

Content Management for Declarative Web Site Design

Richard Cooper [?] and Michael Davidson

[?] Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow G12 8QQ
rich@dcs.gla.ac.uk

Abstract. Web site design and construction is characterised by the use of a number of poorly integrated technologies – mark-up languages, databases, server side middleware, client side scripting languages, etc. The absence of an integrated structure means that reusability and site maintenance are difficult to achieve. Instead of this, we are building a component-based declarative web site design system, in which all aspects of a site (whole sites, pages, style sheets, layouts, data sources, scripts, HTTP objects and XHTML fragments) are described uniformly using a high level component model and the site is generated automatically from the component description. Managing content is clearly a vital part of this and this paper describes how a high level model is used to describe content.

Topics. New Architectures for Data Integration. Declarative Web Design. Relational and XML Data. Component Based Systems

1 Introduction

In many ways the advent of the web, which provides relatively easy ways to program complex distributed applications, has set back the application of software engineering principles considerably. There has been a rapid evolution of programming techniques from static HTML to the use of client-and server-side scripting, dynamic web programming, the inclusion of data drawn from a database and the use of XML[1]. This has meant firstly that web sites have evolved in an *ad hoc* manner and secondly that even the construction of a web site consists of detailed programming using multiple programming systems with little or no emphasis on an overall structure or capability of overall integration. The tasks involved in web site design now include the construction of site structure, the creation of web page structures, the design of a database, programming the interaction between dynamic pages and the databases, providing content management, designing the interaction of the site visitor with the site, and adding client side scripts. Each of these tasks is carried out with varying degrees of separation, little or no integration and with a multiplicity of programming tools. Maintainability and re-use suffer as a consequence.

Alternatively, web design is carried out using tools (such as Dreamweaver and FrontPage) which support the specification of the surface of the site in an unsystematic manner, from which an implementation is generated. Furthermore, the web ac-

cessed through a PC, is just one of an array of emerging channels of interaction, such as PDAs, interactive telephone and interactive TV. To build a system which can only be deployed over the web is a less and less useful achievement. Finally, implementation decisions permeate the design, which is particularly problematic since the specific technology used to implement the site is itself subject to rapid change with new languages, data management systems and middleware products emerging regularly. The only clear benefit from the way sites are currently built, is that there is scope for the separation of concerns, since individuals can concentrate on the development of one aspect of the site.

What is lost in all of this is a coherent and maintainable high level view of the site, capable of being implemented using a variety of techniques. A modern site is essentially a large and complex piece of software and as such should be subject to acceptable software engineering practice. *Declarative Web Site Design* [2, 3, 4, 5, 6, 7, 8] takes the view that the implementation technology used should not drive the design of the web site, but rather the design should start with a high level specification which is easily maintainable and from which one or more implementations can be derived. This means that little time is spent on implementing minor detail and that the site can be re-engineered to take advantage of emerging implementation products without requiring a complete redesign or having to reprogram each component explicitly.

The particular approach taken here is to create the design in terms of *components*, each of which being drawn from one of the aspects listed above. Thus any of the following are components in our model: web pages, XHTML elements, data sources, style sheets, page layouts, scripts and HTTP constructs such as cookies and session variables. Components are created in terms of a component hierarchy, with the most abstract component being at the top and concrete components which take part in the site at the bottom. At any time, a designer can add fresh components by specialising any existing component. For instance, the abstract component for XHTML *table* elements can be specialised to tables particular to football leagues, and further to a concrete component specific to a particular football league.

The main purpose of this approach is that components built using this model will be able totally to describe a web site and be used to generate an implementation in one of a range of implementation technologies. This provides the benefits of re-usability, maintainability and late commitment to specific implementation techniques.

This paper concentrates on the data source aspect of the overall model. We start from the notion that information to be disseminated over the web will normally be stored either as XML or in relational databases (RDBs), but may be held in a variety of other structures. Furthermore, we note that each technology has a different mix of strengths and weaknesses and that perhaps the storage of data arising in an enterprise will typically benefit from using a combination of XML and relational representations. We therefore propose to use an abstract data model for content which can be mapped to any single implementation technology or combination thereof. A secondary aim is for the system to suggest an appropriate mix of implementation techniques.

A brief description of the component model is given as Section 2 in order to provide context for the data source model described in Section 3. Section 4 describes a prototype tool that has been constructed to manage such data sources and Section 5 gives our conclusions and how we plan to proceed.

2. A Component Model for Declarative Web Site Design

In order to provide a more straightforward and tightly integrated mechanism for web site design, the model being developed is both high level and comprehensive. Each constituent part of a web site is described in a uniform way as an instance of a component hierarchy. Each component is parameterised and takes part in the inheritance hierarchy described above. Each component is described in XML and maintains a set of documentary meta-data such as the name of the component, the author, date of creation and a brief description.

The second level of the hierarchy comprises the most general component of each of the main *kinds* of component. For instance, the abstract component, *Web Site*, is described by the parameters which hold the uri, an optional site wide style sheet, a front page and the collection of the other pages in a site. Similarly, *Web Page* is the generic component for all web pages. It has parameters for the title, meta-tags, style sheet, layout and a collection of *Visible* components. *Visible* components are essentially abstract descriptions of XHTML fragments, so the abstract component, *Visible*, generalises over all XHTML fragments, having only a style parameter. Immediately below this lie abstract components representing each of the XHTML element types arranged according to the XHTML DTD [10]. For more details on the component model and the other tools for managing it, see [11].

Data Source is the component type for any source of site content which will be dynamically included into the pages. This may be a database, XML file, spreadsheet or other file structure – or indeed a mixture of these. In web site construction, the *Data Source* component is used when constructing dynamic pages or content management software. As such, it must reveal connectivity details and the range of data that can be used for page content, i.e. a set of queryable tables. In fact, to simplify the automatic generation process and to exploit query optimisation where available, the model expects a *Data Source* to expose a set of views and for all queries embedded in dynamic web pages to be of the form “select * from view”.

Therefore, a concrete *Data Source* component is parameterised by which kind of implementation structure it uses (i.e. relational database, XML file, spreadsheet, etc.), the connection parameters and the set of views which are accessible. The abstract data model underlying a data source is an entity based model capable of being implemented in any of the structures listed above and is described in more detail in Section 3. *View* is the component type for a method of extracting a collection of data from a data source. It includes a query (expressed in the query language described in Section 4), a description of the type of data returned (at present just a list of attribute names and types) and whether one or more entities will be returned.

As an illustration of the Component Model, here are the steps needed to describe a page which is intended to hold a sports league table:

1. A data source is set up, with a view which returns the values found in the table – team name, games played, wins, draws, losses and points, say.
2. In the component hierarchy will be found the abstract component, *Table*, which will be parameterised by a set of column names, and below this is *Dynamic Table*, which is parameterised instead with a view.

3. We create the concrete component *MyLeagueTable*, by instantiating the parameter with the view set up in step 1.
4. This component is embedded in the appropriate page component.
5. The generation software creates a dynamic web page which selects the view data and embeds it into the *Dynamic Table* template.

The next section describes the abstract structure of a data source and how this can be implemented in a mixture of relations and XML, while the subsequent section describes generic content management for this abstract model.

Entity	Property	Domain	Imp	PKey	Card	Null	Unique	Inverse
Book	ISBN	String(20)	R	✓	SV			
Book	Title	String (30)	R		SV			
Book	PubDate	Date	R		SV	✓		
Book	PageLength	Integer	R		SV	✓		
Book	Author	Author	R		MV			Author.Works
Book	PublishedBy	Publisher	R		SV	✓		Publisher.Publishes
Author	ID	Integer	R	✓	SV			
Author	Name	String (20)	R		SV			
Author	Nationality	String (10)	R		SV	✓		
Author	BirthDate	Date	R		SV	✓		
Author	DeathDate	Date	R		SV	✓		
Author	Gender	Gender	R		SV	✓		
Author	Photograph	Image	F		SV	✓	✓	
Author	Works	Book	R		MV			Book.Author
Publisher	ID	Integer	X	✓	SV			
Publisher	Name	String (20)	X		SV			
Publisher	Address	String (50)	X		SV	✓		
Publisher	Publishes	Book	X		MV			Book.Publisher

Figure 1 An Example Model

3. A Typed and Tightly Integrated Structure

The software being developed for data source description uses an abstract model based on entity types. This section describes the model, how it is mapped into the implementation technologies (we choose relations, XML and a mixture of the two, a examples) and an algorithm for automatically determining which technology to use for each fragment of the information to be stored.

3.1 The Abstract Data Model

The data model we have used is a standard entity-based model. A schema consists of a set of entity types each of which has a number of typed properties. Property types can either be base types or other entity types, can have single valued or multi-valued cardinality, and can have inverse relationships enforced on them. As such the model

resembles the ODMG data model [12], among many others. Each property can be indicated to be part of the primary key or separately as non-null or uniquely valued. The structure is flexible so that additional constraints on properties can be added, such as content placement on dynamic web pages in a collaborative web site design tool [13]. Shown here is the use of a special Gender base-level domain used to distinguish pronouns in a data extraction application [14].

Figure 1 shows a sample schema constructed using the model for a repository of book information. Most of the columns have an obvious meaning but the fourth column, labelled *Imp*, indicates the implementation technology expected to be used for this property – R(elational), X(ML) or F(ile). The latter is used for multimedia properties. The entry in this column is not absolutely required, as will be seen, since the schema editor has the capability of automatically determining the appropriate mechanism for each property.

3.2 Mapping to Implementation Structures

The model just described is an excellent starting point for implementing a repository as a relational database or as one or more XML files, or as a hybrid of both.

3.2.1 Mapping to Relations

As the model is similar in many respects to the Entity-Relationship model, the standard technique for mapping an ER model to a set of tables can be used [15]. Entity types become tables, single valued base type properties become columns, while multi-valued properties become tables in their own right. Entity-typed properties become foreign keys, creating intersection tables if the properties are multi-valued in both directions. Thus the schema creation in SQL looks like:

```
create table Book (ISBN Varchar2(20) Primary Key,
                  Title Varchar2(30) Non Null,
                  PubDate Date,
                  PageLength Number2,
                  PublishedBy Number2 references Publisher.ID)

create table Author ( ID: Number2 Primary Key,
                    Name Varchar2(20) Non Null,
                    Nationality Varchar2(10),
                    BirthDate Date,
                    DeathDate Date,
                    Gender Char1 check = "F" or = "M",
                    Photograph BLOB unique)

create table Writes (Book Varchar2(20) references Book.ISBN,
                   Author Number2 references Author.ID,
                   Primary Key (Book, Author) )

create table Publisher ( ID: Number2 Primary Key,
                       Name Varchar2(20) Non Null,
                       Address Varchar2(50) )
```

3.2.2 Mapping to DTD

Alternatively, the schema turns easily into a DTD in a similar manner to that described in [16]. Fundamentally, the mechanism is to turn entity types into XML element types and properties into attributes using ID and IDREF types to manage the foreign keys. A DTD for the example if implemented entirely in XML is:

```
<!ELEMENT Library (Book*, Author*, Publisher*)>
<!ELEMENT Book EMPTY>
  <!ATTLIST Book ISBN ID #REQUIRED>
  <!ATTLIST Book Title CDATA #REQUIRED>
  <!ATTLIST Book PubDate CDATA #IMPLIED>
  <!ATTLIST Book PageLength CDATA #IMPLIED>
  <!ATTLIST Book Authors IDREFS #IMPLIED>
  <!ATTLIST Book PublishedBy IDREF #IMPLIED>
<!ELEMENT Author EMPTY>
  <!ATTLIST Author ID ID #REQUIRED>
  <!ATTLIST Author Name CDATA #REQUIRED>
  <!ATTLIST Author Address CDATA #IMPLIED>
  ... similar for BirthDate, DeathDate, Gender and
  Photograph
  <!ATTLIST Author Works IDREFS #REQUIRED>
<!ELEMENT Publisher EMPTY>
  <!ATTLIST Publisher ID ID #REQUIRED>
  <!ATTLIST Publisher Name CDATA #REQUIRED>
  <!ATTLIST Publisher Address CDATA #IMPLIED>
  <!ATTLIST Publisher Publishes IDREFS #IMPLIED>
```

3.2.3 Mapping to the Hybrid Model

There is a wide range of commercial software that has been produced for integrating XML and relational databases. Oracle, DB2 and SQL Server are typical of products which have sophisticated extensions and SQL/XML and XQuery promises product independent methodologies [9]. What we find when looking at the facilities provide by commercial software falls far short of full integration, covering at best:

- mapping from relations to XML and vice versa,
- input and output using XML data formats,
- storage of whole XML documents within tables,
- support for XML queries over RDBs, and
- support for DTDs and/or XML schema.

The most important weakness of the approach that is taken by these products is that they provide no way to type check a reference to an XML document. Ozone [17] supports a similar hybrid model, but cross-references use a standard type. Thus if one were to create an XML document full of conferences articles, it is not possible to have a cell in the relational database refer to one of those articles or to assert that the column contains only references to XML fragments tagged <Article>. Yet a table such as this might be a very suitable part of the database that is used to manage the information arising out of the conference.

Furthermore, the references between the two types of data storage are essentially one-way. It is not possible to refer to a record in a table from a point in an XML document. Yet, a reference to a supporting record might be the ideal way to add cita-

tions to an article. Taken together, this means that is impossible to enforce relational integrity between data held in an XML document and data held in a relational table using current technology. To integrate the two implementation technologies in this way requires two design decisions – how to represent the references in each direction.

Referring to a relational record in an XML entity is straightforward. The XML file must include a reference to the table and the primary key of the record. This could be achieved by including the reference as a child component using a special tag, but we have decided instead to use a specially named attribute, so that the management software can recognise that the reference is external to the XML file. Thus is the *Book* entity type was implemented as XML and the *Publisher* entity type as a table, then the *PublishedBy* property would appear as follows in the DTD:

```
<!ATTLIST Book RDB$PublishedBy CDATA #IMPLIED>
```

and a typical attribute would appear as:

```
RDB$PublishedBy = "Publisher:23"
```

Referencing XML from a relational database has a wider variety of possibilities. Storing XPATH and XPOINTER expressions are one possibility and storing XQUERY queries another. We have opted here for what is essentially the same technique used above, we place a character string in a conventionally named column which holds references to the XML element type and its ID. Thus the inverse of the above column requires an extra column in the *Publisher* table:

```
:create table Publisher( ID: Number2 Primary Key,  
Name Varchar2(20) Non Null,  
Address Varchar2(50)  
XML$Publishes Varchar2(100) )
```

in which the column *XML\$Publishes* might hold:

```
"Book:1224, 3456, 5678"
```

The management software works by identifying the conventionally named components and redirecting its attention to the appropriate implementation engine to follow links.

3.3 An Algorithm for Automatically Choosing the Implementation Structure

A secondary aim is to avoid the need for the designer to determine which types of data should use relational tables and which should use XML data sources. Database products typically require a database designer to specify which data should be stored in XML formats at design time. It is a central goal to our work that such decisions should be delayed to the last possible moment and to be achieved automatically if at all possible. The Schema Editor used to create a data source can remove the need for the designer to make the decision of where to use XML and where to use relations. We provide an algorithm for making the choice which could be used automatically, but in reality should be regarded as a first approximation to the final choice. We therefore

give the designer the ability to modify the decisions made. Making the choice in some cases the choice between XML and RDBMS tables appears to be very clear-cut:

- Given few elements containing large blocks of unstructured text we want XML.
- Given many elements with small data types we want a relational table
- If we have complex constraints we want a relational table.

However this is very simplistic approach will not work in all cases, for example what if we have large numbers of elements containing large blocks of unstructured text? Similarly, we might expect that the implementation decision made on a particular type be affected by the implementation decisions made on related types. The approach actually taken is that the choice between XML and RDBMS tables is determined using a weighting system. Each type will have a number of XML and RDBMS weights applied to it from the following table:

SQL Property		XML property	
Each non text property in type	5	Each long text element in type	20
Each short text element in type	1	Each reference to another type (1-M)	1
Each simple constraint	1	All element types are text (N = number of types)	2N
Has check constraints	50		

The heuristic also takes into account foreign key references to cut down cross references. A weight of 10 is added for the implementation type for each property implemented in that type. The XML or RDBMS property will be set according to which property gains the greatest weight. In the case of a tie XML will be chosen.

4. Managing the Data Sources

The crux of our proposal are the tools which manage the hybrid data source. These comprise a schema editor which automatically generates the implementation, a browser which allows the data to be viewed and manipulated and a query processor which transforms a high level query appropriately into implementation versions.

4.1 The Schema Editor

A screenshot of the editor is shown as Figure 2. The tree on the left contains one or more schemata, each of which holding a set of entity types. The editor supports the creation and editing of schemata (shown in the left hand pane) – in this case the schema is called *Library* and the entity types which make up the schema appear in the tree. The right hand pane consists of a table used for editing a type selected from the tree, by adding and removing properties and editing their features. Note that in this case, a column has been added to specify the length of particular fields. At the lower right, is a set of radio buttons for selecting whether the entity type is implemented using relations or XML, or the choice is left up to the algorithm (Auto).

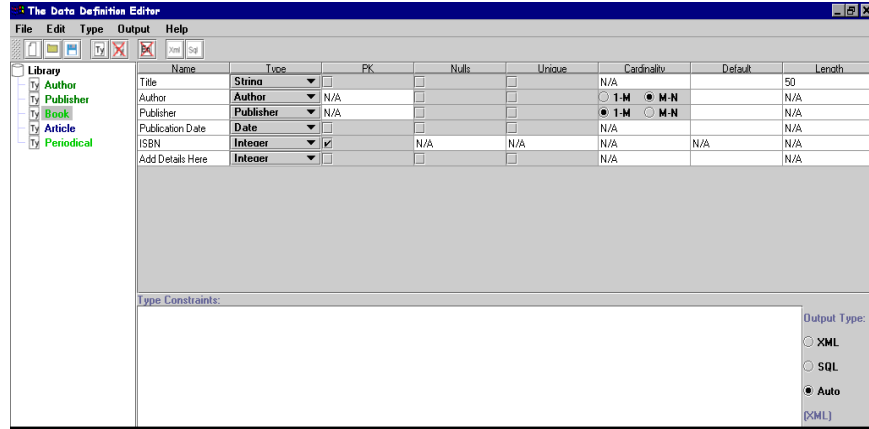


Figure 2 The Schema Editor

4.2 Manipulating the Data

The data is manipulated by a fully hyperlinked, browser interface. Manipulation starts by selecting an entity type, which brings up an editable table, uniform in its use to the Schema Editor. Figure 3 shows a screenshot of this, with an entity type, *Staff*, being displayed together with a hyperlink to a *CV* entity. Each cell is either a simple property value (shown in white) or a foreign key to an entity value (shown in grey). Multiple values are displayed as multiple lines in the cell. Selecting an entity value brings up the display of that entity's data – as shown, the lower frame was brought up by selecting the CV2 button in the upper table. For multi-valued entity properties, an "all" button is placed at the top to bring up all of the values. The data for *Staff* was stored in Oracle, while the *CV* data came from an XML file.

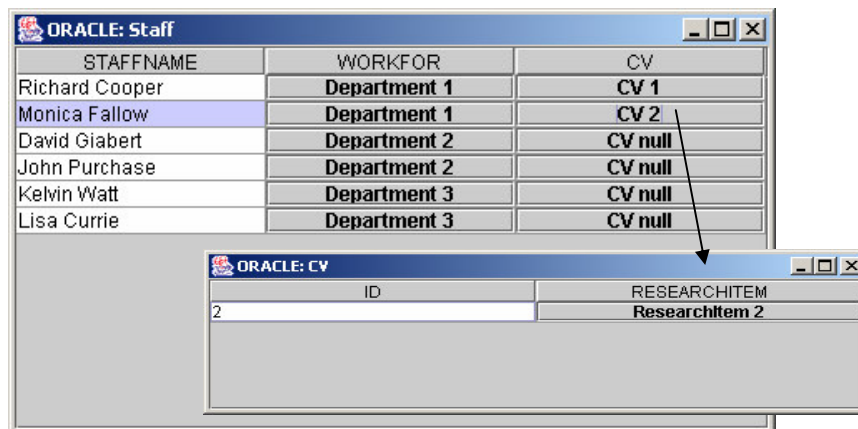


Figure 3 The Browser

4.3 Querying the Data

The abstract model is used as a basis for querying the hybrid data source. An OQL-like query language has been implemented so that queries can be expressed. The queries are then mapped to queries in the implementation systems.

A query in the query language has the following structure:

```
SELECT path1, path2, ..., pathm  
FROM EntityType1, EntityType2, ..., EntityTypen  
WHERE .....logical expression involving paths and constants
```

The paths are dot separated expressions which may start with an entity type or a cursor name or if this is unambiguous these can be assumed. Cursor names can optionally be placed after any of the entity type names if the FROM clause. Here is an example:

```
SELECT Works.Title, Works.PublishedBy.Name  
FROM Author  
WHERE Name = 'Jane Austen';
```

Queries are mapped fairly straightforwardly into each of the implementation mechanisms. Relational SQL is created as a new set of clauses. The SELECT clause is reduced to a cursor name, a dot and the last property name of each path. The WHERE clause is augmented with foreign key join expressions. The FROM clause is augmented to refer to whichever tables are required and cursor names are added with auto-numbering used to distinguish re-use of tables. A complex algorithm is used to merge the various path expressions. The above query becomes:

```
SELECT book1.Title, publisher1.Name  
FROM Author author1, Writes writes1, Book book1, Publisher publisher1  
WHERE author1.Name = 'Jane Austen' and  
and write1.Book = Book1.ISBN  
and Book1.PublishedBy=publisher1.ID;
```

The mapping to XML is to XPATH expressions, as has become standard. For example, the query above becomes:

```
Book[child::author="Jane Austen"]::Title,  
Book[child::author="Jane Austen"]::Publisher::Name
```

4.4 Managing the Data Source Components

The final step of setting up such a data source is to create the data source component itself. This will require first of all a name and secondly any specific parameters. Finally the available queries are expressed in the query language just described.

5. Summary and Conclusions

The paper has described a generalised content management system designed to fit with a component-based declarative web site design system. The goal is to have both an abstract data source description capable of being realised in a variety of implementation techniques and of a content management system also capable of being used in a platform independent manner. The software works by providing a high level, semantically rich model with which the user interacts. This interaction is then mapped down to data manipulation and querying using implementation techniques such as relations or XML as appropriate. The value of this is that the content manager can concentrate on the meaning of the data and not on how it is organised.

In web terms, this work fits into a declarative web site design system based on a hierarchy of components. Components can be XHTML elements, style sheets, scripts and data sources. A data source component is described in terms of the model described here and provides a set of views as its main method of interacting with other components. The views are expressed in terms of the query language described above – the site generation software will generate the low level implementations to install the views and add calls to the views into the server side software which implements the dynamic pages.

This work is in its infancy. The Schema Editor is functioning as well as rudimentary versions of the querying and browsing tools. Complete integration of the whole system is progressing.

More work is required in fine tuning the techniques used here. XML Schema will replace DTDs in our work and the algorithm which allocates the implementation techniques will be adjusted with experience. Finding the optimal method of connecting XML and relations requires further work. The mechanism described here works, but there are other options to explore, notable the emerging XQuery standard. It is also important to include other implementation structures – object oriented databases and spreadsheets. When the whole system has been completed, attention will turn to assessing efficiency issues. There will be a need to carry out experiments to compare our implementation with that of the standard mechanisms that are becoming available in database products. Only then will we know that our approach has proved to be effective.

Acknowledgements

The authors would like to thank Si Ying Meng and Chengcheng Zhou for work on parts of the software.

References

- [1] <http://www.w3.org/> - The World Wide Consortium Web Site

- [2] M. F. Fernandez, D. Florescu, A. Y. Levy and D. Suci, "Declarative Specification of Web Sites with Strudel", VLDB Journal, Vol 9, Number 1, 2000.
- [3] C.R. Anderson, A.Y.Levy, D.S. Weld, "Declarative Web-site Management with Tiramisu", WebDB-99, ACM SIGMOD Workshop on the Web and Databases - WebDB99, 1999
- [4] G. Mecca, P. Merialdo, P. Atzeni, V. Crescenzi, "[The \(short\) Araneus Guide to Web-Site Development](#)", WebDB-99, ACM SIGMOD Workshop on the Web and Databases - WebDB99, 1999.
- [5] G. Holland and K. Kumar, "Component-Based Web Page Composition", Java Center Services, Sun Microsystems, www.sun.com/service/sunps/jdc/component-basedwebpagecomp.pdf .
- [6] G. Graef and M. Gaedke, "Construction of Adaptive Web Applications from Reusable Components", EC-Web 2000, LNCS 1875, Springer Verlag, 2000.
- [7] M. Gaedke and G. Graef, "Development and Evolution of Web Applications Using the WebComposition Process Model", Wb Engineering 2000, LNCS 2016, Springer Verlag, 2001.
- [8] T. Tiedtke, T. Krach and C. Martin, "Infigura: An Integrated Design Tool", *Proceedings of CADUI*, January 2004.
- [9] R.P.Bourret, XML and Databases, <http://www.rpbouret.com/>.
- [10] <http://www.w3.org/TR/xhtml1/> - The Extensible HyperText Markup Language.
- [11] R.Cooper, Declarative Web Site Design Using a Purpose and Component Model, *in preparation*.
- [12] R. Catell *et al.*, The Object Database Standard, ODMG 2.0, Morgan Kaufmann, 1997.
- [13] R.Cooper, An Architecture for Collaboratively Assembled Moderated Information Bearing Web Sites, Web based Collaboration, September, 2002.
- [14] R.Cooper and S. Ali, Extracting Database Information from E-mail Messages, in A.James, B. Lings and M.Younas (eds), "New Horizons in Data Management", Springer Verlag LNCS 2712, July 2003.
- [15] T.Teorey, and J.P. Fry., "A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model", ACM Computing Surveys, June 1986.
- [16] Carsten Kleiner, Udo W. Lipeck: "Automatic Generation of XML DTDs from Conceptual Database Schemas" in Kurt Bauknecht, Wilfried Brauer, Thomas A. Mück (Eds.): Informatik 2001: Wirtschaft und Wissenschaft in der Network Economy - Visionen und Wirklichkeit, Tagungsband der GI/OCG-Jahrestagung, 25.-28. September 2001, Universität Wien, ISBN 3-85403-157-2, Band 1, pp396-405.
- [17] Tirthankar Lahiri, Serge Abiteboul, Jennifer Widom: Ozone: Integrating Structured and Semistructured Data. DBPL 1999: pp 297-323