Information Extraction from the Web

Wolfgang May Georg Lausen
{may|lausen}@informatik.uni-freiburg.de

TECHNICAL REPORT No. 136

Institut für Informatik
Albert-Ludwigs-Universität
Georges-Koehler-Allee, Geb. 51
79110 Freiburg, Germany

March 2000

Abstract

The goal of information extraction from the Web is to provide an integrated view on data from autonomous, heterogeneous information sources. The main problem with current wrapper/mediator approaches is that they rely on very different formalisms and tools for wrappers and mediators, thus leading to an "impedance mismatch" between the wrapper and mediator level. Additionally, most approaches nowadays are restricted to access information only from a fixed set of sources. On the other hand, *generic* Web *querying* approaches are restricted to pure syntactical and structural queries and do not deal with semantical issues.

In this paper, we discuss an integrated architecture for Web exploration, wrapping, mediation, and querying. Our system is based on a unified framework – i.e., data model and language – in which all tasks are performed. We regard the Web and its contents as a unit, represented in an object-oriented data model: the Web structure, given by its hyperlinks, the parse-trees of Web pages, and its contents are all included in the internal world model of the system. The advantage of this unified view is that the same data manipulation and querying language can be used for the Web structure and the application-level model: The model is complemented by a rule-based object-oriented language which is extended by Web access capabilities and structured document analysis. Thus, accessing Web pages, wrapping, mediating, and querying information can be done using the same language.

This integration also allows for data-driven Web exploration which is independent from a given network of individual predefined wrappers and mediators. Thus, in addition to the classical wrapper and mediator functionality, a system with this architecture can be equipped with *Web navigation* and *exploration* functionality. Queries to existing Web indexing and searching engines can also be integrated.

In particular, we present a methodology for reusing generic rule patterns for typical extraction, integration, and restructuring tasks using this framework. In an abstract sense, the system contains a universal wrapper, which can be applied to arbitrary Web pages that the system learns about during information processing. Equipped with suitably intelligent rules, the system can potentially explore initially unknown parts of the Web, thus coping with the steady growth of the Web.

We show the practicability of our approach by using the Florid system [HKL+98]. The approach is illustrated by two case-studies.

Contents

1 Introduction

The Web is now the most popular information repository and there is a strong need for finding relevant information in the Web, and for integration of data from different sources. In general, searching and integration tasks are decoupled: To find information, *search engines* based on semantically simple indexing techniques are often used. *Web querying languages* propose more sophisticated approaches (which have not yet found their way in everyday's use) which allow for SQL-like queries considering the Web structure and keywords when searching for specific information. Here, research is influenced from the area of *semistructured data* in general. Research on *information integration* from given Web sources is often influenced from the area of heterogeneous distributed databases. Since Web sites can be seen as *autonomous* databases that export "unwrapped" semistructured data, there are many exceptions and pitfalls when wrapping such sources. Additionally, data representation can change without notice, and new sources can be added. Thus, flexibility, e.g., supported by rapid prototyping and refinement, is highly desirable for Web data extraction systems. The goal of our work is a system for *spontaneous information extraction from the Web*. In the following we define the relationships of this task with related research areas:

- *Web searching and querying:* Searching for specific information is currently done by *search engines* (e.g., AltaVista [Alt98]) on a very basic level, but with a world-wide search scope. Extended functionality is provided by SQL-like *Web querying languages* that require the urls where the search has to be started to be known in advance. Web querying languages only cover structural and syntactical aspects, neglecting the integration and (semantical) restructuring of the results. Research topics concerning search engines mainly focus on algorithms and technology; research topics concerning Web querying languages focus on models and language concepts.
- *Information extraction from certain Web pages ("Wrapping"):* Given a Web page or a set of Web pages, the problem is to extract the relevant information into an application-level model. This task is in general solved by site-specific wrappers, which are either programmed in a Web querying language, or in a standard programming language (mostly Java) and produce an output in a common data model. Research areas are (i) languages and tools for efficient (manual) wrapper specification, and (ii) concepts for automatic generation of wrappers.
- *Information Integration:* Here, several information sources that provide complementary information have to be integrated at the semantical level. These information sources provide an interface to a common data model with a given signature, already at the semantical level (possibly provided by a wrapper). The integrating module ("mediator") is specified in a language operating on the common data model. It translates user queries into queries on the component sources, and integrates the partial results into an answer for the user. Research topics here are, e.g., language concepts, the use of *ontologies*, and the use of *resource descriptions*.

A standard approach in this area is the mediator architecture [Wie92] which comprises *wrappers* for translating data from different local languages into a common format, and *mediators* for providing an integrated view on the data. Following the wrapper/mediator architecture, most existing approaches have a strictly layered architecture: Every data source is assigned a wrapper, the upper level(s) are built by mediators connected to those wrappers, that provide relevant information for them. The network of sources, wrappers, and mediators has to be fixed a priori. In general, such systems rely on very different formalisms, languages, and tools for wrappers and mediators, leading to an "impedance mismatch" between the wrapper and mediator level. In general, their world model does not contain knowledge about the Web structure itself, which is required for autonomous Web exploration when searching for specific information, and for adaptability. These approaches are well-suited for integration of information from heterogeneous sources, if the Web structure is already explored and interpreted, and all sources are wrapped and stable.

In contrast, *spontaneous* information extraction has to take into account that (i) the Web is steadily growing and (ii) sites are *autonomous*, i.e., their structure can change, or a site can disappear or it can be replaced by another one. Additionally, in the general case, no wrappers for the relevant sites are provided. Thus, Web searching has to be combined with querying and wrapping, and finally, information which is extracted in this way from several sources has to be integrated.

In such cases, the structure of the Web or a site has to be considered and exploited to access its information, i.e., the Web structure also carries relevant information. Here, information extraction and integration also concerns the area of *Web querying*: To extract knowledge in this situation, the *Web skeleton* – formed by Web pages and hyperlinks – has to be considered together with both the *page structure* and the *page contents*.

In [LHL$^+$98], we showed that languages supporting deduction and object-orientation are particularly suited in this context. We presented a formal model for querying structure and contents of Web data: *Object-orientation* provides a flexible and rich data model, e.g., for handling partial information. Using *deductive rules*, the process of exploration, wrapping, mediation, and restructuring can be defined in a high-level, declarative programming style.

In this paper, we propose an integrated architecture for Web exploration, wrapping, mediation, and querying, where the application-level information and the relevant fragment of the Web are represented in a unified world model. By providing an integrated language for wrapper and mediator rules and for querying, an impedance mismatch due to different languages for these tasks can be avoided, allowing also for data-driven Web exploration based on previously accessed pages. The combination of using a Web searching/indexing service such as AltaVista and defining own rules provides a powerful mechanism for searching the Web.

Our architecture is implemented in FLORID, [HKL+98], an implementation of the deductive object-oriented database language F-Logic [KLW95] extended with Web access functionality. F-Logic serves as data model, and as Web access, wrapper, mediator, and query language. The graph-like nature of the object-oriented model is exploited for an integrated model of the threefold space of Web structure, page markup, and contents where these three aspects can interfere and enrich each other. F-Logic rules are used for Web exploration, wrapping, mediation, and information restructuring. Generic rule patterns are used for *structured document analysis* based on an SGML parser, and for processing raw data by pattern matching. The power and flexibility of generic rules can be fully exploited by the language which allows variables ranging also over objects and methods, and path expressions for navigating an object-oriented model of both the database and the Web fragment. Programs are modularly designed by extending a skeleton of generic rules with refinements and application-specific rules. The object-oriented model can then be restructured, e.g., "normalized" into a relational database, or into an XML [XML98] model.

We illustrate our approach by two case-studies: The first one illustrates our world model by extracting data about computer science publications from a well-structured Web site [DBL98]. The second one represents a real-life application that extraxts geographical information from several sites [CIA, Glo] which are not optimally structured and contain overlapping information, partially using different names.

Contributions

We present an approach for spontaneous information extraction from the Web. Our main focus is on (a) an appropriate modeling, and (b) a language and methodology for handling this model. We present a world model which comprises a model of the relevant fragment of the Web as the *carrier of information* and an application-level model of the *carried information* itself [May99b]. We complement the world model by a rule-based declarative language and a methodology for mapping data from the structural model to the application level. The F-Logic language contains very flexible constructs for expressing sophisticated rules (syntactically second-order) that allow the combination of meta-reasoning with generic rules). This shows significant advantages especially for mapping the syntactical level to the semantic level. Programs are in general short and concise in their declarativeness.

The approach is not restricted to F-Logic as a formal framework; nevertheless, its attraction depends much on the flexibility and the expressive power of F-Logic rules. To the best of our knowledge, there is currently no other approach that covers Web exploration, wrapping, and data restructuring and integration in an integrated manner.

Outline

This paper is structured as follows: In Section 2 we present an integrated architecture for Web exploration, wrapping, and mediation. Section 3 gives an overview of F-Logic as the formal framework underlying the system, together with its formal Herbrand-style semantics. Then we present in Section 4 the *Web skeleton* concept that or Web model is based on. Section 5 describes the generic Web interface used to generate the internal representation of the Web model and defines its formal semantics. In Section 6, we extend the Web skeleton by the page structure. This completes the structural aspect of the Web model, illustrated by the first case-study. Section 7 describes how the internal model can be extended by data-driven Web exploration along hyperlinks. The semantical part concerning the mapping of the Web model to an application-level model starts with Section 8. Here, F-Logic provides the formal framework for a rule-based approach to *wrapping*. Section 9 gives a methodology to handle frequent markup elements by generic rules, illustrated by the second case-study. Mediation and data integration in our system are described in Section 10. Section 11 adresses the problem of handling HTML forms; we describe how existing Web indexing engines can be exploited in combination with user-defined rules. In Section 12 we sketch the transfer of information to external database systems or applications. Section 13 contains a comparison and overview of related work in the area of Web data extraction and Web querying. Finally, Section 14 concludes the paper.

2 The Architecture

Our approach is based on an integrated architecture for processing information from the Web as shown in Figure 1 [May00]. Such an architecture is implemented in the FLORID system. It borrows from both *Web querying* systems (cf. W3QL [KS95], WebOQL [AM98]) and systems for *data integration* between distributed (heterogeneous) sources.

Web querying systems are based on generic interfaces. In W3QL [KS95] and WebSQL [MMM97], *query engines* or *remote search programs* operate by evaluating SQL-like or Perl-like conditions on Web pages. In contrast, the Web interface of WebOQL [AM98] maps Web documents to an internal parse-tree representation. Queries are then evaluated against this internal data structure.

(Web) data integration systems mostly follow the common layered wrapper-mediator architecture (cf. [Wie92]) as shown in Figure 2 (the TSIMMIS [GMPQ+97] architectural concept). For this architecture, every data source is associated with a *wrapper* which implements an application-specific mapping to an internal data model. Thus, the system comprises multiple interfaces to individual Web sources via individual, independent wrappers connected to *mediators* that distribute queries and collect the results.

Our architecture is based on a single interface to the Web as a whole. This http/ftp-protocol-based Web interface provides *generic* (i.e., source-independent) operations for fetching and parsing information from the Web into the system. Via this interface, the Web can be seen as a part of the system that can be queried on-demand.

In contrast to pure Web querying systems where the internal functionality is restricted to the querying language itself, FLORID contains a comprehensive data model and powerful logic: The system maintains an integrated object-oriented world model consisting of a model of the relevant fragment of the Web as the *carrier of information*, and an application-level model of the *information itself* [May99b]. This world model is handled by a powerful, user-definable internal logic which combines wrapper, mediator, and querying functionality in a single framework. The system provides the following functionality:

- Exploration and Navigation: Since the internal world model contains information about the Web structure, the system can explore the Web autonomously. Relevant urls can be found and loaded into the system based on hyperlinks on already known pages, or by using Web search machines, such as AltaVista.
- Wrapping: wrapping rules operate on the representation of Web pages as strings or parse-trees, generating an application-level model of the information. Generic rules for standard tasks in wrapping, e.g., handling tables, lists, comma-lists, etc., can be defined [MHLL99]. They are complemented by application-specific rules and refinements for handling excep-

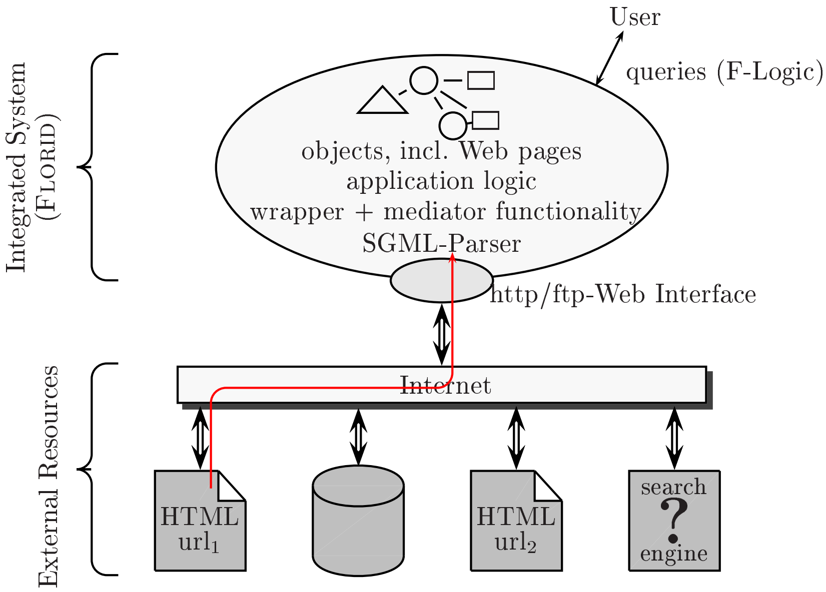


Figure 1: An Integrated Architecture for Web Access

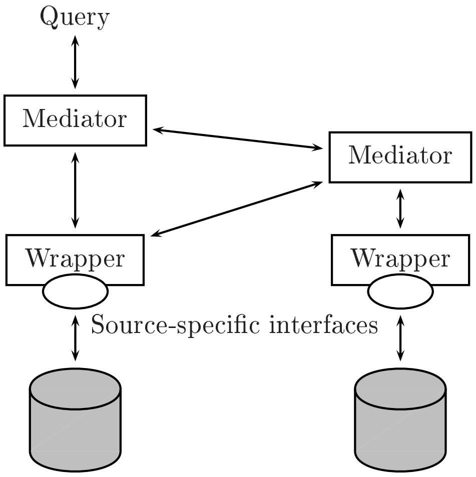


Figure 2: Common Layered Wrapper-Mediator Architecture

tional cases to obtain the required flexibility.

- Integration and Mediation: Using restructuring rules, an integrated application-level model comprising the collected wrapped information from several sources is added to the internal representation of Web pages.

Having navigation rules, access rules, wrapper rules, and mediator rules together, a cross-fertilization by interleaving application of rules between these tasks is possible. This allows especially for data-driven Web exploration based on previously accessed pages as will be shown in Section 7, which can be exploited in a rule-based framework as presented in this work.

In the following sections, we describe the ideas behind the FLORID system [FHK+97][1] as well as its strengths that derive from both the implementation of the architecture just described and the object-oriented deductive language, F-Logic, on which it is also based. Although our implementation is based on the deductive object-oriented paradigm, other implementations (e.g., a "functional" system providing a lisp-like manipulation and querying language using a KIF [Gen91]-like data model) are conceivable, if they provide a Web interface, an internal data model, and abstract functionality as described in Sections 5 and 8.

When dealing with XML sources, the gap between Web structure, syntactic representation, and application-level representation is minimized: the XML Web structure consists of document objects whose hyperlinks directly correspond to XML object references. Also, the markup structure of XML documents directly induces a preliminary object-oriented application-level model.

3 The Formal Framework: F-Logic

By combining the advantages of deductive database languages with the rich modeling capabilities (objects, methods, class hierarchy, inheritance, signatures) of the object-oriented data model, F-Logic [KLW95] provides a suitable framework for handling information in the integrated Web data model. Its main characteristics are:

- rich object-oriented data model for Web structure and application-level model,
- rule-based data manipulation language,
- powerful and flexible language: F-Logic allows variables ranging over objects (including host objects), classes, and methods.

3.1 F-Logic in a Nutshell

The basic syntactic structure is as follows (we omit inheritable methods; for full F-Logic, the reader is referred to [KLW95]). Nested expressions and the formal semantics are presented in Section 3.2.

- The alphabet of an F-Logic language consists of a set $\mathcal{F}$ of *object constructors*, playing the role of function symbols, a set $\mathcal{P}$ of predicate symbols, a set $\mathcal{V}$ of variables, and several auxiliary symbols. By convention, object constructors start with lowercase letters whereas variables start with uppercase ones.
- *id-terms* are composed as terms from object constructors and variables as usual. Id-terms are interpreted by elements of the universe. Ground id-terms play the role of *logical* object identifiers (*oids*).

Let O, C, D, Q_i, S, S_i, SvM, and MvM stand for id-terms.

- An *is-a assertion* is an expression of the form $O : C$ (object O is a member of class C), or $C :: D$ (class C is a subclass of class D).
- A *predicate atom* is an expression of the form $p(Q_1, \ldots, Q_k)$ where $p \in \mathcal{P}$.
- The following are *object atoms*:
  - $O[SvM@(Q_1, \ldots, Q_k) \rightarrow S]$: applying the *single-valued* method SvM with arguments $Q_1, \ldots, Q_k$ to O results in S,
    (for methods without arguments, we write $O[M \rightarrow S]$).
  - $O[MvM@(Q_1, \ldots, Q_k) \twoheadrightarrow \{S_1, \ldots, S_n\}]$: $\{S_1, \ldots, S_n\}$ are possible results when applying the *multi-valued* method MvM with arguments $Q_1, \ldots, Q_k$ to O.

  Similar, *signature atoms* describe the signatures of methods:

[1] available at `http://www.informatik.uni-freiburg.de/~dbis/florid/`.

– $C[M@(C_1,\ldots,C_k)\Rightarrow T]$ (and $C[M@(C_1,\ldots,C_k)\Rrightarrow T]$): applying the method M with arguments of classes $C_1,\ldots,C_k$ to an element of class C results in an element (or elements, respectively) of class T.

- An F-Logic *rule* is a logic rule of the form head $\leftarrow$ body over F-Logic's atoms, i.e., is-a assertions, object atoms, and signature atoms; an F-Logic *program* is a set of rules.

F-Logic supports reasoning about structure and schema by allowing variables at arbitrary positions of atoms, i.e., also at method and class positions (*syntactically higher order*). Such "meta-queries" are unique features of our integration approach.

Example 1 (Publications Database) *In Figure 3, the object-oriented representation (of the application level; later Figures 4 an 7 show the source-level) of the publications database [DBL98] is depicted. The uppermost part shows the schema consisting of the class hierarchy and method signatures (denoted by double-shafted arrows). Below, the instance level is shown. Instance and subclass relationships are denoted by dotted and dashed edges, respectively. Methods are denoted by edges labeled with the corresponding names; following F-Logic's notation, edges are single-headed and double-headed for single-valued and multi-valued methods, respectively. The F-Logic equivalent of the graph is sketched in the lower part of the figure.*

Path Expressions. In addition to the basic F-Logic syntax, the FLORID system also supports *path expressions* [FLU94] in place of id-terms for navigating in the object-oriented model:

- The path expression $O.M@(\ldots)$ is *single-valued* and refers to the unique object S for which $O[M@(\ldots)\rightarrow S]$ holds, whereas $O..M@(\ldots)$ is *multi-valued* and refers to every S_i such that $O[M@(\ldots)\twoheadrightarrow\{S_i\}]$ holds.

Since path expressions and F-Logic atoms may be arbitrarily nested, a concise and extremely flexible specification language for object properties is obtained; its formal semantics is given in Section 3.2.

Example 2 (Path Expressions) *In our example,* $o_{j1}.\mathsf{in\_vol.of} \equiv o_{is}$.
The following query yields all affiliations of VLDB76 editors:

?- P:conf_proc.of[abbrev→"VLDB"], P[year→1976], P..editors[affil@(1976)→A].

Object Creation. Single-valued path expressions in the *head* of rules can create *anonymous objects*. A rule of the form

O.M@(...)[<properties>] ← <body(O,M,...)>

creates an object x such that o[m@(...)→x] and x[<properties>] hold whenever <body(o,m,...)> is satisfied. In our data-driven approach, this feature is used to generate objects representing Web documents. Anonymous objects get an internal identifier; e.g., x is referencable by o.m@(...).

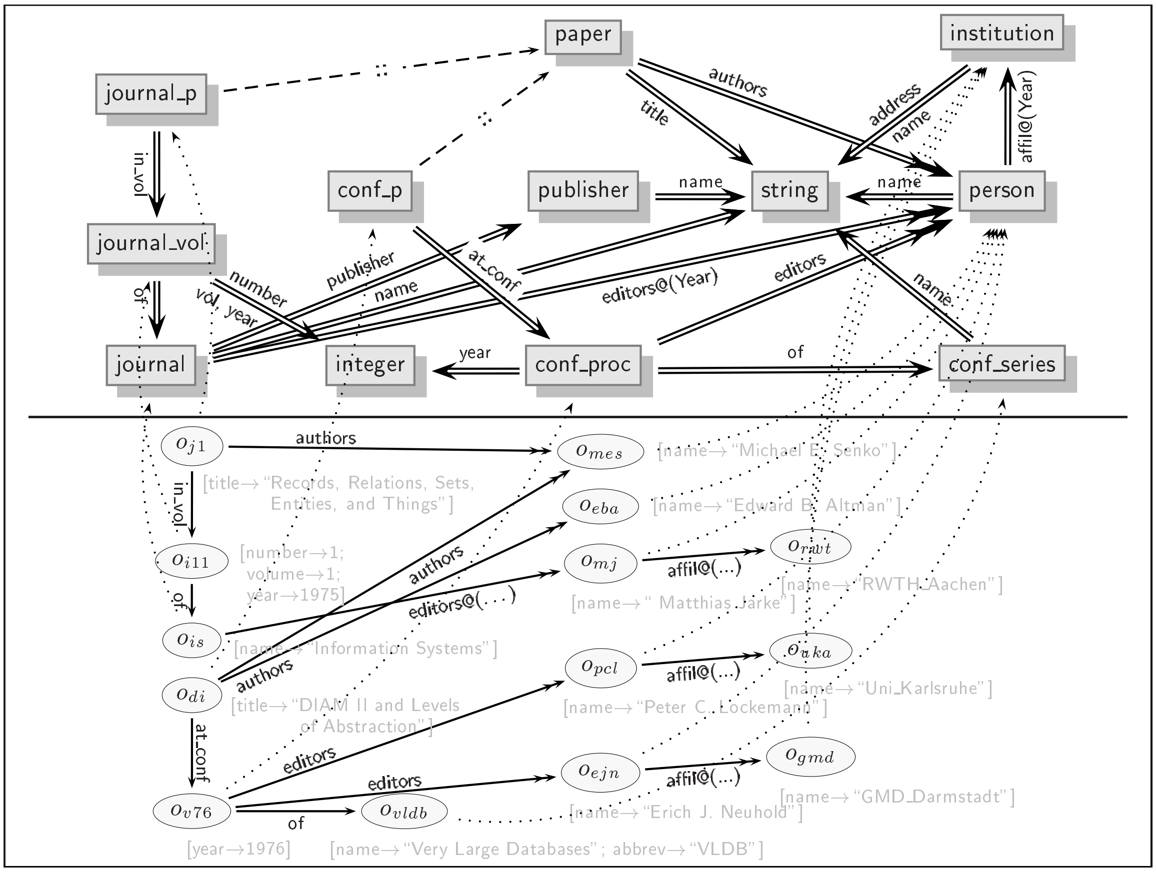
Example 3 *Consider the following situation: When building the database, the publisher objects are created on-demand by*

(J.publisher:publisher)[name→N] ← J:journal[publ_name→N].

Object fusion. Another crucial feature for data integration is the possibility to *fuse* objects which have been generated independently (from different sources), but that denote the same object in the resulting integrated model. F-Logic allows to derive object equivalences ("*object fusion*") by equivalence literals in the head of rules:

$o_1 = o_2$ ← <body(o_1,o_2)>

fuses two objects, i.e., their identifiers (and also all path expressions referencing them) become equivalent and the resulting object exhibits the merged behavior of both original objects (note that this can derive equivalence of other objects, i.e, the above fusion together with o_1[m→v_1] and o_2[m→v_2] would derive $v_1 = v_2$).



journal_p :: paper.   conf_p :: paper.

o_{j1} : journal_p[title→ "Records, Relations, Sets, Entities, and Things"; authors↠{o_{mes}}; in_vol→o_{i11}].

o_{di} : conf_p[title→ "DIAM II and Levels of Abstraction"; authors↠{o_{mes}, o_{eba}}; at_conf→o_{v76}].

o_{i11} : journal_vol[of→o_{is}; number→1; volume→1; year→1975].

o_{is} : journal[name→ "Information Systems"; editors@(...)↠{o_{mj}}].

o_{v76} : conf_proc[of→o_{vldb}; year→1976; editors↠{o_{pcl}, o_{ejn}}].

o_{vldb} : conf_series[name→ "Very Large Databases"].

o_{mes} : person[name→ "Michael E. Senko"]. o_{mj} : person[name→ "Matthias Jarke"; affil@(...)→o_{rwt}].

o_{rwt} : institution[name→ "RWTH_Aachen"].   ...

Figure 3: Publications Object Base in Graph and F-Logic Representation

3.2 Formal Semantics

The semantics of F-Logic extends the semantics of first-order predicate logic. Formulas are interpreted over a semantic structure which gives an interpretation of terms and an evaluation of ground atoms as usual. In contrast to first-order predicate logic, formulas are not only composed by connectives and atoms, but path expressions and F-Logic atoms may be arbitrarily nested: F-Logic atoms can occur at any position where id-terms are allowed in the above syntax definition.

Definition 1 (Pure References) *Expressions which consist only of iterated application of the path constructors "." and ".." to an id-term (e.g., $o_{vldb}.year@(1976)..editors$) are called* pure *references.*

Every ground F-Logic expression has two evaluations:[2]

- a *set of objects* $obj(\_)$, i.e., the objects reachable by the corresponding reference (*object-oriented perspective*), and
- a *truth value* like any other literal or molecule of the language (*deductive perspective*); in particular, a reference r evaluates to *false* if there exists no object referenced by r.

  By assigning a truth value to pure references, they can also appear instead of atoms in rule bodies.

Example 4 (References vs. Truth Values) *o_{vldb} : conf_series is a reference to o_{vldb}, additionally specifying o_{vldb}'s membership to the class conf_series. The truth value of o_{vldb} : conf_series is true.*

o_{vldb} : conf_series[name→ "VLDB"] is again a reference to o_{vldb}, additionally specifying o_{vldb}'s value for name; the truth value of that expression (w.r.t. our database) is true. The (pure) reference o_{vldb}.name evaluates to "VLDB" (as an object) or to true (as a truth value).

Definition 2 (Semantics) *In an F-Logic database $\mathcal{I}$, the object values of ground expressions are given by the following mapping from ground expressions to sets of ground references:*

$$\begin{array}{lcl} obj(t) & := & t \text{ for a ground id-term } t \\ obj(r[spec]) & := & \{o \in obj(r) \mid \mathcal{I} \models o[spec]\} \\ obj(r:c) & := & \{o \in obj(r) \mid \mathcal{I} \models o:c\} \\ obj(c::d) & := & \{c' \in obj(c) \mid \mathcal{I} \models c'::d\} \\ obj(r.m) & := & \{r' \in obj(v) \mid \mathcal{I} \models r[m \rightarrow v]\} \\ obj(r..m) & := & \{r' \in obj(v) \mid \mathcal{I} \models r[m \twoheadrightarrow \{v\}]\} \end{array}$$

Normally, expressions in F-Logic are left-associative; this behavior can be changed by use of parentheses. The $\models$ relation extends from atoms to pure references and nested expressions as follows:

$$\begin{array}{lcl} \mathcal{I} \models r & \Leftrightarrow & obj(r) \neq \emptyset \quad \text{for a ground pure reference } r, \\ \mathcal{I} \models r_1[r_2@(p_1,\ldots,p_n) \rightarrow r_3] & \Leftrightarrow & \mathcal{I} \models o_1[o_2@(q_1,\ldots,q_n) \rightarrow o_3] \text{ for any } o_i \in obj(r_i),\ q_j \in obj(p_j) \\ & & \text{analogously for } \twoheadrightarrow. \\ \mathcal{I} \models r_1 : r_2 & \Leftrightarrow & \mathcal{I} \models o_2 : o_2 \text{ for any } o_1 \in obj(r_1),\ o_2 \in obj(r_2) \\ & & \text{analogously for } :: . \end{array}$$

As an additional abbreviation, the *molecule* $o[m_1 \rightarrow v_1; m_2 \rightarrow v_2]$ stands for the conjunction of $o[m_1 \rightarrow v_1]$ and $o[m_2 \rightarrow v_2]$.

Example 5 (Reference Atoms) *In the rule body (C:conf_series)[name→ "VLDB"], C..editors, C can only be bound to o_{vldb}. Furthermore, o_{vldb}..editors is not defined (since the conference series has no editors). Therefore, the rule body evaluates to false.*
In contrast, in

(C:conf_series)[name→ "VLDB"], C.year@(1976)..editors ,

C can again only be bound to o_{vldb}, C.year@(1976) is bound to $o_{vldb1976}$, and the reference C.year@(1976)..editors has the witnesses P.C.Lockemann *and* E.J.Neuhold *– thus, the rule body evaluates to true with the variable binding $\{C/o_{vldb}\}$.*

Semantics of F-Logic Programs. F-Logic programs are evaluated over Herbrand-style interpretations where the universe consists of ground id-terms. As usual, given the set $\mathcal{F}$ of object constructors, the Herbrand universe U $(=U(\mathcal{F}))$ consists of all ground id-terms over $\mathcal{F}$. In the presence of path expressions, U is not sufficient since single-valued path expressions can

[2] this is similar to LISP: everything evaluating to a non-nil value evaluates to true when interpreted in a boolean context.

create new logical oids when used in rule heads. Thus, single-valued path expressions behave like Skolem function symbols, and U has to be closed w.r.t. application of single-valued path constructors; the resulting closure is denoted by $U^\star$. The extended Herbrand base $\mathcal{HB}$ consists of all ground atoms over $U^\star$, and every subset of $\mathcal{HB}$ induces a Herbrand interpretation.

Definition 3 (H-Universe, H-Base) *Let $\mathcal{F}$ and $\mathcal{P}$ denote the sets of object constructors and predicate symbols, respectively.*

- *The* (extended) H-universe *$U^\star$ (=$U^\star(\mathcal{F})$) is the set of pure references, i.e., the least set $U^\star \supseteq U$ such that:*
  - *if $m, t_0, \ldots, t_n \in U^\star$, then $(t_0.m)$ and $(t_0.m@(t_1, \ldots, t_n))$ are in $U^\star$ for every $n \in \mathbb{N}$.*
- *The* (extended) H-base *$\mathcal{HB}^\star$ (=$\mathcal{HB}^\star(\mathcal{D}, \mathcal{P})$) consists of all ground atoms (i.e., F-Logic atoms and equality atoms of the form $\mathsf{u} = \mathsf{u}'$) and ground pure references over $\mathcal{P}$ and $U^\star$.*

Due to path expressions, the Herbrand base is in general infinite, e.g., all path expressions $adam.\mathsf{wife}.(\mathsf{husband}.\mathsf{wife})^{\mathbb{N}}$ would reference eve; the implementation operates on equivalence classes of objects.

Since $\mathcal{HB}^\star$ contains the set $U^\star$ of pure references, every subset of $\mathcal{HB}^\star$ also assigns a truth value to every $r \in U^\star$. Due to the inherent semantics of the class hierarchy and object identity, we are only interested in interpretations that satisfy the following properties:

Definition 4 (H-structure) *A (possibly infinite) subset $\mathcal{H} \subseteq \mathcal{HB}$ of ground atoms and pure references is an* H-structure*, if the following conditions hold for arbitrary ground pure references u, $u_0, \ldots, u_n$, and u_m:*

- *The* closure axioms *of F-Logic:*
  - *$u :: u \in \mathcal{H}$ (subclass reflexivity),*
  - *if $u_1 :: u_2 \in \mathcal{H}$ and $u_2 :: u_3 \in \mathcal{H}$ then $u_1 :: u_3 \in \mathcal{H}$ (subclass transitivity),*
  - *if $u_1 :: u_2 \in \mathcal{H}$ and $u_2 :: u_1 \in \mathcal{H}$ then $u_1 = u_2 \in \mathcal{H}$ (subclass acyclicity),*
  - *if $u_1 : u_2 \in \mathcal{H}$ and $u_2 :: u_3 \in \mathcal{H}$ then $u_1 : u_3 \in \mathcal{H}$ (instance-subclass dependency),*
  - *if $u[m \to u_1] \in \mathcal{H}$ and $u[m \to u_2] \in \mathcal{H}$, then $u_1 = u_2 \in \mathcal{H}$ (uniqueness of scalar methods).*
  - *reflexivity, transitivity, and symmetry of $=$, substitutability of equal objects (equality axioms).*
- *The* path expression axioms*:*
  - *if $o.m \in \mathcal{H}$ then $o[m \to o.m] \in \mathcal{H}$,*
  - *if $o[m \to r] \in \mathcal{H}$ then $o.m = r \in \mathcal{H}$.*
- *The* active name axiom*:*
  - *for every $r \in \mathcal{H}$ and every $r' \in U^\star$ occurring in r: $r' \in \mathcal{H}$.*

Given a set $\mathcal{H}$ of gound atoms, the operator $\mathcal{C\ell}(\mathcal{H})$ denotes the closure of $\mathcal{H}$ w.r.t. the above axioms.

For an H-structure, the truth of atoms and formulas is given in the usual way [KLW95]. Positive F-Logic programs are evaluated bottom-up by a T_P-like operator including $\mathcal{C\ell}$, providing a minimal model semantics:

Definition 5 (Deductive Fixpoint)
For an F-Logic program P and an H-structure $\mathcal{H}$,

$$
\begin{aligned}
T_P(\mathcal{H}) &:= \mathcal{H} \cup \{h \mid (h \leftarrow b_1, \ldots, b_n) \textit{ is a ground instance of some rule of } P \\
&\qquad\qquad \textit{and } \mathcal{H} \models b_1 \wedge \ldots \wedge b_n\} \ , \\
T_P^0(\mathcal{H}) &:= \mathcal{C\ell}(\mathcal{H}) \ , \\
T_P^{i+1}(\mathcal{H}) &:= \mathcal{C\ell}(T_P(T_P^i(\mathcal{H}))) \ , \\
T_P^\omega(\mathcal{H}) &:= \begin{cases} \lim_{i \to \infty} T_P^i(\mathcal{H}) & \textit{if the sequence } T_P^0(\mathcal{H}), T_P^1(\mathcal{H}), \ldots \textit{ converges,} \\ \perp & \textit{otherwise.} \end{cases}
\end{aligned}
$$

Note that exact knowledge of the formal semantics is not necessary to write programs in F-Logic. In our institute, students have successfully programmed in F-Logic, based “only” on the intuitive ideas of F-Logic expressions and rules.

In the following, F-Logic serves as a framework for building a comprehensive model of a relevant Web fragment and its application-level representation, and for extracting, integrating, and querying data in this model. The FLORID system [FHK+97] provides an implementation of F-Logic extended with Web access functionality as described below.

4 The Web Model

The *topology of the Web* can be represented by a graph (cf. models for semistructured data [Abi97, Bun97]). Every resource in the Web has a unique address, its *url*. Typically, the document associated with a url contains *hyperlinks* to other url's that refer to further Web documents. The nodes correspond to Web documents, and the (labeled) edges correspond to hyperlinks between documents. In this stage, nodes are purely syntactic, i.e., no information apart from the link structure is available; we call this graph the *(Web) skeleton* $\mathcal{SK}$:

$$(x,\ell,y) \in \mathcal{SK} \Leftrightarrow \text{ Web document } x \text{ contains a hyperlink labeled with } \ell \text{ to Web document } y.$$

Example 6 (DBLP Skeleton) *The labeled graph depicted in Figure 4 represents a fragment of the skeleton of the DBLP server [DBL98].*

The *Web skeleton* covers the *structural* aspect of the Web but not its *contents*. Although, the example shows that – with a suitable structure of the Web site – this skeleton corresponds to real world objects (in our example, journals, conferences, books, and authors) and relationships between them.

In our object-oriented world model which will incorporate both a representation of the relevant Web fragment and an application-level model, we represent the Web skeleton by the concepts of url and webdoc [LHL+98]: Every member u of class url provides a special method $u.\mathsf{get}$ that yields the associated Web document $wd : \mathsf{webdoc}$. Hyperlinks are modeled by the parameterized multi-valued method $\mathsf{hrefs@}(\ell)$, where ℓ is a string. $\mathsf{hrefs@}(\ell)$ yields instances of class url:

$$\begin{array}{lcl} u.\mathsf{get}[\mathsf{hrefs@}(\ell) \twoheadrightarrow u'] & \Leftrightarrow & \text{the Web document } u.\mathsf{get} \text{ contains a reference} \\ & & \text{labeled with } \ell \text{ to the url } u'. \\ & \Leftrightarrow & (u,\ell,u') \in \mathcal{SK} \end{array}$$

The description of the model is continued in a finer granularity in Section 6. First, we describe how the Web model is realized in our system.

5 The Web Interface

The Web interface generates the F-Logic representation of the Web skeleton, i.e., it maps the structural information of the relevant fragment of the Web into the unified world model without semantically interpreting its contents. The concepts url and webdoc are implemented as subclasses of string. The above-mentioned method url.get is implemented as an *active* generic method which is based on Internet access using the http/ftp protocols. Calling $u.\mathsf{get}$ makes the associated Web document $(u.\mathsf{get}) : \mathsf{webdoc}$ a part in the world model of the system. By evaluating rules of the form

u.get :- <body> ,

the internal database is extended by a new Web document:

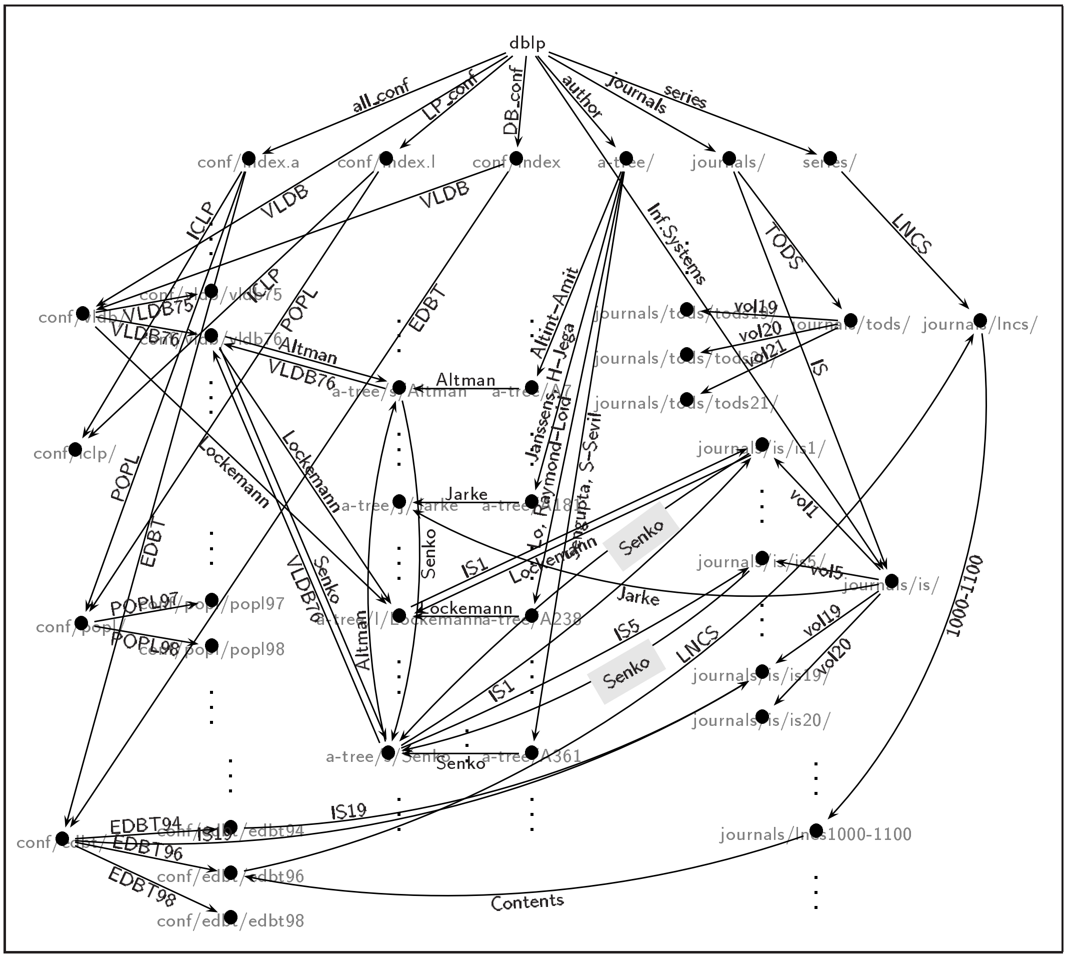


Figure 4: Fragment of the Skeleton of the DBLP Database

- the Web document wd which is accessible via the url u is accessed,
- wd is assigned to the newly created object u.get (conceived as a large string),
- wd is made an instance of class webdoc, and
- several properties are automatically filled in (cf. Figure 5):
  - References to other urls are stored in the built-in attribute hrefs:

    u.get[hrefs@(ℓ) $\twoheadrightarrow$ u'] $\Leftrightarrow$ u.get contains " ℓ " .

  - If an error occurs when a document is accessed via its url, u.get[error $\twoheadrightarrow$ <error_msg>] holds the resulting error message.

By formulating appropriate rules of the form

U'.get :- U.get[hrefs@(L)↠U'], <filter(L,U,U')> ,

where <filter> specifies a *filter* on urls to be followed, the system can autonomously explore the Web (cf. Section 7) and generate the relevant fragment of the Web skeleton. Using u.get..error, fault-tolerant exploration rules can be specified, e.g., accessing alternative data sources. The user can then navigate through the Web skeleton via *path expressions*.

```
url::string[get⇒webdoc].                          % signature of url and
webdoc[url ⇒url; author ⇒string;                  % ... webdoc
       type ⇒string; hrefs@(string) ⇒⇒url ;
       modif ⇒string; error ⇒⇒string].
--------------------------------------------------------------------------
wd1:webdoc[url→"url1"; hrefs@("label")→→{"url2"}; ...]
wd2:webdoc[url→"url2"; type→"html"; ...]
```

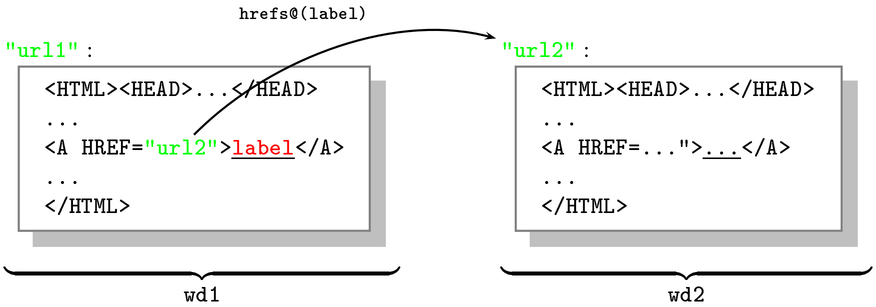


Figure 5: F-Logic Web model: signature and example data

By means of the *Web interface method* u.get, *raw* (i.e., uninterpreted) data from the Web is loaded into the system. This procedure extracts only the very basic properties of Web documents, without further exploiting the document type or structure. At the generic level, the schema can be refined by declaring subclasses of webdoc, e.g., for BibTeX, HTML, XML, etc. In the current implementation, Florid contains a built-in method for handling SGML documents described in the remainder of this section. The contents of the retrieved Web pages has to be analyzed by *wrappers*, using the internal structure of HTML or XML Web documents.

Generic pre-processing of SGML Documents. The active method url.parse generates the F-Logic representation of the parse-tree of an SGML document (in order to do this, Florid uses the SGML-parser *nsgmls* [Cla]). Although parsing already happens inside the system, we consider url.parse as part of the Web interface, since it provides an application-independent functionality tightly coupled with Web-specific notions. When calling u.parse for an url u, the parse-tree is linked into the Web skeleton in the following way:

- the parse-tree is transformed into an object-oriented representation and made an object u.parse : parsetree;
- for every Web document wd, every SGML-tagged group <*tag*> ... </*tag*> is made an object. The class wd.*tag* contains all such objects <*tag*> ... </*tag*> on wd, e.g., x : (wd.table) holds for all tables x on wd;
- each tag induces a method for navigation in a parse-tree: Let x : (wd.*tag*) be a node of the parse-tree, then x.*tag*@(0), ..., x.*tag*@(n) address the distinguished segments inside x, e.g.,
- for suitable $k = 0, 1, 2, \ldots$, wd.html@(k) addresses all distinguished segments of wd between <html> and </html>, e.g., wd.html@(0) is the head, providing wd.html@(0).head@(i) ($i = 1, 2, \ldots$) and wd.html@(1) is the body, providing wd.html@(1).body@(j) ($j = 1, 2, \ldots$).

Example 7 (Tables) *Tables are represented in HTML by the tags* <table>, <tr> *(table row),* <td> *(column elements containing data), and* <th> *(column elements containing header entries), e.g.,*

```
<table>
```

```
  <tr><th>Name</th><th>Birthday</th><th>Affiliation 1998</th></tr>
  <tr><td>D.E.Knuth</td>  <td>10-01-1938</td><td>Stanford Univ.</td></tr>
  <tr><td>...</td>  <td>...</td>  <td>...</td></tr>
  ...
</table>
```

- *all tables whose header contains '1998' in any header row/column are identified by*

  $\mathsf{T}:(wd.\mathsf{table}),\ \mathsf{T.table@(R).tr@(C)[th@(0)}\rightarrow\mathsf{S]},\ \mathsf{substr(S,\text{“1998”})}.$

- *the third column of the 17th row of a given table tab is addressed by tab.table@(17).tr@(3).*

Example 8 (Parsetree in F-Logic representation) *Figure 7 shows the parsetree of the* Information Systems, Vol. 1 *page as given in Figure 6 (the parsetree is integrated into the Web skeleton in Figure 8).*

```
<html>
<head>
<title>Information Systems, Volume 1 </title>
</head>
<body>
<h1>
  <a href="index.html"> Information Systems </a>, Volume 1, 1975
</h1>
<h2> Volume 1, Number 1 </h2>
<ul>
<li><a href="../../indices/a-tree/s/Senko:Michael_E=.html">
      Michael E. Senko</a>:
    Information Systems Records, Relations, Sets, Entities, and
    Things. 3-13
<li><a href="../../indices/a-tree/g/Ghosh:Sakti_P=.html">
      Sakti P. Ghosh</a>,
    <a href="../../indices/a-tree/l/Lum:Vincent_Y=.html">
      Vincent Y. Lum</a>:
    Analysis of Collisions when Hashing by Division. 15-22
<li><a href="../../indices/a-tree/s/Schneider:G=_Michael.html">
      G. Michael Schneider</a>,
    <a  href="../../indices/a-tree/d/Desautels:Edouard_J=.html">
      Edouard J. Desautels</a>:
    Creation of a File Translation Language for Networks. 23-31
</ul>
<h2>Volume 1, Number 2 </h2>
<ul>
<li> [...]
<li> [...]
<li> [...]
</ul>
</body>
</html>
```

Figure 6: Sourcecode of the DBLP *Information Systems, Vol. 1* page

Note the difference between tag instances, tag classes, tag attributes, *and* tag contents*:*

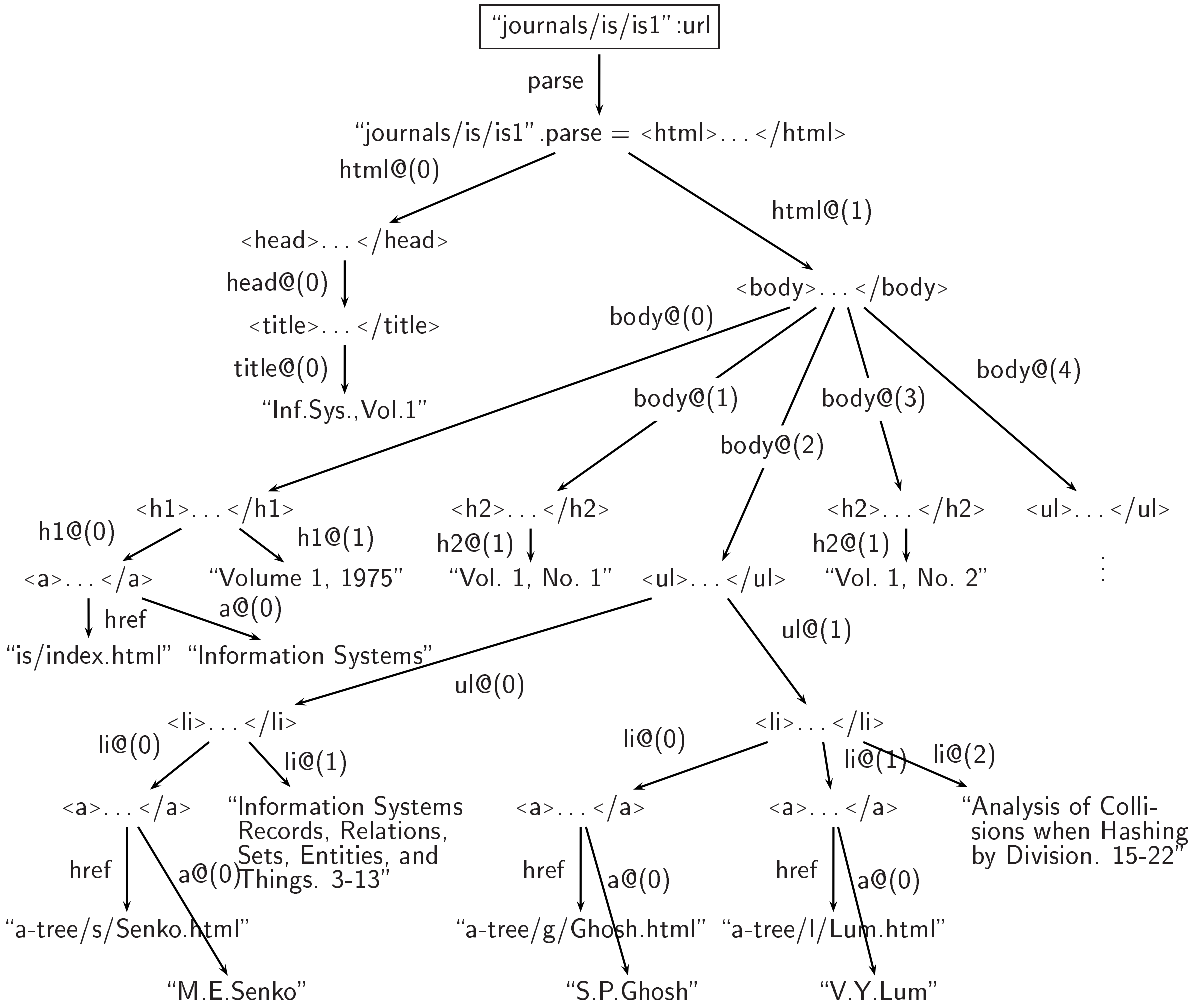


Figure 7: F-Logic representation of the parse-tree of the DBLP *Information Systems, Vol. 1* page

element	class	attribute	contents
"journals/is/is1".parse	html	–	html@(0),html@(1)
"journals/is/is1".parse.html@(1)	body	–	body@(0),. . . ,body@(4)
"...".parse.html@(1).body@(0)	h1	–	h1@(0)
"...".parse.html@(1).body@(0).h1@(0)	a	href	a@(0)
"...".parse.html@(1).body@(0).h1@(0).href	url	–	"is/index.html"
"...".parse.html@(1).body@(0).h1@(0).a@(0)	string	–	"Information Systems"
"...".parse.html@(1).body@(2)	ul	–	ul@(1),. . .
"...".parse.html@(1).body@(2).ul@(0)	li	–	li@(0),li@(1)
"...".parse.html@(1).body@(2).ul@(0).li@(0)	a	href	a@(0)
"...".parse.html@(1).body@(2).ul@(0).li@(0).href	url	–	"a-tree/s/Senko.html"
"...".parse.html@(1).body@(2).ul@(0).li@(0).a@(0)	string	–	"M.E. Senko"

For XML documents, u.parse generates the parse-tree of the XML data source. A generic F-Logic wrapper can be given for mapping the parse-tree directly to an application-level model using the notions given in the DTD.

Formal Semantics of Web Access. F-Logic also provides a formal semantics for its Web access methods that seamlessly extends the original formal semantics. The built-in methods $u.\mathsf{get}$ and $u.\mathsf{parse}$ induce mappings $\mathsf{explore}$ and parse, respectively, from urls to sets of ground atoms (i.e., subsets of the Herbrand base $\mathcal{HB}$):

$$\begin{aligned}\mathsf{explore}: \ & \mathcal{URL} \to 2^{\mathcal{HB}}\\ \mathsf{parse}: \ & \mathcal{URL} \to 2^{\mathcal{HB}}\end{aligned}$$

A Herbrand model $\mathcal{H}$ of an F-Logic program P is a model of P w.r.t. Web access if[3]

- if $\mathcal{H} \models u : \mathsf{url} \wedge u.\mathsf{get}$, then $\mathsf{explore}(u) \subseteq \mathcal{H}$,
- if $\mathcal{H} \models u : \mathsf{url} \wedge u.\mathsf{parse}$, then $\mathsf{parse}(u) \subseteq \mathcal{H}$.

The mappings are integrated into the T_P-operator. For an F-Logic program P and an H-interpretation $\mathcal{H}$,

$$\begin{aligned}T_P(\mathcal{H}) \quad &:= \mathcal{H} \cup \{h \mid (h \leftarrow b_1, \ldots, b_n) \text{ is a ground instance of some rule of } P\\ &\qquad\qquad\qquad \text{and } b_i \in \mathcal{H} \text{ for all } i = 1, \ldots, n\} \ ,\\ T_P^{\mathcal{W},0}(\mathcal{H}) \quad &:= \mathcal{H} \ ,\\ T_P^{\mathcal{W},i+1}(\mathcal{H}) &:= \mathcal{C}\ell(\, T_P(T_P^{\mathcal{W},i}(\mathcal{H}))\\ &\qquad\qquad \cup \bigcup\{\mathsf{explore}(u) \mid T_P(T_P^{\mathcal{W},i}(\mathcal{H})) \models u\!:\!\mathsf{url} \wedge u.\mathsf{get}\}\\ &\qquad\qquad \cup \bigcup\{\mathsf{parse}(u) \mid T_P(T_P^{\mathcal{W},i}(\mathcal{H})) \models u\!:\!\mathsf{url} \wedge u.\mathsf{parse}\}) \quad .\end{aligned}$$

Therefore, for F-Logic programs with Web access, $T_P^{\mathcal{W},\omega}$ is used.

6 The Extended Web Skeleton

When designing a Web site, some real-world objects are represented as individual Web pages, and some are represented by substructures, e.g., lists or tables. By integrating the *parse-trees* of suitably structured Web sites into the Web skeleton, additional objects can be identified. In our framework, the hyperlinks together with the parse-trees form the *extended Web skeleton* [May99b] of the relevant fragment of the Web, representing the information contained in the data sources.

Example 9 (DBLP Server: Extended Web Skeleton) *Figure 8 shows an excerpt of the skeleton from Figure 4 augmented by parse-trees. Dashed lines denote the links from the skeleton, full lines denote the links in the parse-tree representation, and zigzag links denote the hyperlinks originating from parse-tree nodes. Boxed nodes correspond to real-world objects.*

Identifying Real Objects in the Extended Skeleton. The extended Web skeleton is a graph which represents the relevant fragment of the Web in a unified form, but still only based on the syntax and structure of the source. Note that this changes with XML: for well-written XML sources, the parse-tree is already an application-level model. Regarding the object-oriented model of the *application domain*, most of the objects already have a direct counterpart in the extended Web skeleton (in our example, in Figure 8 these nodes are represented as boxes):

- objects are represented as a Web page (here: conferences, journals, and authors). In this case, every reference to this object in the Web is manifested in a reference to the Web page.
- objects correspond to nodes in a parse-tree (here: journal volumes and papers). Here, the object is only referencable in HTML via page#anchor. Often, these objects have counterparts in several parsetrees (e.g., the *Information Systems 1(1)* paper by Senko is represented as a node on the author page of Senko, and on the IS1 page).

[3] recall that the pure reference $u.\mathsf{get}$ evaluates to true if $u[\mathsf{get}{\to}\mathsf{D}]$ holds for some D. Then, D is the Web document accessible at the url u.

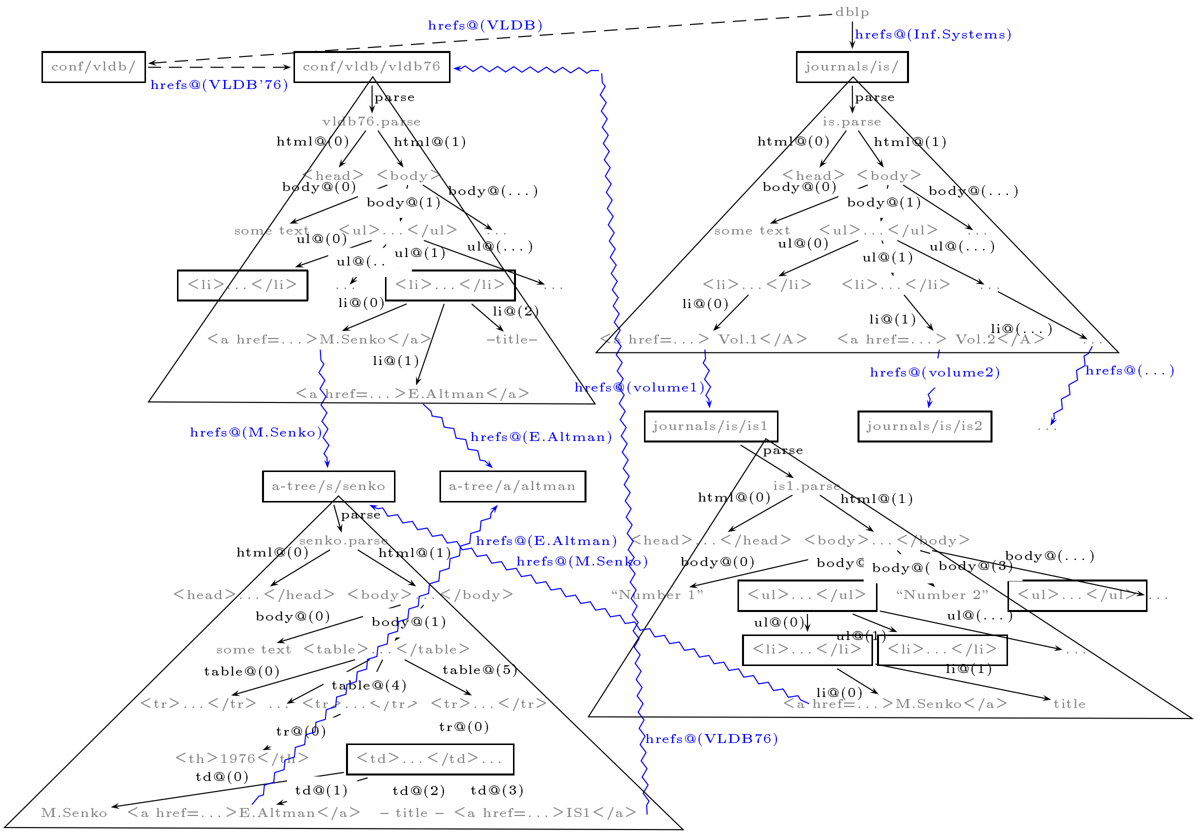


Figure 8: DBLP Database: Skeleton + Parse-trees

Thus, to generate an application-level model, the objects have to be identified, and their attributes have to be retrieved and transformed. Often, a mapping between nodes/arcs of the extended Web skeleton and instances of distinguished classes/relationships of the application model can be given.

Example 10 (DBLP Server: Objects) *From the extended Web skeleton (cf. Figure 8), the members of the application-level classes can be derived. We assume that the F-Logic representation of the extended Web skeleton of the DBLP server is already computed (for its data-driven generation, see Section 7). For conference series, conferences, and journals, only the Web skeleton is used. Note that we cannot yet derive all properties of objects (e.g., titles of papers).*

```
% conference series (e.g., VLDB)
confs(C) : conf_series[name→C;url→U] ←
        dblp.url.get..hrefs@("Conferences").get[hrefs@(C)↠U], substr("/db/conf/",U).
% conference instances (e.g., VLDB76)
conf(U) : conf[series→S; url→U] ← (S : conf_series).url.get[hrefs@("Contents")↠U].
% journals
jrn(J) : journal[name→J; url→U] ←
        dblp.url.get..hrefs@("Journals").get[hrefs@(J)↠U], substr("/db/journals/",U).
```

For journal volumes, the parse-trees (i.e., the extended Web skeleton) are used: every volume is represented in an *tag of the journal's page, containing a hyperlink to the volume's page* *on the server; e.g.,*

Volume 28, 1999 .

```
% journal volumes
vol(J,S):journal_vol[journal→J; url→U] ←
        J:journal, V:J.url.parse.li, V.li@(0)[a@(0)→S; href→U], substr("Volume",S).
```

Every number in a journal volume is a *, the preceding* <h2> *tag indicates the number:*

```
% journal numbers
num(V,N):journal_num[in_vol→V; parsenode→UL; header→H] ←
        V:journal_vol, V.get.parse.html[body@(N)→(H:h2)],
        V.parse.html[body@(N+1)→(UL:ul)].
```

Every paper on a conference or journal volume page is represented in an *tag of the form*

author$_1$, ..., author$_2$: title. pages.

```
% conference papers
paper(P):conf_paper[conf→C; parsenode↠P] ← C:conf, P:C.url.parse.li.
% journal papers
paper(P):journal_paper[num→N; parsenode↠P] ← N:journal_num, N.parsenode[ul@(X)→P].
% papers
conf_paper :: paper.
journal_paper :: paper.
```

The is-author*-relationship between papers and authors is represented by the* <a>*-nodes inside the paper's node, that then contain the author's name and the author's page on the DBLP server:*

```
% authors
author(Name):author[name→Name; url↠U], P[authors↠author(Name)] ←
            (P:paper).parsenode.li@(_)[a@(0)→Name; href→U], substr("a-tree",U).
A[homepage→Url] ← A:author.url.get[hrefs@("homepage")↠Url].
```

The above objects form the instance level *of our example as depicted in Figure 3. The properties of the individual objects are extracted by application-specific wrapper rules (cf. Example 15).*

Thus, the world model provides a unified view of the Web as the data source, and an application-level representation of the information:

- the *topology of the Web* is represented by a graph where the nodes correspond to Web documents, and the (labeled) edges correspond to hyperlinks between documents,
- the *page markup* induces a tree structure to represent the page contents,
- an application-specific interpretation of the extended Web skeleton provides an object-oriented model of the application domain.

The detailed properties (in the above example, e.g., the volume numbers, years, page numbers) of the objects are then extracted by *wrapper* rules that further analyze the contents of the Web pages and the parse tree (cf. Section 8).

In general, the (extended) Web skeleton is generated by accessing the Web via the Web interface (cf. Section 7), starting with one or more initial pages (e.g., the DBLP root page). By analyzing documents, additional urls become known that can be further accessed. In the following section, we describe our data-driven approach to Web exploration. By wrapping and mediating, from the (still uninterpreted) extended Web skeleton, an application-level model is derived and superposed on top of the skeleton.

7 Data-Driven Web Exploration

Systems for integrating data from the Web can be classified based on how data access is controlled:

Warehouse approach: information is first fetched into a local repository to be queried later. In this case, all potential data sources must be known a priori. Here, wrappers have a one-way functionality: mapping Web sources to the common data model. Then, mediators compute the answer to the user's queries.

Virtual approach: Web documents are accessed on-demand, based on a user query. Wrappers transmit queries from the mediator(s) to the underlying sources and receive the answers which are then evaluated by mediator(s). Mediators know about the available data sources and wrappers (using a CORBA-like model, it would be possible to change the set of data sources dynamically).

The above classification is primarily concerned with layered architectures containing separate wrappers and mediators. The modeling of the topology of the Web (in the mediators knowledge) is separated from its contents (represented by the wrapper or an integrated domain data model). Thus, for Web exploration and for identifying appropriate data sources, mediator-side algorithms have to be used (based on explicit source capability descriptions). As a consequence, only data-sources that provide (i) a wrapper and (ii) a suitable source capability description can be used.

Additionally, there are *Web querying* systems that are not restricted to predefined wrappers. There, syntactical and structural queries against the Web are supported (e.g., UnQL [BDHS96], W3QS/W3QL [KS95], and WebOQL [AM98]), although the *semantics* of the contents of the Web pages is not considered.

With our architecture and integrated world model, a hybrid concept between the warehouse and the virtual approach, also offering the flexibility of Web querying systems, is supported. Web pages that become known (by means of a hyperlink referencing their url) can be added on-demand to the internal world model via the Web interface methods. They are then mapped to the internal representation in the extended Web skeleton. By providing suitable generic wrapper rules, arbitrary Web sources can be evaluated. Since Web access itself is defined by rules (containing u.get in the head), a rule- and data-driven Web exploration and navigation is supported. The internal database is generated by iteratively accessing Web documents, analyzing them, and then following the relevant hyperlinks as illustrated in Example 11. Extracted data can be reorganized in this loop or in a separate subsequent step. Thus, the extended Web skeleton can be generated automatically, solely by the Web interface methods url.get and url.parse via filtering rules controlling the exploration process. Relevant hyperlinks can be found on already known pages, or by using Web search engines, such as AltaVista. Therefore, on top of the classical wrapping and mediating functionality, a system using this architecture can be equipped with *Web navigation* and *exploration* functionality.

Example 11 (DBLP Server: Data-driven exploration) *The data-driven exploration of the DBLP server starting with*

dblp[url→ "http://www.informatik.uni-trier.de/~ley/db/"]

is depicted in Figure 9:

1. *starting with the root url, the corresponding Web page* dblp.url.get *is accessed and added to the database.*
2. *From there, the journals page and the conferences page are accessed.*
3. *The pages of conference series and journals, e.g.,* VLDB *and* Information Systems *are accessed.*
4. *The pages of individual conference events and journal volumes, e.g., the* VLDB'76, IS1, *and* IS5 *pages are accessed.*

5. *The individual author pages, amongst them* M.E. Senko*'s and* E. Altman*'s page, are accessed.*

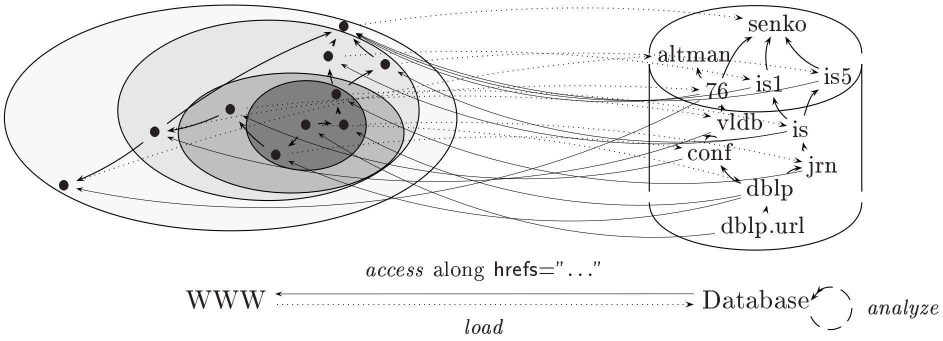


Figure 9: Data-driven exploration of the DBLP Server

The set of pages to be accessed in each step is declaratively specified by the program (cf. Example 12). At runtime, the urls of the pages to be further accessed are extracted from the previously processed data. In our example, if there are any conferences, conference events, or authors added to the DBLP server, the next run of the (unchanged) program will also process these pages. The set of urls to be processed can be restricted using filtering rules of the form

```
u.get :- u:url, <u qualifies as a relevant url> .
```

Example 12 (Exploration of the DBLP server) *In our running example, the DBLP server is explored autonomously with filtering rules. These rules extend the ones given in Example 10 (where the extended Web skeleton was assumed to be given). For a rule-driven generation of the extended Web skeleton, all rules where an object with an own url is identified are extended with the Web access literal u.*get *in the head:*

```
dblp.url.get.
dblp.url.get..hrefs@("Conferences").get.
dblp.url.get..hrefs@("Journals").get.
confs(C) : conf_series[name→C; url→U], U.get ←
        dblp.url.get..hrefs@("Conferences").get[hrefs@(C)↠U], substr("/db/conf/",U).
conf(U) : conf[series→S; url→U], U.get ← (S : conf_series).url.get[hrefs@("Contents")↠U].
jrn(J) : journal[name→J; url→U], U.get ←
        dblp.url.get..hrefs@("Journals").get[hrefs@(J)↠U], substr("/db/journals/",U).
vol(J,S) : journal_vol[journal→J; url→U; header→S], U.get ←
        J : journal, V : J.url.parse.li, V.li@(0)[a@(0)→S; href→U], substr("Volume",S).
author(Name) : author[name→Name; url↠U], U.get ←
                (P : paper).parsenode.li@(_)[a@(0)→Name; href→U], substr("a-tree",U).
```

This program shows how data extraction and Web exploration are combined to allow for data-driven Web exploration.

In the above example, the *structure* of the sites/pages to be visited is already known. Only the actual *set of pages* to be accessed is still unknown. At runtime, *all* pages satisfying a given filter criterion are visited.

Often, semantically related pages are not on a single server and do not reveal a common structure – e.g., homepages of companies or persons. When such pages are accessed with a common goal, e.g., extracting addresses or phone numbers, this can still be possible by means of heuristics. In such cases, Web exploration can be even more autonomous, in the sense that the user did not visit or analyze the evaluated pages before.

Example 13 (Exploration of unknown areas of the Web) *Suppose we are interested in the affiliations, addresses, mail-addresses, photos etc. of all DBLP authors. This information are available from their homepages. These homepages are not part of the DBLP server, and do not share a common structure. Additionally, the address usually not available explicitly on the personal homepage, but on the homepage of the institution the author is affiliated with.*
The url of an author's homepage is given on the DBLP server under the "Homepage" hyperlink:

```
A[homepage→U] ← A:author.url.get[hrefs@("Homepage")→>U].
U.get, U.parse :- A:author[homepage→U].
```

Rules can be given to implement heuristics for the extraction of the address from the homepage, e.g., using the HTML <address> *tag. Similarly,* mailto*-links can be used to extract mail addresses (for the details of the* pmatch *predicate see Section 8):*

```
A[addr→Addr] :- A:author, Addr:(A.homepage.get.addr).
A[email→E]   :- A:author, MailTo:(A.homepage.parse.a),
                MailTo[href@(0)→S], pmatch(S,"/mailto:(.*)/", "$1", E).
```

With our integrated world model and data-driven exploration, also an approach similar to human Web browsing is possible. In human Web "searching", the user usually does not know a priori which sites to visit. Instead, he starts with some overview pages, or with a search engine that provides hyperlinks to documents containing the required information. With a suitable filtering of relevant links, such a strategy can also be implemented in a rule-based Web information extraction system. In order to do this, it is crucial that documents can be selected and accessed at runtime, and that the system is able to process documents whose structure is not known before.

By combining rule-based Web exploration and data extraction, the architecture also allows for the implementation of Web search engines. In contrast to the current index-based search engines (e.g., AltaVista) which allow only simple queries, such a system can extract and process semantical information which can be used to perform semantics-based searching of the Web. To the best of our knowledge, Florid is the only system aware of the Web structure itself in its world model.

The DBLP example represents the best-case for Web data extraction: the source is well-structured by HTML tags and there are no inconsistencies or exceptions. Therefore, we chose it as a canonical example to illustrate our unified world model.

However, reality often looks different: Web pages are in general not well-structured. Wrapping such pages requires powerful mechanisms to deal with differently structured pages. Often, information is spread over several sites, with overlapping and, possibly, inconsistent contents. Especially for wrapping Web pages which are not known a priori (i.e., it is not possible to define a wrapper for them at program design time), the power of the generic "universal" wrapper rules is crucial.

8 Wrapper Functionality

The Florid system is uniformly designed using an object-oriented framework: information is represented in an F-Logic world model; Web exploration, wrapping, mediation, and querying is done by F-Logic rules. Web exploration and access have been dealt with in Section 5.

The gap between the "syntactic" representation in the extended Web skeleton and the application-level representation of its contents has to be bridged by *wrappers*. In the above DBLP example, the wrapping task consisted mainly of identifying objects and properties in the extended Web skeleton. The DBLP server was a best-case example, where information is very well structured and tagged in HTML. In general, Web documents are not so well-structured. To extract information from such Web pages, additional *wrapper* functionality is required.

Depending on the structure of Web pages, *parser*-based or *matching*-based rules are employed; generic rules can be used for standard tasks.

Logical Markup (tagged data): For logical markup such as HTML-lists or tables which is subject to a well-specified grammar, wrapping is based on the *parse-tree.* Generic patterns can be used to decompose lists into their items, and to analyze tables by identifying column headers and table entries. The contents interpretation is then done by application-specific rules. The usage of parse-trees is illustrated in Sections 9.2 and 9.3.

Optical and Syntactical Markup (untagged data): If the data is only partially tagged (paragraphs, line breaks, fonts, commas), the wrapper is based on pattern matching techniques. FLORID provides pattern matching via Perl regular expressions by the built-in predicate

```
pmatch(<string>,"/<regexp>/", "[<fmt-list>]", "[X1,..., Xn]")
```

which extracts substrings by Perl's regular expressions into Perl-variables and binds them to F-Logic variables $X_1, \ldots, X_n$ as specified by <fmt-list> (cf. Example 14).

Additionally, pmatch can be used to search for relevant words in Web documents.

Generic patterns are used, e.g., to decompose "pseudo-lists" which are structured via emphasizing or boldfacing, for comma-lists, and for name-value pairs (cf. Sec. 9.1 and 9.2).

Often, logical markup and textual formatting are mixed, leading to combined generic rules (cf. Section 9.2).

Example 14 (Wrapping by regular expressions) *Assume that the author list of a paper is given only with textual markup as*

$auth_1$, $auth_2$, ... , and $auth_k$: $title$. number n in
Volume v of $series$, pages p_1–p_2, $year$.

Then, the following predicate assigns the relevant substrings to the corresponding variables:

```
pmatch(STRING,
        "/\A ([^:]*): (.*)\.\s
          Number ([0-9]*) in Volume ([0-9]*) of ([a-Z]*), pages ([0-9-]*),([0-9]*)/",
        "[$1,$2,"$4($3)", $5, $6, $7]",
        "[AuthList, Title, Num, Series, Pages, Year]")
```

Here, AuthList *is a comma-list which is further decomposed by application-independent rules (see the subsequent section).* Num *is of the form* volume(number)*.*

Example 15 (DBLP Server: Wrapping) *In Example 10, the objects represented in the DBLP server have been identified in the extended Web skeleton. Now, by* wrapping*, their properties can be completely extracted:*

```
% conference series (e.g., VLDB): done
% conference instances (e.g., VLDB76)
C[year→Y] ← C:conf[url→U].series[name→N],
        pmatch(U,"/([A-z]*)([0-9]*).html/", "[1,2]", "[N,Y]").
% journals: done
% journal volumes
JV[vol→V, year→Y] ← JV:journal_vol[url→U; header→S],
        pmatch(S,"/Volume ([0-9]*), ([0-9]*)/", "[1,2]", "[V,Y])".
% journal numbers
N[num→N] ← N:journal_num[header→H].in_vol[vol→V],
        pmatch(H,"/Volume ([0-9]*), Number ([0-9]*)/", "[1,2]", "[V,N]").
% papers (authors already done): titles
P[title→T; pages→PP] ← (P:paper)..parsenode[li@(X)→TP], string(TP),
        pmatch(TP,"(.*)([0-9]*-[0-9]*—[0-9]*)/", "[1,2]", "[T,PP]").
```

In our example, the application-level model as depicted in Figure 3 is obtained by interpreting the extended Web skeleton by the above application-specific rules.

Additionally, arbitrary predicates (e.g., for string processing) can be defined via FLORID's *Perl* interface: the predicate

perl(<perl-routine-as-string>, X,Y)

binds Y to the result(s) of calling <perl-routine-as-string> with X as argument. Arbitrary procedural algorithms can be implemented in an efficient way. Nevertheless, the most frequent use of the *Perl* interface is probably string reformatting (e.g., formatting date strings, cf. the subsequent section).

In our approach, the construction of wrappers and mediators is based on *generic rule schemata* [MHLL99] for standard tasks which can be complemented by application-specific rules and refinements for handling exceptional cases to obtain the required flexibility. Generic rules are, e.g., given for handling HTML structures such as tables and lists, and syntactical markup such as comma-lists and name-value-pairs. We illustrate the usage of generic wrapping patterns in the subsequent section.

9 Wrapping by Generic Patterns

For the wrapping task, depending on the structure of Web pages, *parser*-based, *matching*-based, or *combined* wrapper rules are preferable. In contrast to wrapping, the mediation and integration task is usually dominated by application-specific rules. However, there are still situations where generic rule patterns apply: objects originating from different sources have to be merged, e.g., identifying the cities of a geographical database with the stations of a train schedule. Similarly, entities can be represented by string-valued attributes in one source, and by objects in another. It is then the task of the integration algorithm to derive a link to the corresponding object.

In general, generic rule patterns are not sufficient to wrap and mediate information from Web pages – due to dissimilarities in the representation and the complexity of the application-inherent semantics. The "skeleton" of the program is assembled using generic rule schemata for standard tasks. Then, in a rapid-prototyping process, it is completed manually [MHLL99]:

- Structural and syntactical dissimilarities are handled by refinements of generic rule patterns which can be tailored to exceptional cases,
- application-specific rules interpret the outcome of the generic patterns in a model of the application domain.
- The rule-based approach allows for modular design of wrappers and mediators.

We illustrate the use of generic wrapping rules by an excerpt of a larger case study [May99a] which creates a geographical database. The main data sources are the *CIA World Factbook* [CIA], which provides information about countries and political organizations, and *Global Statistics* [Glo], a collection of data about administrative divisions and cities. Here, the situation is different from the single-site DBLP server. The skeletons of the data sources are simple. In particular, there is only a minimal number of intra-site references and no inter-site references. Due to the lack of inter-site references, the sites can be evaluated independently with a subsequent inter-source integration step: At the wrapper-level, an individual model for each of the data sources is obtained by instantiating generic patterns. The mediation and integration part consists of fusing corresponding objects across sources and creating references between the partial models. The complete programs can be found in [May99a]. In the following description, all instances of generic extraction patterns are labeled with $\boxed{\text{RULE\_NAME}}$. We present and use generic rules to extract information from:

- repeating patterns of text formatted as a list of properties,

- comma-lists of (name-value-unit)-tuples,
- HTML-lists of properties using also textual formatting, and
- HTML-tables.

9.1 CIA World Factbook: Wrapping Countries [Textual Markup]

The CIA World Factbook provides political, economic, social, and some geographical information about countries. The pages are structured only as formatted text (see Fig. 10), and therefore, the wrapper is based on pattern matching via regular expressions. The regular textual format of the pages allows for very generic and easily extendible patterns for single-valued properties (both numeric and string, such as *population* and *capital*) and for multi-valued properties like border countries, languages, etc., which are given by lists of pairs (*name value*).

Belgium

Geography

Location: Western Europe, bordering the North Sea, between France and the Netherlands
Area: *total area:* 30,510 sq km
border countries: France 620 km, Germany 167 km, Luxembourg 148 km, Netherlands 450 km

People

Population: 10,170,241 (July 1996 est.)
Ethnic divisions: Fleming 55%, Walloon 33%, mixed or other 12%
Religions: Roman Catholic 75%, Protestant or other 25%
Languages: Dutch 56%, French 32%, German 1%, legally bilingual 11% (divided along ethnic lines)

Government

Capital: Brussels
Independence: 4 October 1830 (from the Netherlands)

Economy

GDP: purchasing power parity – $197 billion (1995 est.)

Figure 10: Excerpt of the CIA World Factbook: Countries

We initialize the program with the continent's urls:

```
"Europe":continent[file@(cia)->"eur.htm"].
"Asia":continent[file@(cia)->"asia.htm"].
"North America":continent[file@(cia)->"namr.htm"].
"Australia/Oceania":continent[file@(cia)->"oce.htm"].
"South America":continent[file@(cia)->"samr.htm"].
"Africa":continent[file@(cia)->"afr.htm"].
cia[src->"http://www.odci.gov/cia/publications/factbook/".
```

The continent pages are accessed by the built-in active method url.get:

```
U:url.get :- C:continent[file@(cia)->FN], strcat(cia.src,FN,U).
```

Every continent page contains a list of links to the countries that are automatically stored in the slots u.get..hrefs@(<label>). The country objects are then filled in with their respective urls, names, and continents, and the urls are accessed:

```
cid(cia,C2):country
   [url@(cia) -> U; name@(cia)->> CountryName; continent -> Cont] :-
        Cont:continent.url@(cia).get[hrefs@(Label) ->> U],
        pmatch(Label,"/(.*) \([0-9]/","$1", CountryName).
U:url.get :- _:country[url@(cia)->U].
```

e.g., for Belgium, this rule derives

$o_{belgiumCIA}$: country[url@(cia)→"http://www.odci.gov/cia/publications/factbook/be.html"; name@(cia)↠"Belgium"; continent→o_{europe}] .

Extracting properties of countries. On the CIA pages, country properties are introduced by keywords (cf. Figure 10). Here, scalar properties (such as *capital*) and multivalued properties (such as *religions*) have to be distinguished. Often, values consist of a number and a unit, such as *km*. We define generic rules for handling scalar and multivalued properties. For each individual property, only a tuple containing the required metadata has to be given.

Extracting scalar properties by patterns. Single-valued methods are defined using a generic pattern which consists of the method name to be defined, the regular expression that contains the keyword and the type of the answer:

```
property(country,capital,"/Capital:.*\n(.*)/",string).
property(country,indep,"/Independence:.*\n(.*) [^(]/",date).
property(country,total_area,"/total area:.*\n(.*?) sq km/",num).
property(country,population,"/Population:.*\n([0-9,]+) /",num).
```

The patterns are applied to the country.url.get-objects, representing the Web pages. Note the use of variables ranging over host objects (Ctry), methods (Method), and classes (Type):

```
Ctry[Method:Type -> X] :-                                    TEXT PATTERNS
   property(country,Method, RegEx, Type),                      +REFINEMENT
   pmatch(Ctry:country.url@(cia).get, RegEx, "$1", X),
   not substr("million",X).
Ctry[Method:Type -> X2] :- property(country,Method, RegEx, Type),
   pmatch(Ctry:country.url@(cia).get, RegEx, "$1", X),
   pmatch(X, "/(.*) million/", "$1", X1),  X2 = X1 * 1000000.
```

Here, we see an instance of *refinement*: the (generic) main rule is only valid if it applies to a string or to a number given by digits – but some numbers are given semi-textually, e.g., "42 million". This is covered by the refined rule. Here, e.g.,

$o_{belgiumCIA}$[capital→"Brussels"; indep→"4 October 1830"; total_area→30510; population→10170241]

is derived. Similarly, dates are extracted from strings by further matching:

```
C[M@(date)->D2] :- C:country[M:date->S], perl(fmt.date,S,D),    DATE VALUES
   pmatch(D,"/\A([0-9]{2} [0-9]{2} [0-9]{4})/", "$1", D2).              (*)
```

yielding, e.g., $o_{belgiumCIA}$[indep@(date)→"04 10 1830"].

Extracting multi-valued complex properties by patterns. Ethnic groups, religions, languages, and border countries are given by *comma-separated lists* of *string-value*-pairs, introduced by a keyword; *value* has a unit like "%" or "km". Again, we use generic rules provided with a metadata tuple for every property to be extracted. Then, having the comma-list as a whole in a string attribute, the comma-list is decomposed using the metadata information:

- decomposition of comma-lists into items of the form <name value unit>,

- decomposition of the items into parametrized methods,
- aggregation of items that belong together.

Theses rules can be reused in any situation where comma-lists occur (e.g., again in Section 9.2 to analyze the memberlists of political organizations).

```
commalist(ethnicgroups, "/Ethnic divisions:.*\n(.*)/", "\%", string).
commalist(languages, "/Languages:.*\n(.*)/", "\%", string).
commalist(religions, "/Religions:.*\n(.*)/", "\%", string).
commalist(borders, "/border countries:.*\n(.*)/", " km", country).
commalist(borders, "/border country:.*\n(.*)/", " km", country).
```

Here the matching criterion "border countries" has to be complemented with "border country" (for countries having only a single neighbor). For every pattern, the raw comma-list is extracted, e.g., $o_{belgiumCIA}$[borders : commalist→"France 620 km, Germany 167 km, ..."]; comments (between parentheses) are dropped by a call to the user-defined Perl-predicate drop.comments:

```
C[M:commalist->L2] :-  commalist(M,S,_,_),                     COMMA-LISTS
  pmatch(C:country.url@(cia).get,S,"$1",L), perl(drop.comments,L,L2).
```

The comma-lists are decomposed into multi-valued methods:

```
C[M->>X] :- C:country[M:commalist->S],          DECOMPOSING COMMA-LISTS
            pmatch(S,"/([^,]*)/","$1",X).                          (**)
```

yielding $o_{belgiumCIA}$[borders↠{"France 620 km", "Germany 167 km", ...}].
In the next step, the values are extracted from the individual list elements. There is a refined rule that preliminarily assigns the null value to elements that contain no value (which is often the case for languages or religions). Since comments in parentheses have been dropped, e.g., for borders, *Russia (northwest)*, *Russia (northeast)* (China has two separate borders with Russia) both result in *Russia*.

```
C[M@(X)->>V] :-  commalist(M,_,S,_),                     DECOMPOSING PAIRS
    strcat("/([^0-9]*)\s* ([0-9\.]+)",S,S2), strcat(S2,"/", Pattern),
    pmatch(C:country..M, Pattern, "[$1,$2]", "[X,V]").
C[M@(X)->>null] :- commalist(M,_,S,_), C:country[M->>X], not substr(S,X).
```

This results in $o_{belgiumCIA}$[borders@("France")↠620; borders@("Germany")↠167; ...].
Multi-valued methods with arguments as strings are simply summed up using FLORID's sum aggregate:

```
C[M@(X)->V] :- commalist(M,_,_,string),                          SUMMATION
               V = sum{Val [C,M,X]; C:country[M@(X)->>Val]}, V > 0.
```

Here, e.g., $o_{chinaCIA}$[borders@("Russia")↠3605; borders@("Russia")↠40; ...] are summed up to $o_{chinaCIA}$[borders@("Russia")→3645]. In a refinement step, non-quantified elements of unary lists resulting in percentages are now set to 100%, other entries without percentages remain nulls; here, a count aggregate is used.

```
C[M@(X)->100] :- commalist(M,_,"\%",string),                 REFINEMENT:
          C:country..M@(X)[], not C.M@(X)[],                  EXCEPTIONS
          N = count{X [C,M]; C:country..M@(X)[]}, N = 1.   OF SUMMATION
C[M@(X)->null] :- commalist(M,_,S,string),
          C:country[M@(X)->>null], not C.M@(X)[],
          N = count{X [C,M]; C:country..M@(X)[]}, N > 1.
```

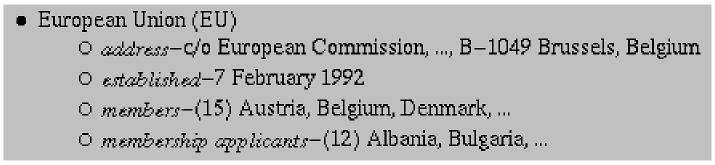
For multi-valued methods with arguments as objects of a class that provides a name method (i.e., an application-level class, here country[borders@(country)⇒number]), the respective argument objects are linked (in fact, M and Class instantiate with borders and country, respectively):

```
C[M@(C2)->V] :- C2:Class[name@(cia)->>X], commalist(M,_, S, Class),
                C:country[M@(X)->Val].
```

resulting in $o_{belgiumCIA}$[borders@(o_{france})→620; borders@($o_{germany}$)→167; ...].

9.2 The CIA World Factbook: Wrapping Organizations [HTML-Lists]

Appendix C of the CIA World Factbook provides information about political and economical organizations. The page consists of tagged nested lists (see Fig. 11). Thus, we exploit the structure of the parse-tree: For every organization, a list of properties is given. The list items are then queried by pattern matching, filling the corresponding methods of the organization. The members are again given as a set of comma-lists of countries (distinguished by the type of membership).


- European Union (EU)
  - *address*–c/o European Commission, ..., B–1049 Brussels, Belgium
  - *established*–7 February 1992
  - *members*–(15) Austria, Belgium, Denmark, ...
  - *membership applicants*–(12) Albania, Bulgaria, ...

```
<UL> ... <LI>European Union (EU)</LI>
 <UL>
  <LI><I>address</I>-c/o ... , B-1049 Brussels, Belgium</LI>
  <LI><I>established</I>-7 February 1992</LI>
  <LI><I>members</I>-(15) Austria, Belgium, Denmark, ... </LI>
  <LI><I>membership applicants</I>-(12) Albania, Bulgaria, ... </LI>
 </UL> ... </UL>
```

Figure 11: Excerpt of the CIA World Factbook: Appendix C: Organizations

```
U:url.parse :- ciaAppC[url->U].
```

For every letter, the page contains a list with organizations. The class u.parse.ul contains all lists in u:

```
L:org_list :- ciaAppC.parse.root[html@(X)->L:(ciaAppC.parse.ul)].
```

The org_lists (<UL>) are then analyzed by navigating the parse-tree: The k-th element of the <UL> is an organization header, the $k{+}1$-th the corresponding data list. In the same step, the abbreviation and the full name are extracted from the header, and the inner <UL> contains the properties of the organization:

```
org(Short):org[abbrev->Short; name->Long; data->DataList] :-
  L:org_list[ul@(K)->OrgHdr], L.ul@(K1)[li@(0)->DataList], K1 = K + 1,
  OrgHdr[li@(0)->Name], string(Name), DataList.ul@(0)[],
  pmatch(Name,"/(.*) \(([^)]*)\)/","[$1,$2]","[Long,Short]").
```

derives, e.g., o_{eu}:org[abbrev→"EU"; name→"European Union";
data→"<UL><LI><I>address</I> ... </LI>
<LI><I>membership applicants</I>- (12) Albania,... </LI>
</UL>"] .

For every organization, several properties are listed, e.g., its address, its establishment date, and a list of members for each member type. For every property, the corresponding x-th item of the <UL> (which itself is an <LI>) is identified by a keyword (which is <I>-ed in the 0-th item of the <LI>), e.g., "address" or "established", or a member type.

```
property(org,address,"address",string).
property(org,establ,"established",date).
property(org,member_names,"member",membertype).
property(org,applicant_names,"membership applicants",membertype).
membertype::commalist.
```

Note that we only use a refinement of a generic rule to skip the leading "-" and the number of members:

```
O[MethName:Type->Value2] :-                              LIST OF PROPERTIES
    O:org[data->D], property(org,MethName,MethString,Type),        REFINED
    D.ul@(X)[li@(0)->_[i@(0)->MethString];li@(1)->Value],
    pmatch(Value,"/\A-\s*(\([0-9]*\))*\s*(.*)/", "$2",Value2).
```

derives, e.g.,
o_{eu}[address:string→"c/o ... Brussels, Belgium"; establ:date→"7 February 1992";
member_names:membertype→"Austria, Belgium, ...";
applicant_names:membertype→"Albania, Bulgaria ..."] .
Again, by an instance of the generic rule pattern (*), the date is formatted:

```
C[M@(date)->D2] :- O:org[M:date->S], perl(fmt.date,S,D),        DATE VALUES
   pmatch(D,"/\A([0-9]{2} [0-9]{2} [0-9]{4})/", "$1", D2).                (*)
```

The address is analyzed by an application-specific matching rule:

```
O[seatcity->City;seatcountry->Country] :-  O:org[address->A],
    pmatch(A, "/(.*)[,0-9] ([^,0-9]*)\Z/", "[$1,$2]", "[City,Country]").
```

derives, e.g.,
o_{eu}[seatcity→"Brussels"; seatcountry→"Belgium"; establ@(date)→"07 02 1992"].
The comma-lists of the member types are again decomposed using the generic rule pattern (**) from the previous section (remember that membertype :: commalist):

```
O[M->>X] :- O:org[M:commalist->S],                DECOMPOSING COMMA-LISTS
            pmatch(S,"/([^,]*)/","$1",X).                               (**)
```

derives, e.g., o_{eu}[member_names:membertype↠{"Austria", "Belgium", ...};
applicant_names:membertype↠{"Albania", "Bulgaria", ...}] .
In order to integrate the CIA countries with the CIA organizations, memberships are represented by references to the respective country objects:

```
O[member@(Tname)->>C] :- property(org,T,Tname,membertype),
                         O[T:membertype->>CN], C:country[name@(cia)->>CN].
```

derives, e.g., o_{eu}[member@("member")↠{$o_{austriaCIA}, o_{belgiumCIA}, \ldots$}].

Flexibility. The flexibility of using a rule-based modular wrapping language proves to be useful if the layout of an information source changes: In the 1997 Factbook, the layout of the organizations page has been changed significantly. Instead of HTML lists, HTML tables were used. These tables have no table row/column semantics, but are only for optical markup.

In the wrapper program, only the first, source-specific rule that decomposes the outermost <UL> has to be redesigned, and instead of the List of Properties rule, another rule Table of Properties for handling tables has to be deployed:

```
O[MethName:Type->Value] :-                               TABLE OF PROPERTIES
  O:org[data->D], property(org,MethName,MethString,Type),
  D.td@(2)[p@(X)->MethString], D.td@(2)[p@(Y)->Value], Y = X + 1.
```

The analysis of the properties of organizations itself does not need to be changed.

9.3 Global Statistics: Wrapping Cities and Provinces [HTML-Tables]

The *Global Statistics* [Glo] pages provide information about administrative divisions (area, population, and capitals) and main cities (population and province). The data is given by HTML tables as shown at the right, but with an irregular column structure. Here, generic rules for analyzing tables are applied. Application-specific rules map the semantics of the columns to the object-oriented model. Additionally, layout specialties have to be dealt with, such as twocolumn layout and multilevel administrative divisions. The generic rules for tables proceed as follows, using metadata information about the properties to be extracted from table columns:

administrative divisions of Belgium

province	capital	area (km^2)	pop. Dec. 31, 1991
Antwerp	Antwerp	2,867	1,610,695
Brabant	Brussels	3,358	2,253,794

main cities of Belgium

cities	pop. 1995 est.
Brussels (Agglomeration)	951,580
Antwerp	459,072

- mapping the tables to a matrix (table, row, column),
- detecting header rows,
- detecting column header entries,
- identifying semantical information of tables and columns using metadata,
- extracting the contents from the matrix into the object-oriented model.

Again, these generic rules can be reused in any situation where HTML tables have to be handled.

As a first step, country urls are extracted from the continent pages. Then, the country urls are accessed and parsed by the built-in SGML parser:

```
U:country_url.parse :- _:country[url@(gs)->U].
```

The class u.parse.table contains all tables of u. Based on the <TR> and <TD> structure, the tables are mapped to a matrix (by several refinements of the following rule):

TABLES: ELEMENTS

```
element(T,Row,Column)[contents->Contents; type->Type] :-
    U:country_url, T:(U.parse.table),
    T.table@(0)[tbody@(Row)->_[tr@(Column)->X[Type@(0)->Contents]]].
```

Here, e.g., the administrative divisions table is the second element in the page body, i.e., $o_{belgiumGS}$.url@(gs).parse.body@(2) : ($o_{belgiumGS}$.url@(gs).parse.table).
Then, element($o_{belgiumGS}$.url@(gs).parse.body@(2),2,3)[contents→"area (km^2)"; type→"th"] and element($o_{belgiumGS}$.url@(gs).parse.body@(2),4,3)[contents→"3,358"; type→"td"] are derived.
Next, information about column headers is extracted:

TABLES: COLUMN HEADERS

```
T[header_row->>HR] :- U:country_url,
        T:(U.parse.table), element(T,HR,_)[type->th].
T[header@(HR,Col)->>String] :-
        U:country_url, T:(U.parse.table)[header_row->>HR],
        element(T,HR,Col)[contents->String; type->th].
```

leading to, e.g., $o_{belgiumGS}$.url@(gs).parse.body@(2)[header_row↠{1};
header@(1,0)→"province";...;header@(1,3)→"pop.Dec.31,1991"].

In order to extract data from the tables, the application-level semantics of tables is described at the meta-data level allowing. In our example, for each country, two tables are given: *administrative divisions* and *main cities*. Both tables contain tuples describing objects (provinces and cities) resulting from the application of a multivalued method on the countries. Each of the columns determines an attribute and can be identified in the table by the corresponding keywords:

```
admdivtab:countrytable[tabheader->>"administrative divisions";
          method->adm_divs; resultclass->province;
          columns->>{prov_name_col,prov_pop_col,prov_area_col,prov_cap_col}].
prov_area_col[attr->area; header->>{"area"}].
prov_pop_col[attr->pop; header->>{"population"}].
prov_cap_col[attr->capital_name; header->>{"capital","hauptstadt"}].
maincitytab:countrytable[tabheader->>{"municipalities", "main cities"};
            method->main_cities; resultclass->city;
            columns->>{city_name_col,city_pop_col,city_prov_col}].
city_pop_col[attr->pop; header->>{"pop"}].
city_prov_col[attr->prov_name; header->>{"prov"}].
```

The tables are now identified via their header entries. Note that the 0.th column is always known to be the column which gives the name of an object:

```
C[TabName -> T[col(name)->0]] :-                          IDENTIFYING TABLES
   C:country[url@(gs)->U], T:(U.parse.table), element(T,_,_)[contents->I],
   TabName:countrytable[tabheader->TH], substr(I,TH).
```

In our example,

$o_{belgiumGS}$[admdivtab→$o_{belgiumGS}$.url@(gs).parse.body@(2)] and
$o_{belgiumGS}$[maincitytab→$o_{belgiumGS}$.url@(gs).parse.body@(3)]

are derived. In the next step, the attribute table columns are identified:

```
 C.TabName[col@(Attr)->Col] :-                           IDENTIFYING COLUMNS
  (C:country).(TabName:table)[header@(HR,Col)->S],
  TabName[columns->>TC[attr->Attr; header->ColHdr]], substr(S,ColHdr).
```

Here, e.g.,

$o_{belgiumGS}$.admdivtab[col@(area)→2; col@(pop)→3; col@(capital_name)→1] and
$o_{belgiumGS}$.maincitytab[col@(pop)→1]

are derived. Now, the data of both tables can be extracted by a *generic* rule:

```
C[Method ->> obj(T,DRow):ResultClass[country->C; Attr->Val]] :-       TABLE
   C:country[(Whichtable:countrytable)->T],                            DATA
   Whichtable[method->Method; resultclass->ResultClass],         EXTRACTION
   T[col@(Attr)->Num],
   element(T,DRow,Num)[contents->Val; type->td].
```

E.g., if C is bound to $o_{belgium}$ and Whichtable is bound to admdivtab, then we have the following bindings:

Method = adm_divs , ResultClass = prov , and T = $o_{belgiumGS}$.admdivtab.

Then, Attr and Num can be bound to (name,0), (capital_name,1), (area,2), or (pop,3). DRow now runs over all table rows which contain <td> data and extracts the corresponding Values. E.g.,

$o_{belgiumGS}$[adm_divs→→{$o_{p\_antwerp}$}] and
$o_{p\_antwerp}$[name_str→"Antwerp"; population→1610695; area→3358; capital_str→"Antwerp"].

are derived. The above rules show that a description of the *table semantics* as meta data proves useful for using generic rules.

In the first step of the mediation process, the references from cities to provinces, and the references from provinces to their capitals are generated:

```
Cty[province->>Prov] :-
    Cty:city[country->C;prov_name->PN], Prov:prov[country->C;name->PN].
P[capital->Capcty] :- P:prov[country->C;name->PN;capital_name->CN],
    Capcty:city[country->C;name@(gs)->>CN;province->>P[name->PN]].
```

10 Mediation and Integration

In the classical wrapper-mediator architecture, the mediators are responsible for data integration as well as for providing an interface to the user (or to other mediators). Typical tasks for mediators are merging information about equivalent objects from different sources, or decoding string(name)-valued attributes into links to appropriate objects from another source. Additionally, the data is mapped to an application-level schema which can be queried or exported.

In our approach, mediators are also specified by F-Logic rules. Again, generic rules are provided to handle typical tasks. Based on the integrated world model, rules can be defined that operate (simultaneously) on the Web skeleton, its hyperlinks, the HTML/XML sources and their parse-trees, and on the application-level model itself. A distinguishing feature of our integrated architecture – using a unified data model and manipulation language – is that these rules can interact to combine information from different data sources. By means of inter-source joins, information already extracted from one source can be applied towards specific extraction of information from other sources (including exploring other sources that provide the required information). After transforming the contents of the Web pages into F-Logic models by wrapper rules, the integration and restructuring of the data in a comprehensive object-oriented model can be performed. This task mainly involves the identification of objects among different sources (e.g., countries on the CIA pages with countries on the GlobalStatistics pages):

- If the objects to be identified are connected by inter-site hyperlinks, these hyperlinks can easily be used to merge corresponding objects (comparable with object references in databases). This is sometimes the case if one of the data sources is a well-known reference site, e.g., it would be reasonable to have links from GlobalStatistics into the CIA World Factbook.
- In most cases – since the sources are independent and autonomous – there are no inter-source hyperlinks. For the integration task, this means that references between objects cannot be derived from hyperlinks, but only by attributes of objects (comparable to primary key/foreign-key conditions in databases; in most cases, the name).

The Case-study. The databases generated by the wrapper rules given in the previous section are integrated into a common object-oriented schema. In our example, there is only a minimal number of intra-site references and no inter-site reference. The partial models generated by the wrappers already provide large parts of the external schema.
First, the countries of CIA and *Global Statistics* have to be matched, either by name

```
C1 = C2 :- C1:Class[name@(A)->N],  C2:Class[name@(B)->>N].   FUSING BY NAME
```

(which would derive $o_{belgiumCIA} \equiv o_{belgiumGS}$) or in case that different names are used, if they belong to the same continent and the capitals' names are the same:

```
C1 = C2 :- C1:country[name@(cia)->>_; continent->CT; capital@(cia)->N],
           C2:country[continent@(gs)->>CT], C2.capital[name@(gs)->>N].
```

To create intersource links, generic rules using *meta knowledge* about the schema (keys, referential constraints) are employed. Here we give only an application-specific rule that links seats of organizations (in CIA) to the corresponding cities (in GlobalStatistics):

```
O[seat->Cty] :- O:org[seatcity->N; seatcountry->CN],
                Cty:city[name->> N], Cty.country[name@(cia)->>CN].
```

deriving, e.g., o_{eu}[seat→$o_{brussels}$].
The rules given here are slightly simplified w.r.t. the original programs [May99a] to illustrate the principles of information extraction from the Web in our approach. In the full program, more exceptions and formatting issues have been faced, that lead to refined rules.

11 Querying Forms and Search Engines

Sections 8, 9, and 10 showed the "canonical" application: wrapping HTML pages and integration of different sources. A large (and increasing) number of Web pages is not stored explicitly on a server, but is created on-demand from an underlying database by a database Web server. These pages are accessible via HTML/CGI *forms* which provide an interface to the database. Here, a special – and very important – case are *Web search engines* which provide a form-based interface to an index. The ability to use such Web searching engines in combination with data-driven Web exploration leads to a powerful tool to search and extract information.

11.1 Querying Forms

When information is provided via *forms*, the wrapper must be able to simulate calls of CGI scripts. The information about the cgi script to be called and how to call it is contained in the contents of the <form> tag as shown below (which is an excerpt of the AltaVista query form):

```
<html>
<head><title>AltaVista HOME</title></head>
<body>
...
<form method="GET" action="/cgi-bin/query">
<input type=hidden name=pg value=q><br>
<input type=text name=q value="">
</form>
...
</body>
</html>
```

The url of the CGI script is given in the <form action=*cgi-url*> attribute. The parameter names are given by <input name=p> tags inside the <form> tag. Note that *hidden* parameters of the form <input type=hidden name=p value=v> are allowed which predefine parameters which cannot be given by the user (cf. Example 16). In order to send parameter values to the CGI script, two different methods can be used: `GET` and `POST`; the method to be applied is given in the method attribute of the <form> tag. With the `GET` method, the parameters are simply appended to the base url of the CGI script. With the `POST` method, the parameters are received by the CGI script via standard input.

Obviously, calls via `GET` are easy to simulate in our framework: parameters and their values are appended to the url of the CGI script, yielding a parameterized url u_{par} of the form

$$\texttt{cgi-url?}p_1\texttt{=}v_1\texttt{\&...\&}p_n\texttt{=}v_n\ .$$

Figure 12 shows a generic wrapper for HTML forms using the `GET` method that proceeds as follows:

- the values to be queried are expected to be single-valued attributes of the object query; e.g., query[author→"M.E.Senko"; year→"1975"].
- the rules iteratively check all tags inside the <form> tag, adding "name=value&" to the url for every <input> tag.

The result page of the query is then automatically associated with u_{par}.get as usual.

```
% initialize generation of query url:
F[createurl@(0)→U1]:- F:(X:url.parse.form),
        F[method→GET; action→BaseURL], strcat(BaseURL,"?",U1).

% incrementally working on the tags inside the <form> tag:
% next tag is a hidden input:
F[completeurl@(M)→U0; createurl@(M)→U1]:-
        F:(X:url.parse.form)[createurl@(N)→U], M = N + 1,
        F.form@(N):(X.parse.input) [type→"hidden"; name→N; value→V],
        perl(strings.concatenate,[U,N,"=",V],U0), strcat(U0,"&",U1).

% next tag is a non-hidden input:
F[completeurl@(M)→U0; createurl@(M)→U1] :-
        F:(X:url.parse.form)[createurl@(N)→U], M = N + 1,
        F.form@(N):(X.parse.input) [type→"text"; name→N],
        query[name→V] % the object "query" must contain the values
        perl(strings.concatenate,[U,N,"=",V],U0), strcat(U0,"&",U1).

% next tag is not an input tag:
F[completeurl@(M)→U0; createurl@(M)→U1] :-
        F:(X:url.parse.form)[completeurl@(N)→U0; createurl@(N)→U1],
        M = N + 1, F[form@(N)→T], not T:(U:url.parse.input)

% no more elements inside the <form> tag: access generated url.
U.get :- F:(X:url.parse.form)[completeurl@(N)→U], not F.form@(N).
```

Figure 12: Generic wrapper for generating form queries

Example 16 *The Web search engine* AltaVista *[Alt98] uses forms with the* `GET` *method as shown above. Figure 13 shows the complete wrapper program for AltaVista.*
Lines 1-3 generate the query url

```
"http://www.altavista.com/cgi-bin/query?q=Freiburg"
```

for the query form. This and all later result urls are requested via the http protocol, and the parse-trees are generated by the Web interface (Line 5): Every result page contains a number of hits, and links to further result pages:

- *each hit – and only the hits – are presented in* <dl> *tags:*

  ```
  <dl><dt><b>1. </b>
    <a href="http://web.mathematik.uni-freiburg.de/Welcome_e.html">
    <b> Faculty of Mathematics, Freiburg University</b></a><dd>
    Albert-Ludwigs-Universitaet Freiburg. Faculty of Mathematics ...
    <b>URL:</b>
    <font color=gray>web.mathematik.uni-freiburg.de/Welcome_e.html<br>
    Last modified 11-May-99 - page size 4K - in English</font></dl>
  ```

  Each hit is made an object and some properties can already be extracted from the information given by AltaVista (Lines 6–13). This information can be used to filter those hits actually accessed.

- *additionally, every result page contains a "Next" hyperlink to the subsequent result page:*

  ```
  <a href="/cgi-bin/query?pg=q&q=Freiburg&stq=10&c9k">[<b>Next</b>]</a>
  ```

  Following this hyperlink, further result pages can be accessed (Line 14-16). All result pages have the same layout such that the above rules apply.

```
 1 altavista[bin_url->"http://www.altavista.com/cgi-bin/query?q="].
 2 altavista[searchtext->"Freiburg"].
 3 U:resulturl :- strcat(altavista.bin_url, altavista.searchtext, U).

 4 resulturl::url.

 5 U.parse :- U:resulturl.

 6 X:hit[url -> HitURL:url; title -> T; mod -> M; year -> Y;
         size -> S; lang -> L] :-
 7   X:(U:resulturl.parse.dl),
 8   X.dl@(0).dt@(1):a[href -> HitURL].a@(0)[b@(0)-> T],
 9   X.dl(0).dt@(4)[font@(3) -> Info:string],
10   pmatch(Info,"/Last modified ([0-9]*-[A-z]*-)([0-9]*) -
11                 page size ([0-9]*)K - in ([A-z]*)/",
12               "[$1$2,$2,$3,$4]", "[M,Y0,S,L]"),
13   string2real(Y0,Y).

14 NextURL:resulturl :-  X:(U:resulturl.parse.a),
15    X[href->LocalNextURL].a@(1)[b@(0)->B], substr("Next",B),
16    strcat("http://www.altavista.com", LocalNextURL, NextURL).
```

Figure 13: Basic Wrapper Program for AltaVista

Note that the pure wrapper+explorer consists only of three rules (Lines 5-16) and that the program does not need any stratification.

The hits can be further filtered, e.g., only pages which are at least from 1998, in English, and not on a .com domain:

```
HitUrl.get:relevant_page :- X:hit[url->HitURL; year->M; lang->"English"],
   M > 1997, not pmatch(HitURL,"(.*)/.com\Z/","$1",_).
```

Then the selected pages can be accessed and processed as usual.

To simulate queries on forms using the `POST` method, additional protocol-level functionality is needed: there, the parameters are received by the CGI script via standard input. Suitable functionality is, e.g., provided by the *perlwww* library that can either be used with FLORID's perl interface, or FLORID could be extended by a suitable built-in. From the *declarative* point of view, this provides no new insight, so it is not (yet) implemented in the research prototype. Note that other approaches, such as WebOQL [AM98] and W3QL/W3QS [KS95, KS98b] do not provide the functionality to handle `POST` forms either.

11.2 Exploiting Web Search Engines

In the case-studies presented above, we showed how a database can be generated from Web pages whose structure was known a priori. The focus was on wrapping and integration; data-driven Web exploration played only a minor role (to range over a set of relevant urls). Motivated by the AltaVista example to handle forms, data-driven Web exploration can be used for powerful Web searching by combining existing Web indexing engines and user-defined rules: the Web indexing engine provides basic searching features and yields a set of potentially relevant urls.

Refining Search Results. Daily experience shows that from the results of a query to a Web searching engine, often only a small subset of the pages is really relevant, and additional selection criteria should be applied to restrict the set of pages that actually have to be checked: for example, obsolete pages that have not been maintained for years are not worth looking, and, for us, pages written in japanese need not be considered (similar criteria have already been used in the previous example). E.g., the following tasks can be programmed to act on web search engines:

- The results of Web searching can be used to start data-driven exploration around the result pages:
  - search all pages on the Freiburg University server that contain the word "Seminar", and try to find the corresponding research group and the responsible assistant.
  - search all pages linking to my homepage and try to find the name and the e-mail adress of the responsible person.
- By combining the results of different Web search engines, a better success rate can be achieved.

Data-Driven queries on Web Indexes. Another interesting application is to query Web indexes depending on previously extracted data. In our example, we could query AltaVista for tourist information pages of all capitals. Additionally, the results can be filtered as described above.

In general, when using Web search engines to find relevant pages, the structure and layout of the resulting pages is not known. Thus, in most cases, only rough heuristics, based on text search can be applied. Of course, the development of intelligent wrapping techniques can lead to better results.

12 Data Transfer to DBMS and Applications

A system using our architecture is especially tailored to Web data extraction and restructuring, but in general does not provide classical database functionality, such as querying in SQL, storage optimization, or security. Furthermore, the extracted data should either be integrated into an existing database in some classical database management system, or submitted to an application. In order to do this, the application-level fragment of the internal model implicitly defines the *output schema*. The communication with applications or database systems can either be controlled by the application, or by the Web extracting system:

Pull-Interfaces: The transfer activity is initiated by the DBMS which states a query to the Web data extraction system and receives the result in a distinguished data format. Here it can be distinguished whether the Web data extraction system already holds the extracted database and the DBMS, or the application queries this database, or if the Web data extraction system is idle and the DBMS or the application submits a program to be evaluated and a query to the system. In the first case, only the answer to the query has to be computed, whereas in the second case, the system evaluates the program and sends the results to the DBMS or to the application.

The query and the answer are basically passed through the F-Logic user interface (note that by using predicates, the results are directly usable by relational databases), although an OQL-like interface to the F-Logic database could also be implemented.

Push-Interfaces: The transfer activity is initiated by the Web data extraction system which sends an excerpt of its database to the DBMS or to the application (either at regular intervals, or whenever the database has changed).

XML Output. In addition to the classical interface for relational DBMS, electronic data interchange (EDI) in XML can be supported by an interface producing an XML model of the internal database. For a given DTD and a mapping of the object-oriented model to that DTD, the matching information is converted into an XML file. Then, further processing by any tools accepting XML input, or a browsable presentation via XSL/CSS is possible.

Implementation. Via output redirection, the contents of an F-Logic database stored in FLORID can be filtered and formatted to any data model. For the above case study, we produced the output in the following data models using F-Logic programs:

- a cleaned object-oriented schema in F-Logic syntax,
- a relational schema used for Oracle 8 training,
- an XML schema (Figure 14 shows an excerpt of the XML representation of MONDIAL).

13 Related Work

Web querying languages. Most implementations of Web query languages follow an integrated approach with a generic Web interface (mainly, HTML parsing) and an internal data model. Web querying languages are not concerned with application-level semantics and integration of data. Some of them provide restructuring functionality within the internal data model.

As one of the first Web querying languages, **WebSQL** [MMM97] modeled the Web by two virtual relations which exactly correspond to documents and hyperlinks. For documents, attributes title, type, modif, length, and text – the latter containing the document source – are defined. Thus, these relations represent exactly what we call the Web skeleton, including the automatically defined document built-in methods. WebSQL allows queries on the Web structure using this "virtual graph". The intra-document structure is not modeled, and therefore, cannot be queried either; only text search in documents using the `contains` predicate is possible. The SQL `SELECT FROM WHERE` syntax is extended with a `SUCH THAT` construct applied to every element of the `FROM`-clause to define the relevant portion of the Web to be materialized. The `SUCH THAT` condition is either an url or of the form `MENTIONS <string>` (which can be checked against an index); thus, the number of root documents for a query is finite. Web access is implemented by a *query engine* which – based on the `FROM` clause of the WebSQL query – either sends a request to an index server, or accesses the document tree specified by the WebSQL query via the http-protocol.

From the same group, **WebOQL** [AM98] additionally considers the internal structure of documents by making parse-trees first-class citizens of the model that can be queried directly. Similar to our approach, they use a unified model of the inter-document level (Web) and the intra-document level (parse-trees). Browsing is regarded as a function (corresponding to our Web interface method parse) that maps urls to *hyper-trees*, or *"webs"*. Thus, all HTML pages are handled by a generic built-in interface that maps an HTML page to a hyper-tree without requiring external wrappers. WebOQL queries are then performed against the current web; additionally, new webs can be built. Furthermore, queries against forms using the `GET` method can be done via string manipluation.

One important distinction between WebOQL and FLORID is in the "weight" of the data model: while the FLORID data model is a "lightweight" data model (i.e., there is almost no fixed schema), the WebOQL hypertrees are based on a fixed generic tree schema.

Example 17 (WebOQL vs. Florid) *In Example 8, the parse-trees in* FLORID *are constructed directly by tags. Thus the "signature" can be queried by syntactically second-order expressions like*

"journals/is/is1".parse[M$\rightarrow$X] .

Binding M to html, the tags give the names of the navigation methods in the parse-tree:

```
<country id="D">
  <name>Germany</name>
  <total_area>356910</total_area>
  <population>83536115</population>
  <encompassed continent="europe">100</encompassed>
  <ethnicgroup name="German">95.1</ethnicgroup>
  <ethnicgroup name="Italians">0.7</ethnicgroup>
  <religion name="Roman Catholic">37</religion>
  <religion name="Protestant">45</religion>
  <language name="German">100</language>
  <border country="F">451</border>
  <border country="A">784</border>
  <border country="CZ">646</border>
  ...
  <province id="prov-Germany-Baden-Wuerttemberg">
    <name>Baden Wuerttemberg</name>
    <area>35742</area>
    <population>10272069</population>
    <city is_state_cap="yes" id="cty-Germany-Stuttgart">
      <name>Stuttgart</name>
      <longitude>9.1</longitude>
      <latitude>48.7</latitude>
      <population year="95">588482</population>
    </city>
    <city id="cty-Germany-Mannheim">
      <name>Mannheim</name>
      ...
    </city>
    ...
  </province>
  <province id="prov-Germany-Berlin">
    <name>Berlin</name>
    <area>889</area>
    <population>3472009</population>
    <city is_country_cap="yes" is_state_cap="yes"
          id="cty-Germany-Berlin">
      <name>Berlin</name>
      <longitude>13.3</longitude>
      <latitude>52.45</latitude>
      <population year="95">3472009</population>
    </city>
  </province>
  ...
</country>
```

Figure 14: Excerpt of the XML representation of MONDIAL

"journals/is/is1".parse.html.body@(2).ul@(0).li@(0)[a@(0)→"M.E.Senko"].

In WebOQL, the tree schema for HTML pages is predefined such that every tree node has attributes Tag *(containing the outermost tag),* Source *(containing the complete subtree), and*

Text *(containing the plain text of the subtree), e.g.,*

browse("journals/is/is1").Tag = "html" ,
browse("journals/is/is1").Source = "<html><head>...</body></html>" ,
browse("journals/is/is1").Text = "Information Systems Vol.1 ..."

Navigation in WebOQL has to be done by navigation patterns *which are regular expressions that specify the structure of the paths to be followed in the parse-tree, e.g.,*

select x.text
from x in browse("journals/is/is1")
via [Tag="HTML"][Tag="BODY"][Tag="UL"][Tag="LI"][Tag="A"]

Since wildcards can be used (e.g., $\wedge$ for internal (i.e., parse-tree) arcs, and $>$ for external (e.g., hyperlinks) arcs), general path expressions *can be expressed in WebOQL queries. In the case of* FLORID*, we have shown in [LHL+ 98] how general path expressions can be expressed via F-Logic rules.*

Similar to OQL, WebOQL is a purely functional language with additional operators for manipulating hypertrees; the results of WebOQL queries are always hypertrees. In contrast, FLORID queries primarily result in variable bindings that can directly be used in rules to generate arbitrary F-Logic models.

W3QL/W3QS [KS95, KS98b] is another early Web querying language/system. The basic data model is rather simple and contains only the Web structure: The Web is mapped to a graph consisting of nodes (Web documents) and links (hyperlinks). For HTML documents themselves, only specific tags are evaluated, such as title, title.content, content-length, anchor[i] (the ith outgoing hyperlink) – (similar to those which are extracted automatically by FLORID's url.get method). The syntax of W3QL is similar to SQL, with the `FROM` clause augmented to handle paths in the Web graph, and the `WHERE` clause augmented to state conditions on the variables denoting Web pages and hyperlinks. In

```
SELECT ... FROM n1, l1, n2 WHERE ...
```

n1 and n2 have to be bound with urls, and l1 has to be bound with a hyperlink from n1 to n2. Special types of general path expressions are also allowed in the `FROM` clause. The `WHERE` clause must contain a *domain condition* of the form n1 in <set-of-urls>, specifying a finite set of urls to be the roots of the search. In contrast to the other described approaches, all other conditions in the `WHERE` clause (concerning the above-mentioned document properties) are not evaluated against an internal data model, but by external W3QL-specific programs (e.g., `SQLcond` and `Perlcond`):

```
WHERE n1,n2: SQLCOND '(n1.format = HTML) AND
                      (n2.title = "Information Systems")'
WHERE l3,n4: PERLCOND
     '(l3.content =~ /href\s*=\s*"(.*)"/i) &&
      (n4.content =~ /$1/)'
```

Thus, in W3QL/W3QS, the facilities for querying the document structure are very restricted. As a special feature, W3QS has special functionality to handle forms/CGI scripts: a DOF (Dictionary of forms) is maintained which contains known forms and their parameters. A special interface is provided to specify arguments to be given to forms using the `GET` method in W3QS queries, e.g.,

```
FILL n1 AS author-search-form WITH name = "M.E.Senko" .
```

If W3QS encounters a form which is not yet known, an interactive program `learnform` is started and the user is requested to provide appropriate data.

In contrast to the other Web querying approaches, W3QS has a non-integrated architecture using external tools. Thus, it can be easily extended to additional file types (LaTeX, BibTeX, postscript), and to more sophisticated condition-evaluating programs; also an integration of forms using the `POST` method in this way seems to be possible.

A future extension of W3QS/W3QL [KS95] to a hierarchical graph model of the Web (HTML and XML), similar to our approach, consisting of the sites, Web pages, and intra-page structuring, is proposed in [KS98a].

WebLog [LSS96] is a deductive Web querying language operating in an integrated graph-based framework. Although its syntax resembles F-Logic [KLW95], it is not fully object-oriented; the only objects are *"rel-infons"* which simulate in some sense the nodes of the parse-tree. Using rel-infons, atomic formulas describing HTML pages are of the form

<url>[<rel-infon-id>: <attr>→<val>] ,

that describe properties of a certain rel-infon on a Web page. Although, the data model does not imply a direct mapping of parse-trees to rel-infons – the actual mapping is left to the user. It it is left open whether or not this can be done in a generic way. Built-in predicates hlink and htext, define the semantics of hyperlinks. Thus, the actual data model is somewhere between the Web skeleton and the extended Web skeleton. Navigation along hyperlinks is provided by the above built-ins, but navigation in the parse-trees is not supported. Since the parse-tree is only partially represented, the evaluation of pages is mainly based on substring matching. There is no reported implementation of WebLog.

The above Web querying languages do not deal with the semantical analysis of document contents, wrapping, or restructuring of information. Although, especially in WebOQL, wrapping in a similar style as in our approach by generic queries should be possible.

Semistructured data. Research in Web querying and Web data extraction (especially, the data models and query languages) is also influenced from research on semistructured data.

The **Lorel** language [AQM+97] for querying semistructured data uses the graph-based OEM data model (Object Exchange Model, originally from the Tsimmis project) that mainly consists of objects and subobjects without any fixed schema. In OEM, data is represented in a graph whose nodes are objects, and whose edges are labeled with attribute names. Complex objects are simply collections of objects; leaf nodes have atomic values.

Example 18 (OEM) *An excerpt of the* Mondial *database in OEM looks as follows:*

<&cont1, continent, set, {&a1, &n1, &c1, &c2, &c3, ...}>
<&n1, name, string, 'Europe'>
<&a1, area, number, 9562488>
<&c1, country, set, {cn1, cc1, ca1, cp1, cap1, ctys1, n1}>
<&c2, country, set, {cn2, cc2, ca2, cp2, cap2, ctys2, n2}>

<&cn1, name, string, 'Germany'>
<&cc1, code, string, 'D'>
<&ca1, area, number, 356910>
<&cp1, population, number, 83536115>
<&cap1, name, string, 'Berlin'>
<&ctys1, cities, set, {cty1, cty2, ...}>
<&n1, neighbor, set, {c2, c3, ...}>

<&cn2, name, string, 'Belgium'>

The Lorel language provides SQL/OQL-like constructs, extended by powerful *general path expressions* which are regular expressions for characterizing paths in an OEM model. E.g., the following query selects all cities in countries which are reachable by land from Germany:

```
select X.city.name
from Mondial.country X, Mondial.country.Y
where Y.name = 'Germany'
  and Y.(border)+ = X
```

Lorel is implemented in the Lore system. In contrast to Web query languages/systems, Lore is not tailored to deal with Web pages – the OEM data is assumed to be provided by source-specific wrappers that can be implemented in any language providing an OEM interface, especially, the TSIMMIS (described below) wrappers can be used.

A similar light-weight data model is used in [AV97], with the main difference that they the children of each vertex are ordered, which is often necessary in data exchange formats. Integration based on (re)construction of data forests is done by declarative rules, where *tree terms* – similar to F-Logic molecules can be used.

The **YATL** language of the **YAT** system [CDSS99] is a rule-based querying and transformation language for XML/SGML wich is also based on a tree representation. Generic trees can be instantiated to be used in mediator programs. The Web structure is not modeled, and the relevant part of the Web cannot be extended at run-time.

UnQL [BDHS96] also uses a graph-based model for semistructured data and defines a language for navigating and querying it. UnQL models can be restricted by *graph schemata* that constrain the allowed paths in the graph, An important application of this is query planning and optimization. UnQL queries also have an OQL-like syntax of the form `SELECT ... WHERE ...`; instead of a `FROM` clause, references from the root node and other distinguished nodes are given in the `SELECT` part. Additionally, *restructuring queries* to conctruct new trees are possible. Here, *structural recursion* is used. In contrast to the classical `SELECT ... FROM ... WHERE`, this allows to express queries and data translations also in the presence of cycles in the graphs.

Query languages for semistructured data, such as UnQL and Lorel, do not support ordered collections, which is often crucial when working with Web pages.

Web site maintenance and restructuring. Web querying languages are also a prominent component of website maintenance and restructuring systems. The structure of a Web site is exploited to define views from which new Web pages are generated. In the **Strudel** system [FFK+98], the Web is also mapped to a unified internal, OEM-like graph representation. The language STRUQL is used both as a query and a transformation language on this graph model. The STRUQL language is based on a `WHERE ... COLLECT ...` statement. The `WHERE` part specifies conditions that determine which objects will be selected. Similar to Lorel and UnQL, regular path expressions are allowed. During the evaluation, the `WHERE` clause generates all variable bindings. The `COLLECT` clause places the results in collections for further use (or returns them as the result of a query). The syntax is extended for Web page generation with `CREATE` and `LINK` commands which generate nodes and links from the variable bindings. Wrappers to the internal Strudel graph model have to be provided independently. As a non-graph based approach, the **Araneus** project [AMM97] uses a hypertext-based model for Web site restructuring. This system uses different languages to extract data and define views on it.

Data integration. Most systems used to extract information follow the layered approach, using separate programs for wrapping (often, an individual wrapper is designed for every source) and mediation. In these layered architectures, the modeling of the topology of the Web is separated from its contents. The projects described in the following implement the virtual approach and use SQL-like query languages:

In the **Tsimmis** project [GMPQ+97], the graph-based *OEM* (Object Exchange Model) is used as a common data model for the extracted, application-level data. In TSIMMIS, the Web structure and the page markup is not modeled. Instead, it is presumed that external *wrappers* are given that map Web pages to the OEM representation. Different languages are used for wrapper and mediator specification (MSL and WSL, respectively), and for querying

(MSL and Lorel). In the wrapper specification language WSL, rules specify which action has to be executed on the resource for a given query, e.g., (possibly using external resources) to yield OEM atoms. E.g., for a relational database or for a query form, rules look as follows:

```
C :- <C country {<code $CODE>}>
     //$$ := "select name
              from country
              where code=$CODE"//

C :- <C country {<code $CODE>}>
     //$$ := "find-code $CODE"//
```

For ASCII or HTML sources, an additional external, non-WSL wrapper layer (e.g., by perl scripts) would be necessary. MSL maps OEM data to OEM views, similar to F-Logic rules. E.g., all cities which are stored as capital (names) in *CIA*, and are also stored in *GlobalStatistics*, are collected by the MSL rule

```
<capital {<name Cap> <country CN> R }>@med :-
<country {<name CN> <capital Cap>}>@cia
AND <country {<name CN> <city {<name Cap> | R}>}>@gs
```

The MedMaker tool [PGMU96] implements MSL as part of the Tsimmis project. [HGMC$^+$97] describe a grammar-based tool for coding wrappers producing OEM output.

In the **Ariadne** project, e.g., [AK97], the Web – i.e., the Web structure and the page markup – is not modeled explicitly. Instead, for every relevant Web page, a grammar to wrap it into the target model is derived. Their approach does not support HTML tables or any textual formatting capabilities – which is a common way for structuring data on Web pages. Thus, the extraction of data as shown in Section 9 is not possible with their wrappers.

Jedi [HFAN98] is a tool to manually specify Web access and wrappers for HTML pages by combining grammars and rules. **W4F** [SA99] is a toolkit for interactively generating wrappers for HTML pages using HEL, a DOM-based language operating on the parse-tree of a document. Here, comma-lists and name-value pairs are regarded as atomic – i.e., the result is not completely mapped to the application domain, requiring a manually specified application-specific postprocessing. HEL focusses on wrapping single Web pages; Web exploration and information integration is left to the application. Another tool for interactive generation of wrappers using the DOM model is presented in **NoDoSE** [Ade98]. [KWD97] present inductive learning methods for wrapping HTML pages with a tabular layout and report a 48% success. [ECY$^+$98] report higher than 90% success for automatical matching-based wrapper generation for *data-rich and ontological narrow* sources.

This shows that automatic wrapper generation is still in its infancy. Indeed, given the many idiosyncrasies of real-world sources, it seems more viable to employ high-level declarative languages for rapid prototyping and refinement of wrappers. In particular, our approach allows the combination of reusable *generic rule patterns* with application-specific rules for interpreting the outcome of these generic patterns. Additional refining rules can easily be added by the user.

Most of the above models focus either on the *Web representation* of information (UnQL, Strudel), neglecting the application semantics of the information, or on Web access and wrapping only (Jedi, W4F), or on mediation (Tsimmis/OEM), leaving the mapping from the Web representation to the data model to external wrappers. The above approaches do not deal with Web exploration.
For a complete overview of existing systems, see [FLM98].

XML querying languages. Recently, languages for querying XML documents (extensible stylesheet language, [XML98]) came up. XML allows to define tags with an application-level meaning (cf. Figure 14). There is not yet a standard for an XML query language, but several querying languages have been proposed; not only for general semistructured data, but also specially tailored towards XML.

XQL [Rob] is a subset of the stylesheet language XSL [XSL98a]. The result of an XQL query is always another XML document containing the selected XML nodes. *Directory expressions* are used to navigate in the hierarchical schema of an XML database. For the XML database given in Figure 14, the query

mondial/country

generates an XML file containing all country elements; // is used for arbitrarily deep search in the hierarchy:

mondial//city

yields all cities in the levels country/city, country/province/city, country/region/departement/city, and so on.

By using a *filter*, conditions can be evaluated; the directory expression selects the value to be contained in the result (similar to F-Logic's *reference semantics*), and the filter states a condition to be evaluated (note the difference between (4) and (5) in the following expressions):

1. mondial/country[population > 1000000] % all countries with more than 1 million inhabitants
2. mondial/country[@car_code = "D"] % the country whose car code is "D"
3. mondial/country[@name = "Germany"]//city % all cities in Germany
4. mondial//city[@name = "Cordoba"] % all cities whose name is Cordoba
5. mondial/country[.//city/@name = "Cordoba"] % all countries where exists a city "Cordoba"

Filters can also be used to express semi-joins, i.e., the element to be selected can be compared to another element: The query

mondial/country[@name = //country//city/name]

selects all countries that have the same name as some city (possibly in another country). It is *not* possible to combine information from different paths through the document in the result. Also, it is not possible to restructure nodes before addding them to the resulting XML document. For a querying and data manipulation language, XQL suffers from serious restrictions which are mainly due to the fact that XQL has no notion of variables or variable bindings.

Another approach is followed by **XML-QL** [DFF+99]. Again, the result of an XML-QL query is an XML document. In contrast to XQL, XML-QL uses variables and variable bindings that allow for joins and reorganisation of data. The selection and querying construct contains an XML pattern that has to be matched against the XML document. Variables are used to bind values that generate the result.

```
WHERE <city name = $n>
        <longitude> $long </longitude>
        <latitude> $lat </latitude>
      </city> IN "www.xxx.de/mondial.xml"
CONSTRUCT <result>
    <city name =$n long=$long lat=$lat/>
```

Variables can also be used for joins:

```
WHERE <country name = $cn1>
       <province name = $pn1>
         <city name = $cty>
         </city>
       </province>
      </country> IN "www.xxx.de/mondial.xml"
```

```
        <country name = $cn2>
         <province name = $pn1>
           <city name = $cty>
           </city>
         </province>
        </country> IN "www.xxx.de/mondial.xml"
        not $cn1 = $cn2
  CONSTRUCT <result>
      <pair country1 = $cn1 countr2 = cn2 name =$n/>
```

selects all pairs of cities in different countries which have the same name. This query is not possible in XQL.

Advanced features in XML-QL allow *tag variables* (<$tagvar>), similar to variables at method positions in F-Logic and regular path expressions in tags; e.g., the following query relates every city to its country, independent of the depth of nesting in levels of administrative divisions:

```
  WHERE <country name = $cn>
         <*>
           <city name = $cty>
           </city> IN "www.xxx.de/mondial.xml"
  CONSTRUCT <result>
      <citycountry country = $cn city = $cty>
```

In [FSW99], XML-QL, YATL [CDSS99], and Lorel [AQM+97] are compared with XQL. This work showed that XML-QL, YATL, and Lorel are nearly equivalent (they all use variable bindings to generate a set of answers which are then formatted into a resulting XML document). XQL is significantly less expressive (as already noted above). These languages do not support navigation along XML reference attributes yet.

The XSL(T) language [XSL98b] provides transformation facilities for XML documents in a functional style. It provides iterative constructs, which can be used to specify joins; generating elements (with unique id) and following references is also supported.

Current work on Florid includes its application to XML data as query and transformation language using F-Logic as internal data model.

Information brokering. Brokerage of information from several sources is investigated in **InfoSleuth** [BBB+97], **Information Manifold** [LSK95], and **Observer** [MKSI96]. In Information Manifold, all data sources provide a site description which describe the sites' capabilities in terms of an object-relational common world model. The sources are collected a priori at design time of an application; the user then queries only the world model. The InfoSleuth architecture is based on agents representing data sources and broker services. Sources become accessible when their agent is registered. Here, source agents contain site descriptions. Similarly, in Observer, the content of each data repository is described by an ontology. Thus, all these approaches presume given site descriptions or source agents for each accessible data source – i.e., they cannot be used for exploring the Web itself. Furthermore, the Web structure is not part of their world model.

14 Conclusion

In the typical layered wrapper-mediator architecture, each wrapper is associated with a data source. Towards the mediator, it provides an interface for exchanging application-level information. The Web structure (both the Web skeleton, and the parse-trees of individual Web pages) is *not* part of the system's world model. The network structure of the wrappers and mediators has to be designed for the relevant Web fragment – which has to be known a-priori.

Instead, our approach is an integrated system incorporating a generic Web interface, so that the Web can be seen as a part of the system, i.e., arbitrary Web pages can be accessed at runtime. Using an integrated internal world model that includes the Web structure, there is no gap between wrappers and mediators. Rules implementing Web exploration, wrapping, and mediating functionality can be defined which operate on the Web skeleton, its hyperlinks, the page HTML/XML sources and their parse-trees, and on the application-level model itself.

We have presented an integrated data model for representing a fragment of the Web and its contents, and for extracting information from this fragment. The link structure (skeleton) of the Web is extended by the parse-trees of SGML-structured Web pages to be a graph model of the Web. The model is then extended to contain an application-level object-oriented model. The unified approach allows us to use the same data manipulation and querying language for querying the Web structure and the application-level model. Here, we apply a rule-based object-oriented language which is extended by Web access capabilities and allows for navigation in the unified model.

We have shown a methodology for extracting and integrating information from the Web. Based on an integrated rule-based language for wrapping and mediation, it allows reuse of generic rule patterns for several applications. By instantiating generic rule patterns to handle logical markup and textual formatting and typical situations of data integration, a program skeleton is assembled which can be then completed by application-specific rules and refining rules. A comparison with (semi-)automatic approaches for wrapper-generation shows that these do not provide (yet) a sufficiently fine granularity to exhaustively extract information from semistructured sources – especially if these use textual formatting. Here, a combination of automatical pre-generation of wrapper skeletons, later manually refined, could be useful. This approach has been implemented in FLORID.

In addition to the classical wrapping and mediation functionality, a system using this architecture can be equipped with *Web navigation* and *exploration* functionality. Therefore, Web pages not known a priori can be detected to contain potentially relevant information. Since wrapper rules can be written on a generic, markup level, such pages can also be processed using suitable heuristics: The generic rule schemata form a universal wrapper which can be applied to arbitrary Web pages when they become known to the system during information processing. This can be interpreted as a dynamical generation and configuration of *agents* inside the main system as follows: rules that apply to an information source automagically form the (wrapper) agent for this source. Equipped with suitably intelligent rules, the system – with its virtual agents – can potentially explore unknown parts of the Web, thus coping with the steady growth of the Web. Therefore, the architecture also allows the implementation of data-driven, hyperlink-oriented Web search engines that can also extract and process semantical information.

Accessing (well-written) XML pages, the impedance mismatch between the extended Web skeleton and the application-level model is reduced: semantic tags directly induce methods that represent relationships in the object-oriented model, making the wrapping task much easier. Even if XML becomes more common, HTML pages will still exist. Often, data which is internally represented is available to non-privileged users only via (XSL/CSS-)script-generated HTML pages. Thus, for spontaneous information extraction, wrapping of HTML sources is expected to remain an important topic.

Current work on FLORID includes its application to XML data as query and transformation language, using F-Logic as internal data model.

Acknowledgements. We want to thank the former team members Rainer Himmeröder, Paul-Thomas Kandzia, Bertram Ludäscher, and Christian Schlepphorst for the fruitful cooperation during the project.

References

[Abi97] S. Abiteboul. Querying Semi-Structured Data. In *Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 1–18. Springer, 1997.

[Ade98] B. Adelberg. NoDoSE – A Tool for Semi-Automatically Extracting Semi-Structured Data from Text Documents. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 283–294, Seattle, Washington, 1998.

[AK97] N. Ashish and C. A. Knoblock. Wrapper Generation for Semi-structured Internet Sources. *SIGMOD Record*, 26(4):8–15, 1997.

[Alt98] AltaVista, 1998. `http://www.altavista.com/`.

[AM98] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Intl. Conference on Data Engineering (ICDE)*, 1998.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *Intl. Conference on Very Large Data Bases (VLDB)*, 1997.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries (JODL)*, 1(1):68–88, 1997.

[AV97] S. Abiteboul and V. Vianu. Queries and Computation on the Web. In *Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, pp. 262–275. Springer, 1997.

[BBB+97] R. Bayardo, W. Bohrer, R. Brice, A. Cichocki, G. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk:. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrandt, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM Intl. Conference on Management of Data (SIGMOD)*, pp. 505–516, Montreal, Canada, 1996.

[Bun97] P. Buneman. Semistructured Data (invited tutorial). In *ACM Symposium on Principles of Database Systems (PODS)*, pp. 117–121, Tucson, Arizona, 1997.

[CDSS99] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators need Data Conversion. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1999.

[CIA] CIA World Factbook. `http://www.odci.gov/cia/publications/factbook/index.html`.

[Cla] J. Clark. SP: An SGML System Conforming to International Standard ISO 8879 – Standard Generalized Markup Language. `http://www.jclark.com/sp`.

[DBL98] DBLP Computer Science Bibliography. `http://www.informatik.uni-trier.de/~ley/db/`, 1998.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML. In *8th. WWW Conference*. W3C, 1999. W3C note, `www.w3.org/TR/NOTE-xml-ql`.

[ECY+98] D. Embley, D. Campbell, Y. Yiang, Y.-K. Ng, and R. Smith. A Conceptual-Modeling Approach to Extracting Data from the Web. In *Intl. Conf. on the Entity Relationship Approach (ER)*, number 1507 in LNCS. Springer, 1998.

[FFK+98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the Boat with Strudel: Experiences with a Web-Site Management System. In *ACM Intl. Conference on Management of Data (SIGMOD)*, 1998.

[FHK+97] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A Prototype for F–Logic. In *Intl. Conference on Data Engineering*

(ICDE), 1997.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3), September 1998.

[FLU94] J. Frohn, G. Lausen, and H. Uphoff. Access to Objects by Path Expressions and Rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 273–284, 1994.

[FSW99] M. Fernandez, J. Siméon, and P. Wadler. XML Query Languages: Experiences and Exemplars. draft manuscript, communication to the XML Query W3C Working Group., 1999. `http://www-db.research.bell-labs.com/user/simeon/xquery.ps`.

[Gen91] M. R. Genesereth. Knowledge Interchange Format. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pp. 599–600, San Mateo, CA, USA, April 1991. Morgan Kaufmann Publishers.

[Glo] `http://www.stats.demon.nl/`.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2), 1997.

[HFAN98] G. Huck, P. Fankhauser, K. Aberer, and E. J. Neuhold. Jedi: Extracting and Synthesizing Information from the Web. In *Intl. Conference on Cooperative Information Systems (CoopIS)*, pp. 32–43. IEEE Computer Science Press, 1998.

[HGMC+97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, and A. Crespo. Extracting Semistructured Information from the Web. In *International Workshop on the Web and Databases (WebDB)*, pp. 18–25, 1997.

[HKL+98] R. Himmeröder, P.-T. Kandzia, B. Ludäscher, W. May, and G. Lausen. Search, Analysis, and Integration of Web Documents: A Case Study with FLORID. In *Proc. Intl. Workshop on Deductive Databases and Logic Programming (DDLP'98)*, pp. 47–57, Manchester, UK, 1998.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, 1995.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World-Wide Web. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 54–65, Zürich, Switzerland, 1995.

[KS98a] D. Konopnicki and O. Shmueli. Bringing Database Functionality to the WWW. In *International Workshop on the Web and Databases (WebDB)*, pp. 49–55, 1998.

[KS98b] D. Konopnicki and O. Shmueli. Information Gathering in the WWW: The W3QL Query Language and the W3QS system. *ACM Transactions on Database Systems (TODS)*, 23(4):369–410, 1998.

[KWD97] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Intl. Joint Conference on Artificial Intelligence*, 1997.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8):589–612, 1998.

[LSK95] A. Levy, D. Srivastava, and T. Kirk. Data Model and Query Evaluation in Global Information Systems. *Journal of Intelligent Information Systems*, 2(5), 1995.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. A Declarative Language for Querying and Restructuring the Web. In *Proc. 6th Intl. Workshop on Research Issues in Data Engineering (RIDE)*, 1996.

[May99a] W. May. Information Extraction and Integration with FLORID: The MONDIAL Case Study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[May99b] W. May. Modeling and Querying Structure and Contents of the Web. In *Workshop on Internet Data Modeling (IDM'99), Proc. DEXA 99 Workshops*, pp. 721–725. IEEE Computer Science Press, 1999.

[May00] W. May. An Integrated Architecture for Exploring, Wrapping, Mediating and Restructuring Information from the Web. In *Australasian Database Conference (ADC 2000)*. Australian Computer Science Communications, Vol. 2, No. 2, IEEE CS Press, pp. 82-89, 2000.

[MHLL99] W. May, R. Himmeröder, G. Lausen, and B. Ludäscher. A Unified Framework for Wrapping, Mediating and Restructuring Information from the Web. In *International Workshop on the World-Wide Web and Conceptual Modeling (WWWCM)*, number 1727 in LNCS, pp. 307–320, 1999.

[MKSI96] E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *International Conference on Cooperative Information Systems (CoopIS '96)*, 1996.

[MMM97] A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the World Wide Web. *Intl. Journal on Digital Libraries (JODL)*, 1(1), 1997.

[PGMU96] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Intl. Conference on Data Engineering (ICDE)*, pp. 132–141, 1996.

[Rob] J. Robie. XML Query Language (XQL) FAQ. `http://metalab.unc.edu/xql/`.

[SA99] A. Sahuguet and F. Azavant. Looking at the Web through XML glasses. In *Intl. Conference on Cooperative Information Systems (CoopIS)*. IEEE Computer Science Press, 1999.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 1992.

[XML98] Extensible Markup Language (XML). `http://www.w3.org/XML/`, 1998.

[XSL98a] Extensible Stylesheet Language (XSL). `http://www.w3.org/Style/XSL/`, 1998.

[XSL98b] XSL Transformations (XSLT). `http://www.w3.org/TR/1999/PR-xslt-19991008`, 1998.