Chapter 4

The HyperView System

In this chapter the HyperView System is presented. It is a platform for implementing virtual Web sites based on the concept of hyperviews introduced in the preceding Chapter 3. The HyperView System is implemented mainly in Prolog. Hence it uses Prolog encodings for the introduced formal concepts like clustered graphs, schemata, queries, rules, and views that are presented in this chapter. In Chapter 5 we present the HVQL language for denoting graphs, queries, and rules in a more intuitive way and the translation of HVQL into the Prolog encodings presented here.

4.1 Encoding of Graphs

We start with the encoding of graphs, beginning with plain graphs and then moving on to clustered graphs. in the HyperView System, graphs are used to represent data and schemata at the different layers of the system as well as further metadata.

4.1.1 Plain Graphs

Plain graphs (cf. Definition 3.1.1 on page 25) are directed vertex- and edge-labeled graphs of the form $G = (V, E, s, t, a)$ where V denotes the set of vertices and E the sets of edges. s and t are functions that map edges to their source and target vertices, respectively. a assigns each vertex and edge its label. Labels are terms from a term algebra $T_\Sigma()$

We represent a vertex u with label $a(u) = x$ as a Prolog fact `vertex(`u`,`x`)` and an edge e with source $s(e) = u$, target $t(e) = v$, and label $a(e) = l$ by a fact `edge(`u`,`l`,`v`)`. Vertices from V are implemented by Prolog atoms. Edges are anonymous in the sense that they do not have identifiers. Hence multiple edges having the same source, target, and label cannot be distinguished. However, since the multiplicity of an edge is not relevant in most applications, this does not pose a problem in practice.

As an optimization, we adopt the convention to use a single edge `edge(`u`,`l`,`x`)` instead of the two facts {`edge(`u`,`l`,`v`)`, `vertex(`v`,`x`)`} to represent an atomic attribute value $u.l = x$ of an object u. To this end, we extend the encoding by admitting vertices that are Prolog terms.

4.1.2 Clustered Graphs

A clustered graph (cf. Definition 3.1.2 on page 26) consists of a *base graph* and a *structure graph*. Each vertex of the base graph is assigned to a *cluster* that is modeled as a vertex of the structure graph. Similarly, each edge of the base graph is assigned to a *dependency*, i.e., an edge of the structure graph. This dependency points from the cluster of the edge source to the cluster of the edge target.

We consider all vertices belonging to the same cluster together with their outgoing edges as a module of the clustered graph. In Prolog, each module constitutes a separate database for facts and predicates. We use the syntax M`:`T to denote a fact, clause, or goal T in the context of module M. We implement each graph module corresponding to a cluster c by a Prolog module of its own. We denote the name of the Prolog module for cluster c by M_c.

We use the encoding for plain graphs introduced above to store the graph module for cluster c in the predicates `vertex/2`[1] and `edge/3` in Module M_c. This has the advantage that lookups for vertices and edges in a known graph module are more efficient than a lookup in a global fact database. Moreover, a modularized graph database is easier to manage.

However, there remains one problem to be solved: edges pointing to vertices in other clusters cannot be represented. Therefore, we introduce the special syntax `edge(`u`,`l`,`v`@`M_d`)` to denote an edge with target v in cluster d.

Since the details of this mapping from graph modules to Prolog modules should be hidden from the user, we introduce two predicates `vertex/3` and `edge/4` that are defined by the clauses `vertex(`M`,`V`,`X`):-` M`:vertex(`V`,`X`)` and similarly `edge(`M`,`U`,`L`,`V`):-` M`:edge(`U`,`L`,`V`)`. These predicates provide a global interface for the whole clustered graph.

[1] The notation p/n means a predicate or term with name p and arity n

4.1.3 Type checking

The formalism presented in Chapter 3 uses strong typing of graphs. Elements of data graphs are mapped to schema elements by interpretation morphisms (cf. Definition 3.1.10 on page 28). In the HyperView System, interpretations are not stored explicitly, only the schema cluster in which a data cluster is to be interpreted is kept in the system. This information allows to check whether a data graph module conforms to its corresponding schema module.

4.2 Encoding of Queries

According to Definition 3.3.1 on page 37, a *query* consists of a *query graph* Q, the *anchor graph* $Q_0 \sqsubseteq Q$, a *query constraint* Γ, and a *typing* τ that maps elements of the query graph to schema elements. Note that all the mentioned graphs are clustered graphs.

Answering a query for a given initial match $m_0 : Q_0 \longrightarrow G_0$ in an initial data graph G_0 amounts to finding matches $m : Q \longrightarrow G$ in an extension of G_0 that coincide with m_0 on Q_0 and satisfy the boolean condition on the label variables occurring in Q specified by the query constraint Γ. A match for a query graph is a graph morphism that assigns each element of the query graph a corresponding element of the data graph such that the query graph element and the data graph element are of the same type.

In the HyperView System, queries are encoded as Prolog goals. The translation into this encoding does not have to be done manually: in Chapter 5, the query language HVQL is presented which allows queries to be denoted in a more intuitive way.

The query graph is encoded in the HyperView System as a conjunction of atomic goals for the predicates `vertex/3` and `edge/4`. Distinct variables instead of atoms are used for the vertices of the query graph. The atomic goals have to be ordered according to a topological ordering. This means that the `vertex` goal representing the source of an edge has to come before the `edge` goal representing this edge which in turn comes before the `vertex` goal for the edge target. This way, the query graph goal implements the traversal of a data graph.

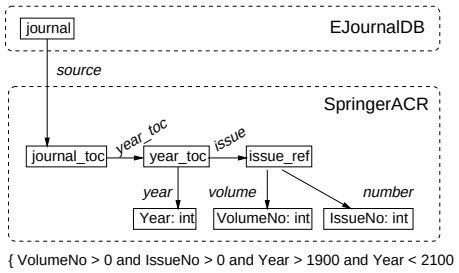
The anchor graph is encoded implicitly within the query graph. When the query is executed, it is required that the variables occurring in the anchor graph are bound to corresponding data graph elements with respect to a given initial match.

The query constraint is expressed as a conjunction of arbitrary Prolog goals. The conjunction of the query graph encoding and the query constraint encoding forms the encoding of the query. The conjuncts forming the query constraints can be arbitrarily mixed with the `vertex` and `edge` goals provided that upon execution, all constraint goals will be sufficiently instantiated. This can be checked statically by taking into account that all variables occurring in previous `vertex` or `edge` goals are already instantiated at the moment that the constraint goal is called. This reordering does not change the semantics of the query goal, but influences its performance. It is most efficient to execute constraint goals as early as possible.

The typing of a query is not explicit in its Prolog encoding; however, it is implicitly determined by the query and can be statically checked against the clustered schema graph. Currently, this is not supported by the HyperView System.

In Figure 4.1, an example for a query and its Prolog encoding is presented. As we have seen in Section 4.1.2, targets of edges pointing from one cluster into another are denoted using the special "`@`" syntax. This has to be taken into account when encoding the query graph as well. For instance, the second goal in Figure 4.1 denotes an edge from a vertex in the module to which the variable `EjournalDB` is bound to a vertex in the module to which the variable `SpringerACR` is bound as a result of calling this goal.

A query is executed by providing an initial match for the anchor graph and then calling the Prolog goal expressing the query. The initial match is given by *unifying* the vertex and cluster variables in the goal that belong to the anchor graph with the identifiers of the corresponding vertices or clusters of the data graph.



$\Longrightarrow$

```
vertex(EJournalDB,Journal,journal),
edge(EjournalDB,source,JournalTOC@SpringerACR),
vertex(SpringerACR,JournalTOC,journal_toc),
edge(SpringerACR,JournalTOC,year_toc,YearTOC),
vertex(SpringerACR,YearTOC,year_toc),
edge(SpringerACR,YearTOC,year,Year),
Year > 1900, Year < 2100,
edge(SpringerACR,YearTOC,issue,IssueRef),
vertex(SpringerACR,IssueRef,issue_ref),
edge(SpringerACR,IssueRef,volume,VolumeNo),
VolumeNo > 0,
edge(SpringerACR,IssueRef,number,IssueNo),
IssueNo > 0
```

Figure 4.1: Example of a query and its encoding in the HyperView System. Note that Prolog variables are always capitalized.

When the initially bound query goal is then called, Prolog enumerates all possible solutions of the goal. A subgoal of the form `vertex(`M`,`X`,`l`)` binds X to each vertex in module M with label l. A subgoal of the form `edge(`M`,`x`,`l`,`Y`)` binds Y to the target vertex Y of each l-labeled edges emanating from x.

Prologs starts the evaluation of a conjunctive goal with the leftmost subgoal and enumerates for each variable binding returned by this subgoal all compatible solutions of the remaining subgoals. This results in a depth-first traversal of the underlying data graph.

Each solution of the query goal corresponds to a match of the query graph in the data graph.

4.3 Encoding of Rules

A HyperView-Rule consists of a *LHS graph* contained in a *RHS graph*, an *anchor graph* contained in the LHS graph, a *typing morphism*, and an *application constraint* (cf. Definition 3.3.4 on page 40). Additionally, there may be a reuse specification with one or more reuse graphs associated with a rule (cf. Definition 3.4.1 on page 47).

We call the LHS the *query part* since it defines (together with the anchor graph and the application constraint) a query against the data graph. We call the elements of the RHS graph that are not contained in the LHS graph the *update part*. Note that the update part is not necessarily a graph, since there typically exist edges with source or target vertices in the LHS graph.

A rule is applied to an initial match for its anchor graph by executing the query defined by its query part, the anchor graph, and the application constraint. For each match of the query part, the graph elements defined by the update part are added to the data graph.

In the HyperView System, the following restrictions apply: the anchor graph must consist of a single vertex, and there is exactly one outgoing edge from the anchor vertex that points to a vertex in the update part. We call this edge the *primary edge* of the rule.

Note that all the graphs in the definition of a rule are clustered graphs. Rules have one or more input clusters and one output clusters. The query part may include all these clusters, while the update part is within the output cluster. In practice most rules will have a single input cluster even though this is not restricted by the implementation.

In Figure 4.2, a schematic diagram of such a HyperView System rule with a single input cluster is presented. The output cluster of the rule is determined by the module in which the rule is placed. Similar to queries, the typing of a rule is not given explicitly, but can be checked statically, by matching the rule against the schema.

A rule as a whole is represented as a clause of the predicate `compute_edge/4` that is called by the rule activation mechanism described in the next section. The query part of the rule together with the application constraint is represented as a query goal as presented in Section 4.2. The

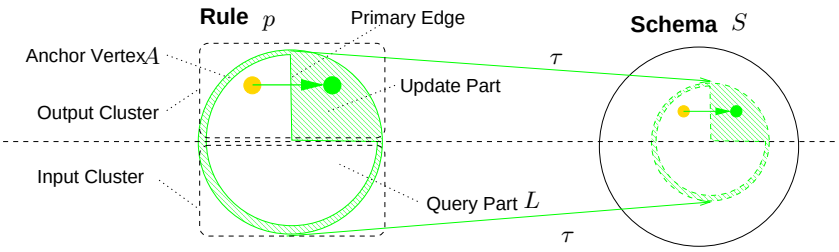


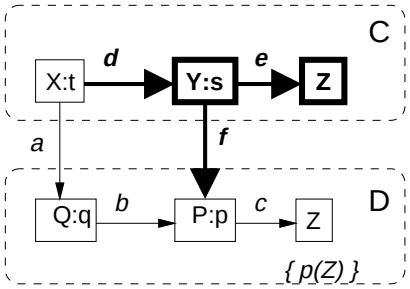
Figure 4.2: Schematic diagram of a rule p in the HyperView System.

encoding of the update part is discussed below. Let Q be a query goal representing the query part and U a goal representing the update part. Let X be the variable denoting the anchor vertex and C a variable denoting the module in which the anchor vertex is stored. Let l be the label of the primary edge and Y the variable denoting its target. Then the rule is denoted by the clause `compute_edge(`C`,`X`,`l`,`Y`):-` Q`,` U.

The update part is encoded in form of statements that insert new graph elements encoded as facts into the Prolog database. The HyperView System provides the meta-predicate[2] `update/1` that takes as argument a goal that is a conjunction of `vertex/3` and `edge/4` goals and executes this conjunction in a special environment where calls to `vertex/3` and `edge/4` are interpreted as insert-statements for vertices and edges, respectively. If the vertex identifier argument in a `vertex/3` goal is a variable, the system will generate and assign a new vertex identifier to it. `edge/4` goals have to be fully instantiated upon execution. This requires a slight reordering of the subgoals, compared to the encoding of queries: the `vertex/3` fact specifying the target of an edge has to be called before the `edge/4` fact in order to ensure that the target is bound to a vertex identifier when the edge is inserted.

Reuse graphs within the update part have to be translated into goals for the meta-predicate `ensure/1` that first executes its argument as a query goal to match the reuse graph in the data graph; only if this fails it will call `update/1` with this argument to add a new instance of the reuse graph to the data graph.

We illustrate the encoding of rules with a trivial example presented in Figure 4.3.




```
compute_edge(C,X,d,Y):-
    vertex(C,X,t),
    edge(C,X,a,Q@D),
    vertex(D,Q,q),
    edge(D,b,P),
    vertex(D,P,p),
    edge(D,P,c,Z),
    p(Z),
    update((
        vertex(C,Y,s),
        edge(C,X,d,Y),
        edge(C,Y,e,Z),
        edge(C,Y,f,P))).
```

Figure 4.3: Example of a trivial rule shown in graph representation (left) and its Prolog translation (right). The rule starts from a vertex `X` and creates for each graph match binding the variables `D`, `Q`, `P`, and `Z` a new vertex `Y` and edges labeled `d`, `e`, and `f` in cluster `C`. `Z` represents an attribute value that is not represented as a vertex of its own, but as edge target only (cf. Section 4.1.1)

[2]Traditionally, predicates that take Prolog goals as arguments are called *meta-predicates*. More precisely, such predicates are higher-order predicates.

When executing the goal `compute_edge(`c`,`x`,`l`,`Y`)` where x@c denotes an initial match for the anchor vertex of the rule, the query goal Q is called with $X = x$ and $C = c$. For each resulting rule match (cf. Definition 3.2.6 on page 33), the rule is applied (cf. Definition 3.2.8 on page 34) by calling U to add the elements of the update part to the data graph.

4.4 Rule Activation

Each cluster c in the graph database corresponds to a unique cluster $c' = \iota(c)$ in the schema graph. How this correspondence is stored in the HyperView System is discussed in Section 4.7. From the schema cluster c', several dependencies can point to other schema clusters $c'_1 \ldots c'_n$. Rules for c' may refer to c' and the c'_i. Most rules will refer to a single c'_i only. In order to support modularization, for each dependency (c', c'_i) there will be a separate implementation module M_i in which the translations of all rules referring to c' and c'_i are stored as clauses of the predicate M_i`:compute_edge/4`. However, this does not preclude a rule stored in M_i to access other input clusters as well.

In order to support virtual graphs that are materialized on demand, the `edge/4` predicate introduced in Section 4.1.2 is redefined to implement the following algorithm:

When solving a goal `edge(`c`,`u`,`l`,`V`)`, it is tried first to find all facts of the form M_c`:edge(`u`,`l`,`V`)` in module M_c. If no such facts can be found, it is tried to materialize the requested edge using an appropriate rule by calling M_c`:dispatch_edge(`c`,`u`,`l`,`V`)`. This predicate is imported into the graph module M_c from the schema module $M_{c'}$. For each implementation module M_i, this predicate contains a clause that calls M_i`:compute_edge(`c`,`u`,`l`,`V`)` in M_i in order to activate the relevant rules in M_i. The relevant rules within an implementation module are selected by matching the label l of the requested edge against the label of the rule's primary edge.

All solutions of all selected `compute_edge/4` clauses from all implementation modules M_i constitute the final solution set for the target V of the edge l starting in u. The graph elements of the rule update parts are created as a side effect of calling the `compute_edge/4` implementations. This side effect materializes the requested edge and thus saves the effort to recompute it the next time. Moreover, additional graph elements created by the rules may be necessary to answer subsequent requests for vertices or edges. For instance, the goal `edge(`c`,`u`,`l`,`V`)` retrieves only the identifier V of the target vertex, but not its label. The label has to be retrieved by a subsequent call `vertex(`c`,`V`,`x`)` that matches the `vertex/2` fact added by the rule as a side effect.

4.5 Query execution

How does the rule activation mechanism described in the last section implement the formal query semantics defined in Section 3.3? The formalism introduces the concept of query execution plans (Definition 3.3.10 on page 44) that cover a query graph with binding graphs that are mapped by binding morphisms to the RHS graphs of appropriate rules.

A straight-forward implementation of this query execution concept would require to determine and execute all possible QEPs. For each QEP, the rules called by this plan would have to be determined and an ordering of the rules would have to be established.

As mentioned before, the HyperView System implements a restricted form of HyperView rules. This restriction allows QEP and their binding morphisms to be built implicitly and incrementally: when calling `edge/4` as a subgoal of a query goal, the binding morphism for the edge is established by unifying the arguments with the respective parameters of the rule clause implementing the requested edge. Similarly, the rule dependencies and the initial rule matches induced by these dependencies are established by unifying the anchor vertex of a rule with a vertex identifier resulting from a previous rule application. Since alternative rule implementations may exist, the query plans corresponding to these alternatives are enumerated via the Prolog backtracking that selects the alternative clauses.

Thus, the underlying Prolog mechanisms of unification and backtracking are employed to find graph matches and to enumerate all solutions of a query in a natural way without explicit management of alternative QEPs and graph matches.

4.6 Complexity and Performance

Graph transformation techniques often have the draw-back of showing a very low performance. This is often excused with the exponential complexity of graph matching. One can easily see that for instance the task of finding all matches for a linear path of length n in a fully connected graph of N vertices takes N^n steps. Hence the worst case complexity of graph matching is indeed exponential. However by using efficient implementation techniques, the performance can be improved considerably. The main strategy for speeding up graph matching is to prune the search space. In the HyperView System this strategy applies as follows:

1. Since the cluster in which a graph element is to be matched is always known, the search space is reduced to a single cluster instead of the whole graph.
2. Most graph matches start from an anchor vertex that has a fixed match. This reduces the search effort by a factor that is equal to the number of vertices in the graph cluster.
3. Since the Prolog fact database is indexed by the Prolog machine, looking up partially instantiated graph elements does not require all graph elements in a cluster to be scanned. This applies for instance if an edge with given source vertex or the label of a given vertex have to be retrieved.

Since the costs of graph matching dominate the performance of query execution and rule application, the performance gains reached by these measures are sufficient to make applications like in those discussed in Chapter 2 and Chapter 7 feasible. This means that response times are typically below a few seconds. Timing experiments suggest that response time is dominated by the response times of the underlying sources and the network connection rather than the graph transformation operations.

4.7 Metadata management

Three kinds of metadata are stored in the HyperView System: schema clusters describing the structure of data clusters, information about all available graph clusters in the system, and data about URLs and Web pages.

4.7.1 Schema clusters

Schemata are represented as clusters whose vertices and edges represent vertex and edge classes, respectively (cf. Definition 3.1.10 on page 28). The labels of schema elements denote types that specify the admissible labels of corresponding instance elements. In the HyperView System, instance labels may be arbitrary Prolog terms. For schema labels, the following convention has been adopted: reserved names such as `integer`, or `atom` denote the corresponding atomic data types. Further names such as `url` denote application specific data types. Text vertices are represented by lists and denoted by the type name `text`. Except of these, all other schema labels denote the class of terms that are equal or more specific [3]. Thus a schema edge labeled `name` denotes instance edges labeled `name`, and a schema edge `select(_)` denotes instance edges labeled `select(`t`)` where t may be an arbitrary term.

Several examples of schema clusters are presented in Chapter 2 and Chapter 7. These examples have been exported by the HyperView System system from the respective applications.

[3] Formally, this means that the schema label subsumes all corresponding instance labels.

4.7.2 The `meta` cluster

The dedicated cluster named `meta` records information about all clusters in the HyperView System system. Each cluster is represented as a vertex whose identifier is the name of the Prolog module in which the cluster is stored. Additionally, the `meta` cluster also records all implementation modules containing rules. Three vertex types are distinguished: `instance`, `schema`, and (implementation) `module`. Each instance vertex has a `schema` edge pointing to the vertex representing its schema. Schemata have `implementation` edges pointing to the implementation modules. This information is crucial for static type checking of graphs.

The `meta` cluster is an instance of the `meta_schema` schema cluster. A schema cluster called `schema` specifies the schema of all other schema clusters. In Figure 4.4, an example of the `meta` cluster is depicted.

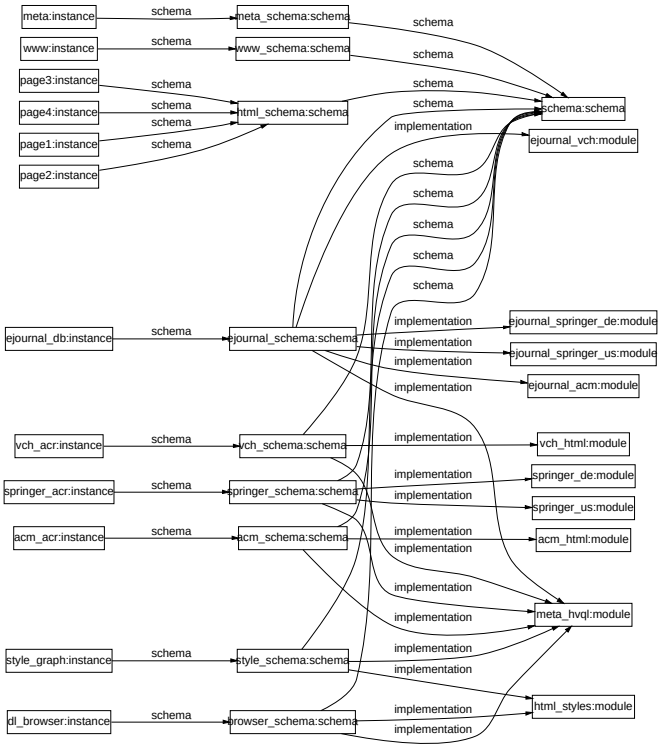


Figure 4.4: Example of a `meta` cluster for a run of the electronic journals application (cf. Chapter 2).

4.7.3 WWW meta data

In the dedicated cluster `www` the HyperView System stores metadata about documents loaded from the Web. It contains two vertex types, `url` and `page`. If the page from a certain URL has been loaded, vertices representing the URL and the module in which the page cluster is stored are created. From the `url` vertex, an edge labeled "`page`" points to the `page` node which has an inverse "`base`" edge. From each `page` node, a "`root`" edge points to the root node of the page cluster. Furthermore, the time at which the pages was loaded and the HTTP headers returned by the HTTP server are recorded as attributes of the `page` vertex.

Page fragments addressed by named anchors within the document text are represented by `page` vertices of their own that have a `fragment` attribute recording the name of the anchor and a `source` edge pointing to the `page` vertex representing the page as a whole.

For the www cluster, computed edges are available that support the loading of a page for a given URL, the conditional loading if the document at a URL has changed after some date, the deletion of a page cluster, and the call of an external Web browser for a given URL.

An example of a small www cluster is presented in Figure 4.5.

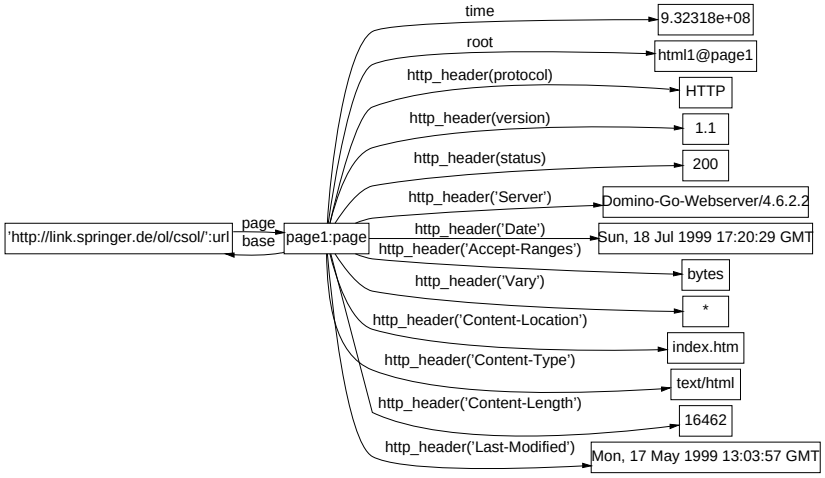


Figure 4.5: Example of the cluster www after loading a single page.

4.8 The HyperView System prototype

The HyperView System (without application-specific code) consists of approx. 250 kilobyte of Prolog code in approx. 9000 lines, divided into currently 49 modules. The application examples presented in this thesis fill another 5000 lines. The HyperView System runs on SWI-Prolog (Version 3.2.6), but can be adapted to most other current Prolog implementations that support modules. The HyperView distribution includes all necessary external tools. It can be easily installed on most Unix platforms since it supports automatic configuration.

The following external tools are used by the HyperView System:

wget: a HTTP client implementation that is used to retrieve HTML pages and XML documents from the Web.

JSDK: virtual Web sites are supported by a Java servlet that is based on the Java Servlet Development Kit (JSDK2.0). Includes the servlet runner (a rudimentary HTTP server). The servlet-based Web interface of the HyperView System is discussed in Chapter 6.

Apache + JServ: the Web interface servlet of HyperView runs inside a servlet-enabled HTTP server. The current distribution supports Apache(1.3.4) with servlet module JServ(1.0b3). This tool is not included in the distribution.

graphviz: a tool box for visualizing graphs. The graph diagrams in this thesis are exported from the HyperView System into the graph format of graphviz and then converted to PostScript by the dot program from this tool box.

HyperDesigner: a graphical editor for HyperView-rules and schemata. It is based on the graph editor VGJ and implemented in Java. From the graphical representation of rules and schemata HyperDesigner generates HVQL code that serves as input for the HyperView System. HyperDesigner has been developed as part of the HyperView project and is described in the diploma thesis [Öksüz, 1999b].

HyperDiscoverer: an interactive graphical tool for analyzing the structure of HTML graphs. HyperDiscoverer offers several algorithms for typing nodes and merging subtrees of similar structure. Together with HyperDesigner, HyperDiscoverer forms a development environment for the HyperView System. It is described in the diploma thesis [Öksüz, 1999a].

The architecture of the whole HyperView System prototype is shown in Figure 4.6. The servlet coupling is discussed in Chapter 6.

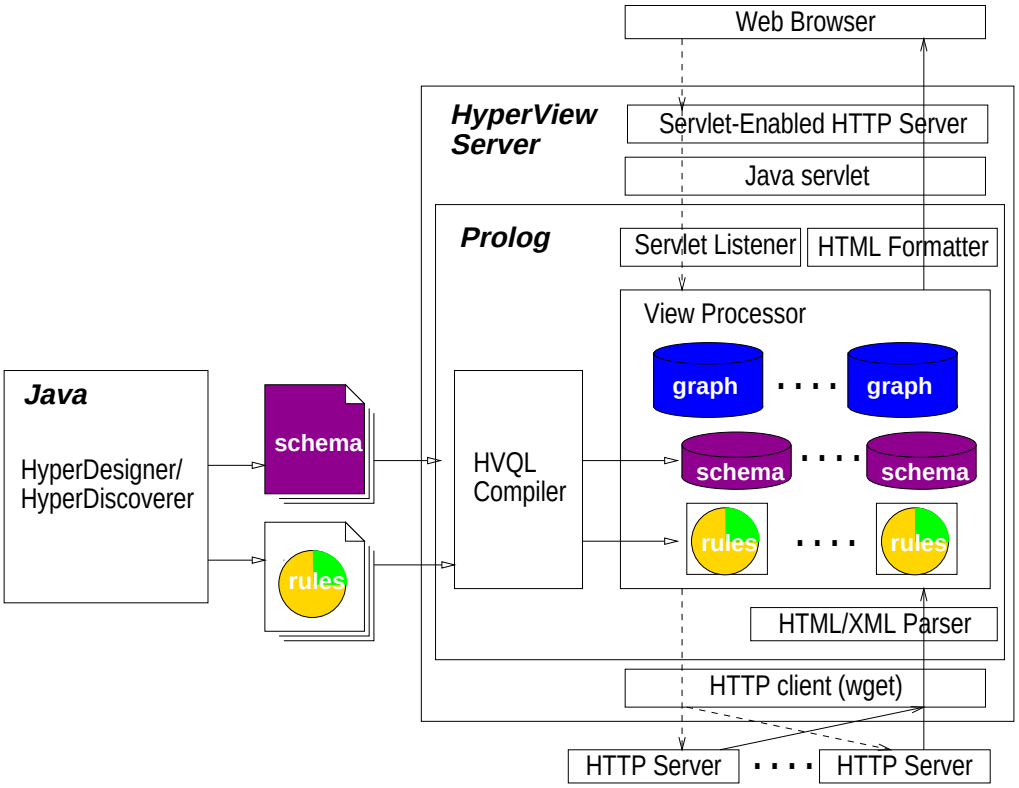


Figure 4.6: Architecture of the HyperView System prototype.

4.9 Summary

The HyperView System provides a software platform for implementing HyperViews and HyperView-based virtual Web sites in Prolog. It maps the concepts defined in Chapter 3 to Prolog constructs. Some restrictions of the formal framework apply in order to ensure the efficient execution of HyperViews.

Graphs are represented in the HyperView System as sets of Prolog facts. Queries are encoded as conjunctive goals over this fact database. Rules are implemented as clauses of Prolog predicates. These predicates return targets for requested edges that are not yet materialized. As a side effect, rule clauses materialize the requested edges and possibly further graph elements by adding them to the fact database.

The HyperView System additionally uses graph clusters to store schemata and metadata on the available graph clusters and the pages loaded from the Web.