The Araneus Guide to Web-Site Development

Giansalvatore Mecca [1], Paolo Merialdo [1,2], Paolo Atzeni [2], Valter Crescenzi [2]

[1] D.I.F.A. – Università della Basilicata
via della Tecnica, 3
85100 - Potenza, Italy

[2] D.I.A. – Università di Roma Tre
via della Vasca Navale, 79
00146 – Roma, Italy

[mecca,merialdo,atzeni,crescenz]@dia.uniroma3.it

Araneus Project Working Report
AWR-1-99
(version 1.0 – March 9, 1999)

Abstract

Web sites are rapidly becoming a world-wide standard platform for information system development. The paper reports on the work conducted in the last few years in the framework of the Araneus project in the field, presenting models, tools, methodologies, techniques for Web design and development, as well as ideas coming from a number of concrete experiences in developing data-intensive Web sites using the system. We also discuss research directions we are following to define a unified framework for data and application management on the Web.

1 Introduction

Web-Site development has recently imposed itself as a new and challenging database problem. This has justified a number of research proposals coming from the database area (e.g., [11, 17, 10, 25, 23]) for data management in Web sites; other relevant works in the field have investigated the extension of design methodologies to these sites, and their interaction with development tools [12, 18]. Indeed, the notion of a Web site has recently evolved from a small, home-made collection of HTML pages into a number of different forms, including rather complex and sophisticated information system. Given the large number and diversity of Web sites, we find useful to classify them in categories, according to their complexity in terms of data and applications (i.e., services), as shown in Figure 1.

1. we call *Web-presence Sites* those sites with low complexity both in terms of data and applications; these sites usually contain a small number of pages (in the order of the dozens), and mainly serve for marketing purposes; we believe that a vast portion of Web sites do fall in this category; however, given the relatively small size, these are usually made by hand, possibly with the help of HTML editors or simple site-manager software;

2. the *service-oriented sites* are the ones mainly dedicated to some specific service. There are several examples of these sites: one example are search engines; another typical example are free email services, like HotMail [3]. In both cases, although the site may have a large back-end database, the structure of the data and of the hypertext is quite simple (typically one single class of objects), and the complexity is rather in the underlying applications that guarantee the service;

3. *catalogue* (or *data-intensive*) *sites* are sites that publish many data, and therefore have a complex hypertext structure, but offer little or no services. Academic sites – with data about people,

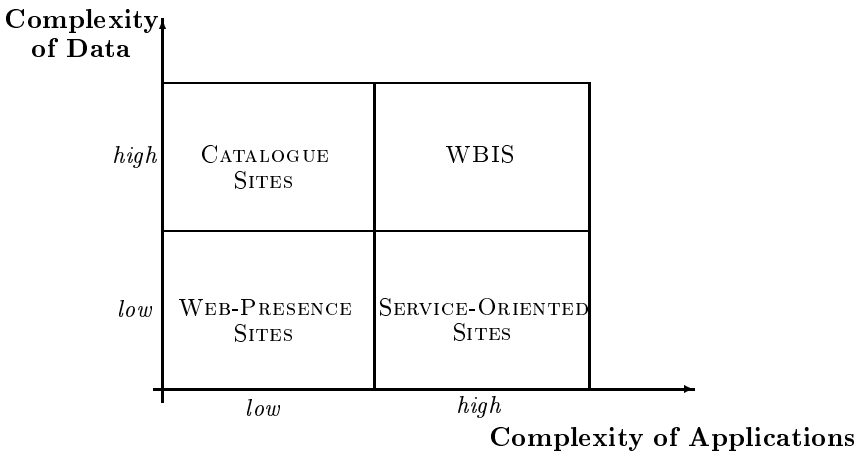

Figure 1: Classification of Web Sites

courses, research – are one example. In all these sites, the focus is mainly on organizing the data in a browsable hypertext form, and on the maintenance of both the underlying data and of the hypertext;

4. finally, the most intriguing class of sites are what we call *Web-Based Information Systems*, i.e., real information systems on the Web that offer access to complex data and at the same time also provide sophisticated interactive services. Large electronic-commerce sites obviously fall in this category, like also company information systems based on intranet platforms.

The classification is of course a bit crude: many sites may fall in between some of the categories above, yet it serves the goal of the discussion. Of these four classes, Web-presence sites can be created and maintained by hand, and usually don't need the development of ad-hoc techniques. Service-oriented sites are to be considered too application-specific to allow for a general treatment. We therefore will concentrate mainly on catalogue sites and WBIS.

If developing and administering a catalogue site can in some way be considered as a typical database problem, on the contrary, it is still unclear what should be the foundation for developing real information systems on the Web, since a full-fledged proposal, encompassing all aspects of Web-based information-system development – namely, data management techniques, application development and the associated methodologies – is still missing.

In this paper, we report on the work we have carried on in the last few years in the framework of the Araneus project, which brought us to develop a number of tools to serve as a basis for developing Web applications. Most of the ideas incorporated in the system stem from a conspicuous development activity we have conducted on large real-life Web sites in different domains, ranging from University sites, to civil engineering, to non-profit organizations, for a total of several thousands Web pages. Such an experience contributed to refine our approach, on the one side by highlighting the real challenges that a system has to face in this field, and on the other side by somehow forcing us to pursue the rapid development of user needs and market technology. In the paper, we first discuss our approach to the development of data-intensive sites, illustrating the main components of our system and our concrete experiences; then, we discuss some directions in which we are extending our work in order to incorporate a broader support for WBIS.

The Web-site development software is part of a larger system, the Araneus *Web-Base Management System* [21], which in addition incorporates tools to *query* and *integrate* both structured and semistructured data; we will not report on this in the paper, but simply want to emphasize that coupling data extraction with hypertext generation allows to develop a number of interesting applications, in which pieces of information are extracted at some sources, re-organized to create new sites,

and these sites are not only browsed, but also possibly queried back by other applications. We refer the reader to [1] for references on other aspects of the system.

2 Overview of the System

The main features of our approach are:

- it is based on a clear separation between four different levels, namely (*i*) *data management*, (*ii*) *hypertext structure* , (*iii*) *graphical presentation* and (*iv*) *application development*; the separation is justified by the observation that these levels are largely independent, and should be possible to change one without affecting the other ones; this makes the architecture of the system highly modular, i.e., made of a number of interacting components, one for each level;

- the Web-site development phase is based on a specific *design methodology*, the ARANEUS *Web-Site Design Methodology* [12], an evolution of the traditional Entity-Relationship database design methodology, which reflects the separation of levels and helps the developer in the design, implementation and maintenance phase;

- we use a formal data model, called ADM, for hypertext description; this has the advantage of giving a compact and effective description of a site structure, in order to better reason on its effectiveness; ADM is essentially an object-relational data model, with untyped links and union types; an interesting feature is that, although developed independently, ADM represents a nice abstraction of XML [7] modeling primitives, thus providing a natural basis for describing XML data sources;

- the system allows for large flexibility in choosing the actual implementation for the site; in fact, it can both produce *virtual pages* (i.e., pages generated on demand), or standard, materialized HTML files; also, due to the separation of levels and the nature of the ADM data model , the system can easily generate both plain HTML or XML sites with XSL style-sheets; the migration from one platform to the other is completely transparent; in fact, of most of the sites we have developed so far, we have both an HTML and an XML version available;

- the system is quite standard and portable; all parts of the system are written in Java; it interfaces to any JDBC-enabled [4] database;

The system architecture we refer to in this paper is show in Figure 2 (dashed boxes correspond to module currently under development). As it can be seen from the figure, the system interfaces with a DBMS and with a HTTP Server. All functionalities can be accessed via a suitable user interface, using which the developer can conduct all tasks related to site management. The core of the system stands in the six internal modules, which we shall discuss in the following Sections.

The ADM OBJECT MANAGER takes care of handling ADM objects and storing them in the database, either relational or object-relational, as discussed in Section 3. The DBMS interface, presented in Section 4, handles all communication with the external DBMS through SQL queries; it "incapsulates" the DBMS in such a way that the overall architecture is independent on the specific DBMS, and can be migrated easily from one platform to the other.

The core tools for site development, PENELOPE and TELEMACHUS, are discussed in Sections 5 and 6. PENELOPE is an SQL-like language based on a nested-object algebra for generating hypertext views on a database; it is used to automatically generate HTML or XML pages starting from the database content. TELEMACHUS handles the presentation, i.e., the graphical layout of documents; it is based on a notion of *style* for pages and attributes inside pages, and can generate either HTML formatting or XSL style-sheets.

The work of all modules above is somehow coordinated by HOMER, our case-tool for Web site design and automatic development; as discussed in Section 7, Web developers can progressively design

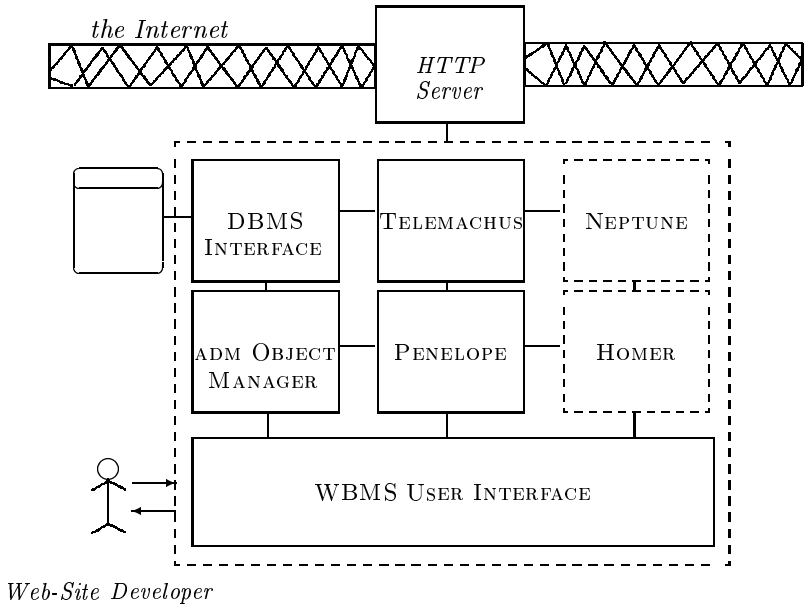

Figure 2: Architecture of the Araneus WBMS

the site using a graphical interface, starting from database design, and then moving successive levels; the design artifacts are then used as an input to the respective tools in order to proceed with the actual site implementation.

Finally, in Section 8, we discuss how we are extending the system in order to design and develop applications in the site. This extension is based on Neptune, a Workflow-Management System: with this approach, each service to be offered through a site is considered as a workflow to be handled by Neptune. The latter coordinates all activities in the workflow, and interacts with Penelope and Telemachus for generating pages. We also discuss how workflow management changes the overall Web-site design methodology.

A distribution package containing part of the software described in this paper is available for download on the Araneus Project Web site [1].

3 adm Object Manager

We use the Araneus Data Model (adm) [11] to give an intensional description of a Web site, abstracting the logical features of Web pages. In adm each page is seen as a complex object, with an identifier, the url, and a set of attributes. Pages sharing the same structure are grouped in *page-schemes*; a set of page-schemes corresponds to a site scheme. Attributes may be optional and have a type, which can be either simple, i.e. mono-valued, or multi-valued. Simple types are `TEXT`, `IMAGE`, and `LINK`. Complex attributes are based on a limited number of primitives, as follows: (*i*) *structures*, i.e. typed tuples; (*ii*) *union types*, i.e., disjunctions of attributes; (*iv*) *lists*, i.e., ordered collections of tuples (possibly nested); (*v*) *forms*; forms are seen in the model as "virtual" lists of tuples, with a number of attributes (the form fields) of different types (text-areas, selections, radios, checkboxes etc.), and an associated action, i.e., a link to some result page; the list is virtual in the sense that values for the form attributes are not physically stored in the page, but rather have to be specified by the user before submitting the form. Filling-out form fields and executing the form action is therefore conceptually similar to selecting one tuple of values in a list of links.

Figure 3 shows a graphical representation of an adm scheme corresponding to (a portion) of a site describing a scientific conference, which we will be used as a reference example throughout the paper. In the scheme, "stacks" are used to represent page-schemes, and edges denote links. Figure 3

also contains an explanation of the other graphical primitives.

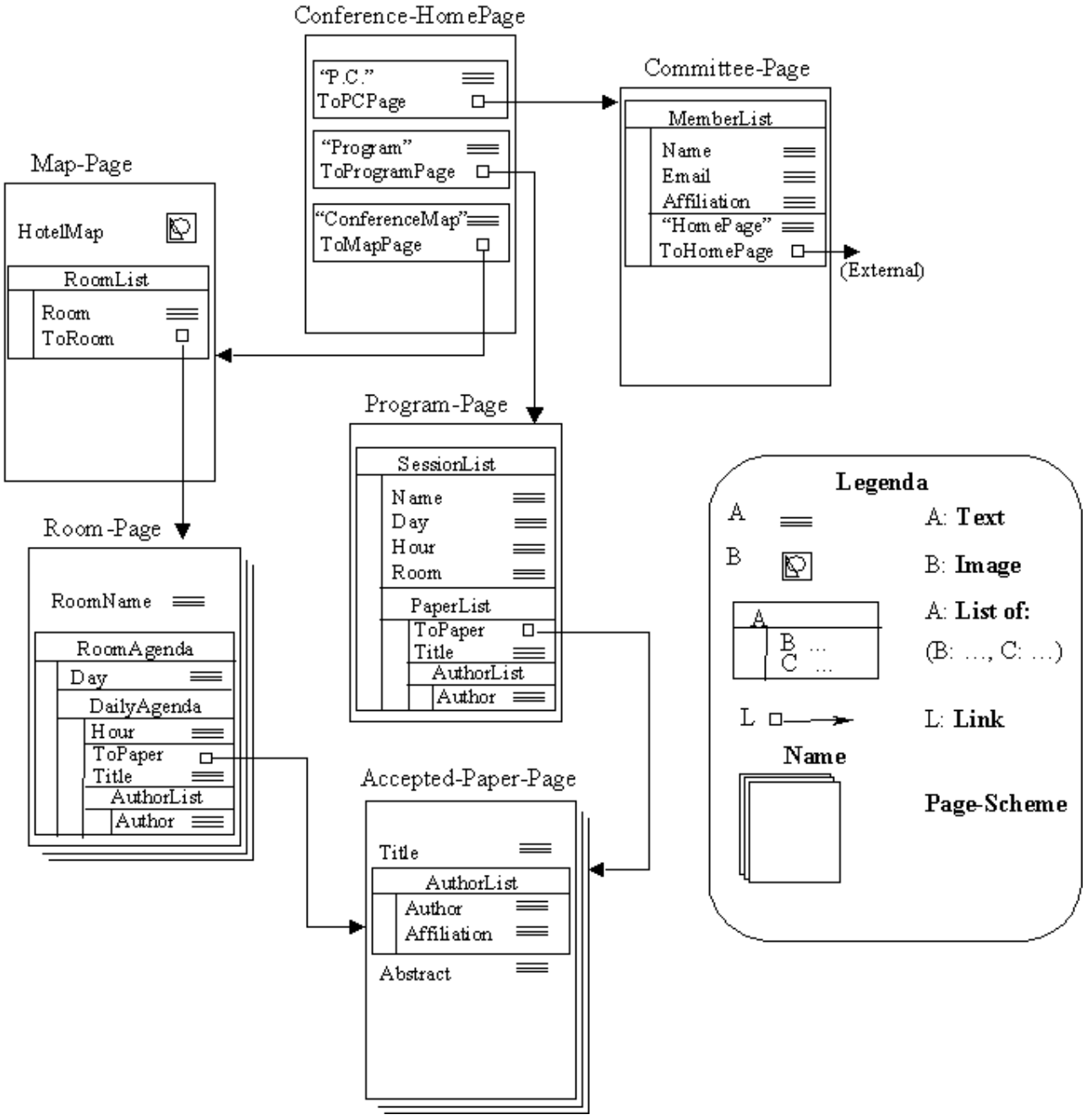

Figure 3: Example of ADM Scheme

There are two points we want to emphasize here. First, the use of the ADM data model plays a cardinal role in our approach; in fact, it allows to give a compact, intensional description of a site structure at an abstract level, and provides a basis for reasoning about the effectiveness of the chosen hypertext organization; also, as it will be clear in the next Sections, the site scheme is essential in all phases of the site design and implementation (all tools are based on that).

Second, it is worth noting that ADM is somehow at the crossroads of traditional database models and XML. In fact, the fundamental modeling primitives of the model have a natural counterpart in the ones that are typically offered by object-database systems, the main differences being the absence of hierarchies and inheritance, and the presence of union types. Thus, ADM modeling primitives might somehow be considered as a subset of ODMG [14] and SQL3 [6] data models, enriched with union

types. At the same time, ADM can be considered as a logical abstraction of XML [7]. If, in fact, XML should rather be considered as a data format than as a data model, yet its modeling primitives do correspond to the ones present in ADM: structures, possibly nested lists, disjunction, links. In this respect, an XML DTD can be seen as a type declaration for a class of documents, which has a natural counterpart in ADM page-schemes.

Special attention is to be devoted to links: links in XML are essentially untyped, since it is not possible to constrain in XPOINTER [9] and XLINK [8] the type, i.e., DTD, of a link destination. Following this we have decided to include both typed and untyped links in the data model, in the sense that a link attribute may both have a give type for its destination, or a generic link type; a generic link may have constraints on it, specifying partial type information. However, these ideas are more relevant in the site querying and integration process than in the site development one, and we refer the reader to a forthcoming paper.

4 DBMS Interface

The DBMS Interface is based on JDBC-ODBC and uses SQL as a language. It is used to store ADM objects in the external database, by decomposing them in flat tables. Note that, although using a relational database as a back-end gives less flexibility than, for example, using a full-fledged object-oriented database, still it has the great advantage of leveraging a wide-spread technology and drastically cutting the site development costs; in this way, an organization willing to develop a Web site doesn't usually need to buy a new DBMS, and can use its own as a back-end.

When generating HTML pages, Penelope uses the interface to issue query to the database and construct hypertext views, which are then translated into HTML or XML. It is worth noting that two different approaches to Web publishing are possible in this context. Database products on the market adopt *pull* techniques, in which pages contain calls to the DBMS, and, when the user requests a new page, such calls are evaluated, the page is generated on the fly and returned to the browser. The main advantage of this approach is that pages always reflect the most recent database state; however, there are at least two limitations associated with it.

- First, if the underlying database has to be used also for other ends—for example, like a repository for a company information system—frequent accesses to the Web site may considerably increase the load on the database and can slow down the overall performance; on the other side, creating a new database especially intended for Web publishing purposes may not be economically feasible and poses further problems to guarantee consistency between the two repositories.
- Second, the resulting Web site is strongly platform-dependent: the HTTP server needs a specific DBMS as a back end to serve pages, which often contain non-standard tags to invoke the execution of scripts; this means, for example, that such a site cannot be mirrored or distributed over the network, nor moved to another platform without also migrating the DBMS.

An alternative is represented by a *push* approach, in which data are materialized in HTML files and 'pushed' to the site. This clearly solves the problems above, since the resulting site is standard and the HTTP server works independently from the DBMS; however, in this case, the management of pages in presence of updates is more complex; in fact, when the database is updated, also materialized HTML files need to be correspondingly maintained to reflect the change.

Since push techniques are becoming increasingly popular on the Web due to the appearance of *channels*, i.e., sites that periodically deliver pages or portions of sites directly to the client machine, we decided to support both approaches: in a site, pages can be kept virtual, and generated on-the-fly, or materialized in HTML files. As it will be discussed in Section 5, maintenance of materialized pages is guaranteed by a suitable *page-update language*. Here we want to emphasize how the approach of adopting a relational database and materializing pages in HTML files can drastically cut the site-development costs. To give an idea of this, we mention that several large catalogue sites we have

developed – like, for example, the Faculty of Engineering Web Site at University of Basilicata [2], a site of several hundred pages and several thousands accesses per year – use Microsoft Access [5] as a back-end database. Pages are materialized, in order to reduce the database workload, and periodically updated using the page-update language provided by PENELOPE.

5 PENELOPE

PENELOPE is a system for automatically generating complex Web sites starting from data managed by a DBMS. Both XML and HTML can be used as target mark-up language. It is also worth saying that the system supports both *push* and *pull* solutions: it can either generate and materialize Web pages starting from the database content, or can be used to dynamically create pages upon request.

The site creation phase using PENELOPE takes as an input an ADM description of the target hypertext, plus page-styles, as generated by TELEMACHUS (see Section 6). The structure of the target Web site and the correspondence with the source database are described to the system by means of a declarative language, the PENELOPE *Definition Language* (PDL). Based on such description, a manipulation language, the PENELOPE *Manipulation Language* (PML), allows to create specific pages, as well as the whole site.

In the following, first we present the basics of the PENELOPE Definition Language by means of some examples, then introduce the language formal semantics; some advanced features of the language are then briefly discussed; a presentation of the main statements of the PENELOPE Manipulation Language concludes the section.

5.1 The PENELOPE Definition Language (PDL)

A PDL definition specifies how pages in the site are to be mapped onto the underlying database. It is made of a site definition statement, followed by a collection of page-scheme definitions, one for each page-scheme of the site. The site definition statement specifies (*i*) the root, that is, the base URL of the target site, (*ii*) the *style* of the site, i.e. a set of presentation directives, as it will be clear in Section 6, and (*iii*) the data source (usually a ODBC or JDBC data source) where data to be published are stored. The general form of a site definition statement is as follows:

```
SCHEME Address
[STYLE Style]
ON     DataSource
```

Each page-scheme definition consists of a `DEFINE-PAGE` statement, which essentially specifies how to fill-out pages based on attributes from tables (base relations or views) of the source database. In order to correlate single pages and generate a complex Web site, a suitable URL invention mechanism, borrowed from object-oriented databases, is used. URL invention is based on the use of *local* URL*s*, which allow to identify new pages; they can be either constant strings, or strings built using the function symbol `URL` from attributes in relations. For example `result.html` is a constant local URL, whereas `URL(<AuthorName>)` denotes a local URL built from values of database attribute `AuthorName`.

A `DEFINE-PAGE` statement has the form:[1]

```
DEFINE-PAGE P
AS          S
USING       R1, R2, ..., Rn
```

where: (*i*) P is the page-scheme name; ; (*ii*) $R_1, R_2, \ldots, R_n$ are tables in the site data source (or SQL views over it); and (*iii*) S describes the page structure, by specifying the page attributes, their type, and their correspondence with database attributes.

[1] The language includes other optional clauses, which however are not presented here for the sake of space.

To illustrate the DEFINE-PAGE statement let us consider some examples. In particular, assume our data source is a relational database, let us call it Conference, dealing with data for a scientific conference, with the following relations (key attributes are underlined):

1. Author(AuthorName, Email, Affiliation, Address);
2. PCMember(PCMemberName, Email, Affiliation, IsChair, HomePage);
3. OCMember(OCMemberName, Email, Affiliation, HomePage);
4. Paper(PaperCode, Title, Abstract, ContactAuthorName, Accepted, SessionName);
5. PaperAuthor(PaperCode, AuthorName);
6. PaperToPCMemeber(PCMemberName, PaperCode);
7. Session(SessionName, Day, Hour, Room);

Consider page-scheme ACCEPTED-PAPER-PAGE in Figure 3. It contains two mono-valued attributes, of type TEXT, i.e., the paper's title and abstract, plus a multi-valued attribute, corresponding to the list of authors. The following PDL statement is used to describe how instances of this page-scheme have to be generated starting from data of the Conference database:

```
DEFINE-PAGE ACCEPTED-PAPER-PAGE
AS          URL(<PaperCode>);
            Title:          TEXT <Title>;
            AuthorList:     LIST-OF (Author:     TEXT <AuthorName>;
                                     Affiliation:TEXT <Affiliation>;)
            Abstract:       TEXT <Abstract>;
USING       PaperAuthor,
            Author,
            AcceptedPaper:  (SELECT *
                             FROM Paper
                             WHERE Accepted = True)
END
```

It is easy to see that the statement closely resembles the page-scheme structure. The main part of the statement is the AS clause, describing how to fill-out data in the page. In particular, this clause specifies how to assign URLs to instances of the target page-scheme, and how to generate attribute values for instances of the target page-scheme.

URLs have to be generated by the system, and each time a page is created, a new, different URL is needed. We use function terms to generate URLs; in the example, term URL(<PaperCode>) specifies that the system has to generate an URL for each page, and that the URL must be uniquely associated with the value of attribute PaperCode;[2] in this way, a different page will be created for each different paper code.

The DEFINE-PAGE statement also describes how pages must be filled-out starting from data stored in the source database. For each attribute of the page-scheme there is an item in the AS clause. For each page instance, attribute values are taken from tables specified in the USING clause. In particular, as it will be detailed below, tables in the USING clause are joined, and each page-scheme attribute is associated with an attribute of the resulting relation, namely the one whose name is enclosed by brackets (e.g. <Title>). For example, the definition of attribute Title of type TEXT specifies that each page will report the paper title, as specified in the ADM scheme, and that the corresponding values come from attribute Title. Note that, in the USING clause, both tables and SQL views can be specified.

Let us consider another example, showing how local URLs are used to link pages together. The following PDL statement specifies how page-scheme PROGRAM-PAGE, shown in Figure 3, has to be created:

```
DEFINE-PAGE PROGRAM-PAGE
AS    URL("Program.html");
```

[2] This technique is inspired by the use of *Skolem functors* to invent new OID's in object-oriented databases [20].

```
      SessionList:  LIST-OF (Name:      TEXT <SessionName>;
                             Day:       TEXT <Day>;
                             Hour:      TEXT <Hour>;
                             Room:      TEXT <Room>;
                             Paper-List:LIST-OF (ToPaper :  LINK-TO ACCEPTED-PAPER-PAGE (
                                                  Title <Title>;
                                                  URL(<PaperCode>));
                                                Author-List LIST-OF (Author:  TEXT <Author>));
USING Session,
      Author,
      PaperAuthor,
      AcceptedPaper:  (SELECT *
                       FROM Paper
                       WHERE Accepted = True)
END
```

In this statement, we specify that, coherently with the page-scheme structure, each page must contain a list of sessions; each session is composed by a list of papers, each one with its title and authors. Note that, for each paper we need a link that leads to the corresponding page—the one defined in the previous PDL statement. In the statement above, for each item in list `PaperList`, we use the paper title as anchor of the link; then, to link the appropriate pages, we use function term `URL(<Title>)` as a value for the link reference; this enforces the correct reference as long as URLs for paper pages are generated using the same function term.

5.2 PDL Semantics

To understand the mapping between the values of relations specified in the `USING` clause, and the values of the pages to be generated, we now develop a formal semantics for the PENELOPE Definition Language.

The semantics of a PDL statement can be easily explained algebraically. In particular, each statement in the language maps to an expression in a nested relational algebra, enriched by a specific operator for URL invention. The nested relational algebra we refer to is that defined by Roth *et al.* in [24] for the class of nested relations. The operators of nested algebra work on nested relations: operators $\cup, \cap, -, \pi, \rho, \sigma, \bowtie$ are natural extensions of traditional operators of the usual relational algebra, and their definition is not recalled here. A specific operator of nested relational algebra is the *nesting operator*, ν, whose definition is recalled hereafter. Let $R(\overline{AB})$ be a nested relation scheme associated with nested relation r, with $\overline{A}$ and $\overline{B}$ denoting sets of attributes. The *nesting* of r along $Y = \overline{B}$, denoted by $\nu_{Y \leftarrow \overline{B}}$ selects the set of tuples of r equals on $\overline{A}$ and compacts them into tuples where the set of their different values on $\overline{B}$ becomes values of the complex attribute Y.

To describe semantics of a PDL statement, we need to extend the nested relational algebra above by a further operator, denoted URL, for describing the URL invention mechanism. URL allows to introduce new attributes into a nested relation as follows. Let r_i be a nested relation with scheme R_i; operator URL applied to relation r_i takes a number of arguments, $c_1, \ldots c_n, B_1, \ldots, B_m$, where each c_i $(n \geq 0)$ is a constant value, and each B_j $(m \geq 0)$ is an attribute in R_i; we use a Skolem function that, for each tuple t in r_i, returns a unique identifier, denoted by $f_{c_1,\ldots c_n,B_1,\ldots,B_m)}(t)$, constructed starting from constant values $c_1, \ldots c_n$ and values $t.B_1, \ldots, t.B_m$. Then, $\text{URL}_{A \leftarrow c_1,\ldots c_n,B_1,\ldots,B_m} R_i$ adds a new attribute, A, to R_i; for each tuple t of r_i, the value of A is $f_{c_1,\ldots c_n,B_1,\ldots,B_m}(t)$.

When the PENELOPE system executes a PDL statement, first it performs a natural join of the tables specified in the USING clause. Second, all the needed URL attributes are added to the resulting relation. The URL values are generated by the URL operator from the values of its actual parameter (i.e. a constant string, or a value of the database). Then, to get rid of all the attributes that are not useful for page instance generation, i.e. those database attributes that are not associated to any page attribute in the page-scheme, an ordinary projection operation is performed. Finally, the resulting relation is transformed in a set of nested tuples: nesting operations are performed starting from inner

ADM attributes of type `LIST-OF`. Each nested tuple of the resulting relation corresponds to a site page.[3]

With respect to the definition of page-scheme `ACCEPTED-PAPER-PAGE` given above, the PENELOPE system, at execution time, performs the following expression:

$$\nu_{AuthorList \leftarrow AuthorName,\ Affiliation}($$
$$\pi_{URL,\ Title,\ AuthorName,\ Affiliation,\ Abstract}($$
$$URL_{URL \leftarrow PaperCode}(PaperAuthor \bowtie Author \bowtie AcceptedPaper)))$$

5.3 PDL Advanced Features

So far, we have seen the main basic features of PDL. Actually, the language also allows for specifying the structure of sophisticated pages, including, for instance, click-map attributes, links to existing pages, and forms. In the following we briefly present these aspects.

Virtual Pages and Modifiers As it was discussed above, PENELOPE supports both push and pull generation. The standard URL-invention mechanism creates file names and materializes pages in HTML files; however, in some cases a better option is to leave pages virtual, and generate them on the fly upon user request. To specify that a page is to be generated by running a script, instead of being materialized in a file, the URL() function has a *modifier*, "cgi". Consider, for example, page `ACCEPTED-PAPER-PAGE` above. If we want it to be a virtual page, the corresponding definition is to be changed as follows:

```
DEFINE-PAGE ACCEPTED-PAPER-PAGE
AS          URL("cgi",<PaperCode>);
            Title:          TEXT <Title>;
            AuthorList:     LIST-OF (Author:       TEXT <AuthorName>;
                                     Affiliation:  TEXT <Affiliation>;)
            Abstract:       TEXT <Abstract>;
USING       PaperAuthor,
            Author,
            AcceptedPaper:  (SELECT *
                             FROM Paper
                             WHERE Accepted = True)
END
```

The only difference with respect to the previous statement stands in the URL() clause. The presence of the "cgi" modifier tells to the system that it does not have to generate the actual HTML or XML code for this page, but rather a script, that will be invoked in order to generate the page upon request. This script will have an input parameter, i.e., the paper code; when invoked, it will extract the data from the database and return the page source on the standard input.

It is worth noting that the use of these virtual pages is completely handled by the system, which automatically produces all code necessary to generate the page. Also, a virtual page may embed links to both virtual and materialized pages, thus making the integration of push and pull quite seamless. Of course, a virtual page can be easily referenced from another page by using the "cgi" modifier in the URL clause of the link attribute. In this way the system knows that a script is to be run in order to retrieve the link destination page.

The use of modifiers, like "cgi" above, allows for a wider range of link types. There is, for example, a "mailto" modifier that can be used in order to produce a link to an e-mail address. A similar syntax can be used in order to introduce links to offsets (i.e., links to some specific portion of a page, also called *internal links*). Another interesting possibility is that of referencing pages that are external to the site. These are called *external pages* in PENELOPE. However, for the sake of space we choose not to develop further on these issues here.

[3] Actually, to improve performances, there is a constraint on nested attributes: for each level of nesting, all non-nested attributes must come from a single table or view in the `USING` clause.

Defining Click-Maps An effective way of rendering links in Web pages can be implemented using HTML click-maps. Note that click-maps, from the data-model perspective, are simply lists of links with an associated image; deciding to implement the links as a click-map is rather an implementation choice. The PENELOPE system allows for an easy and intuitive management of pages containing this kind of linking mechanism. To define a page have to be created with a click-map, a specific syntax is provided by PDL. In particular, PDL allows to specify `MAP` attributes. A `MAP` attribute is made of two parts: (*i*) an image (i.e. an attribute of type `IMAGE`), used as a basis for the map, and (*ii*) a link (i.e., an attribute of type `LINK-TO`), used to associate portions of the image with links to other pages.

Assume we want to generate a page with a click-map showing a map of the building where the conference is held, with links to scheduled events in each room (see page-scheme `MAP-PAGE` in Figure 3. Assume we have stored in our database a table `MapTable` containing the name of the file to use as a map, plus coordinates of box-shaped areas corresponding to rooms, as follows:

MapFileName	*Room*	*Coords*
../icons/5Stars.jpg	CongressHall	20, 13, 22, 15
../icons/5Stars.jpg	NorthHall	18, 8, 21, 15
...	...	...
../icons/5Stars.jpg	SouthHall	9, 33, 11, 35

The following PDL statement will generate the map in the unique instance of page-scheme `MapPage`:

```
DEFINE-PAGE MAP-PAGE
AS          URL("map.html");
            ConferenceMap:  MAP (Map :     IMAGE   <MapFileName>;
                                ToRoom:    LINK-TO RoomPage (
                                                    Coords :  TEXT <BoxCoords>;
                                                    URL(<Room>));
USING       MapTable
END
```

When the PENELOPE system interprets the statement above, a piece of HTML like the following is generated:

```
<HTML> ...
<MAP NAME="ConferenceMap">
   <AREA SHAPE="RECT" COORDS="20, 13, 22, 15" REF="../RoomPage/CongressHall.html">
   <AREA SHAPE="RECT" COORDS="18, 8, 21, 15" HREF="../RoomPage/NorthHall.html">...
   <AREA SHAPE="RECT" COORDS="9, 33, 11, 35" HREF="../RoomPage/SouthHall.html">
</MAP>
<IMG USEMAP="#ConferenceMap" SRC="../icons/5Stars.jpg"> ...
</HTML>
```

The picture of the conference place is displayed as a sensitive map, and by clicking on a region associated with a room the page presenting activities scheduled for that room is reached.

Defining Forms Another fundamental construct in HTML pages are forms. Forms are particularly important in all cases in which a page is used to collect some user input and then run a procedure on the server. They are therefore essential to run a workflow. Embedding a form in a page generated by PENELOPE has a number of subtleties, related to the various kinds of fields that a form can have (text areas, radios, checkboxes, selections etc..). However, the basic idea is the following: the PDL specification of a form is made of: (*i*) a bunch of attributes, one for each field in the form; attributes can either be empty (like text-areas), or allow a selection among a number of values from some database attribute; (*ii*) a number of buttons, one for each action associated with the form; (*iii*) the URLs of a number of scripts to be run when submitting the form, one for each button.

Suppose, for example, we need to have a page in our conference Web site to collect paper reviews by PC members. A form to collect electronic reviews might be described as follows:[4]

```
DEFINE-PAGE REVIEW-PAGE
AS          URL(cgi,"review.html");
            ReviewForm:  FORM (Action:        TEXT   "cgi/CheckReview.bat";
                               Method:        TEXT   "POST";
                               Go:            SUBMIT "Submit Review";
                               Restart:       RESET  ;
                               ReviewerName:  SELECT ( items :  LIST-OF (
                                                       Name :  OPTION = <PCMemberName>;));
                               PaperCode:     SELECT ( items :  LIST-OF (
                                                       Code :  OPTION = <PaperCode>;));
                               GradeReject:   RADIO             "Reject";
                               GradeNeutral:  RADIO             "Neutral" CHECKED;
                               GradeWAccept:  RADIO             "Weak Accept";
                               GradeAccept:   RADIO             "Accept";
                               Review:        TEXTAREA, SIZE "15" "80"; );
USING       PCMember, Paper
END
```

The page will contain a post form that executes a "CheckReview.bat" script in directory "cgi". It will contain two buttons, one to submit and the other to reset. There are essentially four fields in the form: the `ReviewerName` and `PaperCode`, both to be selected among a list of alternatives corresponding to values in the database; the grade, a radio to check (in this simple example, either "Reject" or "Weak Accept" or "Accept"); and finally the actual `Review`, a free text to be inserted in a textarea.

It can be seen that the syntax for describing forms can in some cases be quite elaborate. In fact, forms are hardly coded in PDL by hand. Still, forms are a fundamental mechanism in interconnecting PENELOPE and NEPTUNE, as it will be clear in the following sections.

5.4 The PENELOPE Manipulation Language

The creation of pages, as defined in the PDL source code is performed by means of instructions of the PENELOPE *Manipulation Language* (PML). PML provides two main instructions, `Generate` and `Remove`, which can refer to (*i*) the whole site; (*ii*) all instances of a page-scheme; or (*iii*) pages that satisfy a condition in a `Where` clause.

The general form of a PML statement is as follows:

```
Generate|Remove P
Where           C
```

where P is a page-scheme name, and C is a boolean predicate over some attributes of P.

Let us consider an example; assume we are interested in generating instances of page-scheme `ACCEPTED-PAPER-PAGE`; in particular, suppose we want to generate pages corresponding to papers by a given author; then the following PML statement has to be executed by the system:

```
Generate ACCEPTED-PAPER-PAGE
Where    AuthorName = "Tom Scott"
```

It is important to observe that PML is an effective tool also for maintaining the site. In fact, it allows to update pages, during the life-cycle of the site. A suitable algorithm for incremental page maintenance has been defined by Sindoni in [26]. The page-maintenance algorithm takes as input a database update, and returns a minimal set of PML instructions needed to correspondingly update

[4] The actual syntax has been simplified in some details for clarity's sake.

the pages. In essence, when an update to the database is requested to the system, it automatically generates a *mixed transaction*, in which SQL updates to database tables and PML updates to pages are combined in order to guarantee consistency between the two. The transaction is then atomically executed against the database and the Web site.

6 TELEMACHUS

One of the most difficult and underestimated tasks in developing a Web site consists in handling the graphical layout of pages. Nevertheless, people willing to create their sites are often more worried about having an appealing presentation than about data management issues; this is not surprising, since Web sites are becoming a prominent commercial vehicle, and therefore need to attract customers. This makes design and implementation of presentation a large part of the site life-cycle.

Experience tells that there are at least three fundamental requirements in this field: (*i*) first, it is very useful to have rapid prototyping tools, i.e., tools that to produce some approximate layout for all pages in a site; this allows to concentrate on the other aspects of site design with little initial effort on the layout; (*ii*) then, at a subsequent step, one should have flexible tools to refine presentation details and obtain an appealing final result. Finally, (*iii*) presentation is hardly developed by coding; it is much more convenient to work on *example* HTML *pages*, that can be displayed using a standard browser to get an immediate feedback, and then let the system derive the necessary code from examples.

These ideas have inspired TELEMACHUS, our tool for presentation design and development. In the previous Section we have seen that PENELOPE somehow assumes that the PDL site definition code also associates some form of style to each page; when the system builds instances of a given page-scheme, attribute values are formatted according to the specified graphical directives. These styles are designed with the help of TELEMACHUS.

6.1 Styles and Formatting

Before actually describing how TELEMACHUS works, let us mention what is a style in our approach. A fundamental notion is the one of *attribute style*, which specifies how values of a given attribute must be formatted in a page. To be able to produce sophisticated formatting, an attribute style is made of two arbitrary pieces of HTML code [5], called *prefix format string* and *suffix format string*, between which the attribute values will be enclosed when generating pages.[6] To give an example, consider attribute `Room` in page-scheme `PROGRAM-PAGE` in Figure 3, corresponding to the room in which a conference session will be held. To obtain a simple, boldface style and color red for room names in pages, we may specify the following attribute style:

```
ROOM: [<FONT COLOR="red"><B>] [</B></FONT>]
```

In this way, for each room name – say "*Panoramic Room*" – a piece of HTML code of the form: `<FONT COLOR="red"><B>Panoramic Room</B></FONT>` will be produced in the page. If, however, we want a more elaborate formatting, in which the room name is written in a red and "Arial"-boldface, and preceded by an arrow image, we may use the following style (the HTML table is needed to correctly align image and text):

```
ROOM: [<TABLE CELLPADDING="3" BORDER="0" ROWS="1" COLS="2">
       <TR><TD WIDTH="30"><IMG SRC="arrow.gif" WIDTH="30" HEIGTH="30"></TD>
       <TD><FONT FACE="Arial" COLOR="red"><B>]
      [</B></FONT></TD></TR></TABLE>]
```

[5] We mainly refer to HTML, but TELEMACHUS can easily handle any other mark-up language.

[6] For some types of attributes, TELEMACHUS also allows to specify an *infix format string*; to keep the development simple, we do not elaborate further on this.

```
/* Page-Style for Page-scheme PROGRAM-PAGE */
HEADER:            [<HTML>
                   <HEAD>This is the HTML header</HEAD>
                   <TITLE>This is the page title</TITLE>
                   <BODY BACKGROUND="../icons/na.gif">
                   <A Name="begin"></A>
                   <HR>]
SESSIONLIST:       [<TABLE>] [</TABLE>]
SESSIONLIST.NAME:  [<TR><TD> ...  ]  [ ...  </TD>]
...
SESSIONLIST.ROOM:  [<TD><TABLE CELLPADDING="3" BORDER="0" ROWS="1" COLS="2">
                    <TR><TD WIDTH="30"><IMG SRC="arrow.gif" WIDTH="30" HEIGTH="30"></TD>
                    <TD><FONT FACE="Arial" COLOR="red"><B>]
                   [</B></FONT></TD></TR></TABLE></TD></TR>]
PAPERLIST:         [<UL>] [</UL>]
...
FOOTER:            [<HR>
                   <CENTER>
                   <A HREF="#begin"><IMG SRC="../icons/ToPageBegin.gif"></A> <BR>
                   Site created by the
                   <A HREF="http://www.dia.uniroma3.it/araneus">Araneus WBMS</A> <BR>
                   <P><A HREF="Mailto://Webmaster@www.aaa.com">WebMaster</A>
                   </CENTER>
                   </BODY>
                   </HTML>]
```

Figure 4: An example of style produced by TELEMACHUS for page-scheme PROGRAM-PAGE

It can be seen how such a simple mechanism is in fact very flexible. A *page-style* specifies all format directives for a given page-scheme; it contains a set of attribute styles, one for each attribute in the ADM scheme of the page, plus a *header section* and a *footer section*. Header and footer specify graphical features to be associated with the page itself, rather than with a specific attribute, like, for example, page background and banners. Like attribute styles, also header and footer consist of arbitrary pieces of HTML code. Figure 4 shows a fragment of a possible style for page PROGRAM-PAGE above (the style has been simplified for presentation purposes). When generating instances of a given page-scheme, PENELOPE loads the corresponding page-style[7] and formats data according to it: each page has the header and footer defined in the style, and each attribute value is enclosed in between its format strings.

Actually, the styling mechanism provided by TELEMACHUS is even finer. In order to guarantee a good compromise between rapid prototyping and accuracy in the final product, beside attribute styles and page styles, we also have a notion of *site-styles*. Site-styles are used at the beginning of the presentation design phase, in order to produce a first version of page-styles based on formatting choices that will be common to the whole site. This is very useful in order to speed-up the layout phase: in fact, usually pages in a site are organized according to some common lines – i.e., background color, font face, font color, link format etc. One example of site-style is reported in Figure 5. It can be seen that a site-style is essentially a *generic* page-style, in the sense that it specifies one header and one footer common to all page-schemes in the site, plus a number of formats, each common to all attributes of a give type – text, image, link, list etc. – in the site. A site may have one or more site-styles like the one above. These are used by TELEMACHUS as starting directives in order to automatically generate a first version of page-styles for the different page-schemes. Then, page-styles can be further customized, by changing header, footer, and attribute styles, in order to vary the layout from one page to the other.

[7] For page-schemes that do not have an associated page-style, PENELOPE adopts a default style.

```
/* Site-Style "sample.sty" */
HEADER:               [<HTML>
                      <HEAD>This is the HTML header</HEAD>
                      <TITLE>This is the page title</TITLE>
                      <BODY BACKGROUND="../icons/na.gif">
                      <A Name="begin"></A>
                      <HR>
                      ]
TEXT:                 [<FONT FACE="ARIAL" COLOR="Blue">] [</FONT>]
IMAGE:                [ ] [ ]
LINK:                 [<I>] [</I>]
LIST:                 [<TABLE>] [</TABLE>]
LIST-TUPLE:           [<TR>] [</TR>]
LIST-TUPLE-ELEMENT:   [<TD>] [</TD>]
FOOTER:               [<HR>
                      <CENTER>
                      <A HREF="#begin"><IMG SRC="../icons/ToPageBegin.gif"></A> <BR>
                      Site created by the
                      <A HREF="http://www.dia.uniroma3.it/araneus">Araneus WBMS</A> <BR>
                      <P><A HREF="Mailto://Webmaster@www.aaa.com">WebMaster</A>
                      </CENTER>
                      </BODY>
                      </HTML>]
```

Figure 5: An example of site style

6.2 Working with Telemachus

We are now ready to discuss how Telemachus works. In essence, the presentation design phase goes from rather general and undistinguished formatting (as specified by site-styles) to very particular formatting (obtained by customizing attribute-styles in page-styles). However, as already discussed above, in this process the designer hardly wants to write style code as the one shown in examples above, but rather work with sample HTML pages, to be able to check the chosen layout without the need of generating the actual site.

Telemachus has been conceived to support this process. It makes styles completely transparent to the designer: it allows to write sample HTML pages, from which styles are automatically produced; these HTML pages are called *templates*. A page-template is a prototypical HTML page; it does not contain actual data, but place-holders. For example, a template for page-scheme `PROGRAM-PAGE` above will not contain actual room numbers, but rather strings corresponding to attribute names, of the form `$Room`. Similarly, a site-template is a sample page in which, instead of actual data, strings corresponding to types are reported, like `$Text, $Link` etc. Beside this detail, a template is a fully standard HTML page, which can be edited using the designer's preferred editor (the only limitation being that place-holders cannot be changed) to refine the presentation, and browsed using any HTML browser; since browsing the template closely resembles browsing the corresponding pages, the designer has an immediate feedback on what the corresponding pages in the site will look like. This makes templates a much more convenient work tool than a style.

Based on these ideas, the presentation design process is sketched in Figure 6. In essence, users only work with templates, and Telemachus takes care of generating the corresponding styles.

- The starting point is one (or more) site-templates, i.e., one sample page showing how text, images, links, lists etc. are to be formatted, as specified in the PDL site description; the site-template can be derived from some pre-existing HTML page, or created from scratch, and progressively refined until the layout is satisfactory.
- When the site-template is ready, Telemachus is invoked. It will analyze the template and generate from it the corresponding site-style, according to the syntax shown in Figure 5. This

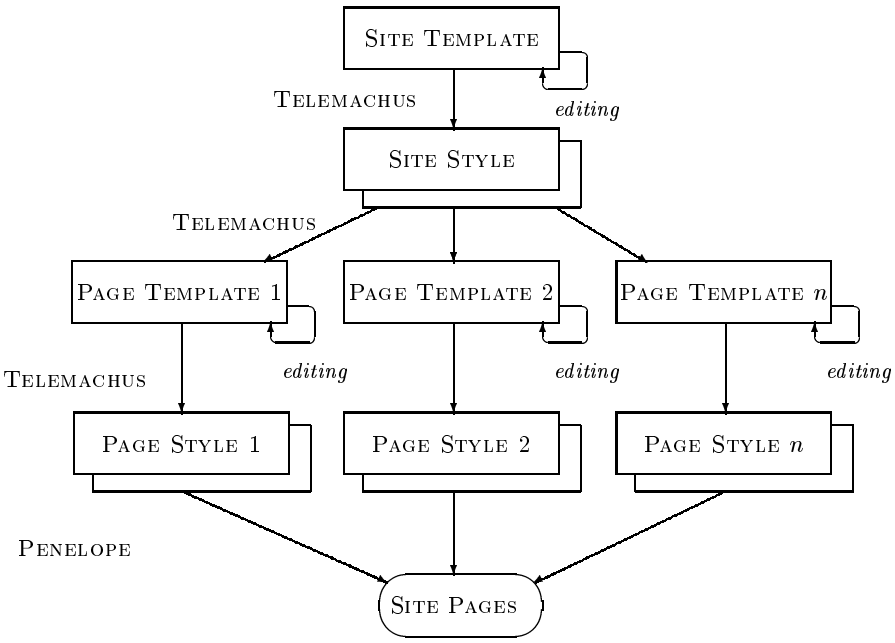

Figure 6: Presentation Design using Telemachus

site-style is then used again by Telemachus in order to produce a first version of the page-templates, one for each page-scheme in the pdl statement. Being all generated on the basis of the same site-style, these templates have a rather homogeneous layout.

- Then, site-templates are analyzed one by one (if needed), and edited in order to refine and customize attribute layout. This may be needed because, for example, some page-schemes must have different headers and footers (different background, different icons), and their attributes specific formatting (like attribute `Room` in page-scheme `PROGRAM-PAGE` above).
- When this editing phase is completed, Telemachus can be invoked again in order to produce page-styles from page-templates. If the target mark-up language is xml, Telemachus also generates an xsl style-sheet for each page-scheme, on the basis of the page-style: for each attribute (i.e., xml element), an xsl rule is generated, that specifies how to embed the value of the element within the pair of formatting strings in the style.

Note that this process is such that presentation maintenance can be handled independently from the other ones in the site. If, in fact, some page-style is to be changed to improve the site layout, it is sufficient to modify the corresponding page-template and then let Telemachus produce the new style, which can then be used to generate the new pages.

The actual prototype of Telemachus also provides advanced features, like styles with parameters and incremental generation of styles. For a detailed treatment of these issue, we refer the reader to the Telemachus user manual.

7 Homer

It can be seen from the previous sections that designing and implementing a site is a rather complex task, that involves several aspects and requires to deal with data under different perspectives, mapping the one onto the others. For large and complex Web sites, the complexity of the design and maintenance process can be reduced only through the adoption of a systematic design methodology,

i.e., a set of models and design steps that lead from a conceptual specification of the domain of interest to the implementation of the actual site.

We have developed a thorough methodological framework for designing data-intensive Web sites, the ARANEUS methodology [12]. A key feature of our approach to Web site design that we want to emphasize here is the clear distinction among different levels: (*i*) *database design*; (*ii*) *hypertext design*; (*iii*) *presentation design*. The separation is justified by the observation that the four levels are largely independent. In essence, by adopting the methodology, designers start from a conceptual description of the site domain (an Entity-Relationship scheme) and through a set of precise steps progressively moves to database logical design (this produces the database relational scheme), then hypertext design (this produces the site ADM scheme), and finally presentation design (producing page-templates).

To simplify this design process, as well as to automate the implementation phase based on the site design artifacts, we have developed HOMER, a case tool conceived to support the designer through the successive design steps. This is a natural complement of our approach, in which the site design evolves through different levels and different descriptions, each based on a formal model. HOMER has two main facilities: first, a graphical user-interface; second, a module to automatically generate code to be run by the different tools in order to implement the actual site. Briefly, HOMER works as follows. First, the system takes as input a declarative specification of the starting conceptual scheme of data and automatically translates it into a logical (relational) database scheme. Then, its graphical interface helps the designer in specifying transformations according to which constructs of the database conceptual scheme have to be manipulated in order to obtain the desired hypertext. By progressively applying these transformations the designer shapes the ADM scheme of the resulting site. Once the ADM description for the site has been generated, based on the specified transformations, HOMER automatically generates the PDL code to be used as an input the page-creation phase.

8 NEPTUNE: Towards Web-Based Information Systems

It can be seen how the tools described in sections above represent a flexible platform for developing data-intensive sites. Still, they provide little support for adding services and application to the site. Our goal is therefore to extend the framework with models and tools to handle application as a further level in the design and implementation phase. In this respect, *workflows* [19] represent a promising direction. Born to automate *business processes* – i.e., coordinated procedures and activities aimed at realizing some business objective – workflow management system are a natural solution to deliver services on the Web [16, 22]. Moreover, in adopting workflows we can leverage on a rather consolidated platform in terms of design and modeling [15, 13]. Our approach is to extend the framework developed in the previous sections with NEPTUNE, a workflow management system conceived to cooperate with other system tools. In this framework, the development of complex information systems is based on the following simple ideas.

A site is made of several intermixed portions: (*i*) a catalogue portion, of *data-access* pages, used to access and browse the site underlying database; (*ii*) one or more *workflow-execution* portions, giving access to one or more services through the execution of a workflow. To give an example, consider the conference site introduced above; most probably the site will have a public part, publishing data about accepted papers, program, organization, and a private part to handle the review process; the latter is naturally implemented as a workflow. A similar argument also holds for most electronic commerce sites. The two different portions are seamlessly combined in the site, in the sense that users may want to browse some data while running the workflow, or starting a workflow after browsing the site.

All the logics of the workflow is handled by NEPTUNE, which generates Java code to coordinate the various activities, assign tasks to actors, and authenticate accesses to the workflow, if necessary; the site is used as an *interface* to the workflow, i.e., all the interaction between actors and workflow management system happens through pages in the site; these pages are generate via a client-server interaction between NEPTUNE on the client side and PENELOPE plus TELEMACHUS on the server

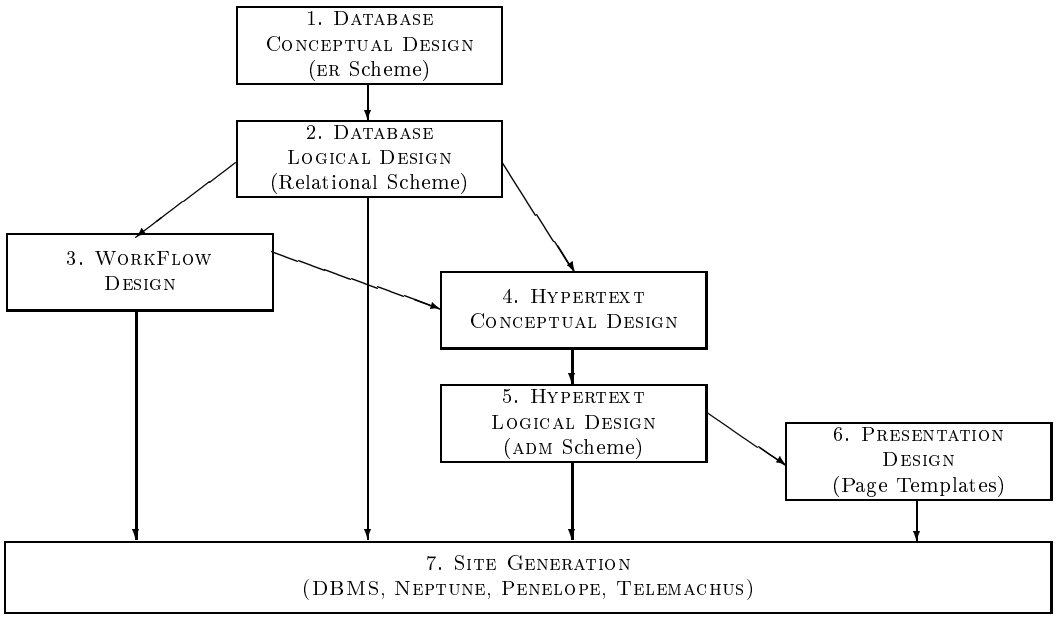

Figure 7: The Araneus Web–Site Design Methodology

side, and contain suitable forms to collect user-inputs and execute tasks. Communication between the site and the workflow management system is based on the database, which is used to store both the workflow state and user inputs.

We want to stress the fact that, as well as the data-access part, also the workflow-execution part of the site needs an hypertext and presentation design phase, and is fully integrated with the data-access part, to which it can be linked. The introduction of the workflow therefore changes the overall design process. Figure 7 shows an extended version of our methodology for WBIS design and implementation. Details about the database and hypertext design phases can be found in [12]. Workflows are designed and described along the lines of [15, 13]. It can be seen how the four different levels (data, application, hypertext and presentation) are clearly distinguished, and at the same time interact in the overall design and implementation process. Although the implementation of Neptune is still under development, our first experiences with the prototype of Neptune have shown the benefits of this approach. The site is in fact a natural platform for implementing the workflow interface, whereas the design and development of a workflow nicely fits inside the design and implementation framework presented above.

References

[1] The araneus Project Home Page.
`http://www.dia.uniroma3.it/Araneus`
`http://www.difa.unibas.it/Araneus`.

[2] Faculty of Engineering at University of Basilicata. `http://www.ing.unibas.it`. In italian.

[3] The HotMail web site. `http://www.hotmail.com`.

[4] JDBC Database Access API. `http://www.javasoft.com/products/jdbc/jdbc.html`.

[5] The Microsoft Web Site. `http://www.microsoft.com`.

[6] The SQL standards page. `http://www.jcc.com/sql_stds.html`.

[7] Extensible Markup Language (XML) 1.0 specification. W3C Recommendation, February 1998. http://www.w3c.org/TR/REC-xml.

[8] XML Linking Language (XLink). W3C Working Draft, March 1998. http://www.w3c.org/TR/WD-xlink.

[9] XML Pointer Language (XPointer). W3C Working Draft, March 1998. http://www.w3c.org/TR/WD-xptr.

[10] G. O. Arocena and A. O. Mendelzon. WebOQL: Restructuring documents, databases and Webs. In *Fourteenth IEEE International Conference on Data Engineering (ICDE'98), Orlando, Florida*, 1998.

[11] P. Atzeni, G. Mecca, and P. Merialdo. To Weave the Web. In *International Conf. on Very Large Data Bases (VLDB'97), Athens, Greece, August 26-29*, pages 206–215, 1997. http://www.dia.uniroma3.it/-Araneus/.

[12] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive Web sites. In *VI Intl. Conference on Extending Database Technology (EDBT'98), Valencia, Spain, March 23-27*, 1998.

[13] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *14th International Conference on Object-Oriented and Entity-Relationship Modelling, (OOER'95) Gold Coast, Australia, December 12-15, 1995. Lecture Notes in Computer Science, Vol. 1021, Springer-Verlag*, pages 341–354, 1995.

[14] R. G. G. Cattel. *The Object Database Standard ODMG-93*. Morgan Kaufmann Publishers, San Francisco, CA, 1994.

[15] Workflow Management Coalition. The workflow reference model. WfMC Document n.TC00-1003, http:/-www.wfmc.org, 1995.

[16] Workflow Management Coalition. Workflow and internet: Catalysts for radical change. WfMC White Paper, http:/www.wfmc.org, 1998.

[17] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'98), Seattle, Washington*, pages 414–425, 1998.

[18] P. Fraternali and P. Paolini. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable Web applications. In *VI Intl. Conference on Extending Database Technology (EDBT'98), Valencia, Spain, March 23-27*, 1998.

[19] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of Workflow Management: From process modeling to infrastructure for automation. *Journal on Distributed and Parallel Database Systems*, 3(2), 1995.

[20] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Sixteenth International Conference on Very Large Data Bases, Brisbane (VLDB'90)*, pages 455–468, 1990.

[21] G. Mecca, P. Atzeni, A. Masci, P. Merialdo, and G. Sindoni. The ARANEUS Web-Base Management System. In *ACM SIGMOD International Conf. on Management of Data (SIGMOD'98), Seattle, Washington*, pages 544–546, 1998. Exhibition Program. http://www.dia.uniroma3.it/Araneus/.

[22] J. A. Miller, D. Palaniswami, A. P. Sheth, K. Kochut, and H. Singh. WebWork: METEOR$_2$'s web-based workflow management system. *Journal of Intelligent Information Systems*, 10(2):185–215, 1998.

[23] F. Paradis and A. M. Vercoustre. A language for publishing virtual documents on the Web. In *Proceedings of the Workshop on the Web and Databases (WebDB'98) (in conjunction with EDBT'98)* http://www.-dia.uniroma3.it/webdb98, 1998.

[24] M.A. Roth, H.F. Korth, and A. Silberschatz. Extended algebra and calculus for ¬1NF relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, December 1988.

[25] G. Simeon and S. Cluet. Using YAT to build a Web server. In *Proceedings of the Workshop on the Web and Databases (WebDB'98) (in conjunction with EDBT'98)* http://www.dia.uniroma3.it/webdb98, 1998.

[26] G. Sindoni. Incremental maintenance of hypertext views. In *Proceedings of the Workshop on the Web and Databases (WebDB'98) (in conjunction with EDBT'98)* http://www.dia.uniroma3.it/webdb98, 1998.