NODAL: Structure and Reference for Web Documents

Lee Iverson[1]

University of British Columbia
Dept. of ECE, 2356 Main Mall, Vancouver, BC Canada V6T 1Z4
leei@ece.ubc.ca,
http://www.ece.ubc.ca/~leei

Abstract. The Web has amply demonstrated the benefits of simple publishing and reference but these benefits are largely limited to the granularity of individual files. The potential for greater benefits that may derive from sub-document reference is currently being explored, but this exploration is limited by the fact that such references can only be used with a small portion of the Web's content (i.e. that encoded in XML). We have developed a system, the Network-Oriented Document Abstraction Language (NODAL), that is designed to provide such a sub-document reference system for documents encoded in any format. Moreover, the data model can be mapped over database schemas and thus provide a common reference and access environment for structured, semi-structured and unstructured data. Finally, we outline a number of ways that this system can be extended to allow for composition, synchronization and reuse of documents and databases in this form and can thus form a hypertextual foundation for interactive application development without inhibiting interoperability with existing systems.

1 Overview

The Web has created an unprecedented ability to publish and distribute documents and other data and to augment this content with hypertextual references. However, the granularity of these references is, for the most part, limited to the file level. The combination of a need to reuse and/or quote sources smaller than a whole document (e.g. Nelson's transclusion[1]) and the advent of new technologies built on hypertextual reference (e.g. RDF[2] and the Semantic Web[3]) has stimulated a desire to provide a means of referencing only parts of a document. In traditional HTML this was accomplished with named anchors, but was thus limited to only those *points* in a document anticipated by the author. XPointer[4], XPath[5] and XLink[6] provide sub-document linking for XML documents but as yet only a small fraction of the Web is available as XML.

Instead of restricting sub-document reference to markup languages, we have designed NODAL, the Network-Oriented Document Abstraction Language, as a simple middleware layer that provides a means of defining URI-referenceable structure for all document repositories and other structured data sources. It

achieves this end by defining a simple data model and schema language that can be used to define fragment references for any document type. A plugin architecture is used to associate format encoder/decoder pairs with MIME types that identify particular document formats. This system thus defines a common language for referencing and accessing the structured contents of arbitrary document formats at arbitrary granularity. Moreover, it can be used in conjunction with other sub-document reference schemes without interference.

Finally, the system provides a base for future development and research towards new software architectures and development paradigms that allow for a much more seamless interconnection between data and documents across applications, formats, database structures and distribution protocols.

2 The Problem: Granular Reference, Content Reuse and Annotation

Consider an engineering student working on a class project. The course has a website that contains notes from classroom sessions as well as a set of links to online resources to learn more about certain subjects. The student exchanges email with other students about his project and asks questions on an online forum and mailing list that help him successfully complete his project. At the end, he has a design document, a project report and an artifact that is a combination of a software system and some hardware described in a CAD model. Through this entire project, he has used a few dozen sources of information (most online), at least half a dozen different software systems, and between his project notes and reports he has written a few thousand lines of text to describe and justify his design. How then is this information structured to facilitate the work itself and to communicate its products? If he was using broadly accepted practices then his working environment consists of a project directory on his desktop system, project and class folders in his email client, and a set of bookmarks for selected information sources in his browser. But how are they connected? Where is the justification for a particular design feature in the CAD model? Where is the connection between his project document and the email conversation and forum entries in which some critical issues were explained?

This kind of scenario is familiar to most hypertext researchers and some of these issues are well described by Ted Nelson in a recent paper[1] summarizing his goals and motivations of the past 40 years and a prescription for "Xanalogical structure". Unfortunately, like many hypertext systems of the past (e.g. Microcosm[7] and Intermedia[8]), he suggests that the solution lies in an incompatible, new system and data model that will only act as its own kind of information silo. Instead we suggest that the only way that this can be made to work (and have any possibility of wide uptake) is if we strive to develop a system that interoperates seamlessly with existing environments and applications (e.g. as in Chimera[9]).

In a separate paper[10], we have analyzed this general class of issues and various kinds of "information silos" (e.g. applications, data formats, operating

systems, database systems). We suggested that a change to the assumed application architecture would allow for the deep linking, content reuse and general annotation facilities necessary to build general personal (and group) information management environments. This application architecture (the DKC Model) contains a persistent storage layer at its base (the *Data* layer) that forms the basis for sharing, linking, reuse and versioning of data and structure. A *Knowledge* layer that can be used to imbue this data with semantic meaning, and then a final, independent layer of *Context* that manages user interface, interaction and modelling. By advocating the independence of *view* and *storage* (in the Context and Data layers), we suggest that this model will not only allow personal and group information management to finally begin to deal with cross-platform, cross-application issues but that we will enable greater innovation and adaptation in the user- and task-oriented spaces often considered by HCI and Hypertext researchers.

3 A Solution: Document Data Modeling and Reference

The requirements elucidated for the Data layer in the DKC model include: a structural data model, both database and filesystem (or document-based) views, and a standard, granular, sub-document reference system. The NODAL system defines a data model that is a superset of the relational model that can be applied to modeling document formats in such a way as to provide a basis for a URI-based fragment references for any kind of modeled document. Moreover, with the architecture outlined below, we show how this can also be extended to a wide variety of data access protocols, some database-like and some filesystem-like.

4 The NODAL Data Model

The challenges are to develop or adapt a data model that has clear application to distributed storage systems, provides a framework for both absolute and relative URI-based reference, and maps naturally to document formats, database schemata and application-level APIs. Moreover, for some of the more advanced functionality we will discuss later, it is important to distinguish which units of data will have metadata associated with them.

The fundamental design constraints we settled on are summarized as follows:

- Use a type system to model the structural and value constraints that characterize the particular data encoded by a particular format. This means that each data format or database schema should be expressable in the NODAL data modelling language with a new schema or via reference to an existing schema.
- Clearly separate data modeling from syntax. A common data model for a variety of data formats requires this independence, although there are certainly situations in which there must be a standard syntax for certain uses of the model (e.g. in coding references as URIs).

– Don't invent new data types or structures. Compatibility of this model with existing programming languages and data storage models is a primary concern. We are primarily interested in providing a minimally sufficient model that can be adapted for a wide range of purposes.
– Encourage standardization and reuse of data schemas by designing a language that encourages granular reuse and composition of schemas.
– Distinguish between *immutable* and *mutable* types. We need both, but immutable objects are more distributable (reference equals copying). The only objects that may need metadata such as change history and access control are the mutable ones.

We believe that these constraints follow naturally from both programming language and data modeling experience and from the overwhelming need for both backward compatibility and extensibility. If we ignore these requirements, our goal of providing a model that can be the basis for seamless information integration will fail.

4.1 Literal Data Types

The literal, immutable data types were chosen by combining the type systems of XML Schema: Part 2[11], the SQL99 standard[12], and modern programming languages. No significant explanation should be required to justify the set of literal data types shown in Table 1. Where appropriate, a reference is provided to a standard that describes the storage format and textual expression of each type. The Name type is the only one of these that may need explanation. It is an immutable sequence of characters with an optional namespace (specified by a URI as in XML namespaces). This type is distinguished from the String type (a Sequence of characters) that allows modification.

Table 1. Literal data types for NODAL data model

Type Name	*Description*	*Standard*
Boolean	A true or false value	
Character	A single character	ISO 10646
Octet	An 8-bit unsigned integral value (0-255)	
Short	A 16-bit signed integral value	
Integer	A 32-bit signed integral value	
Long	A 64-bit signed integral value	
Float	A 32-bit floating point value	IEEE 754[13]
Double	A 64-bit floating point value	IEEE 754[13]
Name	An immutable character string	
Timestamp	A single moment in time	ISO 8601[14]

One characteristic of these atomic types is that their identity is completely described by their content. This combination of content-defined identity and

immutability is usually described as a "value" type in computer language design. Their advantages for distributed computing applications is well known: they are inherently distributable, since all copies are identical.

4.2 Structured Data: Collections as Nodes

To support dynamic, structured data, we then add to these literals a system of structured, modifiable types that we refer to as *nodes*. We define the node types N such that for $t \in N$ we have $t = \{(a_i, v_i)\}$, a finite set of attribute/value pairs where $a_i \in A(N)$ and $v_i \in V_i(N)$. Thus, for any node type N, we have a domain $A(N)$ that constrains the attributes of $t \in N$ and a domain $V_i(N)$ that constrains the values v_i that may be associated with attribute $a_i \in A(N)$. By varying the constraints on $A(N)$ and $V_i(N)$ we can define a variety of different classes of node types, while still maintaining this common attribute/value model. Below, we describe these constraints with set functions that compose new domains (node types) using existing domains (types). See also the UML diagram in Fig. 1.

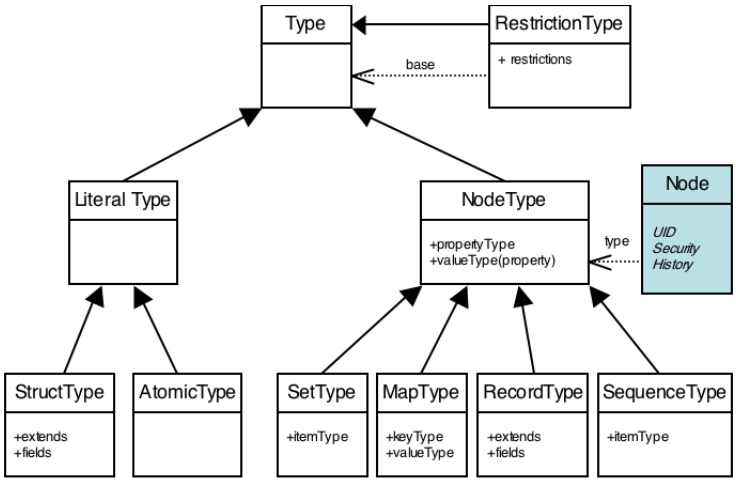

Fig. 1. NODAL Data Model. UML inheritance diagram of basic NODAL type system.

Map The simplest of these is the map $\mathrm{M}(A, V)$ for two domains $A \subset T$ and $V \subset T$ whose instances associate any attribute $a \in A$ with a value $v \in V$.

Set In this formulation, a set $\mathrm{St}(A)$ is a kind of map for which $A = V$ and for which $t \in \mathrm{St}(A) \rightarrow (\forall (a_i, v_i) \in t : a_i = v_i)$. In other words, if we define a kind of map for which the attributes and values are always identical, then we can use this as a set.

Sequence A sequence $\mathrm{Sq}(V)$ where the attributes $A(\mathrm{Sq}(V)) = [0, ..., n]$ form an interval and all values are in the domain $V \subset T$.

Record A record $R = \mathrm{R}(A, \{V_i\})$ is defined by a fixed set of names $A(R) = \{a_i \in \texttt{name}\}$ and the domains of the values associated with each name $V_i(R) \subset T$. We refer to the names a_i as *fields* of the record R and the domain $V_i(R)$ is the *field type* of field a_i in R. We further define an *inheritance* relationship for record types such that if $a \in A(R') \rightarrow a \in A(R)$ and $a \in A(R') \rightarrow V(R, a) \subset V(R', a)$ then we say that R is derived from R'. Clearly, these conditions ensure that $R \subset R'$ even if R has extra fields $a \in A(R)$ such that $a \notin A(R')$. Moreover, this is clearly compatible with the existing relational model as outlined in [15].

For example, a `Document` record has a field `mime-type` for the MIME-type of document that is a Name that matches a particular regular expression. The record type for `XMLDocument` includes an override of the `mime-type` field to a fixed value: "text/xml". Note that this is a restricted but compatible specification for the value of the field. Thus the Record types can be used to form an object inheritance hierarchy where kind-of restrictions can be maintained. And example of the declaration of Record types and field shadowing is shown in Fig. 2, an excerpt from the standard NODAL data model for the basic NODAL types.

Structures In order to support a lightweight, compound literal type, we also provide a means of defining record-like literals, which we call a *structures*: $\mathrm{Str}(A, \{V_i\})$. Interpreted exactly as a record type of the same form (with the additional restriction that all of the V_i types must be literals), these are extremely useful for modelling data that have natural *value* semantics but are still decomposable into meaningful components (e.g. RGB pixels in an image).

Node Identity and Metadata Since the node types are modifiable (and thus not uniquely identified with their contents), we require some sort of external identity to be able to maintain stable references to nodes. These node IDs can be anything unique within an identifiable context (e.g. a database table, a file, a repository). For example, in a native NODAL repository, it is likely that we will have simple node IDs that are unique within the entire repository. In a traditional filesystem-based repository it is more natural to have node IDs uniquely determined within the context of the containing document or file. In a relational database mapping, we can simply use the primary keys of each database table as the unique node ID. The flexibility of this requirement should allow us to find such a system of unique IDs for any of the different kinds of data sources we have proposed supporting within this framework.

Given the recognition of this requirement, and the identification of these nodes as the minimal units of modification, it is also natural to suggest that these nodes be the minimal units for recording change history and access control.

Literal Structures Because of the need to have structured types without the metadata and identity overhead of the Node types (e.g. an RGB pixel in an image), we also allow for the definition of new *structured literal* types, using

```
...
  <!--
  The Document type, an encapsulation of a graph of Nodes
  -->
  <record name="Document">
    <field name="mimeType" type="Name"/>
    <field name="root" type="Node"/>
  </record>

  <!--
  The Directory type, a Document that is simply a mapping of names to
  Documents.
  -->
  <record name="Directory" extends="Document">
    <field name="root">
      <map keyType="Name" valueType="Document"/>
    </field>
  </record>
...
```

Fig. 2. An excerpt from the `baseTypes.nls` schema that defines the basic standard data types in the system. An example of type restriction by record inheritance and field shadowing is shown, with a Directory a kind of Document that contains a mapping from Name to Document.

a mechanism very similar to the Record type. In these Struct types, a set of field names are defined and associated with value types in exactly the same way as with Record types, except that the objects created are immutable and have identity associated with their contents. This is a very useful, lightweight tuple type that has application in a variety of different contexts.

Schemas Given these building blocks, a *schema*, or set of types, can be considered as a model of the constraints on a set of interconnected, structured values. One way of tightening the constraints on a particular type or particular context is to use a restriction language to limit the value space of a particular type. This is a very powerful concept and is the foundation for the XML Schema type system[11] and the ISO standard type system from which it is derived[16]. To this end, we provide just such a type *restriction* facility for atomic types based on a set of matching functions applied to a base type. Some of the restrictions available are: regular expressions for Name and String types; inclusive and exclusive minima and maxima for any ordered type; a namespace restriction for Name objects; and an enumeration list for any atomic type (including the single-valued enumeration or fixed value).

4.3 Documents and Node Graphs

Since this model is clearly a superset of the relational model, it can obviously be used to describe structured data stored in a relational database. How then is it also useful for modelling documents in a filesystem? Simply by associating a particular document format with a root node type in some schema that models the structure of the data contained within those documents. In the Web-based NODAL system, this is done by associating a MIME type[17] that identifies the format with a `DocumentFormat` class that defines the type of the root node and encoder and decoder methods that translate between a bitstream and instances of an associated schema (see Fig. 2). A file then corresponds to the graph of the nodes reachable from the root node. In this way, we can integrate document and database accesses with a common API and, given the reference architecture we describe next, a common reference language.

The choice of the term Node for these collection objects should now be clear. A document in this kind of model consists of set of Node objects with properties that are either literal or references to other Node objects. These Node-Node properties can be considered to be the labelled edges of a graph that we refer to as the *document graph*. Unlike XML though, this graph is not restricted to being a hierarchy. Parent-child relations are one-way, although a structural query can be used to recover the many possible parents (and document containments) of a particular Node.

5 The NODAL Path Language

In order to provide an external reference system for this data model, we adopt the path language approach of XPath[5] and define URI references based on a navigational path through a document graph. A path $\mathbf{p} \in \mathbf{P}$ in NODAL is a chain of path components $\mathbf{p} = [p_1, ..., p_n]$. Using the *concatenation* operator $\mathbf{p} = \mathbf{p}'/p_n = [p_1, ..., p_{n-1}]/p_n$, we can define the *tail* of a path $\mathbf{p}$ as the final component $\mathbf{p}^* = p_n$ and the *parent* of a path $\mathbf{p}$ as the path $\mathbf{p}'$ such that $\mathbf{p} = \mathbf{p}'/\mathbf{p}^*$. Note that neither of these exist if $\mathbf{p}$ is the empty path $[]$, but that the parent of a single-component path is the empty path.

To interpret paths, we need some way of interpreting the path components individually and then in sequence. In this formulation, a path component has two aspects, its path normalization function and its binding function. Each of the components $p \in P$ has a path normalization function $\mathrm{N}_p : \mathbf{P} \rightarrow \mathbf{P}$ that takes a path and produces a *normalized* path $\mathbf{p}$ for which $p = \mathbf{p}^* \rightarrow \mathrm{N}_p(\mathbf{p}') = \mathbf{p}$. Thus, any path component p that appears as the tail of a normalized path has the property that $\mathrm{N}_p(\mathbf{p}) = \mathbf{p}/p$. Thus concatenation is the standard normalizing function. An example of a component that does not always concatenate is the *parent* operator `..` that extracts the parent path. The normalizing function for the `parent` component is

$$\mathrm{N}_{..}(\mathbf{p}) = \begin{cases} [..] & \text{if } \mathbf{p} = \emptyset, \\ \mathbf{p}' & \text{otherwise.} \end{cases}$$

So, the parent operator can only appear as the first component of a normalized path.

But paths are defined to provide a means of accessing values within a node graph, so we need a means to determine the target value of a path $\mathrm{V}_{\mathbf{p}}$. To evaluate this target value, we consider another aspect of a a path component, its *binding function* $\mathrm{V}_p : \mathbf{P} \to T$, which returns the value that a path with p at its tail refers to. We then define the target of a path with respect to its tail as:

$$\mathrm{V}_{\mathbf{p}} = \mathrm{V}_{\mathbf{p}^*}(\mathbf{p}).$$

We distinguish two kinds of path components, the *absolute* components have values that are independent of the containing path $\mathbf{p}$, while the *relative* components have dependent values. A relative path is then a path that contains only relative path components, whereas an absolute path contains at least one absolute component. To reduce path redundancy we require that the normalization function of every absolute path component produce a single component path containing itself:

$$\text{if } p \text{ is absolute then } \mathrm{N}_p(\mathbf{p}) = [p],$$

Thus an absolute path has exactly one absolute component at its head. Examples of some of the available components are show in Table 2.

Table 2. Some of the basic path components as normalization and binding functions. Note that in the evaluation of V_p functions we use the notation $v.foo$ to specify the value of property foo of the node v.

p description	N_p	V_p	Functional Form	Shorthand
document *doc*	$\mathbf{p} \Rightarrow [doc]$	V_{doc}		*doc* URI
node *id*	$\mathbf{p} \Rightarrow [id]$	node_{id}	`nid(`*id*`)`	
parent	$\mathbf{p} \Rightarrow \mathbf{p}'$	none	`parent()`	`..`
fragment root	$[doc]/\mathbf{p} \Rightarrow [doc]/p$	$\mathrm{V}_{doc}.root$	`root()`	`#/`
property g	$\mathbf{p} \Rightarrow \mathbf{p}/p$	$\mathrm{V}_p.g$	`property(`g`)`	g
range of i to j	$\mathbf{p} \Rightarrow \mathbf{p}/p$	$\mathrm{V}_p.\,\mathrm{range}(i,j)$	`range(`i`,`j`)`	

So far, these paths are independent of the data model outlined above. To provide some grounding, it will be useful to consider paths within documents. Remember that a document is modelled as the graph of nodes reachable from a root node. (see Sec. 4.3). Thus, a reference to a document determines the starting point for navigation within the node graph. An absolute path thus has two parts, the document part and the fragment part. If the document part is not empty, then the fragment part is evaluated relative to the specified document's root node. We can now appreciate one of the main advantages of the unification of the node/collection data model in terms of attribute/value pairs: a homogenization of the standard component for selecting a value in a collection, namely `property(`g`)`. From a single absolute root path, we can create paths to

all reachable nodes and values with only chains of these `property` components. The property component is thus the fundamental building block of the relative reference mechanisms in the NODAL path language.

Finally, each path component is expressible as a function or shorthand in text, and a path URI is simply the concatenation of these component expressions with an appropriate separator (in the fragment part of a path, the `'/'` character is used as a separator). As with XPointer[4] fragment URIs, we use a context frame `#ndl(...)` to enclose NODAL fragment expressions. Other fragment expressions are passed to the plugin responsible for the MIME type of the document addressed. Thus, HTML and XML documents can properly handle `#id` fragment ids and even `xpointer(...)` expressions without interfering with the NODAL references.

So, given the components described in Table 2 (a subset of the component operators available in NODAL) we can describe a number of NODAL path expressions as examples:

URI	Description
`http://sf.net/index.html#ndl(/)`	The root node of the document `http://sf.net/index.html`
`#ndl(../foo)`	The property named `foo` of the node that is the parent of the path to be applied to.
`file:/doc.txt#ndl(15/range(4,16))`	The characters between index 4 and 16 on line 15 of the local text file `/doc.txt`

5.1 Anchors

As in HTML and the Dexter Reference Model[18], indirect references to Nodes in other documents or repositories are enabled by the creation of a special kind of Node, the Anchor. An Anchor is essentially a proxy for another node, and is completely specified by a *path*. When the Anchor is encountered, it evaluates the binding of the Anchor's path in the context of the path *to* the Anchor (to handle relative references) and then acts as the node bound to this path. These anchors are thus of the style of Nelson's transclusion operators[1]. We also provide a facility (as a path operator) to interpret any String or Name as a path URI and then extract the binding. This is how we implement HTML-style hyperlinks.

6 The NODAL Architecture

This data model and path language are, of course, only useful in the context of a system to interpret them. The NODAL prototype is a Java middleware implementation of the data model and path language with a small set of proof-of-concept format plugins (text, HTML, XML, the NODAL schema language). This prototype is currently able to interact with data access protocols `file:`,

`http:` and `imap:`. All of these provide documents as bitstreams and thus need to pass through document decoders in the format plugins before being presented via access APIs.

7 Future Directions and Potential

We are currently testing the system in this data consumption mode and formulating a number of pilot projects to assess the ability to build working applications on top of the NODAL APIs. The NODAL type system is already implemented based on an interpretation of the data model described in the `baseTypes.nls` schema excerpted in Fig. 2. One of the most interesting possibilities is to create a personal information management environment that can operate by building semantic indices not only to local and remote files but also to their contents to provide granular annotation.

Future work is planned on the following items:

- Database interaction: Automatically extracting database schema and allowing the NODAL APIs to access relational and object database systems was one of the design goals and must now be implemented and tested.
- Read/write functionality for all data sources: Currently we can consume and reference data from a wide variety of sources but can as yet only write to local file systems.
- Search: It is important to provide mechanisms for data discovery that go beyond the simple exploratory metaphor provided by filesystem and link following. We must be able to search data based on content and structure.
- Versioning: As was stated above, the best unit for attaching metadata and managing version control is the Node. Each node has a specific structure and a limited number of possible changes. We can already generate change records and associate node versions with each transaction. The difficulty comes layering this functionality on top of inextensible data stores such as local filesystems.
- Access control: If versioning is best done at the node level, then perhaps access control is too.
- Synchronization: Once a version management is available, then it should be possible to enable CVS-like[19] synchronization between local copies and remote repositories. We suggest that it will be easier to automatically extract difference charts between local and remote modifications, since most of the node types have a very simple set of modification operators. In fact, the Simias Collection Store[20] being used by the iFolder project has a very similar structure to this one and it is completely designed for synchronization.

With the NODAL data model and path language, we have demonstrated a new paradigm for opening up *all* Web-accessible content (not just HTML and XML) to the advantages of hypertextual information management. We hope to extend it so that it can become a generic Data layer that can be the foundation for a next-generation of fully interoperable, collaborative end-user applications.

References

1. Nelson, T.H.: Xanalogical structure: Needed now more than ever: Parallel documents, deep links to content, deep versioning, and deep re-use. ACM Computing Surveys **31** (1999)
2. Lassila, O., Swick, R.: Resource description framework (RDF) model and syntax specification. The World Wide Web Consortium `http://www.w3.org/TR/WD-rdf-syntax` (1998)
3. Berners-Lee, T.: The semantic web roadmap. `http://www.w3.org/DesignIssues/Semantic.html` (1998)
4. DeRose, S., Maler, E., Jr., R.D.: XML pointer language (XPointer). The World Wide Web Consortium `http://www.w3.org/TR/WD-xptr` (2000)
5. Clark, J., DeRose, S.: XML linking language (XLink). The World Wide Web Consortium `http://www.w3.org/TR/xlink` (1999)
6. DeRose, S., Maler, E., Orchard, D.: XML path language (XPath). The World Wide Web Consortium `http://www.w3.org/TR/xpath` (2001)
7. Fountain, A.M., Hall, W., Heath, I., Davis, H.: MICROCOSM: An open model for hypermedia with dynamic linking. In: European Conference on Hypertext. (1990) 298–311
8. Yankelovich, N., Haan, J.B., Meyrowitz, N., Drucker, S.: Intermedia: The concept and the construction of a seamless information environment. IEEE Computer **21** (1988) 81–96
9. Anderson, K., Taylor, R., Whitehead, E.: Chimera: Hypermedia for heterogeneous software development environments. ACM Transactions on Information Systems **18** (2000) 211–245
10. Iverson, L.: Data, knowledge, context: An application model for collaborative work. (2004 (submitted))
11. Biron, P.V., Malhotra, A.: XML Schema part 2: Datatypes. The World Wide Web Consortium `http://www.w3.org/TR/xmlschema-2/` (2001)
12. ISO/IEC 9075:1999(E): Information technology - Database languages - SQL. International Organization for Standardization (1999)
13. IEEE Standard 754-1985: Binary Floating-Point Arithmetic. IEEE Computer Society (1985)
14. ISO/IEC 8601:2000: Representations of Dates and Times. International Organization for Standardization (2000)
15. Ramakrishnan, R., Gehrke, J.: Database Management Systems. 3rd edn. McGraw-Hill (2003)
16. ISO/IEC 11404:1996: Language-independent Datatypes. International Organization for Standardization (1996)
17. Freed, N., Borenstein, N.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. The Internet Society `http://www.ietf.org/rfc/rfc2045.txt` (1996)
18. Halasz, F., Schwartz, M.: The Dexter hypertext reference model. Communications of the ACM **37** (1994) 30–39
19. Berliner, B.: CVS II: Parallelizing software development. In: Proceedings of the USENIX Winter 1990 Technical Conference, Berkeley, CA, USENIX Association (1990) 341–352
20. Lasky, M.: The Simias collection store model. Novell Corporation `http://forge.novell.com/modules/xfmod/docman/?group_id=1372` (2004)