INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Tamino Experience Report: Advantages and Pitfalls of an XML Database Management System

Sebastian Kraus

Projektarbeit im Rahmen des Fortgeschrittenenpraktikums

Beginn der Arbeit:	01.05.2001
Abgabe der Arbeit:	01.08.2002
Betreuer:	Prof. Dr. François Bry
	Tim Furche
	Tim Geisler

Zusammenfassung

Im Rahmen eines Praktikums an der Fakultät für Informatik der Ludwig-Maximilian Universität München wurde ein Informations System implementiert, dessen Hauptanforderung die Einbindung von XML Technologie war. Dieser Bericht beinhaltet die Erfahrungen, die mit der nativen XML Datenbank Tamino von der Software AG gemacht wurden, welche die Anwendung mit Daten versorgte und behandelt sowohl die Vorteile wie auch Einschränkungen. Am Beispiel eines halbwegs komplexen Datenmodells werden die nötigen Schritte zur Benutzung von Tamino beschrieben und auf die für Datenbanken typischen Bestandteile wie Anfragesprache, Linkkonzepte und Datenmanipulation eingegangen. In vielen Punkte scheint das DBSM Tamino noch zu enttäuschen aber wegen der raschen Etablierung der XML Technologie sind stetige Verbesserungen zu erwarten.

Abstract

Within the scope of a practical course, offered on the Institute of Computer Science of the Ludwig-Maximilian-University in Munich, an information system with the main aim to use XML technology, has been implemented. This paper is about the experiences made with the Software AG's native XML database Tamino, which provided the system with data. At this, the advantages and restrictions are discussed. The needed steps to use Tamino are described with a quite complex data model and typical features like queries, links, and data manipulation capabilities are discussed as far as they were used in the environment of the project. In many points the DBMS Tamino seems to disappoint but with the fast development of the XML technology, improvements in this sector are to expect.

Inhaltsverzeichnis

1 Introduction

This paper describes how the XML database management system Tamino of Software AG[1] has been used for implementing a web-based information system about courses at the Institute for Computer Science of the University of Munich. The development took place during Winter 2000/2001, lasted 4 months, and involved 13 persons working at about 10 hours per week.

The choice of the XML database management system Tamino for this project was motivated as by the following considerations:

- Not only administrative data such as room numbers and course times, but also documents such as descriptions about courses had to be managed. Representing such "mixed" data is considered one of the strength of XML.
- With a native XML database management system like Tamino, data format conversions from and to XML can be avoided.
- Experimenting with one of the first native XML database management systems was one of the goals of the project.
- The idea to work with new not well known software was an important aspect when collecting the applications. Tamino was not known and promising a wide support of XML features. To access the database and to provide various kinds of applications with content, an application server was needed which should be able to integrate services for the purpose of XML processing in different forms. Cocoon[2], an *XML Publishing framework* has been chosen. It is a web client server application promising an easy separation of content, logic and style. Tamino and Cocoon have a clear interface which separates the work on both of them into two tasks.

Basically, the system developed is an *Information Management System*. To access informations, the user is working on a browser based webpage which is providing the following services:

- Informations about university courses and persons related to the courses can be questioned. Similarly to the official webpage of the Institute of Computer Science of Munich, every important fact about events, courses, main researches and general important rules of studying is given.
- Personalized user profiles give the possibility to manage a personal time table, adapted to the individual needs of the study course. Features reach from simple settings of the appearance of the retrieved informations over server based calendar functions as automatic reminders of special events up to online registration to courses by the students. On the other side administrative clients who are offering and teaching the courses should be able to upload the required informations the easy way.

[1] http://www.softwareag.com
[2] http://xml.apache.org/cocoon

- Output on different devices is supported: The system should deliver informations to different media such as PC streams, PDA, and voice. It was realized for HTML, WML and later Voice XML.

A mentioned above, the project has been split into several tasks. Due to the use of marked off software and the resulting interfaces the development of the information system has consisted in the following.

- *Data modeling:* This task includes building up a schema of the data and their administration (i.e. updating of the schema and the data). The schema files and data documents has to be loaded into the Tamino database and administrated. The description of the data modeling task and its experiences with Tamino is the content of this paper.
- *Application logic:* A content management system has to be implemented with the use of Cocoon and its integrated tools. The system is used to collect the data from the database and to convert them to a format which can be used for the output section for further processing. It is the central unit of the application making the interfaces to the two other tasks available.
- *Presentation:* This task is responsible for the layout and design of the user interfaces. Transforming of data, delivered from Cocoon, has to be done via XSLT[Cla99] StyleSheets to achieve respective format for the different media. The media considered were browser based HTML pages and WML for mobile phones.

This paper describes the experiences with Tamino which made during the developing phase of data modeling. At this time, many ideas how to realize features of the specification had to be given up. It is not surprising, that a nearly new software can not integrate the whole bunch of features at all or even in the same way as the specification purports. In this sense the paper could give a overview for critical points which, when planning a similar project, someone could think about.

In developing this information system, the following pitfalls of Tamino in its current version were encountered:

- Tamino does not fully support XML. Features like ID/IDREF and entity references are missing. Both restrictions have serious consequences for data modeling and storing.
- A real data manipulation language is missing in Tamino for the following reason: Tamino's Interactive Interface must be used for populating a database, and for updating. Unfortunately, this interface is not powerful enough and often requires too much steps.
- X-Query, the query language provided by Tamino, is not powerful enough. X-Query is not related to XQuery[CFMR01] of the W3C[3] In particular, XPath[CD99], a component of X-Query, is not fully integrated. For example building join expressions with X-Query is only possible in a restricted way. Furthermore, XML links are not fully supported by Tamino's implementation of X-Query. As a consequence of this later point, Tamino can not process link bases as they are specified in XLink[DMO01].

[3] http://www.w3.org

- Updating large, structured data items is extremely complicated. In some cases, the database management tool Tamino Manager", is not running in a stable manner. Whether update operations are possible depends on the server software. The Microsoft Information Server[4] is more widely supported then the Apache Server[5], which was used in the project.
- Tamino's documentation is often difficult to read and in some cases contains errors.

This paper is structured in the following way.

- *Chapter 2: DTD and Tamino Schemas*
  A overview is given, how a sample XML DTD can be mapped to an Tamino Schema.
- *Chapter 3: Data Modeling*
  This chapter gives a detailed explanation about the motivation and the implementation of the projects data model.
- *Chapter 4: Querying*
  Queries are an essential part of database environments. But many aspects of the Tamino query language are non-satisfying and have to be mentioned.
- *Chapter 5: Data Manipulation and updating large data items*
  Some ways of manipulating the data are shown. Further on, the results of a test how Tamino handles a huge amount of data, are listed.
- *Chapter 6: Insufficiencies of the Documentation*
  In some parts Tamino's documentation is written too superficial.
- *Chapter 7: Advantages of Tamino and a native XML database*
  It is pointed out which way native XML databases like Tamino are superior to traditional databases.

2 DTD and Tamino Schema

2.1 Tamino and its Schema: A short Overview

Tamino is an information server that uses XML to store and retrieve data of different type (for example, relational data, graphics, sound, video, plain text, or of course XML-structured text).

It uses its own schema language to determine where the data is stored and how it is indexed.

Though a well-formed XML object (i.e. an XML document) can be stored without defining a schema the Tamino documentation recommends do so. If a schema is defined before the data objects are loaded into the database Tamino recognizes where the data is stored and how it is indexed to ensure a better performance. When an schema is defined, the XML objects are stored in a special area and on default indexing which allows full text retrieval is applied. As

[4]http://www.microsoft.com
[5]http://www.apache.org/

another advantage should be seen that trough the modeling of a schema by its own a better overview over the datamodel is given and changes are more quickly made.

The Tamino schema language is based on an XML DTD. This DTD consists of three elements, `Collection`, `Doctype`, and `Node`, which stand in the following relation to each other:

```
<! ELEMENT    Collection   (Doctype*)>
<! ELEMENT    Doctype      (Node+)>
<! ELEMENT    Node         EMPTY>
```

The `Collection` elements relate to databases. Collections are used to combine `Doctypes` for the purpose of providing parallel access to multiple doctypes. A `Doctype` is a data definition, comparable to Tables used in relational database systems. A Tamino schema describes one `Doctype`, so that the two terms, (Tamino)- Schema and `Doctype`, are sometimes used in place of each (a instance of a schema is a instance of a `Doctype`).

Node elements express all information items contained in the `Doctype`. Compared to a DTD, where child elements are embedded within their parent element, the node elements are linked by the attributes `ino:key` and `ino:parent` to express parent-child relationships. All node elements are at the same level, corresponding to the DTD above but structured in a tree hierarchy as shown in Figure 1.

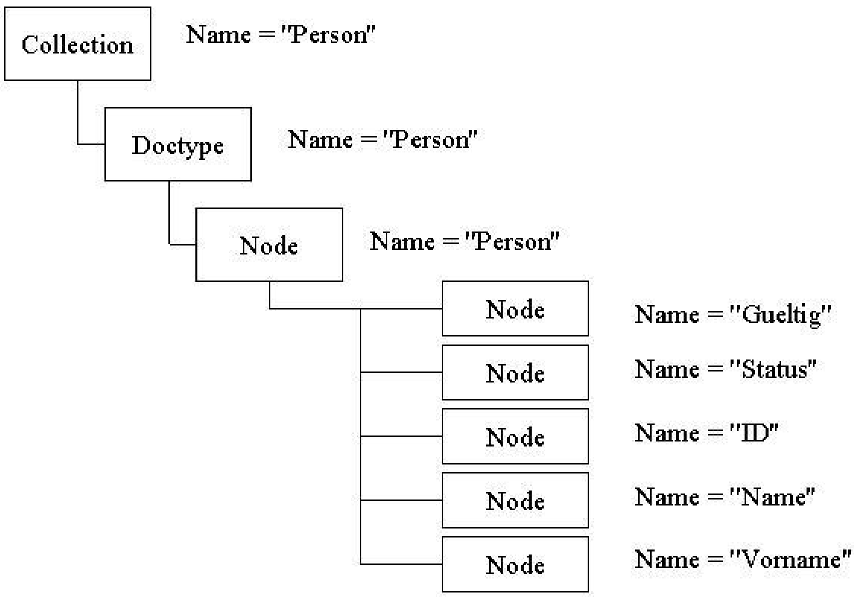

Abbildung 1: The internal hierarchy of a Tamino Schema

2.2 Creating a schema from an existing DTD

The resulting Tamino schema after translating the DTD is a comparatively complex one, including the three standard elements. All information about elements is realized as attributes. This complexity makes it nearly impossible to build a Tamino schema by hand. It requires too much effort as it is illustrated by the following code.

This XML DTD is describing an object `Person`

```
<!ELEMENT     Person   (Name, Vorname)>
<!ATTLIST     Person   ID      ID          #IMPLIED
              Gueltig  CDATA               #IMPLIED
              Status   CDATA               #IMPLIED >
<!ELEMENT     Name     (#PCDATA )>

<!ELEMENT     Vorname  (#PCDATA)>
```

and represents the following in Tamino schema.

```
<?xml version=''1.0''?>
<ino:collection ino:name=''Person'' ino:key=''ID001''>

<ino:doctype    ino:name=''Person''
                ino:key=''Person1''
                ino:options=''READ INSERT UPDATE DELETE''>

<ino:node       ino:name=''Person''
                ino:key=''Person2''
                ino:obj-type=''SEQ''
                ino:parent=''Person1''
                ino:search-type=''no''
                ino:map-type=''Native''/>

<ino:node       ino:name=''Gueltig''
                ino:key=''Person3''
                ino:obj-type=''CDATA''
                ino:parent=''Person2''
                ino:search-type=''text''
                ino:map-type=''No''/>

<ino:node       ino:name=''Status''
                ino:key=''Person4''
                ino:obj-type=''CDATA''
                ino:parent=''Person2''
                ino:search-type=''text''
                ino:map-type=''No''/>
```

```
<ino:node          ino:name=''ID''
                   ino:key=''Person5''
                   ino:obj-type=''CDATA''
                   ino:parent=''Person2''
                   ino:search-type=''text''
                   ino:map-type=''No''/>

<ino:node          ino:name=''Name''
                   ino:key=''Person6''
                   ino:obj-type=''PCDATA''
                   ino:parent=''Person2''
                   ino:search-type=''text''
                   ino:map-type=''No''/>

<ino:node          ino:name=''Vorname''
                   ino:key=''Person7''
                   ino:obj-type=''PCDATA''
                   ino:parent=''Person2''
                   ino:search-type=''text''
                   ino:map-type=''No''/>

</ino:doctype>

</ino:collection>
```

Note that a `collection` is surrounding the `person` element. It is needed, given by the element hierarchy of Tamino schema. In this case, a `collection` named `Person`, an underlying `Doctype` named `Person` and a node have been defined.

Attributes such as ID are defined as subelements of the original DTD-elements. They are labeled with the internal attribute `ino:obj-type` of the Tamino Schema, which is always set to `CDATA`. This can help to identify the attribute elements when reading a Tamino schema file.

These various attributes and so the many different settings for indexing are hard to handle. So the preferred way to define the meta structure is to create a DTD by third party software, adapted to the needs of the application. As it is not intended by the Software AG to write Tamino Schemas by hand, tools for authoring DTD's and translation from DTD to Tamino schema are included when purchasing the database system. In the project the DTD/XML editors XML Authority[6] and XML Spy[7] were used successfully. The different graphical and textual views gives a good overview of the DTD and the underlying data model. As next step the DTD can be transformed to the Schema with the Tamino Schema Editor. Not every DTD can be translated into schema because a lot of DTD features are not supported as shown in the following chapters.

A useful but not essential feature would be some standard Datatypes as the XML-Schema supports. Standard types like numbers and date ease the development of the data model and

[6]http://www.softwareag.com/xml/about/glossary.htm
[7]http://www.xmlspy.com/

further processing.

3 The data model

In the following, the ideas that led to the data model (see Appendix A) are explained. The characteristic features of XML, used in implementing this data model are stressed.

The data model has been influenced strongly by those features which are or aren't supported by Tamino. The attention is turned to the problems encountered, their causes, and in which way they affected the functionality and implementation of the information system, and which solution was chosen in the end.

3.1 Characteristic and Requirements to the project's data model

The main idea of the project has been to build an information system for students, lecturers and civil servants to get current informations and to prepare a simple interface to provide the system with content. One important aspect has been the dynamical character of the system but also several other requirements has been influenced the structure of the data model.

- The system should deliver information about an Institute of Computer Science but later extensions had to been foreseen, so there is need of abstraction, flexibility and an easy possibility for extensions.
- In some cases a very specific modeling of the elements is wished to deliver a clear and detailed structure for a precise evaluation of the element's content.
- Many elements are needed in different context. An element can stand in relation with several elements and for each relationship the element has a different role. So a labeled link concept is needed to prevent redundancies by saving equal informations.
- Different courses of study have different conditions of study. The information system should be able to handle with these conditions and to make suggestions for a personalized time table. This calls for a model expressing this conditions.

The data model includes the following characteristics:

- The data model is composed of a mixture of structured data and plain text ("mixed contents").
- Entities are not used (with the one exception of the reference to an altered HTML-DTD). Tamino can not handle with this typically XML technique.
- To prevent storing data several times, references to elements are used widely, expressed by using ID/IDREF attributes, even if they are not internally supported in Tamino.
- Otherwise, if duplication of information is acceptable, child elements are used.

In the following these features and the problems that occurred are explained, in which way they have been solved and where the functionality of the application had to be restricted.

3.2 Links and link bases

As mentioned in the requirements chapter, elements can have different roles in their relations to other elements. So a person can exist as a teacher for a course, as an employee of a chair, or as a user for the time table. Furthermore there exist elements which content is needed in several other elements. As an example several administrative sections can be located in one building but also several lessons are offered in this building. The fact that an administrative section is represented within a different `element` than the lessons it is more practical the data model does not only consists of embedded elements to avoid redundancies of bigger data blocks. Therefore a link concept was used. A link to the element `Person` (described in Chapter 2.2) can be set by defining an element `Person_Link`. This link was labeled with an attribute `role`, specifying the nature of the relation. To query each element it has an explicit `Id`.

The following code is an example for a link to a person, shown as XML data, embedded within the element `Lehreinheit` (i.e. a teaching unit):

```
<Lehreinheit>
        ...
        <Person_Link Person_ID=''Person_KlausMeier'' Rolle=''Veranstalter''/>
</Lehrerinheit>
```

Embedded within the element `Benutzer`("user"):

```
<Benutzer>
        ...
        <Person_Link Person_ID=''Person_KlausMeier''  Rolle=''Benutzer''/>
</Benutzer>
```

The linking elements are defined in the DTD as link bases:

```
<!ELEMENT      Person_Link EMPTY>
<!ATTLIST      Person_Link   Person_ID IDREF   #IMPLIED
               Rolle      CDATA   #IMPLIED >
```

The following problems with links in Tamino have been observed.

The database Tamino doesn't support the ID/IDREF(S) attributes of the XML working draft. So the information, that one element refers to another is not internally stored in the database. The ID/IDREF attribute gives only the developer the information, that the respective element is linked with another.

As solution to link elements Tamino provides specific data types for the Tamino schema. The process of defining the links is not very user friendly and forces the developer of the data model to define each single link with great effort and always to have the model of the Tamino schema in mind.

As example the modeling of the `Person_Link` part of the `Benutzer`/(user)- doctype is shown as Tamino Schema, in which a link to the `Person` doctype is wished.

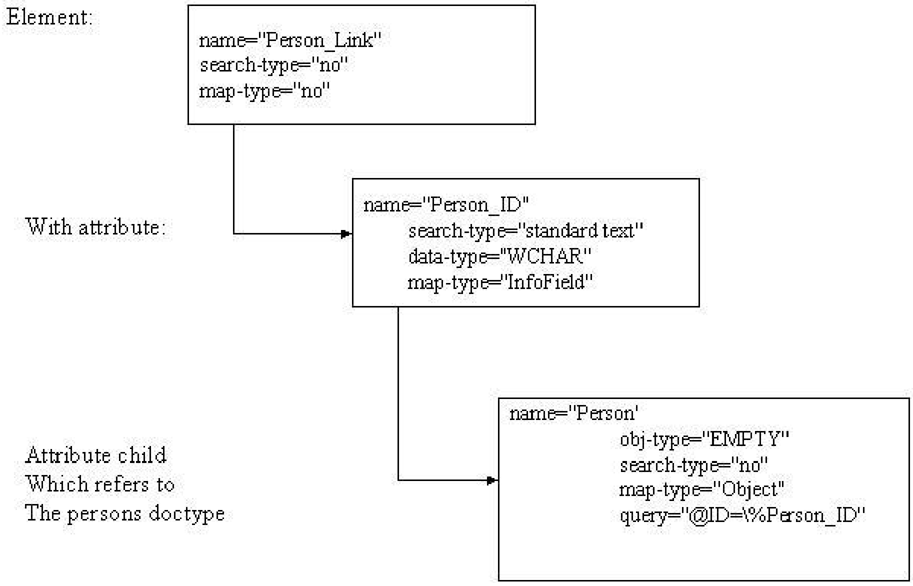

Abbildung 2: Configuration of a link

A user is a person who possesses specific informations so a link has to be built from the user element to the person (Figure 2). This is done with the child element `Person_Link` and its `Person_ID` attribute. To define the link in Tamino the Tamino Schema Editor can be used. After reading the `Benutzer` schema file it shows all elements, here the `Person_Link`, and all its attributes. The attribute `map-type` within the attribute `Person_ID` must be set to the `InfoField` value. The linking attribute below contains `Object` as `map-type`, a query string and `object-ref`. The query tries to match the `Person_ID` value with the `ID` value of person elements located on the `object-ref` path.

This example only shows a link in one direction, from user to person. If a bidirectional link is wished the other direction must be declared in the same way on the other element.

The techniques and problems to query the elements are explained in Chapter 4.

3.3 Abstraction

Abstraction is needed for an universal applicable structure, not only to ensure a data model which is easy to extend but also to prevent huge changes on it. Another benefit is, that the amount of the used elements in the DTD and the schema is lowered. So elements which can be described through elements of the same DTD element and which have similar preferences and semantic meanings, are good candidates. The relations produce a hierarchical tree.

In the application `Lehreinheit` (a teaching or a course unit) and `Verwaltungseinheit`(an administration unit) were identified as this special units. `Lehreinheit` contains all objects which belong to the teaching of the university, `Verwaltungseinheit` all administrative categories.

The idea for abstraction is shown on the element `Lehreinheit` (course unit). The element `Lehreinheit` can stand for a course, but also general for an teaching unit happening within a special period of time. So a hierarchy can be expressed by a multiple link element `Lehreinheit_Link` which refer to another `Lehreinheit` element realized by an IDREF Attribute.

```
<!ELEMENT Lehreinheit   (..., Zeit , ..., ..., Lehreinheit_Link*
                         ,...,..., ..., ... )>

<!ELEMENT Lehreinheit_Link EMPTY>
<!ATTLIST Lehreinheit_Link   Lehreinheit_ID IDREF #IMPLIED
                         ... >
```

The relation is important to express the following example:

A course, say `Informatik I`, consists of various events. There are two dates for the lessons itself which are required courses and in general, more than one exercises are offered for this course. This gives a hierarchy shown in Figure 3:

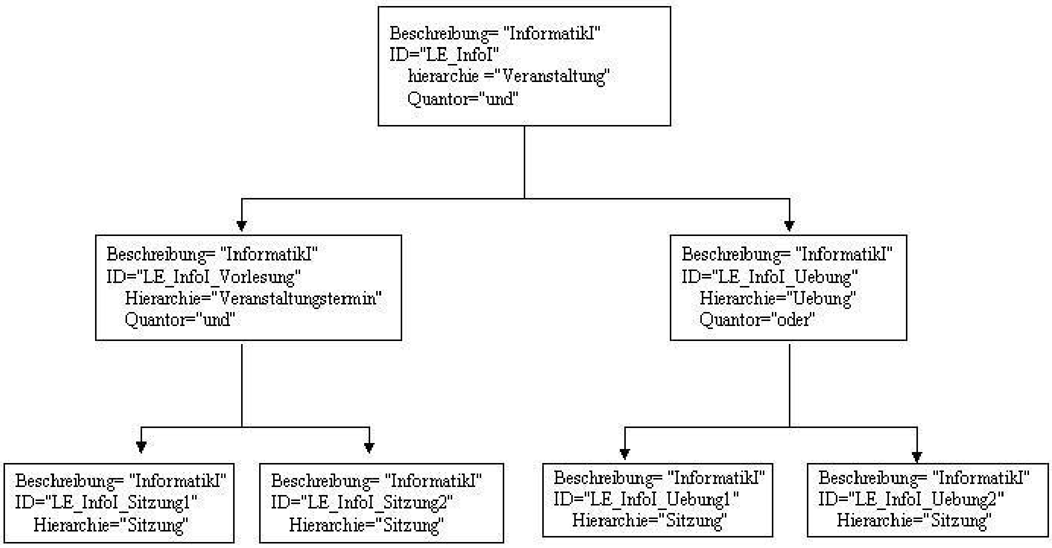

Abbildung 3: Hierarchy-tree of "Lehreinheit"

The conditions and hierarchy between events are expressed by the attributes:

- Quantifier (`Quantor`), which includes the boolean values `und` ("and") and `oder` ("or"). The content management system has to evaluate this conditions given through the semantic meaning of the operators. The and-operator in the hierarchy level `Veranstaltung` tells the application logic, that the underlying elements `Veranstaltungstermin` (dates of courses) and `Uebung` (lesson) are obligatory, the "or"-operator within `Uebung` implies, that one of the two lesson dates has to be visited, to fulfill the requirements of the course as a student.

- Hierarchy (`Hierarchie`), which values describes the level classified in categories adapted to the environment of the data model. These attributes can be easily renamed for extension. So `Veranstaltungstermin` is a course, which consists of two single events (`Sitzung`) the week, `Uebung` is a lesson with two single events and both together represents the proper course `Informatik I` a student have to visit in his first term.

The course `Informatik I` can be linked to a parent element, named for example "basic_studies". This is not a course at all, but it includes all courses offered in the first four terms of a students education.

For more specific queries on the tree the element `Lehreinheitstyp` is embedded within the `Lehreinheit`. It delivers more detailed information about the events. So the events can be categorized as "practical course", "course", "lecture" etc.. These values can be chosen freely adapted to the environment of the information system. Again, it is the task of the content management system to evaluate these values. The initiation of such a type enables a precise differentiation and a selection in the vertical direction (Figure 4).

```
<!ELEMENT     Lehreinheit   (..., Zeit , ..., ..., Lehreinheit_Link*
                   ,...,..., ..., Lehreinheitstyp )>

<!ATTLIST     Lehreinheit      ID            ID    #IMPLIED
              ...
              Quantor          (und | oder)        #IMPLIED
              Hierarchie       (Studiengang |
                               HS_GS |
                               A_PG_ST |
                               Veranstaltung |
                               Veranstaltungstermin |
                               Uebung |
                               Sitzung )           #REQUIRED >

<!ELEMENT     Lehreinheitstyp  (#PCDATA )>
```

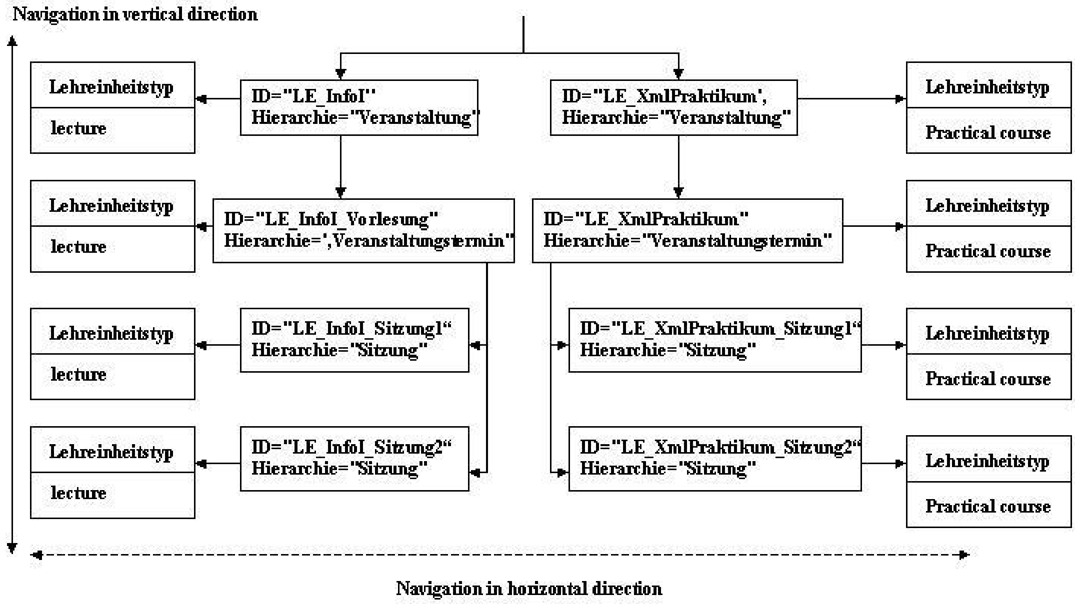

Abbildung 4: "Attributes enable navigation in different directions"

3.4 Flexibility

The free choice to name attributes and elements gives a high flexibility as shown in the preceding chapter by the element Lehreinheit. Further on, `Benutzereinstellung` (preferences), needed to store user profiles, are modeled in a very general manner.

```
<!-- xmlmod is the root element-->
<!ELEMENT xmlmod   ( (...|...|...|...|...|Benutzereinstellung|...|...|  )*  )>

<!ELEMENT Benutzereinstellung  (Benutzer_Link , Einstellung )>
<!ATTLIST Benutzereinstellung  Selektor CDATA  #IMPLIED >
<!ELEMENT Einstellung          (#PCDATA )>
```

Theoretically users can have an arbitrary number of preferences elements. They are linked to the user with the element `Person_Link`. The name of a single preference is defined by the attribute `Selektor` and the value is stored inside the element `Einstellung` (settings). If the information system is extended, the user can easily add new features.

3.5 Structured data

If an XML document is high structured it means single elements are possessing child elements which again have child elements and so on to a arbitrary depth. The leaf element is definitely reached if the possible element's content can no more be split into two different items. This level is theoretically reached if the content is a numeric number or a single letter. In most cases this extreme granular representation of data is not necessary. In applications which mostly use text information the leaves content would be a word or a number and normally the content of the leave node is given by the meaning of the leaves names and the composition with its parent and sibling elements. A high structured schema presume a very careful creation of the respective documents to be valid. But advantage of the strong structural data is that it can be used very well for further processing in program operations like calculations on time or even user adapted suggestions of a time table based on constraint programming. Moreover, queries can deliver more specific results and in practice the creation of the data documents by hand is better manageable due to the support of editors like Xemacs[8] and Xml Spy[9] which support validation with respect to a DTD on the fly and show possible entries when an XML document is created.

In practice the leave elements are not linked. This would give theoretically the possibility, to prevent all redundancies but it was not possible as Tamino has difficulties with the linking mechanism (as discussed in Section 3.2). When linking nearly each element with the other the resulting structure would cause a overhead of administrative work and a schema would need more memory resources than the documents content. Finally a tradeoff has to be found, where elements have to be linked and where one can be embedded within other elements.

In the project the time element is the most complex one. The time element is not only a simple digit moreover it possesses informations about beginning and ending of an event, iterations, date and time. It expresses a space of time which is no restriction because a point of time is a space too. The representation of date and time is not split into its numbers; a standard format has been pinned down at the whole project's environment.

The `Zeit`(time) element:

```
<!ELEMENT Zeit (Startuhrzeit? , Enduhrzeit? , Datum ,
                Wochentag? , Geblockt?, Regelmaessig?)>
<!ATTLIST Zeit Tempore  CDATA  #IMPLIED >

<!ELEMENT Startuhrzeit  (#PCDATA )>
                        <!-- Format: hh:mm-->
<!ELEMENT Datum         (#PCDATA )>
                        <!-- Format: DD.MM.YYYY-->
<!ELEMENT Enduhrzeit    (#PCDATA )>
                        <!-- Format: hh:mm-->

<!ELEMENT Wochentag     (#PCDATA )>
                        <!-- Possible values: Montag,Dienstag, ... -->

<!ELEMENT Geblockt      (Ausschluss+ )>
<!ELEMENT Ausschluss    (Startdatum , Enddatum )>
<!ELEMENT Startdatum    (#PCDATA )>
                        <!-- Format: DD.MM.YYYY-->
<!ELEMENT Enddatum      (#PCDATA )>
                        <!-- Format: DD.MM.YYYY-->

<!ELEMENT Regelmaessig  (Periodizitaet , Enddatum )>
```

[8] http://www.xemacs.org/
[9] http://www.xmlspy.com/

```
<!ELEMENT Periodizitaet (taeglich | woechentlich | monatlich | jaehrlich )>
<!ELEMENT taeglich      (#PCDATA )>
                        <!-- Possible values: 1 - 7 -->
<!ELEMENT woechentlich  (#PCDATA )>
                        <!-- Possible values: 1 - 52 -->
<!ELEMENT monatlich     (#PCDATA )>
                        <!-- Possible values: 1 - 12 -->
<!ELEMENT jaehrlich     (#PCDATA )>
                       <!-- Possible values: 1 - N-->
```

The underlying modeling of time should cover all possible cases of time information about events. Each date in the data model exists of certain elements and can be used flexible to:

- single events
- recurring events
- blocked iterations
- excluded dates
- different period types

Dates consists of following parts:

- `Startuhrzeit`: Begin of an event .
- `Enduhrzeit`: End of an event.
- `Datum`: The date. In a recurring event it acts as the start date for continuous iterations.
- `Enddatum`: This is the date of the last possible iteration
- The elements `Geblocked` and `Regelmaessig` are optional and used for blocked or recurring events.
- `Tempore`: This attribute indicates the exact beginning of the event. `s.t.` stands for *sine tempore* and means the event starts at the exact stated time, `c.t.` stands for *cum tempore*, i.e. the event starts fifteen minutes later. `<woechentlich>2</woechentlich>` means an iteration of every 2 weeks.

Block events are important if a recurring event can not happen at a specific date or period (holidays, vacations, sickness). For this case the element `Geblockt` (blocked) is used. A recurring event is automatically split in one or more blocks with the embedded element `Ausschluss` (exclusion), which can split the block event.

For example a course is modeled which happens every Tuesday and Friday from 8 c.t to 10 o'clock and the date of beginning is January the first 2002 and it ends on April the fifth. On March the fifth there is no event. Looking at the data model will explain for what reason the end date is April the second and the Friday event is not modeled. A course with more than one events possesses two underlying course elements, each with its own time element. So the example only shows the content of the Tuesday event.

Than the time part of the document part looks as follows:

```
<Zeit Tempore="c.t.">
        <Startuhrzeit>08:00</Startuhrzeit>
        <Enduhrzeit>10:00</Enduhrzeit>
        <Wochentag>Tuesday<Wochentag>
        <Datum>01.01.2002</Datum>
        <Geblockt>
            <Ausschluss>
                <Startdatum>02.04.2002</Startdatum>
                <Enddatum>02.04.2002</Enddatum>
            </Ausschluss>
        </Geblockt>
        <Regelmaessig>
            <Periodizitaet>
                <woechentlich>1</woechentlich>
            </Periodizitaet>
            <Enddatum>02.04.2002</Enddatum>
        </Regelmaessig>
</Zeit>
```

3.6 The reference to an XHTML DTD

In the project only a few elements contain data of mixed content. These are descriptions about persons, courses and buildings and for this purpose XHTML[W3C] format seems to be a good choice. The structure, given from a DTD, allows a more simple handling for further processing to different media like HTML or WML. For example a course can posses two different descriptions, one with detailed content for HTML and one compact for WML which delivers only a small display capacity. The description elements are furnished with types to identify its kind of media.

To be able to validate the XML document including this descriptions, an entity reference to the XHTML DTD were added within these elements.

The problem is that the XHTML DTD includes various recursive calls in indefinite depths. The Tamino Schema Editor, transferring the DTD to Tamino Schema has problems to handle with to complex DTD's. Though the editor provides a function to define the deep of the recursion, but it is only used to specify the level of indexing the data. Also a altered XHTML DTD with no recursive calls was not accepted. Tamino simply can not cope with parameter entities and so it is not possible to combine DTD's. Only copying the modified XHTML DTD into the data model's DTD worked, but this was not on our purpose.

As result, working with the DTD and the mapped Tamino Schema always implied a special treatment. So two DTD's were used. One with parameter entities for the XHTML-DTD to validate the date inside the XML documents and the other without the entities, use to define the Tamino structure.

Entity declaration in the DTD used for validation:

```
<!ENTITY % HTMLlat1 PUBLIC
    ''-//W3C//ENTITIES Latin 1 for XHTML//EN'' ''XHTML-lat1.ent''>
%HTMLlat1;
<!ENTITY % HTMLsymbol PUBLIC
    ''-//W3C//ENTITIES Symbols for XHTML//EN'' ''XHTML-symbol.ent''>
%HTMLsymbol;
<!ENTITY % HTMLspecial PUBLIC
    ''-//W3C//ENTITIES Special for XHTML//EN'' ''XHTML-special.ent''>
%HTMLspecial;
<!ENTITY % xhtml1-strict-par SYSTEM
    ''xhtml1-strict-par.dtd''>
%xhtml1-strict-par;
        ....
<!ELEMENT Beschreibung  (%Block.xhtml; )>
<!ATTLIST Beschreibung  Typ CDATA  #IMPLIED >

Element ''Beschreibung'' in the DTD, used to define the database structure:

<!ELEMENT Beschreibung  ANY>
<!ATTLIST Beschreibung  Typ CDATA  #IMPLIED >
```

4 Queries

4.1 The Tamino query language X-Query

To retrieve the Data from the database, Tamino uses the query language X-Query the query language of Tamino version 1.2 and a successor of XQL. It is developed by the Software AG itself and still under improvement. As mentioned above X-Query is not related to XQuery, the query language of the W3C. X-Query is based on the XPath specification of the W3C, a language to select parts of a document, the nodes of an XML-document tree.

In following points Tamino X-Query differs from XPath. Not all XPath functions are directly supported. The basic functions are `last, position, id, name, string, boolean, not, true, false, number, count, sum, floor, ceiling, round.`

Others have to be implemented by using the Tamino Server Extensions (SXS) which are including `local-name, namespace-uri, starts-with, substring, substring-before, lang,-substring-after, string, length, normalize-space, translate`.

Compared to XPath, X-Query is extended by the functions `adj,after, before, betw, between, intersect, near,` and `sortby,` and a binary equality operator for text retrieval inside text nodes.

XPath offers two syntax styles to locate nodes in the path. An unabbreviated and a shortened Syntax.

For example the expression, `//person` is short for `/descendant-or-self::node()/child::person` and selects all `person` elements in the document.

Tamino only includes the shorted one.

4.2 The problem with the missing nested queries

Working on a database always implies working with objects which are connected in some way. In relational databases the relation is expressed by keys, in XML the use of the ID/IDREF attributes volunteers. This was the basic idea to describe relations in the data model.

Assuming a user wants to visit in a study period some courses and he wants to register these in his personal time table. In the information system the booking of a course is modeled in the way that for every booked course there exists an element `Stundenplan` (time table) which includes a `Benutzer_ID` (user-id) and a `Lehreinheit_ID` (course-id) attribute. These attributes are IDREF(S) to the elements `Benutzer` (user) and `Lehreinheit` (course).

```
<!ELEMENT    Benutzer       (Login, Passwort, Person_Link?)>
<!ATTLIST    Benutzer       ID               ID        #IMPLIED>

<!ELEMENT    Stundenplan  EMPTY>
<!ATTLIST    Stundenplan    Benutzer_ID      IDREF     #IMPLIED
                            Lehreinheit_ID   IDREFS    #IMPLIED>
```

A typical request, using nested predicates, can be the retrieval of all time table elements for a special user.

```
Stundenplan[@Benutzer_ID=//Benutzer[Login='maier']/@ID]
```

This query should deliver all time table elements which user-id attribute matches with the unique user-id of an user element, where the login matches with ”maier”.

But this kind of queries are not possible. X-Query is not capable to climb a path up from an intermediate node towards the root element. It does not matter if the nested query is performed only inside one single doctype or on another as shown in the example above where the user-id attribute of the time table has to match with the id of an user. The only solutions is to avoid nested queries. At this point X-Query differs from the XPath specification which restricted the implementation of the information system hardly and was nearly one reason to think about using a relational database management system.

4.3 Joins

The consequence of the missing nested queries is the implementation of a special kind of semi-joins in the Tamino schema language which gives possibility to dereference by attributes.

Example: A possible query is to return the `Person_ID` of all lectures of the courses `Logik`. Then the query on the DTD part

```
<!ELEMENT    Lehreinheit    (Person_Link*,...)>
<!ATTLIST    Lehreinheit    ID               ID        #IMPLIED>

<!ELEMENT    Person_Link   EMPTY>
<!ATTLIST    Person_Link    Person_ID        IDREF     #IMPLIED
                            Rolle            CDATA     #IMPLIED>
```

```
<!ELEMENT    Person    (Name,...)>
<!ATTLIST    Person    ID          ID     #IMPLIED>
```

has the appearance:

```
Lehreinheit[@ID=''Lehreinheit_Logik'']
/Person_Link[@Rolle='lecturer']
/@Person_ID
```

But it is not possible to retrieve all lectures and their respective informations without using nested predicates. To realize this query Tamino offers a semi-join concept. The way of defining it is explained in Chapter 3.2 for querying statically joined XML documents and for selecting a subset of the result node set X-Query uses a trailing slash. The use of a trailing slash returns the queried persons and all their belonging informations.

```
Lehreinheit[@ID=''Lehreinheit_Logik'']
/Person_Link[@Rolle='lecturer']
/@Person_ID/
```

Regrettably, this semi-join is very restricted in selecting combined parts of the database.

4.4 Querying the results of the joins

One of the tasks of information systems is to deliver data in a form as compact as the user requests. Obviously the possibility of a good selection of information is an important part of any user friendly application. As more compact query result are as less transformation steps are necessary within the application logic to provide the different browser.

The following example will demonstrate the weakness of the query language.

```
<!ELEMENT Lehreinheit  (Person_Link*,...)>
<!ATTLIST Lehreinheit  ID               ID      #IMPLIED>

<!ELEMENT Person_Link  EMPTY>
<!ATTLIST Person_Link  Person_ID        IDREF   #IMPLIED>

<!ELEMENT Stundenplan  EMPTY>
<!ATTLIST Stundenplan  Benutzer_ID      IDREF   #IMPLIED
                       Lehreinheit_ID   IDREFS  #IMPLIED>
```

To get all Person elements and its content to which exists a link from a Lehreinheit to which exists a link in the Stundenplan, a possible and within query should be:

```
Stundenplan/@Lehreinheit_ID/Person_Link/@Person_ID/
```

It is presumed that the first dereferencing happens after `/@Lehreinheit_ID/`. This kind of queries are often needed in the working environment of a database but in Tamino such queries are not possible. After dereferencing the `Lehreinheit_ID` attribute, no further processing is possible. There is no construct in the X-Query Syntax to differentiate between the dereferencing slash and the slash to climb down the element/attribute hierarchy. So the trail slash may only appear in the end of the query string.

One solution could be the use of variables as it is realized in query languages like XML-QL[DFLS99] or Quilt[CRF00]. So the join could be executed by two successive queries. The first is used to store results like ID's or general values of elements and attributes, and the second could query a semi-join construct, specified by the variable. But, as mentioned, variables are not supported, by Tamino X-Query.

Software AG recognizes in this point a non-conformity with XPath and promises the support of sub-queries in the next Tamino version. The integration of the W3C XQuery is planned and should produce relief. At the moment the Software AG is working on a prototype implementation of XQuery, called QuiP and can be downloaded on the Software AG's webpage for free, available on Windows 32 bit platforms. It conforms to the 7-June-2001 draft of XQuery and includes a large number of sample queries to get familiarize with it.

The restrictions given through the missing nested queries and the semi-joins implies that natural joins are not possible and link bases can not be implemented in Tamino. As consequence only large parts of the document tree have to be queried and afterwards joined "by hand" if such requests as in the examples above are needed. This expensive task has to be performed within the content management system (building the interface between database and output). This point is the weakest in the Tamino version as it makes nearly no differences to use a set of doctypes or one single XML document. The functionality of known relational databases given through the established query language SQL is not reached yet.

5 Data Manipulation and updating large data items

5.1 Desired DML characteristics

A couple of characteristics are important to manipulate persistent XML data in a controlled way. Based on the paper "System Desiderata for XML Databases"[ST01] some of its suggestions are listed below. The manipulation requirements are bounded closely to the underlying Data Definition Language (DDL) of the Database Management System, so only aspects, relevant for the project, are mentioned. Tamino provides the definition of multiple DTD's by organizing them into manageable collections. But as mentioned in Section 3, there is no possibility to use entity references to link from one DTD to another. So the whole data model was built in one huge DTD which had to be completely updated several times. The redefinition of the schema seems not very well-engineered which is explained in a succeeding section.

Following Data Manipulation Language (DML) characteristics are important for our information system having Taminos's restricted definition capabilities.

- Update:

  Updating database content includes the standard operations "delete", "insert", and "replace". The DBMS should be able to affect a whole document, parts of it, a style sheet or any other unit. But not only basic data, even metadata, such as a DTD, are imaginable candidates for the update operations. When updates are executed, the system has to ensure the referential integrity. All kinds of links as internal ID /IDREFs, entity references links to documents within the database must be checked such that there are existing targets.

- Querying:
  - Each item of data, however, may provide important information for managing parts of a database. These parts are not only information items of the data by itself but also entities, URI's, tags, schemas, and other metadata. Metadata is data about the documents, and can be an important means for users to manage a repository. An XML database should provide metadata in XML form and allow querying and manipulating through the DTD.
  - An XML database having a rich collection of DTDs or schemas must be able to access DTD information and it can be interesting to test the equivalence of documents or their parts and also of DTDs.

    For example, when adding a new document to the database, first you want to find out if the same document is already in the database or when defining a new schema which parts of the existing schema would not have to be changed. The last point can be helpful to avoid the procedure of updating all documents which have to be valid on the schema.
- Transformation:
  - *Schema evolution:* Because changes in DTDs are quite common, there is often need of transform the existing data, to correspond to the new DTD. Changes in Tamino on the DTD/ Schema always implies first to delete the instances. So it makes no sense to ask, if the instances can be kept by transforming it to the new schema.
  - *Views:* View definition capability is an important means to provide data independence in the presence of database growth and restructuring, to allow data be seen in different ways by different applications, and to provide security for hidden data. Whereas SQL and other database systems specify views through their query languages, it may be far more appropriate in an XML database system to base view definitions on a transformation language. An XML view specification will typically include other transformations beside hiding data, for example to change element and attribute names.

    Neither physical views are supported in Tamino, nor the questioned data can be restricted. The result is always the whole document below the questioned element. The query language X-Query is too weak to get more complex parts of the XML tree.

Tamino does not support the requirements in all points. The next two chapters gives a overview in which way manipulation actions are performed.

5.2 The manipulation capabilities of Tamino

The communication between Tamino and the clients happens via HTTP as protocol for data exchange. The client's requests a sent to the X-Machine, the Tamino engine. Its basic function is to store XML objects in and retrieve them from the respective data sources. The X-Machine offers a set of commands for storing, retrieving and deleting X-Machine objects, creating or erasing collections or schemas, performing transaction processing and diagnostic testing. These commands are sent to the X-Machine either as parameters that are appended to an URL in an HTTP GET request (parameterized URL addressing), or as multi part form data in an HTTP POST request.

The X-Machine commands are:

```
_admin       Perform an administration function.
_commit      Commit a transaction.
_connect     Start a database session.
_define      Create a collection or schema; modify an existing schema.
_delete      Delete one or more objects.
_diagnose    Perform a diagnostic test.
_disconnect  Terminate a database session.
_encoding    Specify the encoding to be used for the current request.
_process     Store one or more objects into a collection; or modify
             an existing XML object.
_rollback    Roll back a transaction.
_undefine    Delete a collection or schema.
_xql         Retrieve one or more objects using the X–Machine query
             language.
```

All the actions mentioned above can be submitted using the methods in the programming languages Java[10] and Javascript. Further on there exists an ActiveX component including these methods. The developer has to implement the illustration of the retrieved document by his own. But for building up or manipulating quickly a database and its content, as it is for the purpose of administration, the interactive interface is the first and only choice. It is not the task of an database administrator, to write his own manipulation application.

The Interactive Interface is a simple browser-based programming interface to the Tamino database server which allows the following actions.

[10] http://java.sun.com/

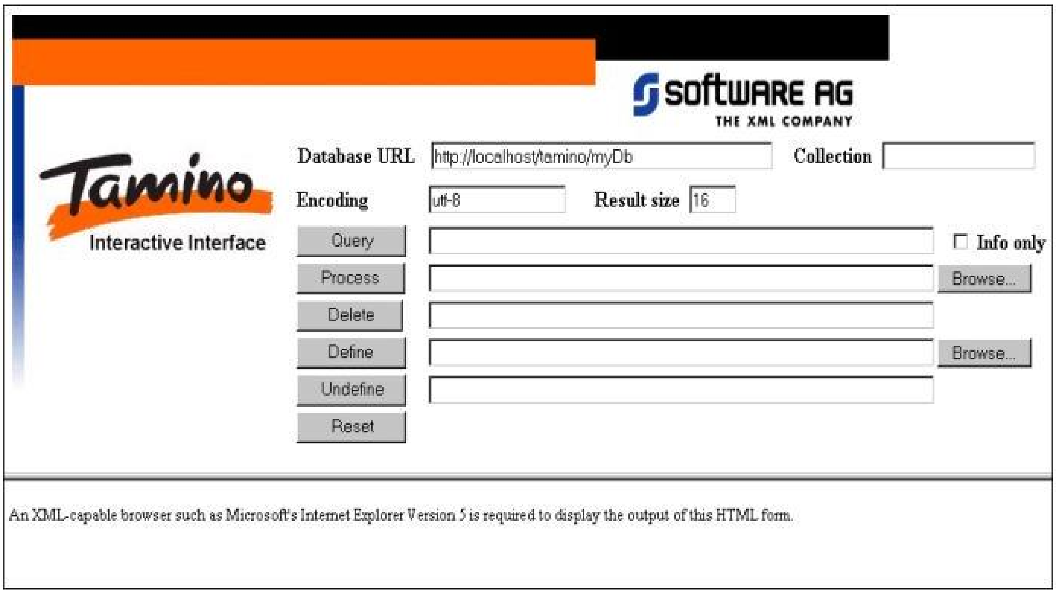

Abbildung 5: Screenshot of the Interactive Interface

- Definition of `Collections` within schemas.
- Loading XML instances of a schema into a database.
- Querying a database using the Tamino XML Query Language X-Query.
- Delete XML instances from a database.
- Delete schemas and collections.

The Interactive Interface is more a handy tool for testing. So it is easy to define a collection and its respective schema and to load sample data into the database. Queries, embedded within applications are generally performed through HTTP requests inside Java code or scripting languages.

Requests from the Tamino Interactive Interface to the X-Machine cause the X-Machine to return an XML response document. To display this response document properly, the Tamino Interactive Interface must to be running in an XML-capable browser. If not using an XML-capable browser, the actions can still be performed without restriction, but the response document will not be displayed properly in the browser. The same result can be achieve by typing in a complete URL in the address line of the web browser. If, for example, a database k-ustomer-dbïs defined on the local machine, and the database contains a collection Personsthat in turn contains a schema Person", the following URL can be typed in the web browser:

```
http://localhost/tamino/customer-db/Persons?_xql=Person
```

This will retrieve all documents that belong to the schema Person". The documents will be displayed directly in your web browser, provided that your web browser is XML-capable.

5.3 Performing manipulation actions

When mapping the data model from the DTD to the schema, several doctypes were identified. The Tamino Manager can save the doctypes as single schema files and can define them to the database at once.

Defining the schemas is not a problem so far the semi-joins, described in Chapter 4.3 are not implemented. This only happens at the first defining sequence, then the joins are set to make the data model usable.

5.3.1 Defining a new Schema

Often there is need of changing parts of a schema. This changes may not necessarily affect the existing data. For example, an attribute is added to the schema which is not a must have for the respective element. So it should be not a problem simply to override this part of the schema.

Tamino, or more exactly the Interactive Interface shows an error, that there is still data and it has to be delete first before updating. But even, if the instance of the schema is deleted, there occurs error massages that the schema could not be redefined. The consequence is to undefine the old one, defining the new one and then to perform the read in of the data.

When defining a new schema two problems can occur. First, if the new schema or a part of it, for example a single doctype, was translated from a DTD, which is the normal way to build it up, the links to other doctypes, have to be set again; the semi join linking mechanism is a special feature of the schema language. In the second case, changes on doctypes are made with the Tamino Schema Editor. This is recommend if small changes have to be done like adding elements or attributes to a doctype. In this case, an existing schema file is read from a local disk or directly from the database by the Schema Editor so the links still exist. All old schemas have to be deleted, the new one can be defined with the Schema Editor. You have to know, that the semi joins are still part of the schemas. If there exists an bidirectional link from one single element to another, you have no chance to perform the update. Tamino is tracking the links and for the first schema, there is no target. So first you have to delete one join-definition to have a starting point for updating. After finishing the update, the join has to be set again. As updating example for an unidirectional link in the datamodel, the doctype `verwaltungseinheit` includes the embedded `adresse` element which involves 2 semi-joins to `gebeaude` and `raum`. If `verwaltungseinheit` and `gebeaude` has to be updated, it is important first to define the element to which a link is set, in this case the `gebeaude` element.

5.3.2 Populating a Tamino Database

The capability of Tamino's link concept is not satisfying, it is of importance to prevent to much doctypes. Instead, embedded elements, the basic feature of XML, should fulfill the specification task of dependencies. Otherwise the administration overhead of the links would increase to much.

So only nine doctypes have been identified in the project, linked by Tamino joins as shown in Figure 6. But again it has not been known at the projects beginning, that the Apache server which was used, is not supported in the same way as the Microsoft Information Server[11]. So if working on an Apache Server only complete instances of doctypes can be manipulated. This means, deleting, updating and inserting of data on partial trees of a doctype can not be performed, not even single nodes. There is no entry about this fact in the documentation. This problem has been encountered on an advanced stadium of the project and have made large demands on time until the support of the Software AG has been questioned.

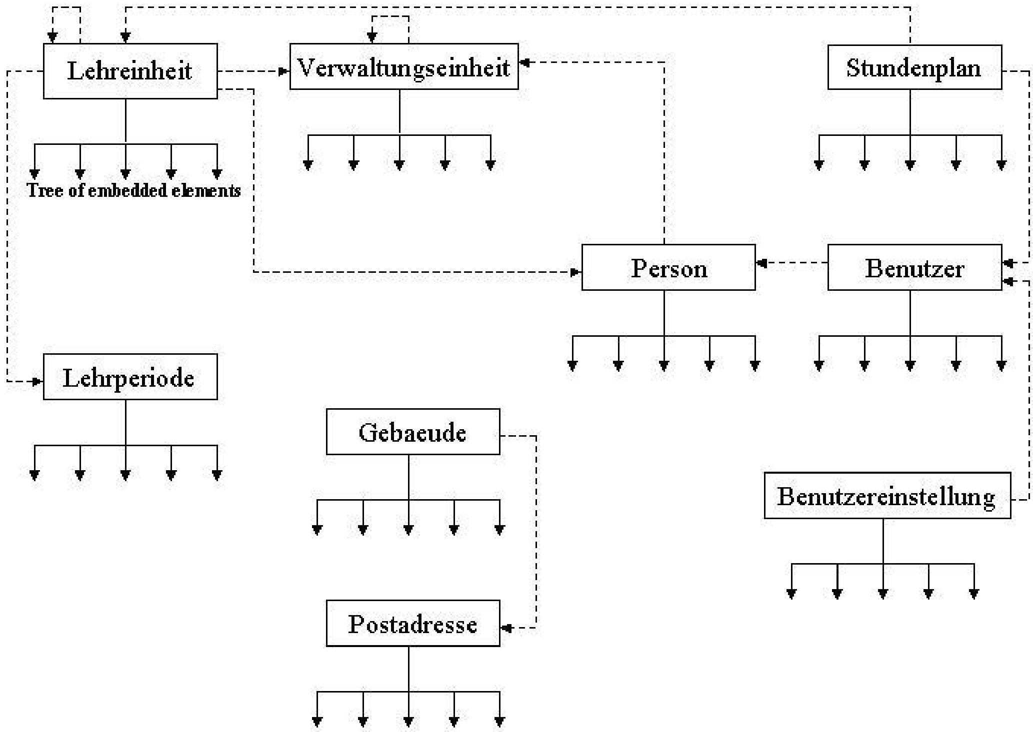

Abbildung 6: The links between the doctypes

This is not a too serious problem for database administration, more it affects the application and the content management. For example, if a user wants to change his preferences the whole "user preferences" instance of this user has to be deleted and defined again. This cause a considerable slowdown of the application performance and efficiency.

Another important aspect for administration is, that the ID attribute values of the data are not checked. Tamino does not map the ID attribute of a DTD to its internal ID's, named `ino:id`. Every data set of an element is furnished with a new `ino:id` value. So if the same data, with same ID's is defined twice, Tamino does not recognize the equivalence and the same data is stored twice. When performing queries on names and attributes and not on the `ino:id` value, the resulting data can be redundant. To prevent complex query expressions redundant data has been removed in the project. But obviously this can be done only on a prototype with small amount of data.

As conclusion Tamino is missing a real data manipulation language. More, there exists only an insufficient manipulation tool with very restricted manipulation capabilities, the Interactive Interface.

[11] http://www.microsoft.com

5.4 Test with large amount of XML data

5.4.1 Tamino utilities

By developing the projects database the data was often affected after a schema update. Using the Interactive Interface for updating more than a few data sets is to slow and unconvincing. Tamino offers data loader utilities to fasten this process. These tools a capable to load XML data from external sources into the Tamino database and are implemented as command-line utilities.

Currently, the loaders have been implemented in Java and Perl.

The Java loader also offers an unload facility for copying data from a Tamino database into a file on the host file system.

Rather than loading XML documents online, the Mass Loader prepares the documents to be loaded offline to increase the efficiency.

5.4.2 Restrictions and Considerations

The following considerations and restrictions should be taken into account when using the XML Mass Loader:

- The Mass Loader needs an active Tamino server with the appropriate `Collection` and `Doctype` defined for it to load documents into.
- The `Collection` and `Doctype` need to be defined before the mass loading can be started.
- The Mass Loader may run remote from a Tamino server, but it requires that its input file is local to the Mass Loader.
- All XML documents must be stored within one input file.
- There must be sufficient space on the Tamino server's temporary working space.
- On error, the XML data of the doctype, which has to be updated, will be left in the same state as before the mass load was executed to warrant the integrity.
- The Mass Loader exclusively locks the complete `Collection` in which the `Doctype` resides.

  This means that queries and other Tamino operations that go against this `Collection` will be refused.

5.4.3 Requirements to the document format

Normally, Tamino's mass load processing will tolerate the following documents:

- Documents that are not well-formed.
- Documents in which closed content is turned on.

- Documents that are not part of the doctype.
- Documents in which parsing and schema level checks fail.

  In these cases, the document is ignored by default, and processing is then continued. An error message is returned, stating that the document was rejected.

5.5 The data loader in use

It is obviously that the final application would not increase beyond the stadium of a prototype forced by the time limit of one term. The amount of data finally loaded into the database has been stored in an XML-document smaller than 10 MB. So it is of interest to further extensions of the application to know, in which way Tamino can handle with large data sets.

5.5.1 The data source

The test data has been chosen from the DMOZ Open Directory Project[12] representing an open source web catalog similar to Yahoo[13]. The DMOZ data is very suitable for mass uploading, because the offers the possibility to serialize DRDF dump to XML and the resulting structure DMOZ structure consists of a very simple DTD. The result of the serialization was a content file with a size of 678 MB including 312642 topic elements (i.e. categories) and 2177335 external page elements (i.e. links to web sites) and a 182 MB sized structure file.

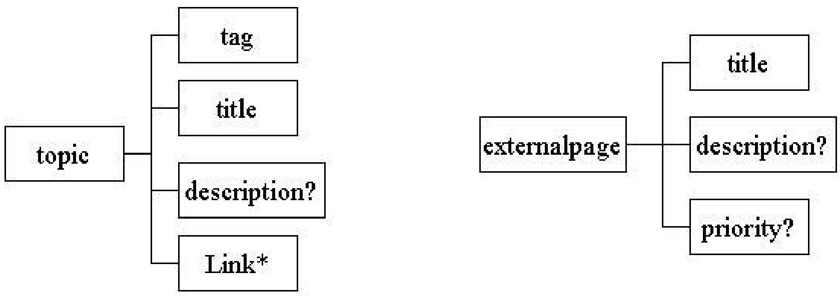

Abbildung 7: Structure of the example

```
<!ELEMENT topic (tag, title, description?, link*)>
<!ATTLIST topic id ID #REQUIRED>
<!ELEMENT externalpage (title, description?, priority?)>
<!ATTLIST externalpage about ID #REQUIRED>
<!ELEMENT tag EMPTY>
<!ATTLIST tag catid ID #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT link EMPTY>
<!ATTLIST link resource IDREF #REQUIRED>
<!ELEMENT priority (#PCDATA)
```

[12] http://www.demoz.org/
[13] http://www.yahoo.com

```
<topic id="Top/Computers">
   <tag catid = "4"/>
   <title>Computers</title>
   <link resource="http://www.cs.tcd.ie/FME/"/>
   <link resource="http://pages.whowhere.com/computers/Timeline.html"/>
</topic>

<externalpage about="http://www.cs.tcd.ie/FME/">
   <title>FME HUB</title>
   <description>Formal Methods Europe (FME) is an European organization
     supported by the Commission of the European Union (via ESSI of the
     ESPRIT program), with the mission of promoting and supporting the
     industrial use of formal methods for computer systems development.
   </description>
</externalpage>

<externalpage about="http://pages.whowhere.com/computers/Timeline.html">
   <title>Computer Timeline</title>
   <description>A brief description of the eras in computing.</description>
</externalpage>
```

5.5.2 Upload of DMOZ content

With the Perl Script based mass loader the data was uploaded via HTTP request inside a LAN. The involved hardware consisted of a Pentium 3-processor with 1GHz and 256 MB RAM.

Messured values are:

- The simulated upload time for 970 MB of data, calculated by the Perl Script was 36 minutes.
- In practice the estimated time would be more the 50 hours. A concrete value could not be obtained; the upload stopped after proceeding 40 percent of the data, probable due to less availability of memory.
- After uploading 40 percent of the data the database used 1,6 GB data space and 0.8 GB index space.
- Queries performed on the data required answering time in the range of some seconds.

6 Insufficiency of the documentation

During the projects development a lot of questions has appeared which could not be answered by the Tamino documentation. The most features had to be figured out by try and error as the exact functionality of the Interactive Interface and the Schema Editor. In many parts the kind of explanation were to circumstantial, sometimes whole features were not treated. Examples are:

- The Interactive Interface is described in an unsufficient manner. It is not clear, in which way correct path expressions should be build. So sometimes, the specification of a path needs the database name followed by `Collection`, sometimes only `Collection` is excepted.
- The functionality of the schema types and attributes is not documented in a comprehensible way. In fact the explanation is straitened to some examples of configuration possibilities.
- A list of non-supported XML features is missing. So the Tamino Schema differs in many points to the possibilities of a DTD.
- The support of ID/IDREF(S) is contended but it is not clear in which way.
- The differences between the support of the Apache Server and the IIS are not mentioned.
- The Linking mechanism of the semi joins is only explained by an example. It was difficult to find out how to use it especially there were printing errors in the source code. Testing the example by simply inserting it to Tamino was complicated.

7 Advantages of Tamino and a native XML database

Many in this paper as problems discussed topics can serve as a help to develop an application in a more structured way by avoiding the critical points of Tamino in a early phase. Nevertheless Tamino accomplishes the fact to be an pure XML database management system. It stores XML structured data.

In this sense Tamino meets the expectations. Some ideas, realized in Tamino are useful features.

- The web based interface for administration gives a flexibility in a great extend.
- The communication with Tamino via HTTP makes an easy interface available.
- The Tamino software is partially useful, specially the Tamino Manager is an comfortable and concise tool.
- Implementing of a DTD or schema is supported in a nice way with the third Party software Xml Authority.

It must be pointed out, that not all features Tamino is providing were used or tested within this project. So there might be more parts to spot.

But a more interesting fact is, that Tamino is a pure native XML database. It is for sure, that native XML databases are going more important in future. XML has not only been adopted as the universal format for documents and structure data on the internet, even as today's demand for speed and reliability increases, more and more industries are turning to XML, not just as a web publishing solution, but as the basis for their applications.

Native XML databases satisfy the need for a robust XML storage method by offering a solution that is specifically designed to handle and optimize XML's unique structure.

An XML database offers major advantages over other XML storage methods in both development and actual usage.

- *The benefits during development:*
  - There is no need to change to non-XML data structures like tables when the XML documents have to be stored so the complexity of programming time is reduced.
  - Everything is designed for the storage of XML documents, including access and search methods, specialized functions, etc. Basically, the developer can think and work in XML from the beginning to the end.
- *Usage benefits*
  - With a native XML database, there is no performance penalty caused by the translation of data structures to and from XML. The speed and performance of an application is increased. This certainly depends on further transformations and the performance behavior of the used processors.
  - The structure of XML documents can be very flexible and diverse, while relational databases require a fixed schema describing all the tables and the relations between tables. This implies that for every XML document with a new or revised structure the database schema needs to be changed, which is a time-consuming process. By storing XML in its native structure, users do not have to make actual changes to the database schema. The database can be simply supplied with a DTD or XML Schema and the records can be stored into. It is not necessary to create tables or normalize the data like in a traditional relational database. The structure provided by the DTD or XML Schema is the structure the database will use.

A The data model of the project

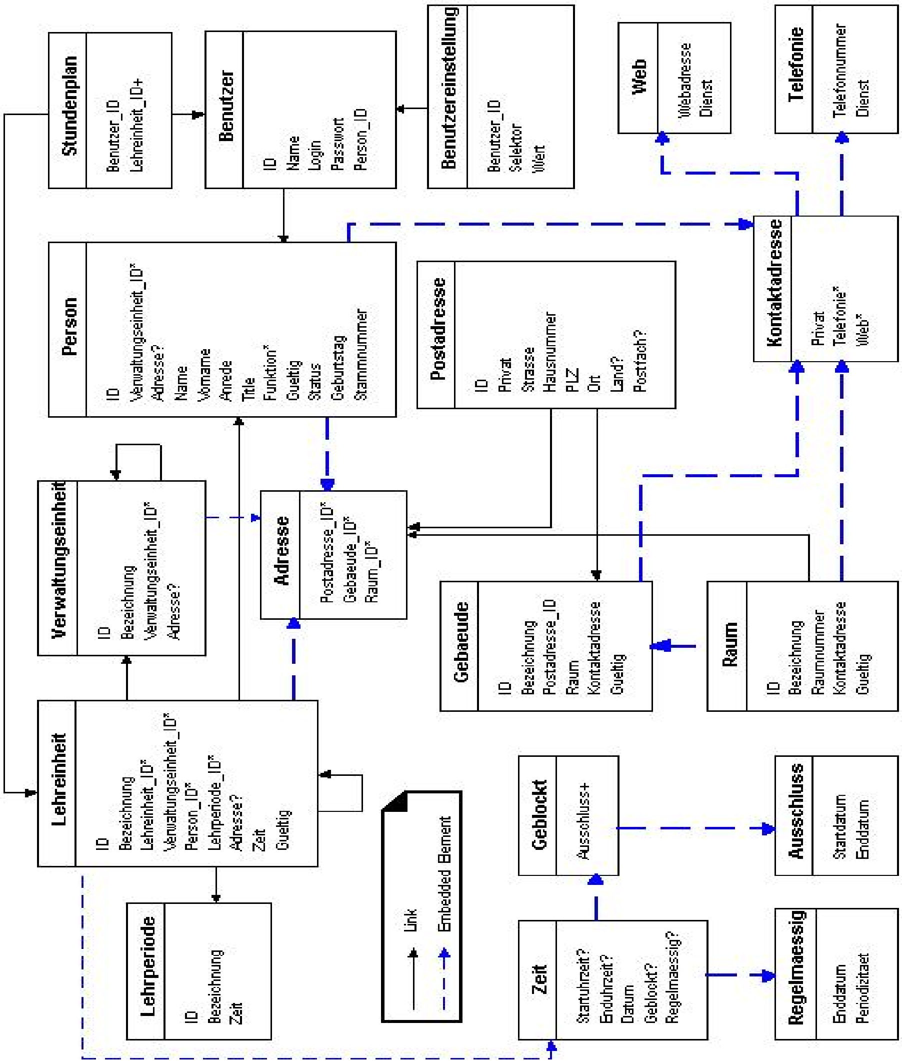

Abbildung 8: "The data model"

B DTD of the project

```
<!ELEMENT    root
             ((Postadresse | Verwaltungseinheit | Person |
             Lehreinheit | Lehrperiode | Benutzer |
             Benutzereinstellung | Stundenplan | Gebaeude)*)>

<!--******* Time *******************************************-->

<!ELEMENT    Zeit (Startuhrzeit?, Enduhrzeit?, Datum,
             Geblockt?,Regelmaessig?)>
<!ATTLIST    Zeit
             Tempore CDATA #IMPLIED>

<!ELEMENT    Geblockt(Ausschluss+)>
<!ELEMENT    Ausschluss(Startdatum,Enddatum)>
<!ELEMENT    Regelmaessig (Periodizitaet, Enddatum)>
<!ELEMENT    Datum (#PCDATA)><!ELEMENT Wochentag (#PCDATA)>
<!ELEMENT    Periodizitaet (taeglich | woechentlich | monatlich | jaehrlich)>
<!ELEMENT    taeglich (#PCDATA)>
<!ELEMENT    woechentlich (#PCDATA)>
<!ELEMENT    monatlich (#PCDATA)>
<!ELEMENT    jaehrlich (#PCDATA)>
<!ELEMENT    Startuhrzeit (#PCDATA)>
<!ELEMENT    Enduhrzeit (#PCDATA)>
<!ELEMENT    Startdatum (#PCDATA)>
<!ELEMENT    Enddatum (#PCDATA)>

<!--******* Fax, Email,etc. *********************************-->

<!ELEMENT    Kontaktadresse (Web*, Telefonie*)>
<!ELEMENT    Web (Webadresse)>
<!ATTLIST    Web
             Dienst CDATA #IMPLIED>

<!ELEMENT    Telefonie (Telefonnummer)>
<!ATTLIST    Telefonie
             Dienst CDATA #IMPLIED>

<!ELEMENT    Webadresse (#PCDATA)>
<!ELEMENT    Telefonnummer (#PCDATA)>
<!ELEMENT    Postadresse (Strasse, Hausnummer, PLZ, Ort, Land, Postfach?)>
<!ATTLIST    Postadresse
             ID ID #IMPLIED Privat CDATA #IMPLIED>

<!ELEMENT    Strasse (#PCDATA)>
<!ELEMENT    Hausnummer (#PCDATA)>
<!ELEMENT    PLZ (#PCDATA)>
<!ELEMENT    Ort (#PCDATA)>
<!ELEMENT    Land (#PCDATA)>
<!ELEMENT    Postfach(#PCDATA)>
```

```
<!--******* Verwaltungseinheit ******************************-->

<!ELEMENT   Verwaltungseinheit (Bezeichnung+, Beschreibung*, Adresse?,
            Verwaltungseinheit_Link*)>
<!ATTLIST   Verwaltungseinheit
            ID ID #IMPLIED>

<!ELEMENT   Adresse (Postadresse_Link*, Raum_Link*, Gebaeude_Link*)>

<!--******* Person ******************************************-->

<!ELEMENT   Person
            (Name, Vorname, Anrede, Titel?, Bezeichnung+, Funktion*,
            Geburtstag?, Beschreibung*, Adresse?, Kontaktadresse*,
            Verwaltungseinheit_Link*)>
<!ATTLIST   Person
            ID ID #IMPLIED
            Gueltig CDATA #IMPLIED
            Status CDATA #IMPLIED>

<!ELEMENT   Name (#PCDATA)>
<!ELEMENT   Vorname (#PCDATA)>
<!ELEMENT   Anrede (#PCDATA)>
<!ELEMENT   Titel (#PCDATA)>
<!ELEMENT   Funktion (#PCDATA)>
<!ELEMENT   Geburtstag (#PCDATA)>
<!ELEMENT   Stammnummer (#PCDATA)>

<!--******* Lehreinheit *************************************-->

<!ELEMENT   Lehreinheit
            (Bezeichnung+, Zeit, Adresse?, Beschreibung*, Lehreinheit_Link*,
            Person_Link*, Verwaltungseinheit_Link*, Lehrperiode_Link*,
            Lehreinheitstyp*)>
<!ATTLIST   Lehreinheit
            ID ID #IMPLIED
            Gueltig CDATA #IMPLIED>

<!ELEMENT   Lehreinheitstyp (#PCDATA)>
<!ELEMENT   Lehrperiode (Bezeichnung+, Zeit)>
<!ATTLIST   Lehrperiode
            ID ID #IMPLIED>

<!--******* Modeling of the user ****************************-->

<!ELEMENT   Benutzer (Login, Passwort, Person_Link)>
<!ATTLIST   Benutzer
            ID ID #IMPLIED>
```

```
<!ELEMENT    Login (#PCDATA)>
<!ELEMENT    Passwort (#PCDATA)>
<!ELEMENT    Benutzereinstellung (Benutzer_Link, Einstellung)>
<!ATTLIST    Benutzereinstellung
             Selektor CDATA #IMPLIED>

<!ELEMENT    Einstellung (#PCDATA)>
<!ELEMENT    Wert (#PCDATA)>
<!ELEMENT    Stundenplan EMPTY>
<!ATTLIST    Stundenplan
             Benutzer_ID IDREF #IMPLIED
             Lehreinheit_ID IDREFS #IMPLIED>

<!--******* Buildings and Rooms ******************************-->

<!ELEMENT    Gebaeude (Bezeichnung+, Raum*, Beschreibung*,
             Kontaktadresse, Postadresse_Link)>
<!ATTLIST    Gebaeude
             ID ID #IMPLIED
             Gueltig CDATA #IMPLIED>

<!ELEMENT    Raum (Bezeichnung*, Kontaktadresse)>
<!ATTLIST    Raum
             ID ID #IMPLIED
             Raumnummer CDATA #IMPLIED
             Gueltig CDATA #IMPLIED>

<!--******* Links *******************************************-->

<!ELEMENT    Verwaltungseinheit_Link EMPTY>
<!ATTLIST    Verwaltungseinheit_Link
             Verwaltungseinheit_ID IDREF #IMPLIED
             Rolle CDATA #IMPLIED>

<!ELEMENT    Lehreinheit_Link EMPTY>
<!ATTLIST    Lehreinheit_Link
             Lehreinheit_ID IDREF #IMPLIED
             Rolle CDATA #IMPLIED>

<!ELEMENT    Person_Link EMPTY>
<!ATTLIST    Person_Link
             Person_ID IDREF #IMPLIED
             Rolle CDATA #IMPLIED>

<!ELEMENT    Postadresse_Link EMPTY>
<!ATTLIST    Postadresse_Link
             Postadresse_ID IDREF #IMPLIED>

<!ELEMENT    Benutzer_Link EMPTY>
<!ATTLIST    Benutzer_Link
             Benutzer_ID IDREF #IMPLIED>
```

```
<!ELEMENT    Raum_Link EMPTY>
<!ATTLIST    Raum_Link
             Raum_ID IDREF #IMPLIED>

<!ELEMENT    Gebaeude_Link EMPTY>
<!ATTLIST    Gebaeude_Link
             Gebaeude_ID IDREF #IMPLIED>

<!ELEMENT    Lehrperiode_Link EMPTY>
<!ATTLIST    Lehrperiode_Link
             Lehrperiode_ID CDATA #IMPLIED>

<!--******* Global Elements ***********************************-->

<!ELEMENT    Beschreibung (#PCDATA)>
<!ATTLIST    Beschreibung
             Typ CDATA #IMPLIED>

<!ELEMENT    Bezeichnung (#PCDATA)>
<!ATTLIST    Bezeichnung
             Typ CDATA #IMPLIED>
```

Literatur

[CD99] James Clark and Steve DeRose. Recommendation: XML Path Language (XPath) Version 1.0. 1999. http://www.w3.org/TR/xpath. 4

[CFMR01] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. W3C Working Draft: Xml Query Requirements. 2001. http://www.w3.org/TR/2001/WD-xmlquery-req-20010215. 4

[Cla99] James Clark. Recommendation: XSL Transformations (XSLT) Version 1.0. 1999. http://www.w3.org/TR/xslt. 4

[CRF00] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. *In Lecture Notes in Computer Science, Springer-Verlag*, 2000. http://www.almaden.ibm.com/cs/people/chamberlin/quilt. 21

[DFLS99] Alin Deutsch, Mary Fernandez, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. 1999. http://www.w3.org/TR/NOTE-xml-ql/. 21

[DMO01] Steve DeRose, Eve Maler, and David Orchard. Recommendation: XML Linking Language (XLink) Version 1.0. 2001. http://www.w3.org/TR/2000/REC-xlink-20010627. 4

[ST01] Airi Salminen and Frank Wm. Tomba. Proceedings of the 27th VLDB Conference: System Desiderata for XML Databases. 2001. 21

[W3C] W3C. Recommendation: XHTML Version 1.0. http//www.3.org/MarkUp/#xhtml1. 17