Building HyperView Web Sites

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

Technical Report B 9/99

May 31, 1999

Abstract

In this report a framework for building "virtual" web sites using the HyperView system is presented. Virtual web sites are web sites that offer information extracted and integrated from other web sites on the fly. The HyperView system already supports the demand-driven integration of information from different semistructured information sources into a graph database. The problem we are dealing with here is to query the database and generate HTML pages from the results as a response to HTTP requests received from the user. The returned HTML pages should hide the aspects of data extraction and integration and should give the user the impression of a single, coherent web site.

We show first how HyperViews comprised of graph-transformation rules can be defined that generate HTML pages from the database. This way web sites for individual application schemata can be designed. In the second part we present a generic rule set that defines a web interface for HyperView graph databases with arbitrary schemata. This generic web interface can be customized for the particular application by annotating the database schema and chosing appropriate styles.

The work presented in this report completes the HyperView approach in the sense that it closes the circle of extracting and integrating information from the web by again publishing the integrated data on the web. Our approach applies as well to the integration and generation of XML documents on the web.

1 Introduction

The goal of the HyperView methodology is not only to extract and integrate information from heterogeneous web sites but to make this information again available as a web site. This web site is generated dynamically in response to HTTP requests. This involves loading pages from other web sites, extracting and combining the relevant data, and finally generating the requested pages from these integrated data. Hence we call it a *virtual web site*. From the client's point of view, a virtual web site behaves just like a normal web site. Thus the virtual web site can be explored by traversing hyper-links and filling out forms. A standard HTML browser is sufficient to access the virtual web site.

In order to support virtual web sites, the functionality of a HTTP server must be added to the HyperView System. There exist several technologies for coupling existing software systems with HTTP servers. In Section 2 we describe the architecture of the HyperView web interface which is based on the Java Servlet technology. A conceptual model of the virtual HyperView web site is presented in Section 3. In Section 4 we show how the graph transformation based view mechanism of HyperView can be employed to generate HTML pages for a specific application with a given database schema. Since it takes some effort to define individual HTML layouts for each node type in the graph database, it is useful to offer a generic solution for the HTML presentation of graph databases. With the help of several customization features, this generic database browser can be adapted to the needs of a particular web application. In Section 5 we present rules for a simplified version of the generic database browser implemented in the HyperView System. The HyperView web interface is a prototype. For large real-world applications, several extensions (discussed in Section 6) will become necessary.

2 Architecture of the HyperView Web Interface

The HTTP server of a virtual web site has to implement the following loop: it receives a HTTP request for a certain URL from a HTTP client, triggers the generation of the HTML page corresponding to this URL, and transmits the resulting page back to the HTTP client. Instead of implementing a HTTP server from scratch it makes more sense to extend the HyperView System with a module for generation of dynamic HTML pages and to couple this module with an existing HTTP server implementation that will take care of the HTTP communication. The resulting overall architecture of the HyperView server is depicted in Figure 1.

The HyperView System has two key requirements:

1. demand-driven operation, i.e., only those information necessary to answer a user's request are materialized

2. data caching for subsequent requests, i.e., data materialized in response to one request are kept and reused in future requests to minimize the loading of external pages

Currently, there exist several technologies for add HTTP server functionality to an information system:

Batch page generation: in principle, the HyperView System could also be used to generate static HTML pages in advance which are then served by a standard web server. However, this violates the requirement of demand-driven operation stated above.

Embedded server: the HTTP server is linked together with the information system into a single executable. The server triggers the generation of pages by calling appropriate (callback) functions of the information system. This solution is most efficient, but also least flexible and requires some programming.

Common Gateway Interface (CGI): for each incoming HTTP request the HTTP server starts an executable selected by the requested URL which writes the document text into its standard

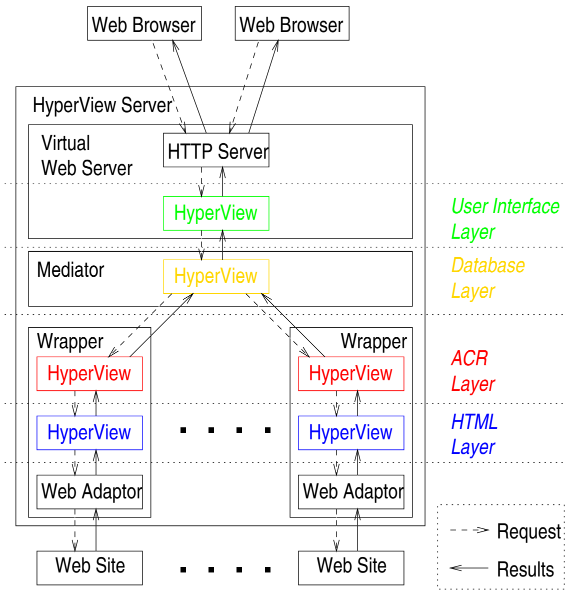

Figure 1: The HyperView architecture.

output and then terminates. The HTTP server then forwards the generated text to the HTTP client. Requirement 2 rules out the possibility of running the HyperView System inside of a CGI process. Instead, the CGI processes would have to act as temporary intermediaries using interprocess communication to communicate with a longrunning HyperView server process. This incurs a high overhead for startup of the CGI processes and setting up the contact with the HyperView server.

Fast CGI: this is an extension of CGI which supports longrunning CGI processes that serve multiple requests. Writing Fast CGI programs is different and more complex than writing standard CGI scripts. Moreover, the technology is not very widespread.

Java Servlets: A Java Servlet[1] is a Java object that runs in a separate thread within the Java virtual machine of a servlet-enabled Web server. It can handle successive requests and change its state in response to these requests. Moreover it can start the HyperView System as a subprocess that runs in parallel to the servlet.

The technology of Java Servlets satisfies both requirements stated above. Furthermore it encapsulates the HTTP protocol nicely in an object-oriented API. Third, this technology has become quite popular and it can be expected to become a quasi-standard for dynamic web documents in the future. For all these reasons it has been chosen as the method for coupling the HyperView System with a web server. The HyperView System is run as a separate process by the HyperView servlet. This architecture is shown in Figure 2.

Since the communication between servlet and HyperView server simply consists of page requests and returned HTML texts, it seems unreasonable to employ any complex middle-ware. Instead we use stream-based standard I/O for this purpose. Thus the HyperView servlet sends the URL and the query attributes of a HTTP request to the standard input of the HyperView server, where the Page Trigger module reads the request and issues a HVQL query that produces a HTML graph. The HTML Printer modules formats this HTML graph as a stream of HTML text which is written to the standard output. The servlet reads this output and passes it to its own output from where it is transmitted to the client browser by the HTTP server.

[1] <http://java.sun.com/products/servlet>

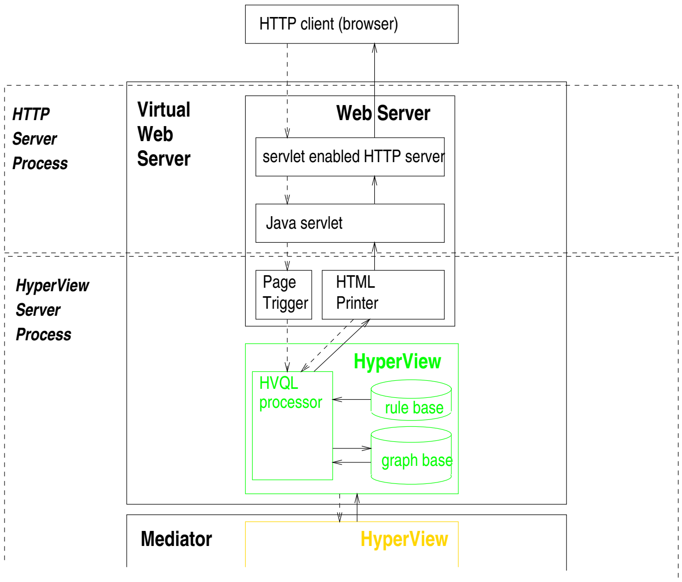

Figure 2: Architecture of the HyperView web interface. Details of the HTML and ACR layers are ommitted.

The use of the Java Servlet technology solves the problem of the *communication* between the HyperView System and the users' web browsers. In the next section, an abstract model for the *structure* of the resulting virtual web site is given.

3 Conceptual model of the virtual HyperView web site

The main desired features for a virtual web site are the following:

Look-and-feel of a conventional web site: the pages of the virtual web site must be connected by links and HTML forms to support navigation.

Multiple views on the database: the virtual web site should not be limited to a single HTML presentation of the database, but should support alternative views on the database. Adding new views should be easy and work independently of other pre-existing views.

Independent specification of content and layout: this is a requirement critical for the maintainability of the web site.

Configurable layout by uniform use of styles: the layout specification should be configurable by exposing styles that can be chosen and parameters that can be adapted. It is crucial that those styles are used throughout the whole web site since otherwise some parts may not be configurable.

Personalized presentation: the web site should support user sessions, i.e., the contents and layout of the pages served to the user may depend on the sequence of pages requested by the user up to this point. A session mechanism can be used to record and respect user preferences, but also for typical electronic commerce applications to keep track of the state of a negotiation process between a customer and the web site of an online vendor.

The concepts introduced above are now modeled in form of a graph schema which forms the basis for implementing the virtual web site as a view on the database graph of the HyperView system.

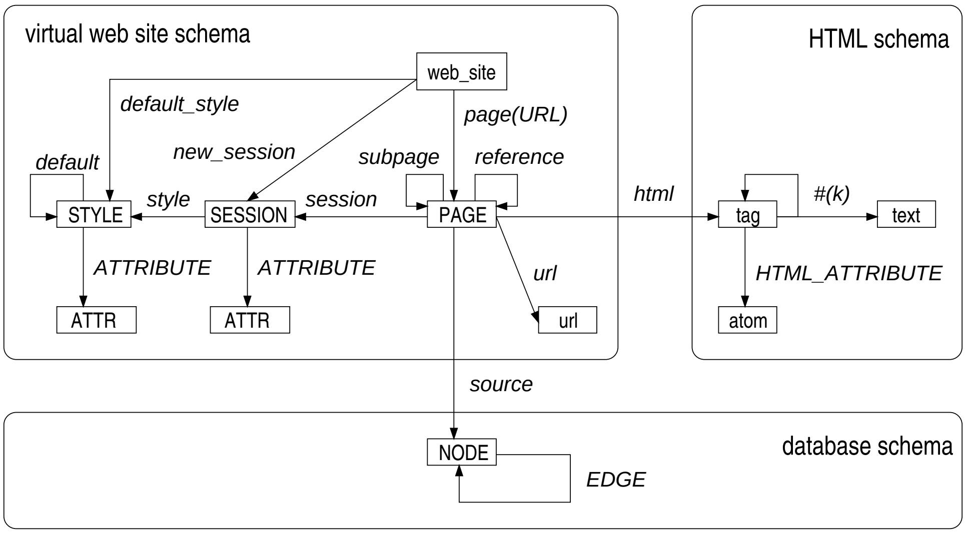

Figure 3: Abstract schema of the HyperView virtual web site.

This schema is depicted in Figure 3. It is abstract in the sense that for a particular web application the abstract node types PAGE, SESSION, STYLE, ATTR, and NODE have to be replaced by concrete node types. The schema shows that database nodes are represented by pages which belong to sessions. Session node may store settings which are global to all pages of a session and may be changed in response to subsequent page requests. In particular, each session has a style which controls the HTML code generation. Style nodes store settings in various attributes. Default values are found along a chain of default edges which provides a mechanism for cascading styles.

The HTML code generated for a page is represented as a graph conforming to the generic HTML schema. This schema models ordered trees of HTML tags whose leafs are text nodes or attribute values. Links between pages are modeled by reference edges between page nodes and later translated into anchor tags in the HTML graph. Pages can also contain other pages, indicated by subpage edges. Rules for generating such pages and the HTML formatting of these pages is discussed in the next section.

4 HTML Code Generation

A HTML page generated as response to a HTTP request typically contains the results of one or more queries against the database. In principle, the HTML code of a page can be generated by a single rule which has a query against the database graph as its body and creates vertices and edges of a HTML syntax graph. For each solution of the query a different HTML page will be generated. To collect all solutions of a query in a single page, the rule has to be applied exhaustively and the resulting HTML pages have to be merged to a single syntax graph.

As we shall see, a better solution is to use a set of rules each of which creates HTML fragments. These fragments are composed to a HTML page by a "dump" algorithm which completely materializes the HTML graph of a page by recursively traversing it and triggering a generic rule for computing the children of a HTML node. This generic rule in turn fires specific rules for the HTML representation of the involved database nodes.

A HTTP request to the HyperView web site is processed in three phases which are discussed next.

4.1 Phase 1: Preparation

The HTTP request for an URL U is received. Optional query attributes (as supplied by a HTML form) may be encoded in this URL. A page node P corresponding to U is found by following the page(U) edge from the web_interface root node (see Figure 3). If the requested page node P does not exist yet, it is created on the fly. In this case, also a new session node is created by following the new_session edge. This new session inherits the default style reachable via the default_style edge.

P may maintain one or more links to some database nodes. A new empty HTML cluster $page(P)$ is now created.

4.2 Phase 2: Generation of a HTML skeleton

The root and other fixed parts of the HTML cluster $page(P)$ are generated by requesting the edge html emanating from P. The rule implementing this edge will typically generate all fixed parts of the HTML graph such as the <html>, <head>, <title>, <body> tags and other more. This may include values of singleton attributes of database nodes associated with P.

In Figure 4 a minimalistic example of such a rule is depicted. For reasons of simplicity, this example makes no use of styles.

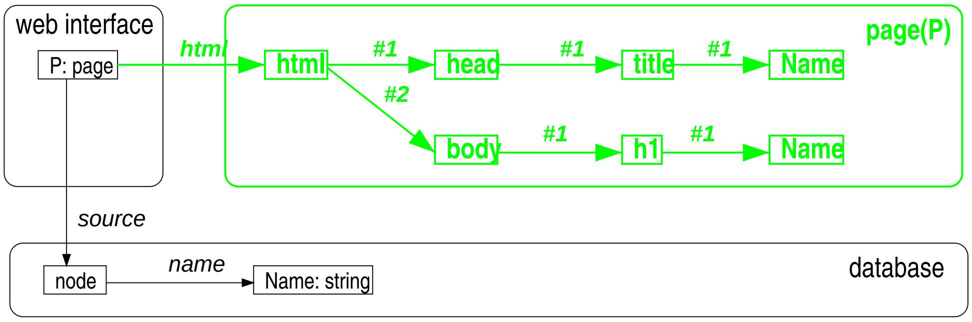

Figure 4: Rule for a minimalistic HTML page representing a database node of type node.

Attributes of database nodes which are optional or may have multiple values cannot be translated into HTML by this rule since for each value this would yield a different version of the page (or none, if the value is missing). Therefore the corresponding parts of the HTML graph have to be generated by separate rules. To indicate where these rules have to be applied and which information they should present, artificial edges are introduced to the HTML graph. These *marker* edges will not show in the generated HTML code, but guide the application of further generation rules.

4.3 Phase 3: HTML dump and generation of variable HTML code

In this phase, the HTML graph of the result page is "dumped", i.e. converted into HTML text. This is done in a uniform way by generating a pair of enclosing HTML tags for each HTML node, listing all attributes of the node in the opening tag, and recursively including the HTML text of all children reachable via its #i edges, sorted by i. These children may either have been be created in Phase 2 or they are computed on the fly by separate rules. Similarly, certain HTML attributes such as the href attribute of anchor tags can be computed on the fly as well. Thus, the HTML graph will be exhaustively materialized by the dump algorithm.

To illustrate this with an example, let's suppose that we extend the type node from Figure 4 by introducing an edge successor leading to another node. The HTML page generated for such an node should then contain a list of hyper-links to the successors of this node. First we extend the rule for the fixed part of the HTML page by adding the tag <ul> for an unordered list and a successors edge back to the node. This edge both marks the place in the HTML cluster where children have to be added and indicates the database node whose successors are to be presented on the page. The modified rule is shown in Figure 5.

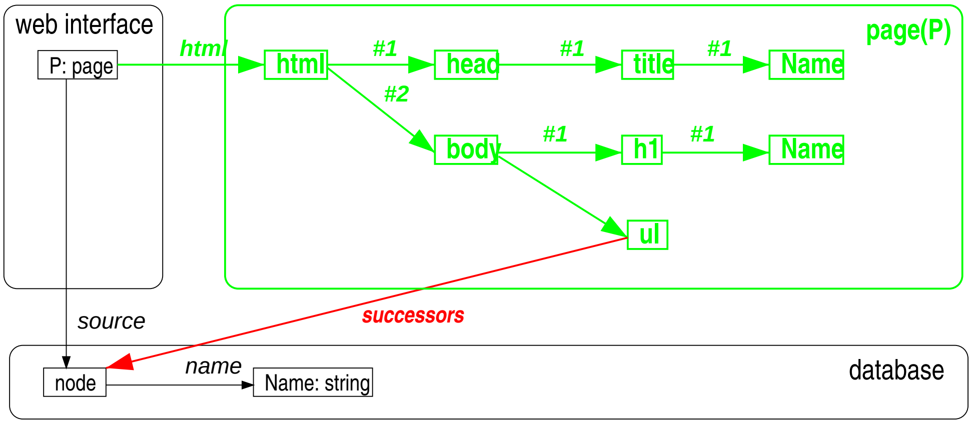

Figure 5: Extended rule for the HTML page of a node with successor edges.

Now we add the rule shown in Figure 6 for generating the hyper-links comprising the items of the mentioned unordered list of successor nodes. Each list item consists of a <li> tag being a child of the unordered list tag <ul>. The constraint new_index(I) ensures that the edges leading to the children are numbered in ascending order. Each child contains an anchor tag <a> with an href_page attribute specifying the target page of the hyper-link. The reuse specification (i.e., the loop enclosing the page node) indicates that an already existing page node for the successor node is to be reused. We assume the existence of a rule which materializes the href attribute of an <a> element as the URL of the page node referenced by the href_page edge. The anchor is labeled by the name of the successor node.

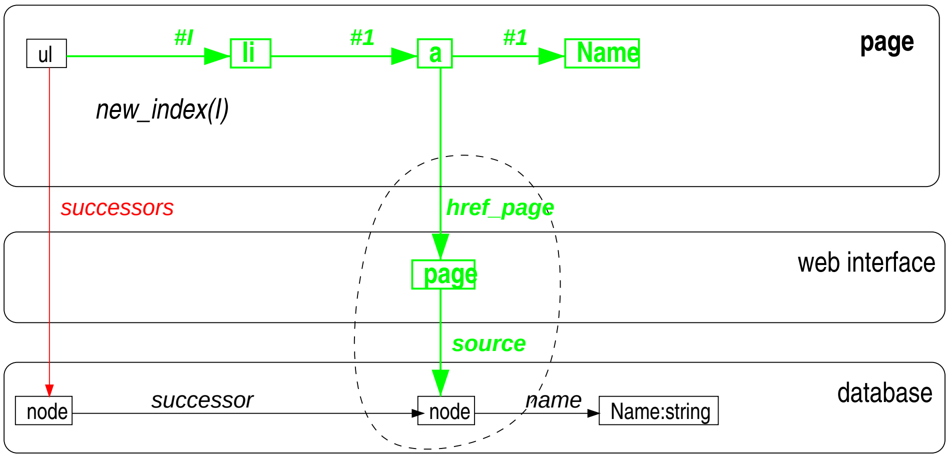

Figure 6: Rule for the successor edges of a node.

4.4 HVQL notation for HTML rules

Using the plain syntax of HVQL for expressing HTML rules leads to extremely verbose rule definitions. For instance, the rule depicted in Figure 5 is denoted by:

```
Page @ web_interface => html =
    HTML:html @ Page -> [
        #(1) = Head:head -> [
            #(1) = Title: title -> [
                #(1) = [Name]]]
        #(2) = Body: body -> [
            #(1) = H1: h1 -> #(1) = [Name],
            #(2) = UL: ul -> successors = Node @ DB ]]
     <== source = Node @ DB => name = Name.
```

To make the notation of HTML fragments easier a special syntax for HTML code has been introduced. It encodes a HTML element <*t*> by a prolog term with functor t and its children as arguments of this term. If the HTML element has attributes, these are simply added as additional arguments. Using this abbreviated syntax, the rule above has the following notation:

```
Page @ web_interface => html =
    html(   head(   title([Name]))
            body(   h1( [Name]),
                    ul( successors = Node @ DB))) @ Page
    <== source = Node @ DB => name = Name.
```

5 The HyperView Browser

The HyperView Browser consists of a set of HyperView rules which define for a database node of any type a HTML page presenting this node. This page lists all edges emanating from the node as well as links to pages which trigger the computation of edges which are not materialized (yet). Parameterized edges are represented by HTML forms in which the user can fill in the parameter values before requesting the corresponding page. The browser provides several features (discussed in the next section) to support a customized HTML presentation.

Here we present a simplified version of this browser which displays the target set for a given edge label on a separate page. Thus the target set is computed only when the page is requested which prevents an exhaustive materialization of the database graph. The rules for this browser can be derived from the rules presented in Figure 5 and Figure 6 by simple modifications.

The rules depicted in Figure 7 and Figure 8 define for a database node of arbitrary type a generic page showing a list of hyper-links labeled with the potential edge labels of this node. A label is called a potential edge label if there exists an edge carrying this label which emanates from the schema vertex representing the type of the node. The existence of a computed edge name which points to the name of a database node is assumed. In the second rule, a page of type edge_page identified by the database node and the edge label is reused or created.

In Figure 9 and Figure 10, the rules necessary for displaying the targets of all edges with a given label sharing the same source node are shown. The edge label is stored as an attribute of the page node. An edge labeled targets(L) connects the list node with the source node of the L-labeled edges whose targets are to be presented as items of the list. The rule depicted in Figure 10 adds for each target node a new list item containing a hyper-link which points to a page for this target node.

5.1 Customization

A uniform presentation of all database nodes results in a hyper-text which is tedious to read. The customization facilities of the browser allow to alleviate the following problems:

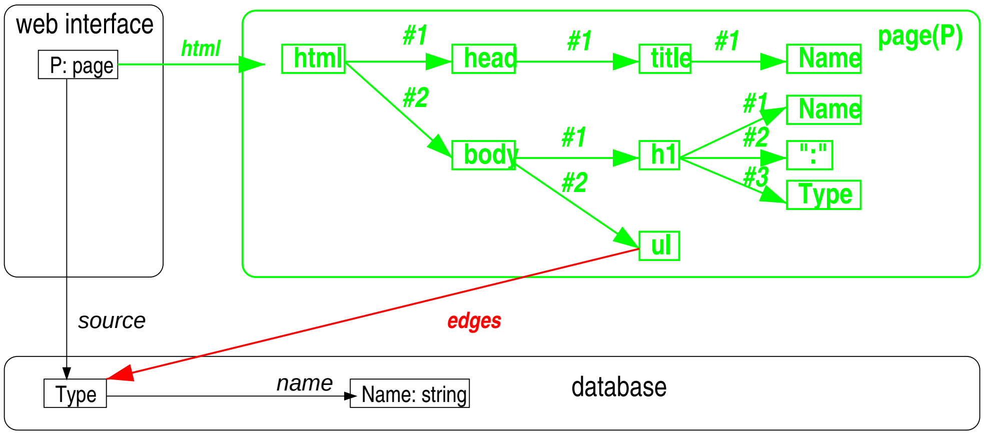

Figure 7: Rule for the generic HTML page of an arbitrarily typed database node.

Identifiers: a hyper-link to a database node is labeled by the node's type and the internal identifier. Since the identifier has little significance, the user has to visit the target page to find out which real world entity is represented by this identifier. The database browser allows to register for each database type a query returning a more informative label for the database node. For instance, journals might be labeled by their title, authors by their name and so on.

Materialization strategies: since the materialization of a computed edge may take time and involve the loading of external pages, the browser requires the user to explicitely request the computation of an edge by following a link. There are, however, computed edges which can be computed without much effort. There are also materialized edges which have so many targets such that the page becomes very large, resulting in an information overload. Therefore the default display strategy can be overridden for each edge in the schema. For instance, it could be specified that on the page for a journal issue the list of its articles is presented by default even though it has to be computed first. This mechanism also allows to specify that certain edges have exactly one target which is to be presented as a singleton value instead of a list.

Atomic formats: by default, atomic values in the database are presented as strings, using the implicit conversion facilities of the prolog output operators. This is clearly inadequate if the atomic value is for instance a URL or a calendar date in some encoding. To overcome this problem, a number of additional output formats for atomic values are available in the browser and can be selected for certain database types. Thus, URLs can be represented as hyper-links which allow the user to reach the URL by a single mouse-click.

Expand/collapse: following a link to a database node has the disadvantage that the context of this node is not displayed. As an alternative mechanism, it is therefore possible to expand links to database nodes in place and in arbitrary depth by including subpages into the current page. If the information is not needed anymore, the subpages can be collapsed again. Similarly, target lists of edges can be expanded or collapsed by the user. Thus, the user can select an arbitrary connected subgraph of the database graph to be displayed on a single page. This mechanism is based on manipulating the web interface graph by adding or removing edges annotating the page nodes.

Styles: in most applications, the HTML layout should follow a certain style. To adapt the HyperView web interface to such a style, the actual HTML generation is delegated to certain rules of style nodes. These rules are parameterized by attributes of the style node which can be

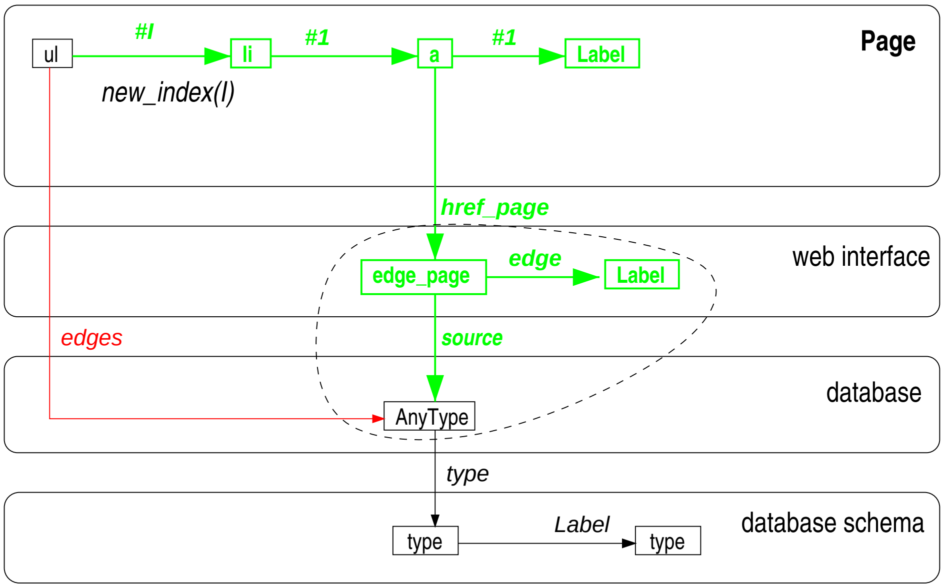

Figure 8: Rule for the list items showing the potential edge labels of an node.

overridden by chaining style nodes. This mechanism allows for instance to set colors, use certain icons, or choose between alternative HTML notations like ordered vs. unordered lists.

The customization facilities could be enhanced in the future by the following features:

Customizable content: by allowing the user to change the display strategies associated with the schema edges, the user can control on a per node type level which information is included in the generated pages.

Customizable styles: The style mechanism discussed above may be used to offer the user at runtime the choice of different layouts and to customize the layout by changing style parameters. To implement this, the browser should be extended with a mechanism for editing graphs. This requires a mechanism for specifying access rights in order to avoid unwanted changes to the HyperView graphs.

Sorting: sometimes the order of edge targets is significant. Hence the specification of sorting criteria should be supported.

Ordering: The ordering of the potential edges of a node depends on the internal ordering of the schema edges which can be influenced by the ordering in the schema specification. Nevertheless, it would be helpful to allow an explicit specification.

Parameter selection: In the current version, parameterized edges are already presented as forms with text fields. If the set of admissable parameter values is finite and is known in advance or can be computed, alternative input elements such as pop-up menus should be provided.

In Figure 11, an example screen shot of the generic browser in the electronic journal application is shown.

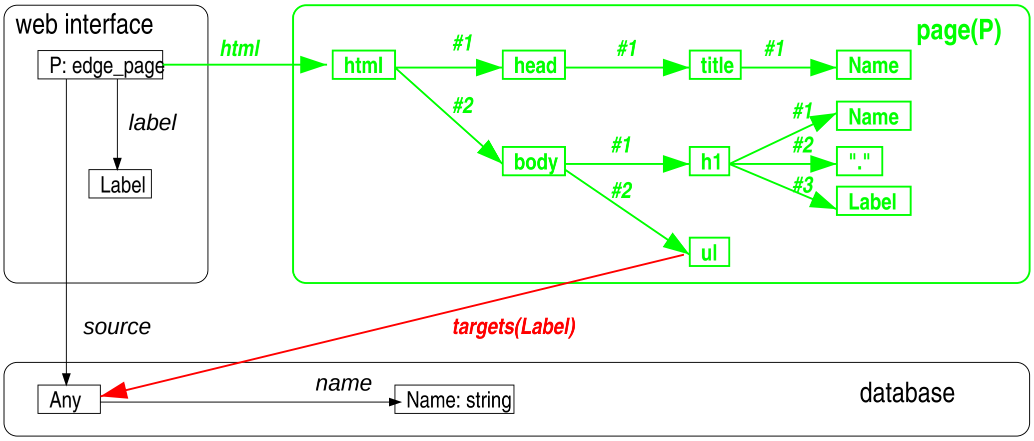

Figure 9: Rule for the generic HTML page showing the targets of edge labeled Label starting from a database node.

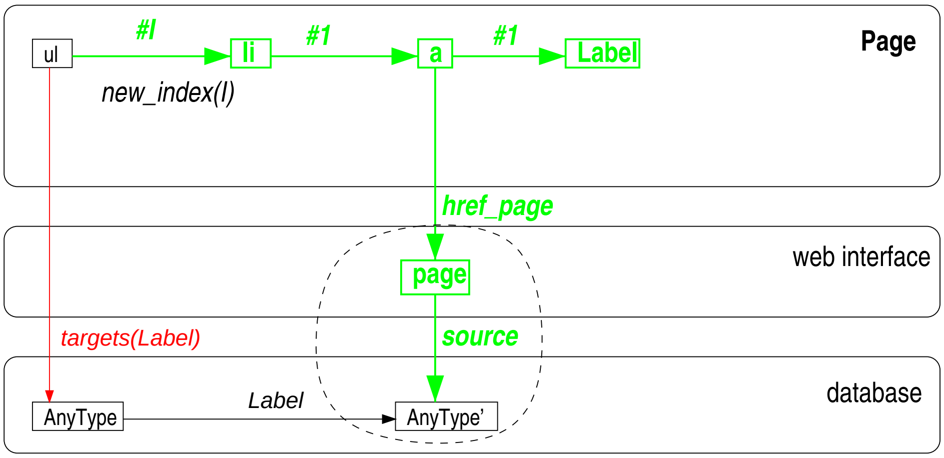

Figure 10: Rule for the list items showing the targets of the edges labeled Label.

6 Future extensions and discussion

Besides the future extensions of the generic database browser discussed above, there are also several extensions to the web-interface prototype as a whole which are necessary to support large real-world applications.

Garbage collection: since more and more data may be materialized with each page request, a garbage collection mechanism is necessary if the HyperView System is to run over longer periods of time. Since the garbage collection of Prolog does apply to the fact database, a collection mechanism based on page timestamps is planned. As a first solution, the system can be restarted if the occupied memory exceeds a certain limit.

Persistent storage: The state of a user's traversal in the HyperView web site is represented by a session node and information associated with it. User profiles can be stored in such session nodes as well, but will not persist when the HyperView System has to be restarted. A persistent storage facility for selected parts of HyperView graphs is therefore necessary to solve this problem.

Figure 11: A screenshot of the database browser. It shows the VLDB journal, with the current issue, its article list, and one of the articles expanded.

Cache warming: the execution of queries requiring the loading of a lot of external pages can exceed the response time tolerated by the user. To speed up such queries, the HyperView System could be forced to materialize often used parts of its database by executing some general queries like "find out which journals are available". This will force the relevant pages to be loaded in advance. In our example, the retrieval of a journal by its title can be executed in-memory, without any external page accesses.

Error recovery: the HyperView servlet has to detect the case that the HyperView System crashes or is blocked by a non-terminating query. This should be implemented as a time-out mechanism which shuts down and restarts the HyperView System process if necessary.

Multithreading: since the Prolog machine on top of which the HyperView System is implemented does not support multithreading, only one page request can be executed at a time. Therefore the HyperView servlet should be extended to support parallel incarnations of the HyperView server. The Java multithreading facilities and the session support offered by the Java Servlet API should make this feasible.

The proposed extensions can be implemented in a straight-forward way. Even without them, the HyperView web interface prototype is already usable for small applications. More important, the existing prototype demonstrates the feasibility of a virtual web site based on the HyperView approach. With the generic database browser it provides a rapid and flexible solution to the problem of presenting graph databases in the web.

7 Conclusion

With the HyperView web interface, the main goal of the HyperView System implementation is reached. The prototype now implements a complete virtual web site. It demonstrates the feasibility of the hyperview mechanism for each of the steps of information extraction, integration, and presentation.