Automatic Development and Maintenance of Semi-dynamic Web Sites through Declarative Specifications

Diomidis Spinellis and Vassilios Karakoidas
Department Management Science and Technology
Athens University of Economics and Business
Greece
email: {dds,bkarak}@aueb.gr

Abstract

Traditionally, the realization of Web sites involves either static content developed using web authoring tools like Microsoft's Front Page and DreamWeaver, or dynamic content delivered by a database driven front-end, where the structured content is organized in a relational schema and dynamically generated on the fly. The limitations of statically-authored web pages are easy to discern and for a number of applications, the use of a database introduces a level of additional complexity that makes the choice a part of the problem space rather than the solution space. We introduce a different approach, which is suitable for managing middle-sized semi-dynamic web sites. The technological requirements of this approach are well-known open source technologies, such as CVS and XML.

Keywords

Web site implementation, Semi-dynamic web sites, XML, make, Bibtex, XSLT, X-Schema, HTML.

1 Introduction

Traditionally, the realization of Web sites involves either static content developed using web authoring tools like Microsoft's Front Page and DreamWeaver, or dynamic content delivered by a data-oriented front-end, where the structured content is stored in a relational database and dynamically generated on the fly. When our group faced the successive failure of both the above approaches, we decided to adopt the task of exploring ideas for a radically different implementation style, based on the declarative specification of all the site's elements. We believe that our approach and many of the lessons we learned can be applied in numerous similar situations, leading to a lightweight, structured, consistent, and maintainable web site building method.

The limitations of statically-authored web pages are easy to discern. The content is entered in an unorganized manner, and, as a result, can be inconsistent in both structure and presentation. While the use of cascading style sheets can help one obtain a consistent look, their use still requires discipline. The authored pages are however still unstructured and the resulting site can be difficult to modify and reorganize. Furthermore, the static authoring model often imposes a centralized management and maintenance style; all additions and changes have to go through a single person, creating a bottleneck, often leading to outdated content.

Adopting a database driven approach is supposed to solve the aforementioned problems. Separating the source data from its (dynamically generated) marked-up version (HTML code) leads to a consistent yet flexible generation of web pages. In addition, the database's relational model will impose its structure on the source data being stored. Finally, the use of a database, allows concurrent updates by different users.

However, for a number of applications, the use of a database introduces a level of additional complexity that makes the choice a part of the problem space rather than the solution space. A database-driven web site requires the implementation of a front-side interface to transform the web site's content into HTML code, and a back-end interface to allow stakeholders enter, review, and update data. The back-end client interface typically requires setting up and maintaining appropriate access permissions. These may need to be integrated into an organization wide single login facility, or operated under a specific security policy. In the second case procedures for setting up passwords, resetting them, and revoking them need to be established for a number of applications, the use of a database introduces a level of additional complexity that makes the choice a part of the problem space rather than the solution space.and followed. A properly running database also requires a skilled database administrator to install it, maintain it, organize backups, and perform modifications to the database schema. In addition, because marked-up content is generated by a front-end program accessing the database, both the front-end and the database must be extremely robust, running on a 24×7 schedule. The front-end, being an executable program working on untrusted data (the web page requests) can become the target of malicious attacks, and must therefore be inspected to ensure its robustness. To minimize the risk of an attack against the database (that would jeopardize the organization's data) the database server has to be installed on a machine separate from the web server, behind a (properly configured) firewall. Finally, the extraction of content from a database often induces the web site's designers and stakeholders adopt a query-style interface. Such an interface is typically less usable than browsable web pages, and the served content is often ignored by search engines, leading to reduced visibility of the (meticulously structured) content. All in all, a database-driven approach appears to be suitable only for those with ample resources to justify the full development and appropriate maintenance of a sophisticated infrastructure.

A new needed for this problem and we volunteered to set up a project for finding it. We had already undergone the two aproaches we aforementioned. The ad-hoc authoring tool–based approach was abandoned, because it led to an inconsistent look and stale content, while the maintenance of a subsequent database-driven design approach was proving intractable for the resources that our group could afford. Our goal was to experiment with a different approach, proving its suitability for managing middle-sized semi-dynamic web sites.

2 Requirements

The functional requirements for our center's site were simple, but not trivial, and had already been satisfied twice in a slightly simpler form. The site's pages should represent the content and the relationships we illustrate in Figure 1. The center consists of multiple research groups. Members of our center and our research projects are associated with the center as a whole, and, typically, also with one or more research groups. Publications, such as journal articles and books, are also associated with the center, individual members (the authors), projects that funded the corresponding work, and groups that performed the work. Note that members, publications, and projects associated with one or more groups are aggregated are typically associated again with the research center as a whole. For the sake of simplicity, we have omitted from our description and the diagram a number of additional relationships, such as the member directing a group or managing a project. As an example of the type of content we were looking for, the research center and each member, group, or project should have a web page with a list of the corresponding publications; the research center and each group should have pages listing their projects and members.

As we hinted in the previous paragraph, the problem with the previous implementations was not the creation of the site satisfying the functional specifications, but the lack of a number of important non-functional requirements. Before embarking on our third attempt, we articulated for the first time those non-functional requirements we thought important, to ensure that our third attempt would produce a result with a longer life. The following is a list of the non-functional properties we deemed important enough to guide our design.

Openness The tools used in the realization of the web site should be available as open source, or supported by multiple vendors. We wanted to avoid becoming tied with a particular proprietary tool. We reasoned that openness would mitigate two risks: finding a maintainer trained to use a particular proprietary tool, and obtaining resources for upgrading and maintaining the tool.

Observability The semantic distance between the specification of an element and its implementation should be minimal [1]. The site's look and content should be maintainable using standard tools and techniques. If possible, the site's maintainer should not be required to learn a scripting language like PHP, or Perl, or a framework like Java2EE or .NET. An approach based on declarative specifications and domain-specific languages would allow end-users, or members close to end-users be involved in maintaining the site, without risking the bottleneck of going through multiple intermediaries.

Robustness The web site should not depend on any programs other than the web server for serving its content. Users updating the data, should be able to author, validate and review, generating localy the HTML pages, their changes without requiring network connectivity. This would allow them to work productively

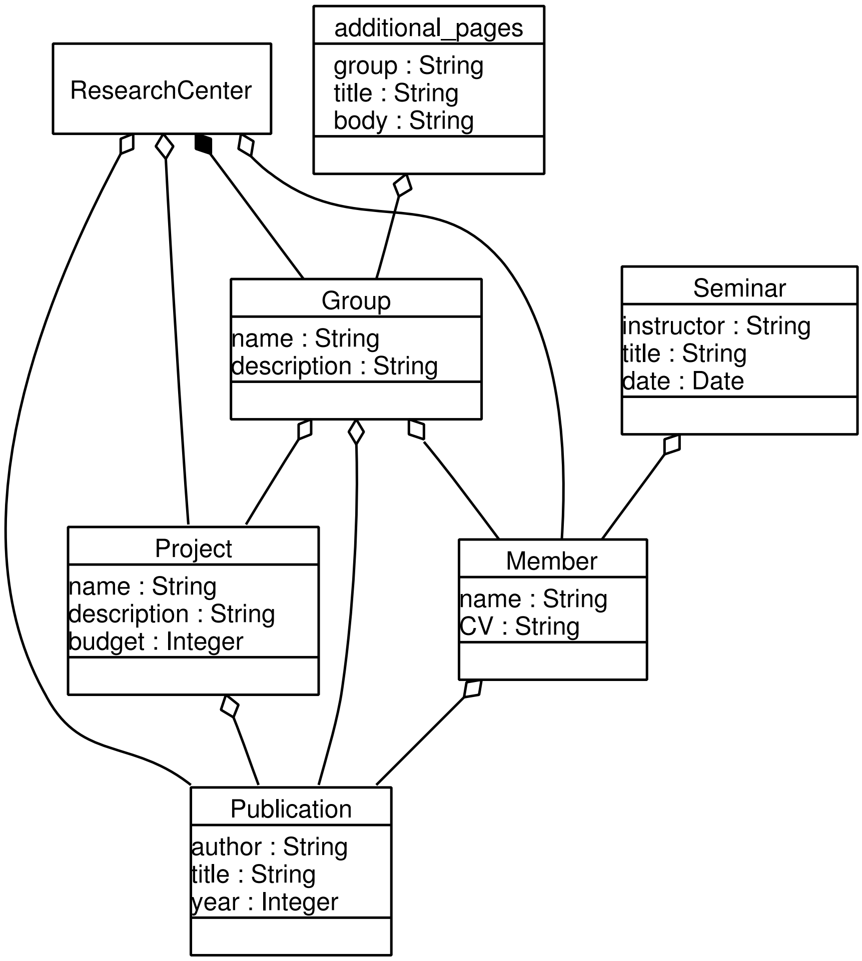

Figure 1: Overview of data element relationships.

over dial-up connections or while on the road. In addition, all the editing can be done with a simple text editor and the each end-user can work both in win32 and unix oriented systems. Minimizing the dependencies on additional servers (such as a database or an application server) and on the network should result in a more robust and easier to maintain system.

Parsimony The implementation effort for implementing the system should be minimal. This would minimize errors and maintenance costs. We reasoned we could satisfy this requirement by using existing tools, if their choice satisfied the other non-functional properties.

3 Design Process

3.1 Design Concepts

Our main guiding principle for was to create a continuous multi-person The guiding paradigm we adopted for implementing the web site was a continuous multi-person development activity. Live web sites continuously evolve; adopting the content authoring paradigm implied by the first approach was a mistake. Empirical evidence supports this observation. Figure 2 illustrates the changes in rankings given to 21 Greek government department web sites between the years 2002 and 2003. The X shape in the rank changes between the one year and the next is, we believe, the result of statically authored web pages degrading to a point of irrelevance, and then being overhauled from scratch. A database-driven approach also hinders evolution. Changes to the content's presentation require the modification and installation of the front end page generator; not a typical lightweight operation. Changes to the data schema are even more intrusive requiring a synchronized modification of the data, the front end, and the back end.

Continuous multi-person development projects are quite common in software development. Numerous developers contribute and coordinate their work through a version control system, like CVS that maintains a master repository of the source code. Concepts like the daily build [2] or the *current* and *stable* branches as practiced by numerous open source projects allow the maintenance of a known-good product. What we needed for our approach were appropriate declarative language-based formalisms for expressing our data, its transformation into HTML pages, and the generation process. We were clear that we wouldn't support the content in a *Makefile* [3].

3.2 Processes

In Figure 3 is depicted a UML use case diagram of our system. We have three actors which are interacting with the system:

ContentDeveloper A content developer is responsible for uploading data in the system. In our case is typically a person for each research group. He can upload XML data files and import new bibliography entries.

ContentAdministrator The content administrator is responsible for the maintenance and the evolution of the system. He corrects problems with the XML Data,

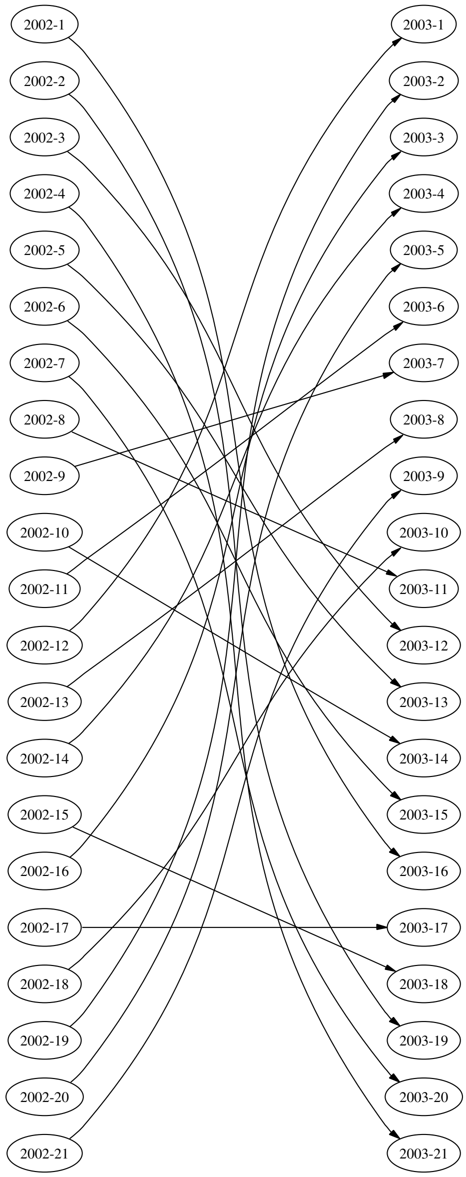

Figure 2: Yearly changes in web site rankings

develops new features or corrects existing ones in the XSLT scripts and the X-Schemas.

ContentConsumer The content consumers are outside our system and they have access only in the generated web site.

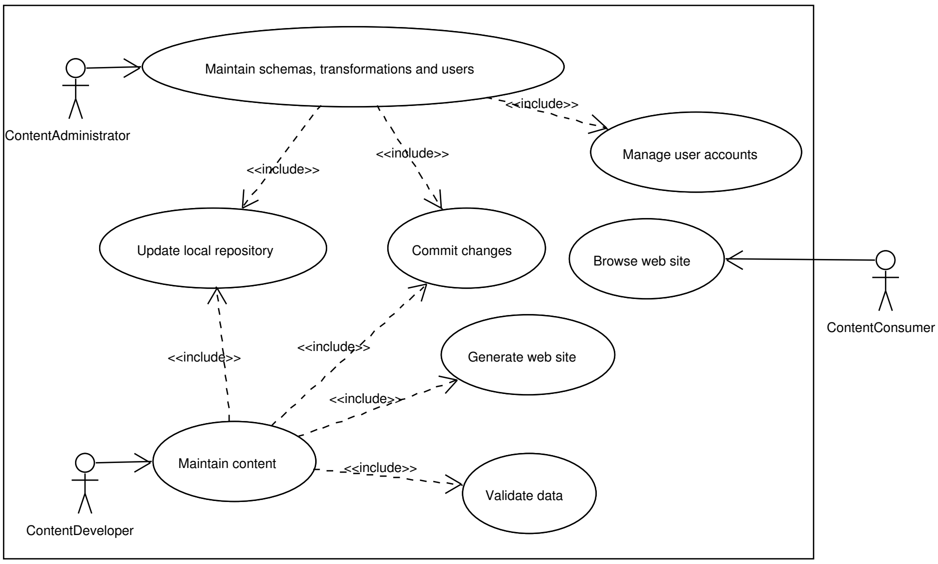

Figure 3: UML Use Case Diagram of the System

The use cases that comprise our system are (Figure 3):

Maintain schemas, transformations and users This use case describes the maintainence and development procedures of the system. Each Content Developer can update the DTD schemas, XSL scripts and the make file in order to correct errors or add new features.

Update local repository Each user in the system must update the local repository each time he uses the system. To do so, they must execute a CVS update command. If there is a conflict, the users must correct the problem and rerun the update command.

Commit changes Content developers and administrators must commit the changes from their local repository to the CVS repository in the web site server.

Maintain content This is the overall use case that allows Content Developers to maintain the content of the web site. Content consists of XML data and bibtex collection files.

Generate web site Content administrators and developers can generate the web site localy for preview. This procedure is performed with the execution of a make file command. Upon completion, Content administrators can review their content and

Validate data Content developers can validate the data in the local repository before the final commit.

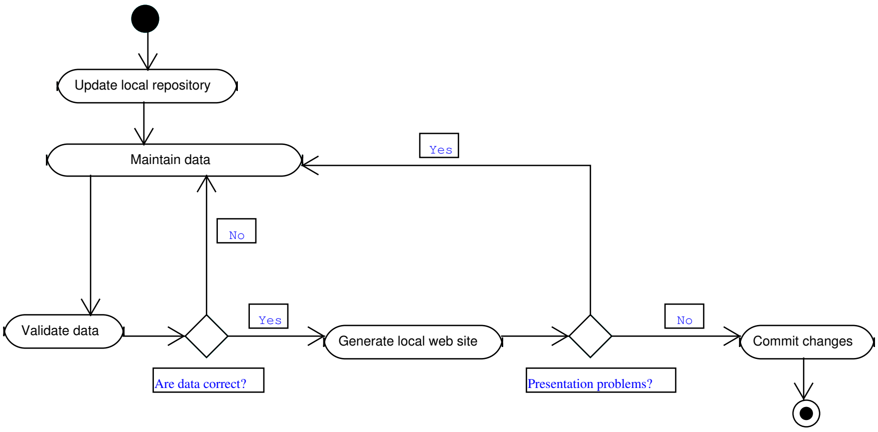

Figure 4: UML activity diagram for Maintain Content

Manage user accounts Content Administrators can perform user management in the system.

Browse web site Normally everyone who is willing to access our web site. They can only browse the generated HTML pages.

In figure 4 we show the activity diagram of the most complex use case in the system. The Content Developer first updates the local repository and then begins the initiation of data maintanance by adding, removing or modifying XML files and bibtex entries. Upon completion data vaildation must be performed before the commit to the CVS repository. If data are valid, then local web site generation must be performed to correct possible presentation problems. After that, the data are ready to be submitted to the main CVS repository. After updating the CVS

4 Implementation

4.1 Key Technologies

Once the design was finalized, implementation proved to be an almost "hollow"' activity, since it did not involve almost anything of what is typically described as coding.

The first step involved selecting and setting up the appropriate tools. In order to meet the requirements we set in section 2, we decided to use open and popular technologies as key elements of our system. We adopted the concurrent versions system (CVS) [4] [5] to coordinate the distribution and update of all the system's components. Authentication for managing content was handled by the Unix group membership mechanism of the host where the CVS repository was stored. We also used *BibTeX* [6] [7] and *bib2xhtml* [8] for transforming the publications into HTML and *xmlstarlet* [9] for validating and transforming all other XML-based data [10]. For data transformations we implemented a system based on XSLT [11]. XSLT is a language for tranforming XML documents. Finally, GNU *make* [12] and a couple of shell script constructs were used for handling the project's *Makefile*. The complete setup including all tools proved to be portable between Unix and Microsoft Windows, with team members working on machines running different versions of Windows, GNU/Linux, and FreeBSD.

4.2 System Development

The next step was a series of iterations where we modeled the data's schema on representative XML files and concurrently wrote the validation DTDs and the transformation XSLTs. The version control system was already proving its value at this point for coordinating the work between the two paper's authors. Because many page elements, like a project's description, could contain content more elaborate than plain text, we used W3C's modular XHTML specification for importing existing XHTML elements editorin our DTDs. This helped us keep our schema description simple, but the corresponding schema expressive.

The automated validation and generation of content was expressed as *Makefile* rules. The individual files XML files are merged in a single XML file for cross validating identifier reference attributes (IDREF). The same file is used to extract the identifiers of all projects, members, and groups into *Makefile* variables. A simple loop then generates the HTML files corresponding to each of the above elements.

The HTML content is by default generated on the local machine, where its maintainer can verify it. After the new content is validated and verified, the maintainer can commit the change to the central CVS repository. A separate *Makefile* rule can then be used, to execute an update command on the host serving the content to the web. The command retrieves the updates from the CVS repository and regenerates the pages on the web-server's file area. All components of our system are under version control, all pages are automatically tagged with identifiers denoting their source, helping the traceability of changes. All exchanges between the developers' machines and the CVS and web host are performed using the secure session shell (SSH) protocol guaranteeing the data's integrity and confidentiality.

You can see representative samples of a project's DTD schema description in figure 5. XML data in figure 6. XSLT transformationn in figure 7, and HTML result in figure 8.

Our system has a distribution that works in both win32 and Unix Enviroments (Linux and FreeBSD). The directory structure of a typical local repository is depicted in figure 9. Most of our users are working in win32 enviroments and as a result we decided to upload a win32 version of the needed tools in the main repository under the directory "bin". The installation procedure is very simple, and the bootstrap tools that requires are only cvs for the initial check out and a ssh client. The needed keys for the secure shell session are provided once in each user. A full version of the system now demands 8 megabytes of space in local machine, plus some extra for the local content store and generation.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="seminar">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="sem_date">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:pattern value="[0-9]{8}" />
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="sem_time" type="xs:string" />
    <xs:element name="sem_title" type="xs:string" />
    <xs:element name="sem_room" type="xs:string" />
    <xs:element name="sem_url" type="xs:string" minOccurs="0" />
    <xs:element name="sem_duration" type="xs:string" />
   </xs:sequence>
   <xs:attribute name="by" type="xs:string" />
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 5: The projects X-Schema file

```
<?xml version="1.0"?>
<seminar by="m_bkarak">
 <sem_date>20040314</sem_date>
 <sem_time>19:00</sem_time>
 <sem_title>Regular Expressions</sem_title>
 <sem_room>906</sem_room>
 <sem_url>http://www.eltrun.gr/seminar/presentations.ppt</sem_url>
 <sem_duration>3 hrs</sem_duration>
</seminar>
```

Figure 6: A typical seminar XML file

5 System Adoption

After we had finished the implementation we were somewhat concerned by how the system would be received by those who would be maintaining the pages. Our research center is multidisciplinary: under its roof are both hard core software engineers using the same tools we adopted in their everyday work, and researchers whose background is management science, marketing, or finance who are comfortable with GUI interfaces.

Our fears were justified. The first presentation of the system to its users ended

```
<xsl:template match="seminar" mode="full">
 <h3><xsl:value-of select="sem_title" /></h3>
  <xsl:element name="a">
  <xsl:attribute name="name">
   <xsl:value-of select="sem_date" />
  </xsl:attribute>
</xsl:element>
 Presenter: <xsl:apply-templates
             select="/eltrun/member_list/member [@id=current()/@by]"
             mode="simple-ref" /><br />
 Date: <xsl:call-template name="date">
        <xsl:with-param name="date" select="sem_date" />
       </xsl:call-template> <br />
 Time: <xsl:value-of select="sem_time" /><br />
 Duration: <xsl:value-of select="sem_duration" /><br />
 Location: <xsl:value-of select="sem_room" /><br /><br />
 <xsl:element name="a">
  <xsl:attribute name="href">
   <xsl:value-of select="sem_url"/>
  </xsl:attribute>
  Download the presentation
 </xsl:element>
</xsl:template>
```

Figure 7: The XSLT transformation file for seminars

almost in a revolt. Non-technical users expressed their inability to comprehend what an XML document was, while technical members helpfully argued for providing a GUI front end. By targeting the users with the least technical experience, promoting our system's "vogue" attributes, such as the use of open source software tools, and convincing them to try to enter a few elements into the system, we were able to overcome the initial reservations and start the data migration process.

The next round of problems surfaced when users began entering malformed or invalid data into the system. This resulted in all users acquiring the copies of the malformed XML files, and strange error messages given to unsuspecting users. As is the custom in a number of development efforts, we had expected the users to verify the changes they made before committing them into the master repository. Non-technical users were however not aware of this etiquette and were committing their changes with the hope they were correct. We used the shared list we had established to explain the importance of following the correct procedures when committing changes. After a few days we got the impression that non-technical users were becoming confident in their work, even proud of sharing sophisticated tools and processes with software engineers.

Two weeks after the initial system presentation all users where able to upload and maintain their data. The inexperienced users learned to edit XML files, importing

```
<tbody valign="top">
<tr>
<td align="left" width="82%">
<h2>Eltrun Seminars</h2>
<a href="index.html#20040314">14 March 2004 - Regular Expressions</a>
<br><br><br>
<h3>Regular Expressions</h3>
<a name="20040314"></a>
Presenter:
<a href="../members/m_bkarak.html">
Mr. Vassilios Karakoidas
</a><br>
Date: 14 March 2004<br>
Time: 19:00<br>
Duration: 3 hrs<br>
Location: 906<br><br>
<a href="http://www.eltrun.gr/seminar/presentations.ppt">
Download the presentation
</a>
```

Figure 8: A generated HTML seminar file

bibtex entries into the system and commiting to the master CVS repository. They just followed steps in a procedure that they see as a black box. Our technical persons experienced more problems than the inexperienced ones and that was came as a suprise for us. Some of them were already familiar with the key technologies and they tried to change the tools proposed with others more user-friendly. These initiatives resulted in corruption of the local repositories, malformed XML data and wrong bibliography entries. After a few false attempts they decided to follow the proposed system..

6 Lessons Learned

We believe our design satisfies the non-functional properties we listed in Section 2, and that our approach stands a higher chance to succeed where the two other approaches failed. The initial user reaction was not favorable, but this can be explained by the significantly higher requirements we placed on our users. Typical users are not well acquinted with command line tools, and often see these as a threat to their productivity. Instead of giving instructions by email to an unfortunate web site maintainer, they now had to become active members of an evolving web site maintenance effort. Not all members of our research center proved ready to take this responsibility. Many groups delegated the maintenance to a single person. Still, however, we succeeded in distributing the previously entirely centralized maintenance effort across our groups. Summarizing, we believe that adopting a software development metaphor and tools for developing and maintaining semi-dynamic web site is a practical worthwhile approach.

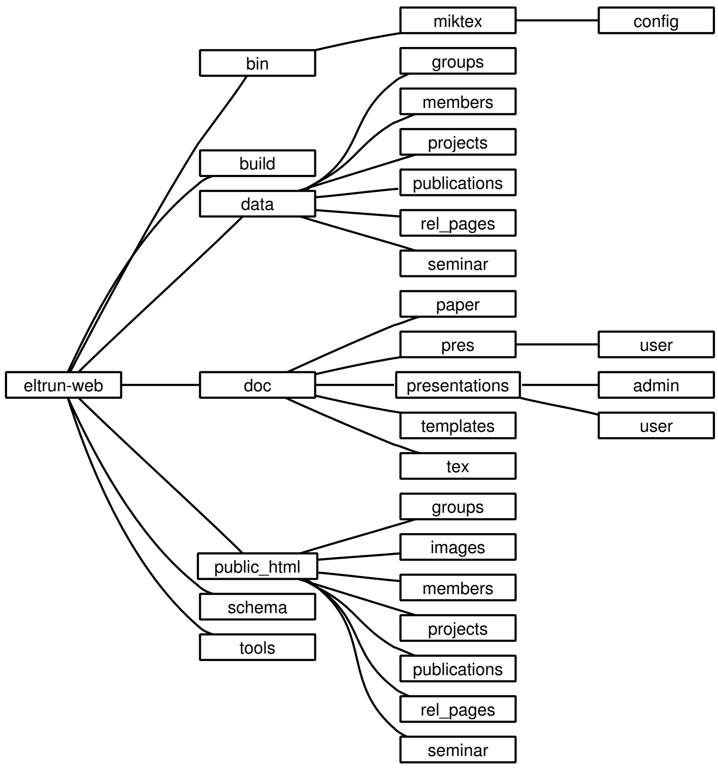

Figure 9: Directory structure of local repository

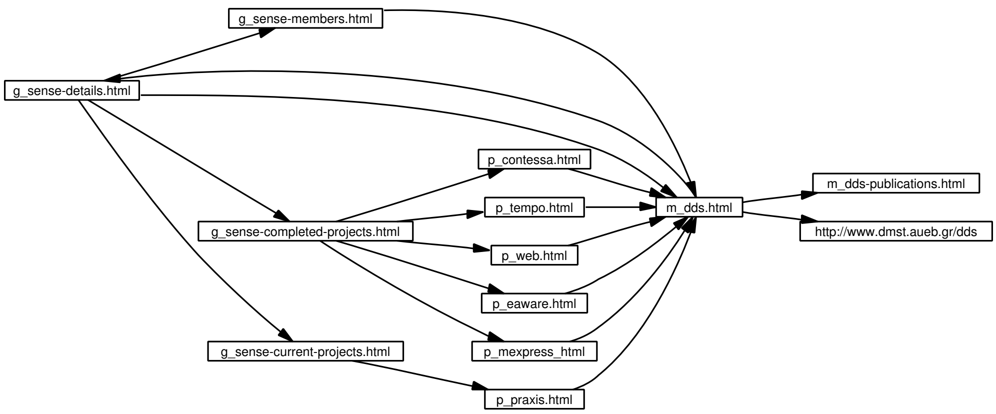

Figure 10: Dependency graph of web site

7 Acknoledgements

We would like to thank Mr. Manoli Skordalaki for his very perceptive comments during the compilation of this paper. Special thanks to Diamianos Chatziadoniou who reviewed early versions of this paper.

References

[1] Diomidis Spinellis and V. Guruprasad. Lightweight languages as software engineering tools. In J. Christopher Ramming, editor, *USENIX Conference on Domain-Specific Languages*, pages 67–76, Berkeley, CA, October 1997. Usenix Association.

[2] Micheal A. Cusumano and Richard W. Selby. *Microsoft Secrets*. The Free press, 1995.

[3] Andrew Oram and Steve Talbott. *Managing projects with make, 2nd Edition*. O'Reilly and Associates, Sebastopol, CA, 1991.

[4] Moshe Bar and Karl Franz Fogel. *Open Source Development with CVS*. The Coriolis Group, Scottsdale, AZ, 2001.

[5] Derek Robert. Cvs–concurrent versions system v1.12.5, December 2003. Available online at http://www.cvshome.org/docs/manual/cvs-1.12.5/cvs.html.

[6] Oren Patashnik. Bibtexing, February 1988. Available online at ftp://sunsite.unc.edu/pub/packages/TeX/biblio/bibtex/distribs/doc/btxdoc.tex.

[7] Dana Jacobsen. Bibtex, December 1996. Available online at http://www.ecst.csuchico.edu/ jacobsd/bib/formats/bibtex.html.

[8] Diomidis Spinellis. bib2xhtml - convert bibtex files into html, June 2004. Available online at http://www.spinellis.gr/sw/textproc/bib2xhtml/.

[9] Mikhail Grushinskiy. Xmlstarlet command line xml toolkit, February 2004. Available online at http://xmlstar.sourceforge.net/.

[10] World Wide Web Concortium. Extensible markup language (xml), August 2003. Available online at http://www.w3c.org/XML/.

[11] World Wide Web Consortium. Xsl transformations (xslt) version 1.0, November 1999. Available online at http://www.w3.org/TR/xslt.

[12] Free Software Foundation (FSF). Gnu make, April 2002. Available online at http://www.gnu.org/software/make/.