View Definition and DTD Inference for XML

Bertram Ludäscher Yannis Papakonstantinou Pavel Velikhov Victor Vianu

`ludaesch@sdsc.edu` `{yannis,pvelikho,vianu}@cs.ucsd.edu`

1 Introduction

We present a query language and mediator architecture for XML data [XML98b]. The query language, called XMAS (***X**ML **M**atching **A**nd **S**tructuring Language*) uses vertical and horizontal navigation. The horizontal navigation provides a powerful novel mechanism for using the order in the semistructured model underlying XML. This couples nicely with the well-known regular path expressions for vertical navigation [Suc, Abi97, Bun97, AQM^{+}97, FFLS98, AM98, CDSS98, XML98a, KS95]. XMAS also provides versatile mechanisms for constructing XML documents as answers to queries, including powerful `group-by` and `order-by` constructs. We briefly describe a prototype for an XML-based information mediation system, built around XMAS.[1]

DTDs can have multiple uses in creating integrated views and querying XML data. Our prototype uses a QBE-style query interfaces driven by the view DTD. The mediator will use DTDs to optimize the queries it sends to the sources. Finally DTDs may guide the production of style sheets, such as XSL scripts [XSL98], that display XML documents as browser-friendly HTML documents. DTDs may also help in the design of the storage structures. Thus, it is clear that DTDs of integrated views will be particularly useful. However, creating a view DTD "manually" by delving into the details of the source DTDs is error-prone and may become the bottleneck of the integration software development, compromising much of the advantage of semi-automatic data integration. A central component of our mediator is a DTD inference module.

After outlining the architecture of the mediator and illustrating XMAS, we present work in progress and results (submitted for publication [PV98]) on DTD inference in views defined by XMAS queries. We point out several strong limitations of DTDs and the need for extending them with (i) a subtyping mechanism and (ii) a more powerful specification mechanism than regular languages, such as context-free languages. With these extensions, we show that one can always infer *tight* DTDs for views defined by *selection queries*, which extract a list of elements from the input. The tight DTD precisely characterizes the documents in the view on sources satisfying given DTDs. We also show interesting special cases where we can derive a tight DTD without requiring the extension with subtyping. Finally we consider related problems such as checking conformance of a view definition to a predefined DTD.

2 XML-Based Information Mediation

We begin with an overview of the architecture of the prototype and the underlying XML query language XMAS. The description follows a query from formulation to processing.

Prototype Architecture The mediator architecture is depicted in Fig. 1: BBQ[2] is a DTD-driven user interface which allows to construct queries in an intuitive graphical way. A design goal of BBQ

[1] The prototype should be fully operational around Christmas.
[2] ***B**lended **B**rowsing and **Q**uerying*

is to provide a seamless blend of browsing and querying modes, in the spirit of GARLIC's PESTO interface [CHMW96]. Fig. 2 shows a snapshot of query composition with BBQ: The *condition* and *answer windows* are used for defining the tree templates of the *head* and the *body* of the generated XMAS query, respectively.

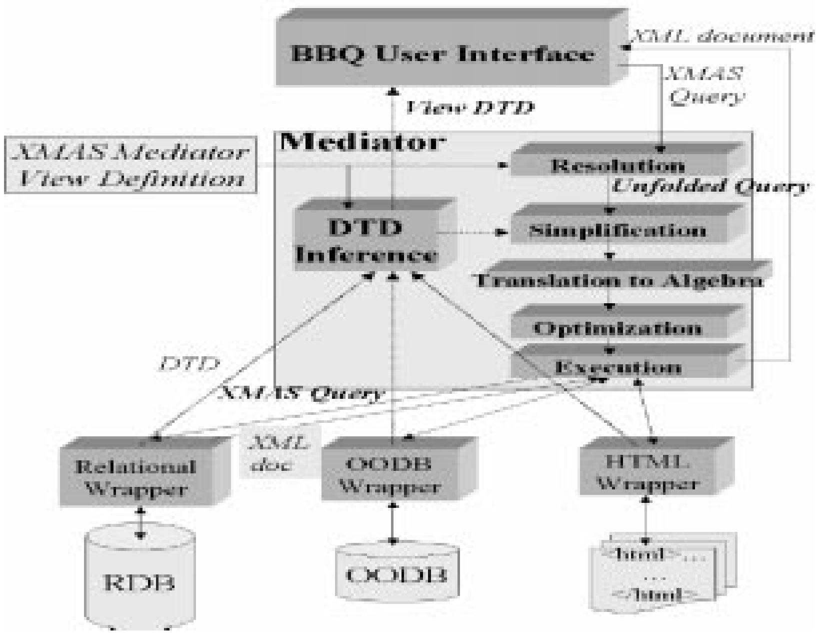


Figure 1: Mediator architecture

Once the condition of the query has been defined, the query output can be constructed by dragging subtrees from the condition window and dropping them into the answer window. Moreover, the output can be restructured and new element names can be created. A special feature of BBQ is the `COLLECT` operator (on the right in Fig. 2), which allows to create data collections simply by clicking on the corresponding element subtree. In this way, all elements of that type which are retrieved by the query body will appear as a *collection list* in the answer. The crux of this feature is that it induces an *implicit grouping semantics*. By default, elements of a collection are returned in the order they were found, but other element orders can be easily defined, e.g., based on attribute values or by applying list functions to them.

The mediator comprises several modules to accomplish the integration; its main inputs are XMAS queries generated by BBQ, and the mediator view definition (also in XMAS) for the integrated view. The latter is provided by the mediator administrator, and prescribes how the integrated data combines the wrapper views. The resolution module resolves the user query with the mediator view definition, resulting in a set of unfolded XML queries that refer to the wrapper views. These queries can be further simplified based on the underlying XML DTDs.

A central component is the DTD inference module: It allows to derive view DTDs from source DTDs and view definitions, thereby supporting the integration task of the mediator administrator.[3]

[3]In the mediator, DTD inference is performed in a separate off-line step, i.e., before the user interacts with the mediator.

The translation module maps the simplified queries into the XMAS algebra which can then be further optimized. Finally, the execution engine issues XMAS queries against the wrappers, and returns the requested XML data to the user, after integrating the retrieved data according to the mediator view.

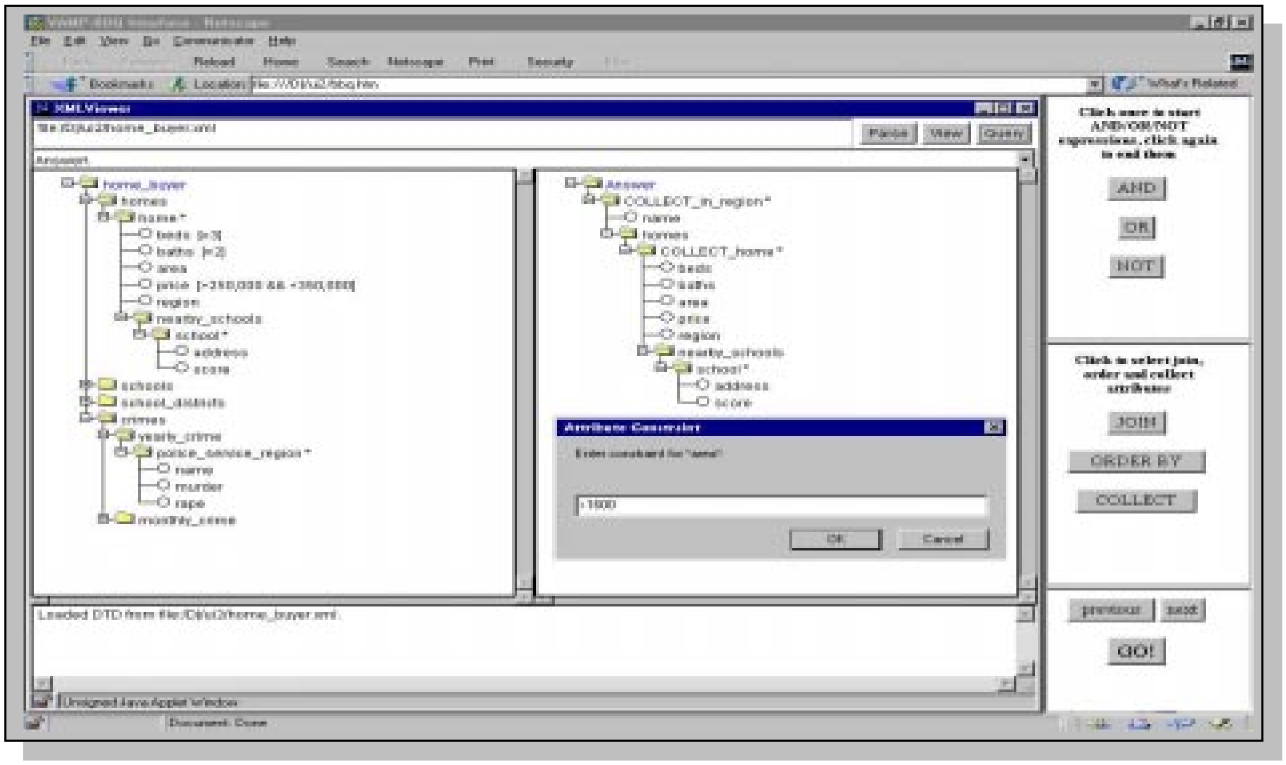


Figure 2: XML query composition with BBQ
(left: *condition window*, middle: *answer window*, right: *operators*)

2.1 The XMAS Query Language

The basic format of a XMAS query is follows:

query	=	`CONSTRUCT` *head* `WHERE` *body*
body	=	*template* `IN` *source*
	|	(*body op body*)
	|	*predicate*
op	=	`AND` | `OR` | `FUSE`

Here, *template* is an XML tree template for extracting data found in *source*. The *predicate* may specify conditions on the variables occurring in the body. Variable bindings returned by the body are used to construct the output XML document: The *head* defines the details of this construction, most notably, grouping and ordering of elements (see below).

Example 2.1 *Consider an XML bibliography database (adapted from [XML98a]), where the structure of a* `book` *element is defined by* `<!ELEMENT book (author+, title, publisher)>`*. Assume that we* *want to collect the titles of all books published by Addison-Wesley, and for each title $T we want to collect all authors $A. In* XMAS *this is expressed as follows:*

```
CONSTRUCT <answers>
              [ <addison_wesley_book> $T
                                      [ $A ]
                 </addison_wesley_book> ]
          </answers>
WHERE <book>
          $T: <title/>
          $A: <author/>
          <publisher name="Addison-Wesley"/>
      </book>  IN "www.somewhere.net/bib.xml"
```

The expression in the WHERE-clause serves as a template for matching data in the XML input: We are looking for book elements having at least a title, author, and publisher subelement. No constraints are imposed on their internal structure; we only require that the name attribute of publisher is "Addison Wesley". The variables $T and $A are bound to the corresponding subelements.

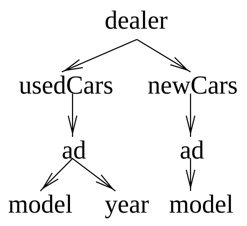
The expression in the CONSTRUCT-clause defines the resulting XML document, i.e., an answers element with addison_wesley_book subelements. The subelements of the latter correspond to the bindings to $T and $A, in this case title and author elements. The bracketed expressions [expr *] are a special feature of* XMAS *called* collection lists *and may occur in the head only. They provide a simple and intuitive means for specifying collections, i.e., lists of elements: Within the outermost answers element, we have a collection of addison_wesley_books, each of which contains a title element (the binding of $T), followed by a collection of authors. See the appendix for a more complex* XMAS *query that corresponds to the query constructed in Fig. 2.*

3 Inferring DTDs for XMAS Views

To develop the DTD algorithm, we use in [PV98] a simplified formal framework that focuses on document *structure*. An XML document is modeled as a labeled ordered tree, called a *loto*[4]. Nodes correspond to XML elements and their labels provide the type names of the elements. The children of a node are totally ordered.

Example 3.1 *Consider the following "dealer" XML document and the corresponding loto.*

```
<dealer>
 <UsedCars>
  <ad>
   <model>Honda</model>
   <year>92</year>
  </ad>
 </UsedCars>
 <NewCars>
  <ad>
   <model>BMW</model>
  </ad>
 </NewCars>
</dealer
```



(L3.2)

[4]Loto stands for "labeled ordered tree object".

Note that the loto retains the element name structure of the XML document. However the string content of the document is discarded since it plays no role in the inference problem.

A DTD is modeled as a *loto type definition* (ltd), that associates with each type name a language on the alphabet of type names, as illustrated next.

Example 3.3 *The loto (L3.2) satisfies the ltd below. For readability, in ltd examples we denote concatenation by comma. The examples specify the root type and the languages associated to each type name.*

$$\text{(LTD3.4)} \quad \begin{array}{l} root : dealer; \\ dealer : (UsedCars, NewCars); \\ UsedCars : ad^*; \\ NewCars : ad^*; \\ ad : (model, year) + model; \end{array}$$

To study DTD inference for views we introduce in [PV98] a formal view definition language that abstracts the core of the XMAS language. The language queries and constructs labeled ordered trees. A query extracts variable bindings from the input using a tree pattern involving *regular expressions* to navigate both vertically and horizontally in the tree. Horizontal regular expressions provide a powerful way to query the order of the nodes. The variable bindings extracted by the tree pattern are used to construct an answer document. Like XMAS, the language has a powerful group-by feature allowing to construct nested lists (details are omitted).

Given a source ltd and a view definition, we study the problem of constructing a *tight* ltd for the view, i.e., an ltd that precisely characterizes the type structure of trees in the view. Clearly, tight ltds are most desirable, as they provide the maximum information on the structure of documents in the view.

Our quest for the tight ltd quickly highlights two severe limitations of current DTDs in XML. The first is that DTDs lack a subtyping mechanism; the repercussions are numerous. For example, there is typically no tight DTD for a single document, such as the one in Example 3.1. Neither is there a tight DTD for the set of documents from two sources, each with their own DTD, as illustrated next.

Example 3.5 *Consider the two sets of lotos defined by the following ltds.*

$$\left[\begin{array}{l} root : UsedCars; \\ UsedCars : ad*; \\ ad : model, year; \end{array}\right] \qquad \left[\begin{array}{l} root : NewCars; \\ NewCars : ad*; \\ ad : model; \end{array}\right]$$

Now consider the "union" of documents satisfying the UsedCars and NewCars lotos. The tightest ltd for the union (i.e. the most restrictive ltd to which the documents conform) is listed below; but it is not tight, as it clearly allows documents not in the union.

$$\begin{array}{l} root : result; \;\; result : UsedCars + NewCars; \\ UsedCars : ad^*; \;\; NewCars : ad^* \\ ad : (model, year) + model; \end{array}$$

The following illustrates again the shortcomings of ltds in describing views, due to the lack of subtyping.

Example 3.6 *Consider the following source ltd and a view that collects all dealers that sell at least one used car and groups them under a "used-car-dealers" node. The tightest ltd for the view is identical with the source ltd — modulo renaming dealers to UsedCarDealers. Thus, the ltd cannot cxapture the fact that at least one used car dealer "ad" must be for a used car.*

$root : dealers;\ \ dealers : dealer^*;$
$dealer : ad^*;\ \ ad : UsedCarAd + NewCarAd;$

The shortcomings illustrated above have a common source. They are due to the inability of ltds (and DTDs) to carry typing information across multiple levels of the trees (lotos) they describe. Intuitively, overcoming this limitation requires the ability to define special cases of a given type. We do this by enhancing DTDs with a simple subtyping mechanism, called *specialization*, which is in the spirit of union types. Despite their simplicity, specialized DTDs encompass the expressive power of formalisms such as dataguides and graph schemas. We illustrate specialized ltds with an example.

Example 3.7 *The following specialized ltd is tight for the singleton set* $\{(L3.2)\}$.

$root : dealer;$
$dealer : (UsedCars, NewCars);$
$UsedCars : ad^{used};\ \ NewCars : ad^{new};$
$ad^{used} : model, year;\ \ ad^{new} : model;$

Informally, ad^{used} and ad^{new} are two specializations of ad: one for used car ads, and another for new car ads. A loto (over the un-specialized set of types) *satisfies* a specialized ltd if the types of the loto can be specialized so that the resulting loto satisfies the ltd. Checking satisfaction of a specialized ltd is less straightforward than satisfaction of standard ltds, and can be reduced to checking membership of a word in a context-free language.

A second important limitation of DTDs is that the languages they specify are restricted to regular languages. This is not sufficient to describe even very simple views:

Example 3.8 *Consider the set of lotos described by the following ltd.*

$root : section;$
$section : intro, section^*, conclusion;$

Now consider a query that collects all intro and conclusion nodes of a given loto and groups them under a root named result, in exactly the same order in which they appear in the input. It is easy to see that the type structure of the result cannot be described by a regular language (but it can be described by a context-free language).

The main result on ltd inference is that the extension of ltds with specialization and context-free languages suffices for selection queries. In [PV98] we show that one can effectively construct, from every source ltd and view definition by a selection query, a tight ltd for the view that uses context-free languages and specialization. We illustrate the technical issues involved in the inference algorithm using three examples from [PV98], included in the Appendix. The above result provides a solution to the tight ltd inference problem for selection queries, but comes at the cost of using the extended ltds. In some applications it is sufficient to provide ltds that are simply *sound* for the view, i.e. ltds that are satisfied by all trees in the view but may also allow trees that are not in the view. If one prefers to give up tightness in return for using regular ltds (corresponding to existing DTDs), there is good and bad news. The bad news is that it is undecidable if a view has a tightest regular ltd (i.e., a most restrictive regular ltd to which the view conforms). The good news comes in several flavors:

- It can be checked whether a view defined by a selection query conforms to a predefined regular ltd; this makes use of the tight specialized context-free ltd we can infer.

- Tight specialized regular ltds can be inferred for selection queries in several special cases of practical interest. For example, one case is when the source ltd is *stratified*, i.e. no type uses itself in the ltd directly or indirectly. Another is when the view is defined by a query involving only non-recursive vertical navigation. If

  specialization cannot be used, one can still infer in these cases a tightest regular ltd for the views.

- As a last resort, one can use heuristics driven by the semantics of the particular application to loosen the inferred tight specialized context-free ltd to a regular ltd to which the view conforms, with no guarantee of it being the tightest.

References

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

[AM98] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proc. ICDE Conf.*, 1998.

[AQM$^+$97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1), 1997.

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

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, 1998.

[CHMW96] M. J. Carey, L. M. Haas, V. Maganty, and J. H. Williams. PESTO: An Integrated Query/Browser for Object Databases. In *Intl. Conference on Very Large Data Bases (VLDB)*, pp. 203–214, 1996.

[FFLS98] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Catching the Boat with Strudel: experience with a Web-site Management System. In *Proc. ACM SIGMOD Conf.*, 1998.

[Gin66] S. Ginsburg. *The Mathematical Theory of Context-Free Languages.* McGraw-Hill Book Co., 1966.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[KS95] D. Konopnicki and O. Shmueli. W3QS: A Query System for the World Wide Web. In *Proc. VLDB Conf.*, pp. 54–65, Zürich, Switzerland, September 1995.

[PV98] Y. Papakonstantinou and V. Vianu. Type inference for views of semistructured data, 1998. Submitted.

[Suc] D. Suciu. An Overview of Semistructured Data. To appear in SIGACT News.

[XML98a] XML-QL: A Query Language for XML. W3C note, `http://www.w3.org/TR/NOTE-xml-ql`, 1998.

[XML98b] Extensible Markup Language (XML) 1.0. W3C recommendation, `http://www.w3.org/TR/REC-xml`, 1998.

[XSL98] Extensible Stylesheet Language (XSL). W3C working draft, `http://www.w3.org/TR/WD-xsl`, 1998.

Appendix

An Example XMAS Mediation Query

The following query is formulated using the BBQ interface and corresponds to Figure 2. It is taken from an actual "home buyer" mediation scenario, where the user wants to find certain houses in "good" regions (e.g., with high-ranked schools and low crime). The matching houses should be grouped by region and ordered by price:

```
CONSTRUCT <Answer> [ <in_region name=$R>
                        <homes> [ $H ] ORDERBY $H.price  </homes>
                     </in_region> ]
          </Answer>
WHERE <home_buyer>
          <homes> $H: <home region=$R beds=$BE baths=$BA area=$A price=$P>
                          <nearby_schools>
                              $S: <school score=$SC/>
                          </nearby_schools>
                      </home>
          </homes>
          <crimes>    <yearly_crime>
                          <police_service_region name=$R murder=$CM rape=$CRA/>
                      </yearly_crime>
          </crimes>
      </home_buyer> IN "www.sdsc.edu/VAMP/MED-VIEW"
  AND   $BE=3 AND $BA=2 AND $A>1600 AND $P>250000 AND $P<350000
  AND   $SC>=70 AND $CM+$CRA=<15 .
```

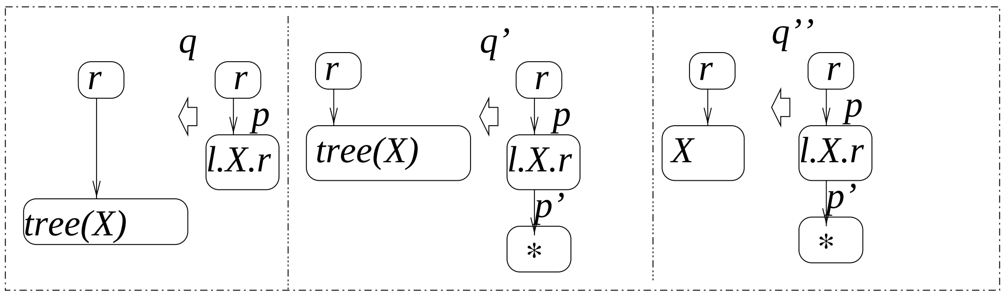
The `WHERE`-clause defines conditions on the home and crime data in the mediator view (e.g., the region `$R` of a given home `$H` must coincide with the name of the police service region of `crimes`, thereby defining a join).

The `CONSTRUCT`-clause defines the resulting XML document. Note the use of collection lists (of the form `[`*expr*`]`) in the head: The `Answer` element contains a list of `in_region` elements, each of which in turn contains a `home` collection which is ordered by the `price` attribute. XMAS allows to apply (built-in) ordering functions to act upon lists. If—like above—no ordering function is given, a default order is used (based on tree positions in the input). A collection induces an implicit *grouping* of data. Roughly speaking, the group-by variables (in the SQL sense) are those variables in the head which do *not* occur in *expr*. Thus, for each binding of these "surrounding" group-by variables, exactly one *expr*-list is generated. In our example, this means that only one `Answer` element is created (since no head variables are outside the outermost list), and that *for each* region `$R`, only *one* `homes` element containing a *list* of `home` elements is defined. We believe that this grouping semantics is very natural and intuitive (e.g., it avoids the use of Skolem functions for grouping).

Illustration of DTD Inference Algorithm[5]

We outline the ltd inference algorithm for three queries, q, q', and q'' which exemplify various aspects of the algorithm. We assume familiarity with basic notions of language theory, including (nondeterministic) finite-state automata ((n)fsa), context-free grammar (CF G) and language (CFL), homomorphism, substitution, and transducer (e.g., see [HU79, Gin66]). We will use basic facts such as closure of regular and context-free languages under homomorphism, inverse homomorphism, intersection with regular languages, substitution with languages of the same kind, and transducers.

[5]The following development is from [PV98].



In query q, the pattern of the body says that the parent of node X in the input loto is reachable from the root by a path spelling a word in a regular expression p. The node X is extracted from the content of the parent by matching the expression $l.X.r$ (l and r are regular expressions), so that X is the last letter of a word in l and it is followed by a suffix in r. The head of the query specifies a loto consisting of a root of type r. The children of r are the lotos rooted at the nodes that bind to X, in the order of occurrence in the input loto. For each binding of X to a node, the entire subtree $tree(X)$ rooted at that node is included in the answer. Query q' is the same as q, except that there must also exist a downward path in p' originating at X. Query q'' differs from q' in that only the bindings for X are retained in the answer rather than the entire subtrees.

We begin with a brief digression to consider a technical problem that arises in all three cases. This relates to the condition requiring the existence of a downward path from nodes of a given type. It occurs as an explicit condition in the bodies of q' and q'', but also arises in a more subtle form in q.

Consider an ltd d and a regular expression p over an alphabet Σ of type names. Let a be in Σ, and consider the question of whether there is a path in p from nodes of type a in lotos satisfying d. There are three possibilities:

- there is a path in p originating at nodes of type a in *some* lotos satisfying d (and then we say that p is *satisfiable* at a),
- there is *never* a path in p originating at a node of type a in a loto satisfying d (and we say that p is *unsatisfiable* at a)
- there is *always* a path in p originating at a node of type a in any loto satisfying d (and we say that p is *valid* at a).

We can show the following useful fact:

Lemma 3.9 *Given a regular ltd d, a regular path expression p over Σ and $a \in \Sigma$: (i) it can be checked in* PTIME *whether p is (un)satisfiable at a; (ii) it can be checked in* PSPACE *whether p is valid at a.*
Satisfiability of p at a can be checked in PTIME by testing non-emptiness of the language accepted by an nfsa constructed from p, d and a; validity is less straightforward and can be tested in PSPACE using an alternating polynomial-time Turing machine.

We now return to the example queries q, q', and q''. Consider first query q. Suppose q and the input ltd d are over alphabet Σ. The ltd d_q for the view defined by q is the following. The type of the root is r. Suppose for simplicity that $r \notin \Sigma$ (the other case is handled using specialization). For $a \in \Sigma$, let $d(a)$ denote the language associated with a in the ltd d. The ltd d_q for the view is defined as follows. Foe $a \in \Sigma$, $d_q(a) = d(a)$. The language $d_q(r)$ is a language over Σ, denoted L_X and defined below. Recall that L_X is the set of sequences of type names of nodes in the input that X can bind to, in the order of occurrence in the input (the order is determined by a depth-first, left-first traversal). To define L_X, let us first focus on the nodes which are parents of X, i.e. they are reachable by a path in p. Let us denote the corresponding language by L_p. The language L_p is defined by the following CFG G (for convenience, we use productions whose right-hand sides are regular expressions over terminals and nonterminals, with the obvious meaning).

Let f_p be an fsa over Σ accepting p; its state transition function is δ. For each state h in f_p, let p_h be the regular language accepted by f_p with start state h. The nonterminals of G are the pairs $\langle h, a \rangle$ where h is a state of f_p and $a \in \Sigma$. The start symbol is $\langle s, d(root) \rangle$ where s is the start state of f_p. The set of terminal symbols of G is Σ. The productions of G are:

- $\langle h, a \rangle \rightarrow \sigma(d(a))$ where h is a state in f_p, $a \in \Sigma$, and σ is a substitution defined as follows. For $b \in \Sigma$ and $h' = \delta(h, b)$:
  - $\sigma(b) = \{\langle h', b \rangle\}$ if $p_{h'}$ is valid at b,
  - $\sigma(b) = \{\epsilon, \langle h', b \rangle\}$ if $p_{h'}$ is satisfiable but not valid at b, and
  - $\sigma(b) = \{\epsilon\}$ if $p_{h'}$ is unsatisfiable at b.
- $\langle f, a \rangle \rightarrow a$ for each accepting state f of f_p.

It is easily verified that $L(G) = L_p$. To obtain L_X from L_p, we need one more step, where the pattern $l.X.r$ is matched against the content of nodes reachable by p. For each $a \in \Sigma$, let r_a be the language which is the image of $d(a)$ under a nondeterministic finite-state transducer which on given input word w outputs the last letter of each prefix of w which is in l and for which the remainder suffix is in r (the nondeterminism arises in guessing that the suffix from a current position is in r, and acceptance requires checking that all guesses along the way were correct). Since $d(a)$ is regular and regular languages are closed under transducers [Gin66], r_a is regular. Lastly, $L_X = \rho(L_p)$ where ρ is the substitution $\rho(a) = r_a$ for each $a \in \Sigma$. Since CFLs are closed under substitution [HU79], L_X is context-free. Indeed, a CFG for it can be effectively constructed from d and q in EXPTIME (polynomial in d and exponential in q). q on inputs satisfying d has a tight context-free ltd constructible[6] in EXPTIME.

Next, consider the query q', which highlights the role of specialized ltds. The nodes of type $a \in \Sigma$ to which X binds must restricted to ensure the existence of the downstream path in p'. This can be done using a specialized ltd, constructed as follows. Let $f_{p'}$ be an fsa accepting p', with start state s and transition function δ. Recall that p'_h denotes the regular language accepted by $f_{p'}$ with start state h. In the specialized ltd we will specify an alphabet Σ' of specializations of type names, and a mapping μ that indicates, for each $a \in \Sigma'$, the type $\mu(a) \in \Sigma$ of which a is a specialization. (Note that μ also defines a homomorphism from Σ' to Σ that can be extended to words over Σ' and lotos with type names in Σ'.) The specialized alphabet Σ' will be:

$$\Sigma' = \Sigma \cup \{\langle h, b \rangle \mid h \in states(f_{p'}), b \in \Sigma, \text{ and } p'_h \text{ is satisfiable at } b\}.$$

The specialization mapping μ is defined by $\mu(\langle h, b \rangle) = b$ and $\mu(b) = b$ for $b \in \Sigma$ and $h \in states(f_{p'})$. The ltd d' over the specialized alphabet Σ' is defined by

- $d'(r) = \sigma(L_X)$ where σ is the substitution defined by
  - $\sigma(b) = \{\langle s, b \rangle\}$ if p' is valid at b,
  - $\sigma(b) = \{\epsilon, \langle s, b \rangle\}$ if p' is satisfiable but not valid at b, and
  - $\sigma(b) = \{\epsilon\}$ if p' is unsatisfiable at b.
- $d'(b) = d(b)$ for $b \in \Sigma$;
- $d'(\langle f, b \rangle) = d(b)$ if f is an accepting state of $f_{p'}$;

[6]The construction can be done in PSPACE by using an nfsa for p rather than the fsa we used for simplicity of presentation.

- $d'(\langle h, b\rangle) = \mu^{-1}(d(b)) \cap \Sigma^*\Sigma'_{next}\Sigma^*$ if h is not accepting, where $\Sigma'_{next} = \{\langle h', a\rangle \mid h' = \delta(h, a)$ and $p'_{h'}$ is satisfiable at a $\}$

Note that the last item simply ensures that the content of $\langle h, b\rangle$ has at least one type allowing to continue successfully along a path in p. It is easily seen that a loto iover Σ satisfies the specialized ltd just defined iff it is in the view defined by q' on inputs satisfying d.

Finally, consider query q''. In this case specialization is not needed. The ltd d'' for q'' is defined as follows. Since the bindings of X have no children, $d''(a) = \emptyset$ for all $a \in \Sigma$. The type r of the root is obtained by applying a substitution τ to L_X as follows:

- $\tau(a) = \{a, \epsilon\}$ if p' is satisfiable but not valid at a,
- $\tau(a) = \{a\}$ if p' is valid at a, and
- $\tau(a) = \{\epsilon\}$ if p' is unsatisfiable at a.

The above discussion contains in a nutshell the basic ingredients for the ltd inference algorithm for selection queries provided in [PV98].