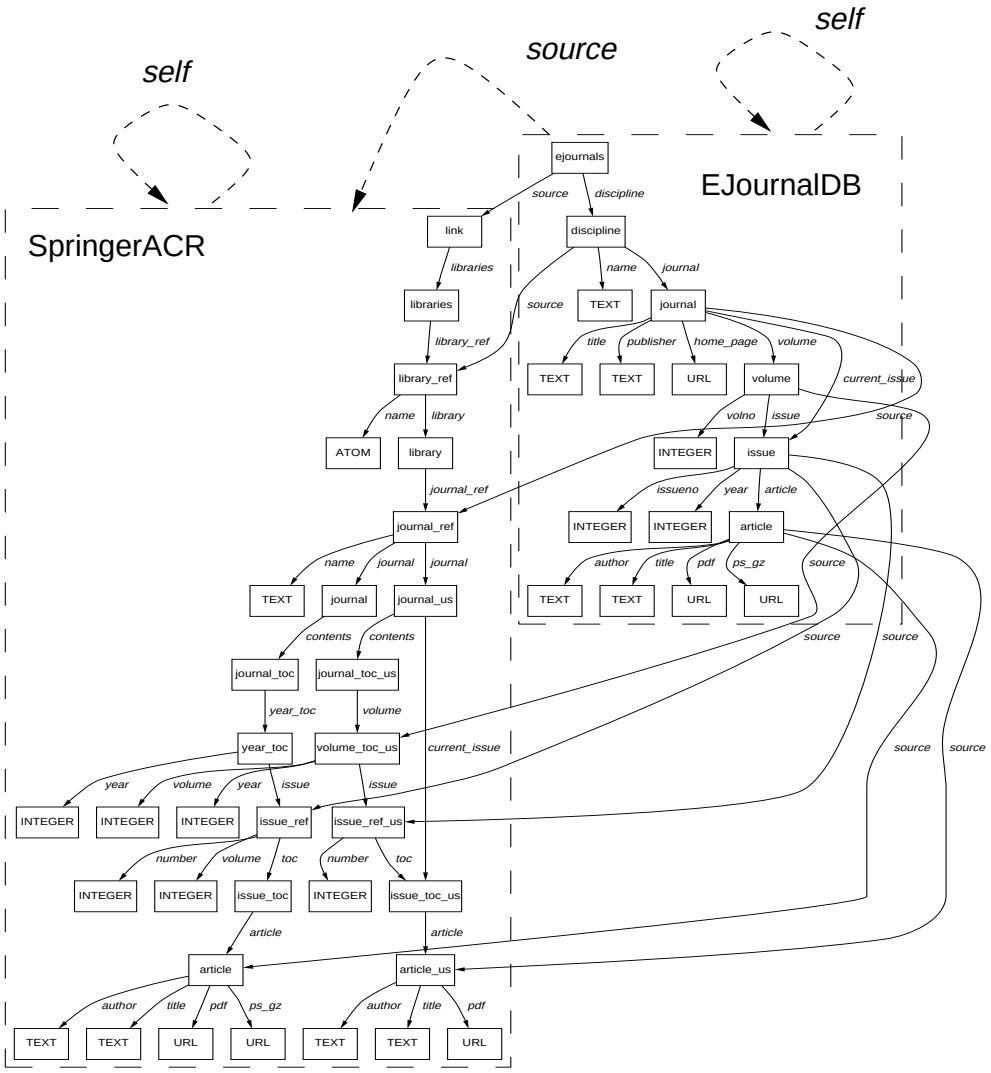

Figure 3.2: Clustered schema for Database and Springer ACR. Uppercase labels denote atomic data types (e.g. INTEGER), lowercase labels denote singleton types consisting of a single element (e.g. journal).

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.

Definition 3.1.11 (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 $a_S(\tau(x))$ 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 $\rho : G \longrightarrow S$ 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 [Heckel *et al.*, 1996] to attributed graph structures. It has the following implications:

- 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 [Buneman *et al.*, 1997] a schema concept is presented which is based on schema graphs labeled with unary predicates. Conformance depends on the existence 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 3.1.12 (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 $\rho \circ f = \tau$. □

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

Definition 3.1.13 (Layered Graph) Let $G = (V, E, C, D, A, s, t, a, c)$ be a 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., $\forall d \in D : l(s(d)) \in \{l(t(d)), l(t(d)) + 1\}$. □

In particular, schemata are layered graphs. The following corollary states that an interpretation of a data graph with respect to a layered schema induces a layered structure on the data graph as well:

Corollary 3.1.2 Let S be a layered schema graph and $\rho : G \longrightarrow S$ an interpretation of a data graph G. Then G is a layered graph with the level function $l_G = l_S \circ \tau$ induced by ρ.

3.2 Rules

A HyperView 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 3.3, we describe how HyperViews can be materialized on demand by invoking appropriate rules. When a rule is fired, 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 [Löwe, 1993] and [Heckel *et al.*, 1995]. In this approach, a rule is modeled by a single partial graph morphism that maps the left hand side of a rule to its right hand side. The application of a rule to a data graph is implemented by a single category-theoretic operation, a so-called *pushout*. Informally speaking, a subgraph matching the left hand

side is cut out and replaced by a new subgraph matching the right hand side of the rule. Besides its simplicity the SPO approach 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. Moreover, the SPO approach can be easily adapted to match our need for non-deleting rules that extend existing data 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 Pushout graph transformation with application conditions on attributes* (cf. citegKoc99), applied to clustered graphs.

In Section 3.3 we will enhance our rule concept further. Hence the following definition is preliminary.

Definition 3.2.1 (Rule) — *preliminary definition*[4]

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

1. a pattern graph R (see Definition 3.1.10), 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.

□

This definition is illustrated in Figure 3.3. Rules are intended to be applied to one or more input clusters and an output cluster and to add new graph elements to the output cluster. Although this is not specified in Definition 3.2.1, but will be formalized in Definition 3.3.4, this intention is indicated by dividing the diagram into a lower and an upper half.

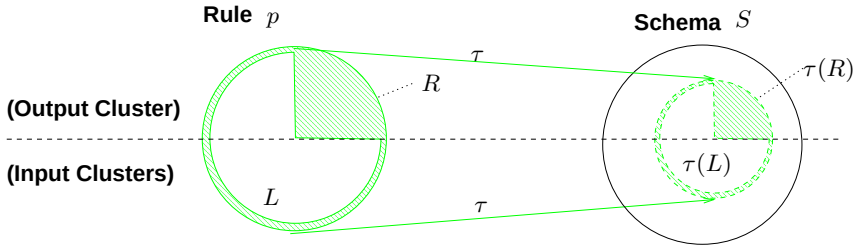

Figure 3.3: Schematic diagram of a rule p (depicted by the left circle) and its typing τ in schema S (right). The LHS graph L (white) is contained in the RHS graph R (green hatching).

Example 3.2.1 An example of a rule is shown in Figure 3.4. The rule get_issue introduced on page 19 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. We use boldface to distinguish new graph elements from those in the left hand side of the rule. For comparison with Figure 3.3, the LHS and RHS are additionally indicated using the same style as there.

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 3.2. If the concrete label of a vertex is of interest, then a variable name can be introduced using

[4]Full definition on page 40

the notation “Variable: Type”. This is important for restricting labels by application constraints and for assigning labels to new vertices in the right hand side.

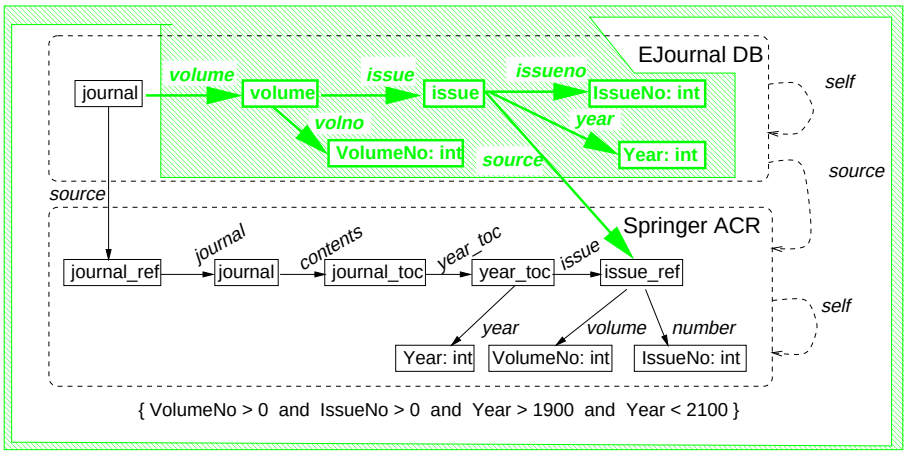

Figure 3.4: ACR Rule get_issue

□

Definition 3.2.2 (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 $t\sigma$. □

Corollary 3.2.1 Substitutions enjoy the following 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 settings 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 by definition variables are substituted only with terms of the same type.
5. Applying a substitution to a term is an idempotent operation.

Proof: These properties follow immediately from the *disjointness* of $\mathbb{V}$ and $\mathbb{V}'$. For instance, cyclic variable settings of the form $\sigma(X) = f(Y), \sigma(Y) = f(X)$ are not possible since $f(X) \in T_\Sigma(\{X\})$ which leads to the contradiction $X \in \mathbb{V} \cap \mathbb{V}' = \emptyset$. □

Definition 3.2.3 (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 $G\sigma$. □

Corollary 3.2.2 (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 $G\sigma$.

Proof: The structure of G is not affected by the application of σ. As pointed out in Corollary 3.2.1, applying σ to the labels of G does not change their type, hence the conditions $\tau_{attr} = type$ and $\forall x \in G \ : \ a_S(\tau(x)) = type(a_G(x))$ from Definition 3.1.11 are satisfied and therefore $\tau : G\sigma \longrightarrow S$ is a typing for $G\sigma$ as well. □

Definition 3.2.4 (Induced substitution) 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 called the **substitution induced by** f.

Let Q and G be $T_\Sigma(\mathbb{V})$-labeled pattern graphs and $m : Q \longrightarrow G$ a graph morphism. Then the substitution σ induced by m_{attr} is called the **substitution induced by** m. □

Example 3.2.2 Let Q consist of a singleton vertex u labeled by $f(X, Y)$ and G consist of two vertices v, w labeled by $f(c, Z)$ and $g(Z)$, respectively. Assume that all variables and terms have the same type.

Then the only possible morphism $m : Q \longrightarrow G$ maps u to v. The induced substitution σ consists of the bindings $X = c$ and $Y = Z$. □

Definition 3.2.5 (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 $\rho : G \longrightarrow S$ and $\tau : Q \longrightarrow S$, respectively.

A type-compatible (cf. Definition 3.1.12) 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 3.2.1 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 problem cannot arise. □

Definition 3.2.6 (Rule Match) Let S be a schema graph. Let G be a data graph conforming to S with interpretation $\rho : G \longrightarrow S$.

Let $p = (L, R, \Gamma, \tau)$ a rule. A **match** for p is a match $m : L \longrightarrow G$ for L in G such that there exists a solution of Γ for the substitution induced by m.

A **partial match** is a match for a subgraph L_0 of L for which a solution of Γ exists.

A **full match** is a match $\bar{m} : R \longrightarrow G$ for R such that the substitution induced by $\bar{m}$ is a solution of Γ. □

Remark 3.2.2 Note, that a full match for a rule is a by-product of applying this rule. Rule application will be defined in Section 3.2.1. 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, let L contain variables X and Y and $\Gamma \equiv X + Y = Z$. Then the value of a variable Z occurring in R outside of L is determined by the linear constraint Γ. □

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 3.2.7 (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. □

3.2.1 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 3.5 illustrates this description.

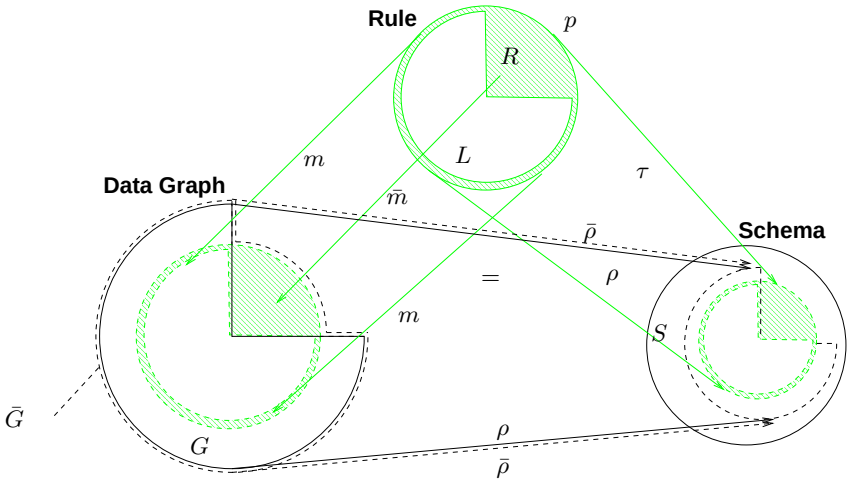

Figure 3.5: Application of a rule p to a data graph G.

Definition 3.2.8 (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 $\bar{\sigma}$ be an extension of the substitution σ induced by m which binds all variables occurring in R and satisfies Γ. We use the notation $\bar{R} := R\bar{\sigma}$ and $\bar{L} := L\bar{\sigma}$.

Let $r : \bar{L} \hookrightarrow \bar{R}$ be the inclusion morphism from the instantiated left hand side $\bar{L}$ into the right hand side $\bar{R}$. Let $\bar{m}, \bar{r}$ be the pushout of m, r and the data graph $\bar{G}$ be the corresponding pushout object (see Figure 3.2.1).

We call $\bar{G}$ the **result** of the **application** of rule p to match m w.r.t. substitution $\bar{\sigma}$.

We denote by $Apply(p|m)$ the set of all full matches $\bar{m} : \bar{R} \longrightarrow \bar{G}$ (with extended data graphs $\bar{G}$) resulting from the application of p to m with respect to different substitutions $\bar{\sigma}$. □

Remark 3.2.3 The match $m : L \longrightarrow G$ is a match for $\bar{L}$ as well since the application of $\bar{\sigma}$ to L has the same effect as the substitution σ induced by m and the application of substitutions is an idempotent operation (cf. Corollary 3.2.1). The morphism $\bar{m}$ is a full match for the rule p. □

Theorem 3.2.1 The graph $\bar{G}$ resulting from applying a rule p to a data graph G at match m (w.r.t. to a substitution $\bar{\sigma}$ that binds all variables in R and is compatible with m) is defined *uniquely up to isomorphism*.

Proof: The instantiated rule $\bar{p} := p\bar{\sigma}$ defines a single variable-free pushout rule $r : \bar{L} \longrightarrow \bar{R}$ where r is the inclusion morphism $r : \bar{L} \hookrightarrow \bar{R}$ embedding $\bar{L}$ in $\bar{R}$. The graph morphism m is a match for r in terms of SPO graph transformation theory.

In [Heckel *et al.*, 1995] and [Koch, 1999] 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

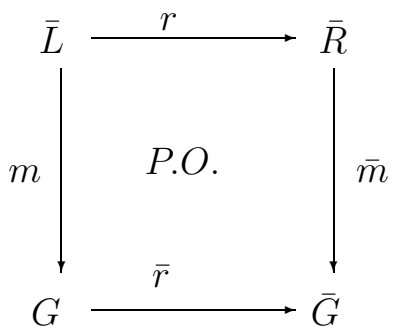

Figure 3.6: The pushout defining the result of the application of rule $p = (L, R, \Gamma, \tau)$ (represented by the inclusion morphism $r : \bar{L} \longrightarrow \bar{R}$) to a match m as specified in Definition 3.2.8.

clustered graphs is an instance of this category, this result applies also to clustered graphs. By definition, a pushout object (here $\bar{G}$) 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 $m : \bar{L} \longrightarrow G$ and $r : \bar{L} \longrightarrow \bar{R}$ in the domain of attributed graph structures is created by taking the disjoint union of G and $\bar{R} - \bar{L}$[5]. The attributes of old elements (in $\bar{r}(G) = G$) are kept. The attribute of a new element $x \in \bar{m}(\bar{R})$ is the attribute of its preimage in $\bar{R}$. Since $\bar{\sigma}$ 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. [6]

Construction 3.2.1 (Rule Application) Let $p = (L, R, \Gamma, \tau)$ be a rule and $m : L \longrightarrow G$ with $m = (m_{vertex}, m_{edge}, m_{cluster}, m_{dep}, m_{attr})$ be a match for p in G. Let $\bar{\sigma}$ be a substitution assigning ground terms to all variables occurring in R such that $\bar{\sigma}$ 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 V_G, edges E_G, clusters C_G, and dependencies D_G), 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 $X_{R-L} = \tilde{m}_X(X_{\bar{R}} - X_{\bar{L}})$ of $X_{\bar{R}} - X_{\bar{L}}$ under an arbitrary bijection $\tilde{m}_X$ which ensures $\tilde{m}_X(X_{\bar{R}} - X_{\bar{L}}) \cap X_G = \emptyset$. Therefore the respective component $m_X : X_{\bar{L}} \longrightarrow X_G \in \{m_{vertex}, m_{edge}, m_{cluster}, m_{dep}\}$ of the match m can be extended to a function $\bar{m}_X : X_{\bar{R}} \longrightarrow X_{\bar{G}}$ by defining $\bar{m}_X := m_X \uplus \tilde{m}_X$.

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

We define each of the source, target, labeling, and clustering function of $\bar{G}$ as extension of the respective function of G that is compatible with the respective function of R. Let $u_{\bar{G}} : X_{\bar{G}} \longrightarrow Y_{\bar{G}} \in \{s_{\bar{G}}, t_{\bar{G}}, a_{\bar{G}}, c_{\bar{G}}\}$. For $x \in X_G$ we define $u_{\bar{G}}(x) = u_G(x)$. For $x \in X_{R-L}$ the respective component $\bar{m}_X$ is bijective, hence we define $u_{\bar{G}}(\bar{m}_X(x)) := \bar{m}_Y(u_{\bar{R}}(x))$ which guarantees the commutativity condition $u_{\bar{G}} \circ \bar{m}_X = \bar{m}_Y \circ u_{\bar{R}}$ on X_{R-L}. In particular, this definition reconnects "dangling edges" since $u_{\bar{R}}(x)$ may be in $\bar{L}$ in which case $u_{\bar{G}}(\bar{m}_X(x)) = \bar{m}_Y(u_{\bar{R}}(x))$ is in G.

Altogether this ensures that $\bar{m} : \bar{R} \longrightarrow \bar{G}$ where $\bar{m} = (\bar{m}_{vertex}, \bar{m}_{edge}, \bar{m}_{cluster}, \bar{m}_{dep}, \bar{m}_{attr})$ is a graph morphism that forms an extension of the match m to a full match.

The morphism $\bar{r}$ is defined as the inclusion morphism $\bar{r} : G \longrightarrow \bar{G}$. □

It has to be shown now that Construction 3.2.1 indeed yields the pushout of (m, r). But before that we first introduce the following lemma:

[5] Note, that the difference of two graphs is not a graph since it may have "dangling edges". Hence care has to be taken to reconnect those edges properly when merging the difference graph with some other graph.

[6] An alternative method that is used in [Löwe, 1993] is to take a disjoint union of copies of G and $\bar{R} - \bar{L}$ and glue all elements with common preimages in L together.

Lemma 3.2.1 Let F, G, H be graphs and $g : F \longrightarrow G$, $h : F \longrightarrow H$ such that $g(x_1) = g(x_2) \Longrightarrow h(x_1) = h(x_2)$ for all $x_1, x_2 \in F$. Then $f(y) := h(x)$ for $y = g(x) \in g(F)$ defines a unique graph morphism $f : g(F) \longrightarrow H$ that satisfies $f \circ g = h$.

Proof: Define $f(y) := h(x)$ for $y = g(x)$. For $x' \in F$ satisfying $g(x') = y$ it follows that $h(x') = h(x)$, hence $f(y)$ is independent of the choice of the preimage x of y.

Let u denote either the source, target, clustering, or attribute function of a graph. In order to verify that f is a graph morphism we have to show that u commutes with f just as it does with the graph morphisms g and h: $u_H(f(y)) = u_H(h(x)) = h(u_F(x)) = f(g(u_F(x))) = f(u_G(g(x))) = f(u_G(y))$. Hence $u_H \circ f = f \circ u_G$.

f is uniquely defined since $f \circ g = h$ implies $f(y) = h(x)$ for $y = g(x)$. □

Theorem 3.2.2 The pair $(\bar{m}, \bar{r})$ as defined in Construction 3.2.1 forms a pushout with pushout object $\bar{G}$.

Proof: We have to show first that the diagram in Figure 3.2.1 commutes. Both r and $\bar{r}$ are inclusion morphisms. Moreover, $\bar{m}(x) = m(x)$ for all $x \in L$. Hence $\bar{m}(r(x)) = \bar{m}(x) = m(x) = \bar{r}(m(x))$ for each element $x \in L$ which proves $\bar{m} \circ r = \bar{r} \circ m$.

Second we have to show that for any alternative pair $(\bar{m}', \bar{r}')$ of morphisms $\bar{m}' : \bar{R} \longrightarrow \bar{G}'$ and $\bar{r}' : G \longrightarrow \bar{G}'$ that satisfies $\bar{m}' \circ r = \bar{r}' \circ m$ there is a unique morphism $d' : \bar{G} \longrightarrow \bar{G}'$ such that $\bar{r}' = d' \circ \bar{r}$ and $\bar{m}' = d' \circ \bar{m}$.

We use Lemma 3.2.1 to define graph morphisms $d_1 : G \sqsubseteq \bar{G} \longrightarrow \bar{G}'$ and $d_2 : \bar{m}(\bar{L}) \sqsubseteq \bar{G} \longrightarrow \bar{G}'$ and show than that $d' := d_1 \cup d_2 : \bar{G} \longrightarrow \bar{G}'$ is well-defined and satisfies the required properties.

Since $\bar{r}$ is the inclusion morphism, Lemma 3.2.1 can be applied immediately and hence $d_1(x) = d_1(\bar{r}(x)) := \bar{r}'(x)$ is the unique graph morphism on G satisfying $d_1 \circ \bar{r} = \bar{r}'$.

For d_2 we have to show the premise of Lemma 3.2.1 using $g := \bar{m}$ and $h := \bar{m}'$. Let $x_1, x_2 \in \bar{R}$ such that $\bar{m}(x_1) = \bar{m}(x_2) =: y$. Since $\bar{m}$ is by construction injective on $\bar{R} - \bar{L}$ and its range on $\bar{R} - \bar{L}$ does not overlap with its range $\bar{L}$, the only nontrivial case is $x_1, x_2 \in \bar{L}$. In this case $x_i = r(x_i)$. We use this and the commutativity $\bar{m}' \circ r = \bar{r}' \circ m$ to derive $\bar{m}'(x_i) = \bar{m}'(r(x_i)) = \bar{r}'(m(x_i)) = \bar{r}'(y)$. Hence $\bar{m}'(x_1) = \bar{r}'(y) = \bar{m}'(x_2)$. Now we can apply Lemma 3.2.1 and conclude that $d_2(y) := \bar{m}'(x)$ for $y = \bar{m}(x)$ is well-defined and is the unique graph morphism on $\bar{m}(\bar{R})$ satisfying $d_2 \circ \bar{m} = \bar{m}'$.

Finally we have to show that d_1 and d_2 coincide on the overlap of G and $\bar{m}(\bar{R})$: An element y is in the intersection of G and $\bar{m}(\bar{R})$ iff there exists a $x \in \bar{L}$ such that $y = \bar{m}(x)$. By using again the commutativity $\bar{m}' \circ r = \bar{r}' \circ m$ and $\bar{m}|_{\bar{L}} = m$ we yield $d_1(y) = \bar{r}'(y) = \bar{r}'(\bar{m}(x)) = \bar{r}'(m(x)) = \bar{m}'(r(x)) = \bar{m}'(x) = d_2(\bar{m}(x)) = d_2(y)$. Hence $d' := d_1 \cup d_2$ is well-defined and unique.

Therefore $(\bar{m}, \bar{r})$ is the pushout of (m, r) and $\bar{G}$ is its pushout object. □

Example 3.2.3 In order to demonstrate this construction, we show the effect of applying the rule get_issue (depicted in Figure 3.4) 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 3.7 only fragments of these clusters and disregard the HTML clusters on which the ACR cluster depends.

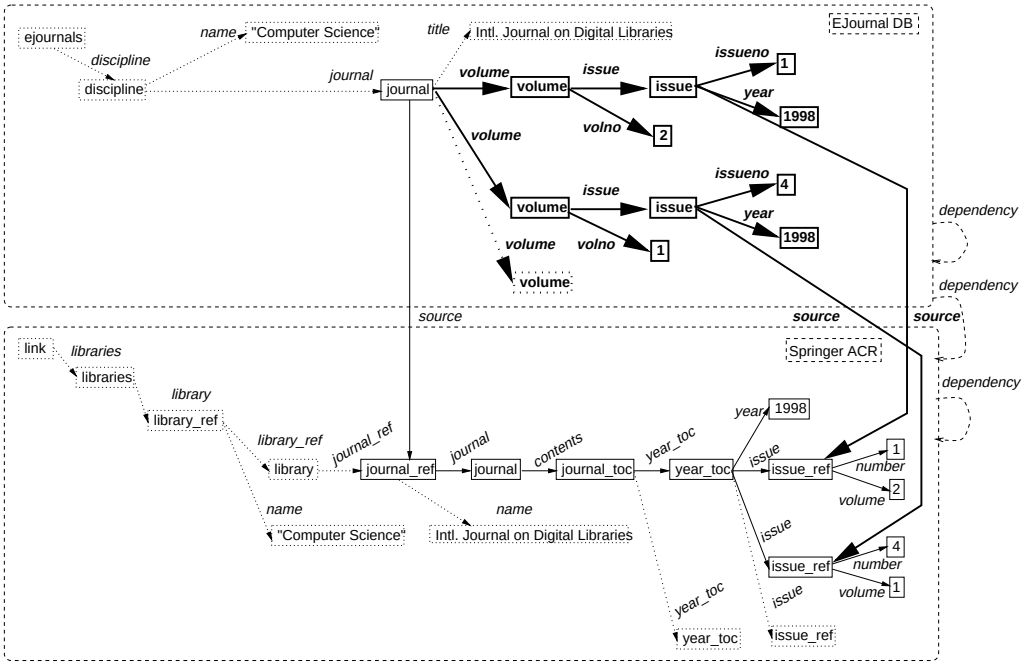

Figure 3.7: Two applications of the rule get_issue, corresponding to the issues 1/4 and 2/1 of the "International Journal on Digital Libraries". New elements are indicated by bold lines, elements matched by the left hand side by normal full lines, and irrelevant elements which are not matched and further matching elements which have been omitted are denoted with dotted lines.

□

3.3 Queries and Oracles

In the previous 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 3.3.1 (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 3.3.2 (Solution) Let G_0 be a data graph with interpretation $\rho_0 : G_0 \longrightarrow S$. Let q be a query as defined above.

A **solution** for q is a triple (G, ρ, m) consisting of a supergraph G of G_0 having an extension $\rho : G \longrightarrow S$ of ρ_0 as interpretation 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. □

We now introduce the concept of an operator that takes a query and a data graph and returns the solutions that result from applying this query to the data graph. We treat this operator as a black box that has to satisfy certain properties, but can be implemented arbitrarily. A similar concept exists in complexity theory [Davis and Weyuker, 1983]: an *oracle* is a black box that is assumed to compute a certain (typically uncomputable) function. We borrow this concept, but use it strictly for *computable* functions. Moreover, we show later how more powerful oracles can be build on top of existing oracles using the rules of a hyperview.

Definition 3.3.3 (Oracle) An **oracle** Φ is an operator which takes a query $q = (Q, Q_0, \Gamma, \tau)$ and a data graph G_0 (with interpretation $\rho_0 : G_0 \longrightarrow S$) and returns a set $\Phi(q, G_0)$ of solutions (G, ρ, m) for q w.r.t. G_0.

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

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 vertices of Q (except of those in Q_0) are typed by vertices of one of the schema clusters c_i, i.e., $\forall v \in V_Q : v \notin V_{Q_0} \Rightarrow c_S(\tau(x)) \in \{c_1, \ldots, c_n\} \subseteq C_S$. □

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 3.8.

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.

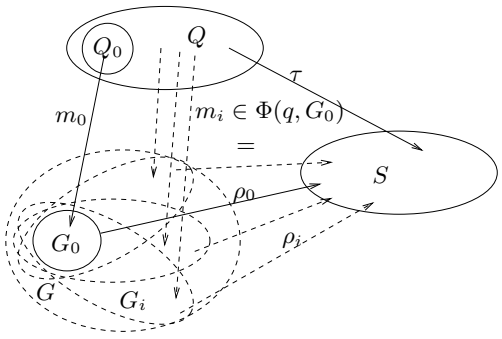

Figure 3.8: A query $q = (Q, Q_0, \Gamma, \tau)$ against an oracle Φ w.r.t. an initial data graph G_0. The solution set is indicated by dashed lines.

Example 3.3.1 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 tries to find matches for Q_0 in the already materialized HTML clusters and for each of these matches it tries to complete this match to a number of matches for the whole query graph Q. To do so, it loads HTML pages from the WWW, triggered by the attempt to match edges representing hyper-links, e.g., the href_target edge in the query depicted in Figure 3.9. The WWW oracle supports sub edges to denote a transitive descendent relation among HTML page elements.

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

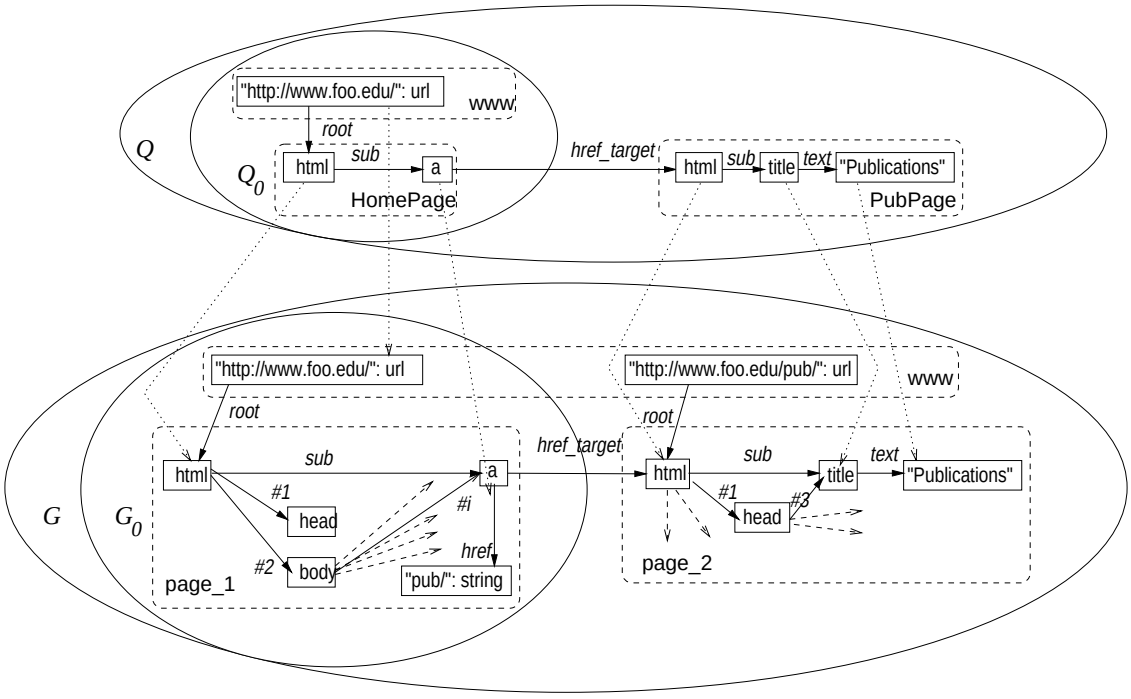

Figure 3.9: Query against the WWW oracle matching the hyper-link from a home page to the publications page of www.foo.edu. The home page is assumed to be already materialized in a cluster page_1. The match indicated by dotted arrows binds the query cluster labeled with the variable HomePage to the HTML cluster labeled page_1 and the query cluster labeled PubPage to the newly loaded HTML cluster page_2.

□

3.3.1 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, for instance by requiring that L has a unique root vertex which forms a singleton anchor graph.

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.

We now also formalize the concept of *input* and *output clusters* of a rule. Input and output clusters of a rule are schema clusters. The idea is to allow new graph elements only to be created in data graph cluster that corresponds to the output cluster.

Thus a rule gets the following form:

Definition 3.3.4 (Rule) *— final definition*

Let S be a schema and $c_0, c_1, \ldots, c_n \in C_S$ be clusters of S.

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

1. a pattern graph R (see Definition 3.1.10), 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 $\tau : R \longrightarrow S$. We require that all vertices of L are mapped by τ to schema vertices belonging to the clusters $c_0, \ldots, c_n$ and all vertices of R that are *not* in L to be mapped to schema vertices belonging to c_0.
5. a boolean term Γ from $T_\Sigma(\mathbb{V})$ interpreted as an **application constraint** for p.

We call c_0 the *output cluster* and $c_1, \ldots, c_n$ the *input clusters* of the rule p. □

Remark 3.3.1 Although the definition does not explicitly restrict the typing of edges in R, the properties of morphisms ensure that τ maps edges of R to schema edges belonging to dependencies that connect the clusters $c_0, \ldots, c_n$. The creation of vertices or edges within data graph clusters that correspond to an input cluster is thus excluded. However, edges to or from such a data cluster are permitted. □

Definition 3.3.4 is illustrated by Figure 3.10. In Figure 3.11 the rule get_issue already introduced in Figure 3.4 is depicted with anchor graph.

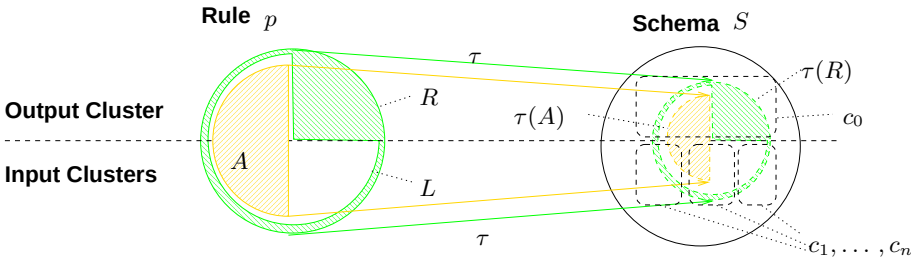

Figure 3.10: Schematic diagram of a rule p with anchor graph A and its typing τ in a schema S.

The query associated with a rule $p = (A, L, R, \Gamma, \tau)$ now 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 $m_0 : A \longrightarrow G_0$.

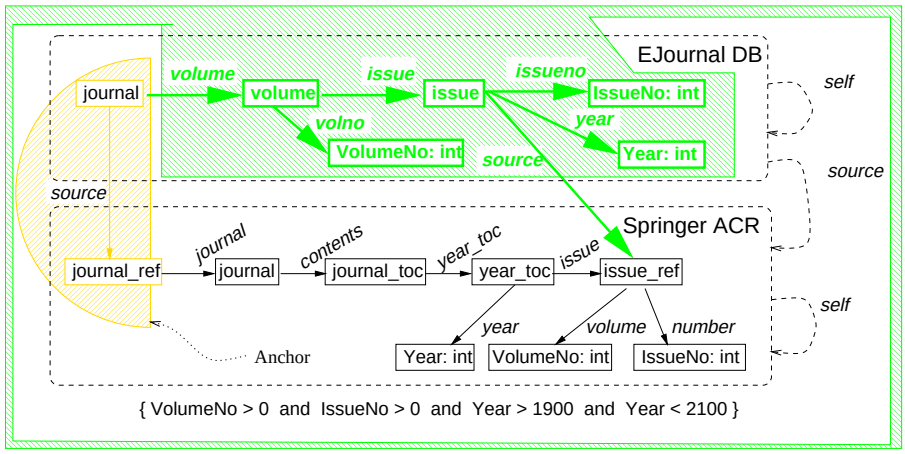

Figure 3.11: ACR Rule get_issue with anchor graph. Typically the source edge created by the right hand side will provide a match for the anchor graph of another rule to be called after get_issue.

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 3.12.

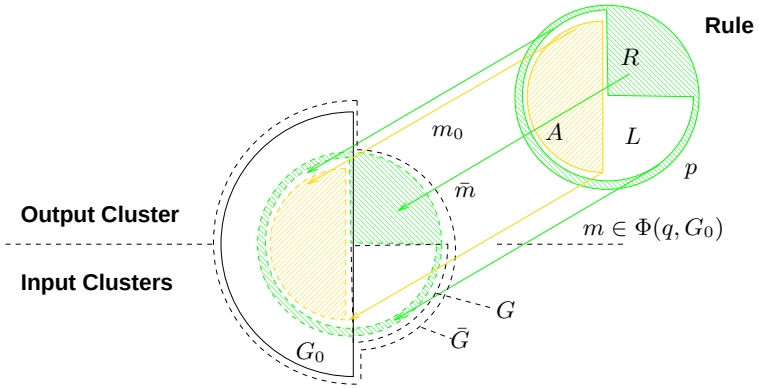

Figure 3.12: Applying a rule to a *virtual* data graph with oracle Φ. The match for A is $m_0 : A \longrightarrow G_0$, the match for the left hand side $m : L \longrightarrow G$, and the full match for the right hand side is $\bar{m} : \bar{R} \longrightarrow \bar{G}$.

Definition 3.3.5 (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 operator** for oracle Φ. □

The rule application operator $Apply^{\Phi}(.|.)$ 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 operator $Apply(.|.)$ for rule application without oracle as defined in Definition 3.2.8.

3.3.2 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 3.3.6 (Hyperview) Let S be a schema. Let $c_0, c_1, \ldots, c_n$ be disjoint clusters of S such that there exists in S a dependency from c_0 to each c_i, $i = 1, \ldots, n$. Let $S_0 \sqsubseteq S$ be the subschema constructed by omitting all other clusters of S and their dependencies.

Let Π be a set of rules. We call Π a **hyperview** with input clusters $c_1, \ldots, c_n$ and output cluster c_0 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 c_0, i.e., $c(\tau(v)) = c_0$
3. each edge $e \in E_R - E_L$ is typed by a schema edge belonging to a dependency emanating from c_0, i.e., $s(c(\tau(e))) = c_0$
4. the variable set of p does not overlap with the variable set of any other rule in Π

□

3.3.3 Using a rule to answer a subquery

Let Π be a hyperview (cf. Definition 3.3.6) 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 3.3.7 (Binding Morphism) Let $q = (Q, Q_0, \Gamma_q, \tau_q)$ be 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 3.3.2 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 3.13.

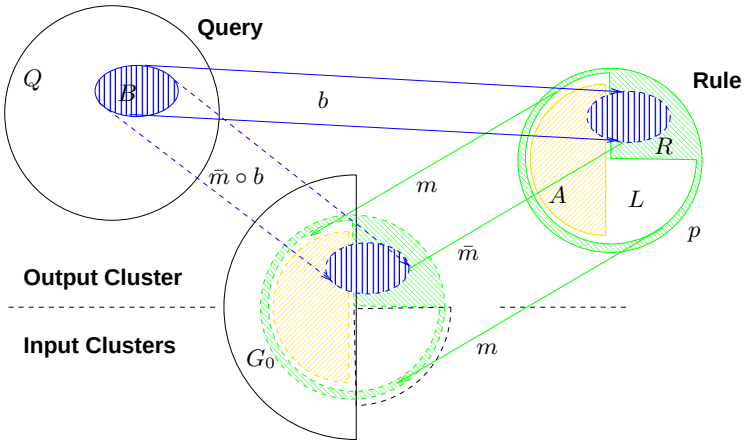

Figure 3.13: Using a binding morphism to obtain a match for a subquery B.

3.3.4 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 there is a match for its anchor graph in the existing data graph, care must be taken to activate rules in the right order. If an anchor match does not exist in the initial data graph it has to be materialized by a preceding rule. Only if this is guaranteed for all rules in the query execution plan, a query can be answered completely.

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 3.3.8 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 3.14 and can also be seen in the example of the query execution plan shown in Figure 3.15.

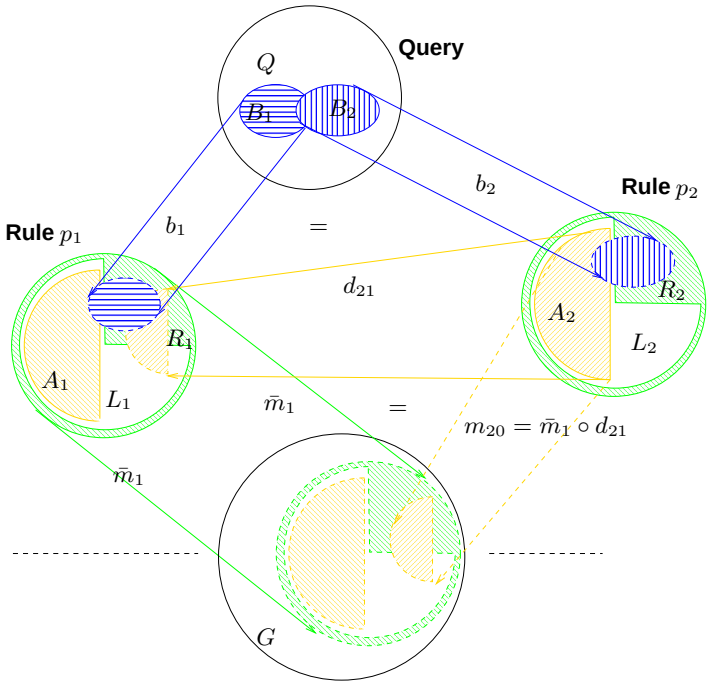

Figure 3.14: Chaining of rules p_1 and p_2. Rule p_1 has already been applied, yielding a full match $\bar{m}_1$. Rule dependency morphism d_{21} maps the anchor graph of p_2 to the right hand side of p_1 and thus lifts the full match $\bar{m}_1$ for p_1 to a partial match m_{20} for p_2.

Definition 3.3.8 (Rule Dependency) Let p_1, p_2 be rules $p_i = (A_i, L_i, R_i, \Gamma_i, \tau_i)$ for $i = 1, 2$.

A **dependency** of rule p_2 from rule p_1 is a type-compatible morphism $d : A_2 \longrightarrow R_1$ mapping A_2 to the right hand side R_1 of p_1. □

Having defined rule dependencies, we require that only binding morphisms compatible with existing rule dependencies are considered. For the following definition we need the concept of compatible morphisms introduced in Definition 3.1.7. Two morphisms are compatible with each other if they coincide on the intersection of their domains.

Now we can define two binding morphisms to be admissible with respect to a rule dependency if elements in the two rules which are bound to the same query element are also connected by the rule dependency. This requirement will ensure that only consistent matches will be chosen for these rule elements and in consequence for the query element.

Definition 3.3.9 (Admissible Binding Morphisms) Let p_1, p_2 be rules $p_i = (A_i, L_i, R_i, \Gamma_i, \tau_i)$ for $i = 1, 2$ and $d : A_2 \longrightarrow R_1$ mapping A_2 a dependency of p_2 from p_1.

Let Q be a query graph and $b_i : B_i \longrightarrow R_i$ be binding morphisms for $i = 1, 2$, respectively. Let $F = B_1 \cap B_2$.

Then b_1, b_2 are called *admissible* with respect to d iff

1. F is nonempty
2. F is mapped completely into A_2, i.e. $b_2(F) \sqsubseteq A_2$
3. b_1 and $d \circ b_2$ are compatible, i.e., $b_1|_F = d \circ b_2|_F$ holds.

□

Definition 3.3.10 (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 $B_i \sqsubseteq Q\sigma$, $p_i = (A_i, L_i, R_i, \Gamma_i, \tau_i) = p_{i0}\sigma$ for some $p_{i0} \in \Pi$
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 $d_i : A_i \longrightarrow R_j$ for which b_i and b_j are admissible, 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. □

Query execution plans for a query can be generated automatically. By using the schema information and the rule typings, critical pairs of overlapping rules and the dependencies between these rules can be identified. We do not go into details of plan generation here, but rather define the notion of plan generator as a black box:

Definition 3.3.11 (Plan Generator) A **plan generator** $Plans^{\Pi}$ is an operator which assigns to a query q the set $Plans^{\Pi}(q)$ of all minimal query execution plans for q with respect to hyperview Π. □

In Figure 3.15 a simple QEP is shown. It involves only two productions, the rule p_1 =get_issue shown in Figure 3.4 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 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.

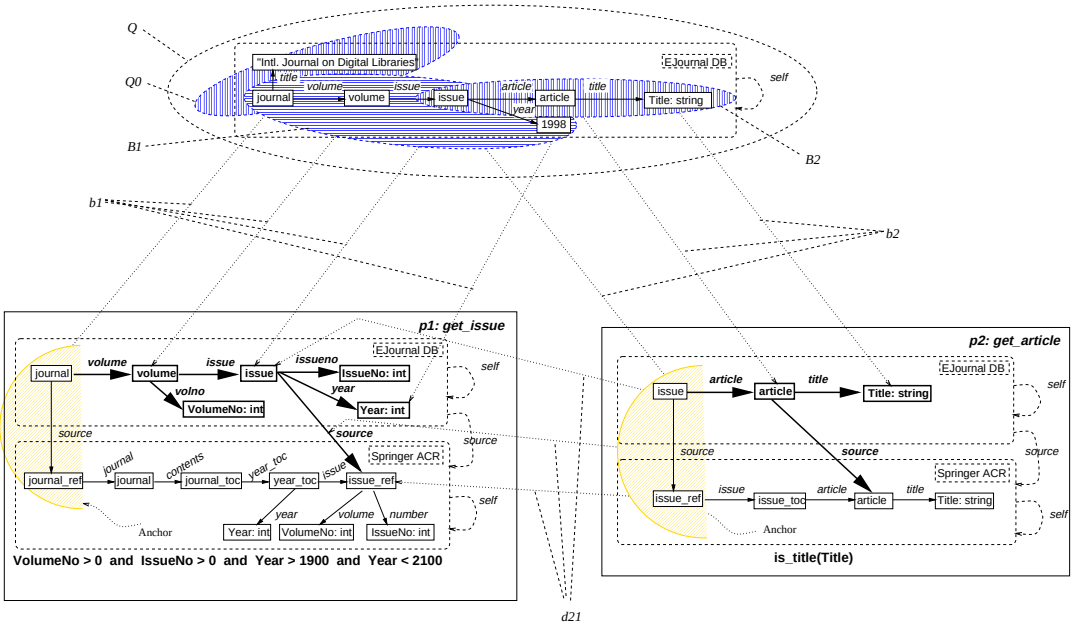

Figure 3.15: Example of a query execution plan

Recall, that the solution of a query is a match for the query graph into an (expanded) data graph that contains the initial match of the query anchor in the initial data graph. Using a QEP, a solution for the query can be constructed from matches for the rules referenced in the QEP.

Definition 3.3.12 (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 $m_0 : B_0 \longrightarrow G_0 \in Matches(B_0, G_0)$ 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 $PlanOracle^{\Phi}(P|m_0)$. □

The following 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. The rule dependencies guarantee that matches for the anchor graphs are provided, thus avoiding the problem that the data graph is not sufficiently materialized to fire a rule.

Construction 3.3.1 (Solution for a QEP, Plan Oracle) Using the names of Definition 3.3.12, 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 $m_{i0} \in Matches(A_i, G_{i-1})$ 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 $\bar{m}_i : R_i \longrightarrow G_i \in Apply(p_i|m_{i0})$ 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 $PlanOracle^{\Phi}(P|m_0)$. □

Remark 3.3.3 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. □

Remark 3.3.4 The term QEP is used here slightly differently than in the field of databases: executing a QEP 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. □

We come now to a construction which is central for the HyperView architecture since it provides the formal foundation for the composition of HyperViews. We construct an oracle that uses a HyperView Π and an oracle Φ for the input clusters of Π to answer queries against the HyperView.

Definition 3.3.13 (HyperView Oracle) Let $q = (Q, Q_0, \Gamma, \tau)$ be a query.

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

Let $Plans^{\Pi}$ be a plan generator for hyperview Π.

Then we define the query match operator $Oracle^{\Phi,\Pi}$ which returns all matches for q when starting from initial data graph G_0 with respect to the oracle 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 3.3.5 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 3.3.3 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. □

3.4 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 3.4.1 (Reuse Specification) Let $p = (A, L, R, \Gamma, \tau)$ be 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. □

The application of a rule p with reuse specifications $K_1, \ldots, K_n$ to a match m in a data graph G is specified by the algorithm in Figure 3.16.

This algorithm starts with the match $m : K \longrightarrow G$ where $K = L$ (1). It sequentially checks for each reuse graph K_i whether the current match m can be extended to a match m' that covers also the reuse graph (3). In this case, m' becomes the current match (4).

Otherwise the data graph G is extended (8) with an isomorphic copy K_i' of K_i under a isomorphism $\tilde{m}$ (6) that is compatible with m and allows K_i' to overlap with G only in the part matched by K (7). The current match is then extended with $\tilde{m}$ (9). This second case is similar to the normal rule application step, except that R is replaced by K_i.

In both cases, K_i is included in K and the algorithm proceeds with the next reuse graph.

This approach can be expressed in a declarative way by using rules with negative structural application conditions. A rule with a reuse specification is translated into a set of rules each of which covers the case that a certain subset of the reuse graph can be matched in the data graph. However, the number of resulting rules will be 2^n for n reuse graphs, and only one of these rules can be applied. Hence it would be completely inefficient to use this translation as an implementation technique.

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 would 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.

3.5 Bibliography on Graph-Transformation

The formal framework of the HyperView approach is based on the *algebraic approach to Single Pushout* (SPO) graph transformation as treated in [Löwe, 1993]. This approach applies not only

Input: $L, K_1, \ldots, K_n, m : L \longrightarrow G$
Output: K (includes L and some K_i),
extended match $m : K \longrightarrow G$

(1) K **:=** L
(2) **for** i = 1 **to** n **do**
(3) **if** $\exists m' : (m' : K \cup K_i \longrightarrow G \ \wedge \ m'|_K = m)$ **then**
(4) m **:=** m'
(5) **else**
(6) **let** $\tilde{m} : K_i \longrightarrow K_i' = \tilde{m}(K_i)$ **be** a graph isomorphism
(7) **such that** $\tilde{m} \nabla m \ \wedge \ G \cup K_i' = m(K)$
(8) $G := G \cup K_i'$
(9) m **:=** $m \cup \tilde{m}$
(10) **end**
(11) K **:=** $K \cup K_i$
(12) **end**

Figure 3.16: Algorithm for applying a rule with reuse specifications.

to classical graphs, but to so-called *graph structures* in general. A graph-structure is an universal algebra with unary operators only. In [Heckel *et al.*, 1995], this approach is extended to *attributed* graph structures, a generalization of labeled graphs.

In the SPO approach, a graph transformation rule is a (possibly partial) graph morphism which maps a LHS graph to a RHS graph. The intuitive meaning of applying such a rule to a graph is the following: Graph elements matching LHS elements are transformed into elements matching the images of these LHS elements in the RHS. If the graph morphism is not defined for a particular LHS element, the corresponding graph element will be *deleted*. If a RHS element is not in image of the LHS, then a corresponding element will be *added* to the graph. Using category theory, the resulting graph can be characterized as the pushout object of the graph morphism defining the rule and the graph morphism specifying the match of the LHS in the original graph.

An alternative graph transformation technique is the *Double Pushout* (DPO) approach [Ehrig *et al.*, 1991] in the category of partial graph morphisms. There, a rule is expressed as a pair of graph morphisms which map a common *interface graph* to the LHS and to the RHS, respectively. All LHS elements which are in the image of the interface graph denote graph elements which must be retained. All remaining LHS elements denote graph elements that must be deleted. Similarly, all RHS elements not in the image of the interface graph denote graph elements to be added. Additional constraints ensure that the result of applying a rule is again a graph. The name of the DPO approach indicates that the category theoretic construction of the result graph involves two pushout diagrams. There are several extensions to the DPO approach, such as allowing labeled or incomplete graphs, using more powerful types of productions or support parallel and distributed rule application[Taentzer, 1996].

There is a number of other approaches to graph transformation such as hyper-edge replacement, higher-order replacement systems etc., the discussion of which is beyond the scope of this work. The currently most complete source on the various approaches to graph transformation is [Rozenberg, 1996, Rozenberg *et al.*, 1999a, Rozenberg *et al.*, 1999b]. Surveys of relevant literature are also given in [Ehrig and Taentzer, 1996, Nagl, 1979]. A short tutorial to the SPO and DPO approaches is presented in [Ehrig *et al.*, 1991].

3.6 Summary

In this chapter 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 Out rules with application conditions on attributes. The main contribution of this formalism 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 is published in [Faulstich, 1998] and [Faulstich, 1999b]. This framework forms the theoretic basis on top of which the HyperView System is implemented. The storage and manipulation of graphs by the HyperView System is presented in Chapter 4. The language HVQL which is used to formulate rules and encode schemata is presented in Chapter 5. There, the translation of HVQL into Prolog is discussed.