XATA2008

XML: Applications and Associated Technologies

XML: Aplicações e Tecnologias Associadas

6^{th} National Conference

Editors:

José Carlos Ramalho
João Correia Lopes
Salvador Abreu

February 14-15th, 2008

Design: Benedita Contente Henriques
Editors: J. C. Ramalho, J. C. Lopes and Salvador Abreu
Composition: J. C. Ramalho, J. C. Lopes and Salvador Abreu
Copies: 100
Edition: February 2008
ISBN: 978-972-99166-5-6

Preface:

This volume contains the papers presented at the Sixth Portuguese XML Conference, called XATA (*XML, Aplicações e Tecnologias Associadas*), held in Évora, Portugal, 14-15 February, 2008. The conference followed on from a successful series held throughout Portugal in the last years: XATA2003 in Braga, XATA2004 held in Porto, XATA2005 in Braga, XATA2006 in Portalegre and XATA2007 in Lisboa.

Due to the research evaluation criteria that are being used to evaluate researchers and research centers, national conferences are becoming deserted. Many did not manage to gather enough submissions to proceed in this scenario. XATA made it through, however, with a large decrease in the number of submissions. In this edition a special meeting will join the steering committee with some interested attendees to discuss XATA's future: the internationalization, the conference model, ...

We think XATA is important in the national context. It has succeeded in gathering and identifying a community that shares the same research interests and has promoted some collaborations. We want to keep "the wheel spinning".

This edition has its program distributed by first day's afternoon and next day's morning. This way we are facilitating travel arrangements and we will have one night to meet.

I hope everyone finds these days interesting.

See you in Évora,

José Carlos Ramalho

Organising Committee:

- Salvador Abreu — Departamento de Informática da Universidade de Évora (chair)
- José Saias — Departamento de Informática da Universidade de Évora
- Pedro Salgueiro — Departamento de Informática da Universidade de Évora
- Vítor Nogueira — Departamento de Informática da Universidade de Évora
- João Correia Lopes — INESC Porto/Faculdade de Engenharia da Universidade do Porto
- José Carlos Ramalho — Universidade do Minho

Scientific Committee:

José Carlos Ramalho	Universidade do Minho — Chair
Ademar Aguiar	Universidade do Porto e INESC Porto
Alberto Brandão Simões	Universidade do Minho
Alberto Rodrigues da Silva	Instituto Superior Técnico
Alda Lopes Gançarski	Institut National des Télécommunications
Ana Paula Afonso	Universidade de Lisboa
Andreia Malucelli	Pontifícia Universidade Católica do Paraná, Brasil
Benedita Malheiro	Instituto Superior de Engenharia do Porto
Carlos Damásio	Universidade Nova de Lisboa
Cristina Ribeiro	Universidade do Porto e INESC Porto
Francisco Couto	Universidade de Lisboa
Gabriel David	Universidade do Porto e INESC Porto
Giovani Librelotto	Centro Universitário Franciscano, Brasil
José João Almeida	Universidade do Minho
José Luís Borbinha	Instituto Superior Técnico
José Paulo Leal	Universidade do Porto
João Correia Lopes	Universidade do Porto e INESC Porto
João Moura Pires	Universidade Nova de Lisboa
Luís Carriço	Universidade de Lisboa
Luís Ferreira	Instituto Politécnico do Cávado e do Ave
Luís Moura e Silva	Universidade de Coimbra
Marta Jacinto	Instituto das Tecnologias de Informação na Justiça
Mário Gaspar da Silva	Universidade de Lisboa
Miguel Ferreira	Universidade do Minho
Nuno Horta	Instituto Superior Técnico
Pedro Almeida	Universidade de Aveiro
Pedro Antunes	Universidade de Lisboa
Pedro Henriques	Universidade do Minho
Rui Lopes	Universidade de Lisboa
Salvador Abreu	Universidade de Évora
Stephane Gançarski	LIP6, University P. & M. Curie Paris 6

Additional Reviewers:

Anália Lourenço
Hugo Manguinhas
Marcos Sunye
Nuno Freire

Acknowledgements:

The editors wish to thank all those who contributed with content to XATA2008 proceedings which is the sixth volume in this series.

Firstly we thank all the authors for the paper production that can be found in this volume.

Secondly, we thank all reviewers for the excellent work they have done in reviewing their assignments helping to increase the overall quality of the works being published.

Moreover, the comments they made about the submitted works will contribute to improve the research related to the application of XML technologies.

Finally, we finish with a word of appreciation to all participants who decided to attend an event where the exchange and discussion of ideas is paramount. Their presence will definitely enlarge the debate, generating new ideas, which will trigger new works that may be reported in the next edition of this conference.

José Carlos Ramalho
João Correia Lopes
Salvador Abreu

Sponsoring Organisations:

The Organising Commitee is very grateful to the following organisations for their support:

- Departamento de Informática da Universidade de Évora
- Departamento de Informática da Universidade do Minho
- Departamento de EngenhariaInformática da Universidade do Porto
- Centro de Investigação em Tecnologias de Informação da Universidade de Évora
- Delta cafés
- Microsoft Corporation Portugal

Contents

IV Semantic Web and Ontologies

V Applications of XML

VI Index

A Tutorial on XProc: An XML Pipeline Language

Rui Lopes

LaSIGE
University of Lisbon
rlopes@di.fc.ul.pt

Abstract. This tutorial will present the syntax and semantics of a new XML processing language specification, XProc, which is being standardised by the World Wide Web Consortium, for the description of sequences of operations to be performed on XML documents.
An XProc pipeline specification processes a set of inputs through an ordered set of operations, resulting on a set of outputs. Each operation performs an atomic transformation on inputs, feeding its results to the next operation. Such operations include well-known technologies, such as XSLT and XInclude, as well as special purpose micro-operations, e.g. insert and delete.
The purpose of this tutorial is the dissemination of this new XML technology, including step-by-step examples on how to create pipelines, and how to leverage the language's constructs and operations, easing the task of building simple and complex XML processing applications.

1 Synopsis

XML processing languages have come a long way. With the increasing use of such languages, including XSLT 2.0 [1], XQuery [2], building complex and full-fledged XML applications entirely in XML is starting to become a necessity. Moreover, background support through XML Schema [3] and other data validation technologies enforce the correctness of XML applications. However, a last piece has still being missing, the glue for all of these technologies. Typically, developers connected these self-contained technologies either through programming APIs, or by hacking custom shells scripts or makefiles. XProc [4] is W3C's answer to the standardisation of XML pipeline processing technologies. With XProc, XML developers can specify entire applications without leaving the XML syntax, thus leveraging existing knowledge on already existing standards and tools (e.g., XPath). This tutorial will provide an overview on how to build simple XML processing pipelines, and dive into some more advanced concepts, showing the expressive power of XProc.

1.1 Pipeline Concepts

The first part of the tutorial will present a short background on the different concepts inherent of XML pipelines, namely *pipelines*, *steps*, *inputs*, *outputs*, *options*, *parameters*, and *XPath contexts*.

1.2 Syntax Overview

After the brief introduction to the main concepts inherent of XProc, a syntax overview will be presented. Small examples will build upon each others, in order to leverage its core concepts, including the different namespaces of the language, scoping, and binding.

1.3 Steps

The core of this tutorial will be spent by presenting the different core steps of XProc, i.e., its main elements. These include *p:pipeline*, *p:for-each*, *p:viewport*, *p:choose*, *p:group*, and *p:try*.

1.4 Other Pipeline Elements

Complementing steps, other pipeline elements provide additional value to the language. Such examples include using inputs and parameters, debugging and documentation helpers, declaring steps, as well as building complex XML applications with pipelines and libraries.

1.5 Standard Step Library

The last part of this tutorial will provide a flavour of the standard step library of XProc. The concept of micro-operations will be introduced, as a practical way to define simple procedures to be applied onto XML documents. Well-known processors will be approached too, such as XSLT and XInclude.

References

1. Kay, M.: XSL Transformations (XSLT) Version 2.0. Technical report (2007) http://www.w3.org/TR/xslt20.
2. Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J.: XQuery 1.0: An XML query language. Technical report (2007) http://www.w3.org/TR/xquery.
3. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: Xml schema part 1: Structures second edition. Technical report (2004) http://www.w3.org/TR/xmlschema-1/.
4. Walsh, N., Milowski, A., Thompson, H.: XProc: An XML Pipeline Language. Technical report (2006) http://www.w3.org/TR/xproc.

Creating a National Federation of Archives using OAI-PMH

Luís Miguel Ferros[1], José Carlos Ramalho[1] and Miguel Ferreira[2]

[1]Departament of Informatics – University of Minho
Campus de Gualtar, 4710 Braga – Portugal
{lmferros, jcr}@di.uminho.pt

[2]Department of Information Systems – University of Minho
Campus de Azurém, 4800 Guimarães – Portugal
mferreira@dsi.uminho.pt

Abstract. This paper describes the planning stages of the creation of a national level federation of archives. The Open Archives Initiative Protocol for Metadata Harvesting will be used for collecting metadata being produced by local archives using any compliant records management software. The first stage of implementation will harvest metadata produced with the DigitArq [1-3] platform in a pilot group of Portuguese Regional Archives. DigitArq relies on Encoded Archival Description (EAD) metadata to describe its collections. However, the overwhelming flexibility and complexity of EAD make the harvesting operation more complex than usual. This paper addresses possible ways of exchanging EAD records using OAI-PMH as the basis for a central repository of metadata that will enable the creation of advanced information services at the national level.

Keywords: Metadata, harvesting, data provider, service provider, interoperability, repository, digital archives, OAI-PMH, EAD, XML, Dublin Core

1 Introduction

The use of standards for describing items of information introduces one major benefit in the globalised information society that we live on: it enables information systems to be interoperable. The arrival of the Open Archives Initiative [1] enabled repositories and other types of information systems to share and concentrate information, and more often metadata, allowing a great deal of new information services to be developed, such as centralised search engines as Google Scholar [4] or OAIster [5] and global statistics such as the ones provided by the Registry of Open Access Repositories (ROAR) [6].

The purpose of this paper is to discuss the major obstacles one will face while attempting to centralize archival metadata coming from different regional

repositories. More specifically, we will address the problematic of using OAI-PMH to transfer metadata encoded in EAD (Encoded Archival Description) [7]. Due to its flexibility, hierarchical nature and complex structure, EAD presents several challenges to anyone trying to exchange this type of information in an efficient way.

This paper is organized as follows: in the section 2 we provide a short description of the EAD standard. Section 3 describes OAI-PMH and its common requests; in section 4 we describe the architecture of the proposed system and problems we expect to face during its implementation; and finally, in section 5 we draw some conclusions and outline some points of future work.

2 EAD as the archival standard for descriptive metadata

EAD [2] is a non-proprietary standard for encoding archival of finding aids. The purpose of EAD is to provide information about archival resources in standard syntax and normalized language. An instance of an EAD document is composed of three parts: a header, a front matter and the archival description of collections (a collection of documents created by a single person, family or organization).

The header section contains information about the EAD document itself [2]. The front matter embeds information convenient for publishing or rendering the finding aid. The archival description contains the bulk of an EAD document instance, which describes the content, context, and extent of a body of archival materials, including administrative and complementary information that facilitates the use and the discovery of the material.

Information in an EAD instance is organized in unfolding hierarchical levels that account for an overview of the whole collection to be followed by a more detailed view of its constituent parts, e.g. sections, classes, documents, etc [1] (Fig. 1). Each level of description contains information that roughly follows the ISAD(g) model [8]. Examples of descriptors that are commonly found at one of these description levels are: title, range of dates, biographic history, archival history, scope and content, existence and location of originals and copies, physical characteristics, etc.

EAD can be used to describe all sorts of archival material, these being physical, like books, reports and photographs, or digital, such as databases, Web pages or spreadsheets.

```
<ead>
  <eadheader>
    <eadid />
    <filedesc>
      <titlestmt>
        <titleproper />
      </titlestmt>
    </filedesc>
  </eadheader>
  <archdesc level="otherlevel" otherlevel="F">
    <did>
      <abstract/>
      <unitid countrycode="PT" repositorycode="ADPRT">EMP/BM</unitid>
      <physdesc>
      ...
      </physdesc>
      ...
    </did>
    ...
    <dsc>
      <c level="otherlevel" otherlevel="SC">
        <did>
          <unitid countrycode="PT" repositorycode="ADPRT">CR</unitid>
          <physdesc>
          ...
          </physdesc>
        ...
        </did>
        ...
        <dsc>
          <c level="otherlevel" otherlevel="SR">
            <did>
              <unitid countrycode="PT" repositorycode="ADPRT">001</unitid>
              <physdesc>
              ...
              </physdesc>
            ...
            </did>
            ...
          </c>
        </desc>
      ...
      </c>
    </dsc>
  </archdesc>
</ead>
```

Fig. 1 - Extract of an EAD instance

3 OAI-PMH as the standard for metadata exchange

The Open Archives Initiative Protocol for Metadata Harvesting (OAI-PMH) [9] plays the important role of enabling repositories to become interoperable. The main goal of OAI-PMH is to allow geographically separated repositories to exchange metadata thus allowing the creation of repository federations.

The OAI-PMH defines a communication protocol that defines how the transference of metadata should be performed between two basic entities: data providers and service providers.

Data providers support the OAI-PMH as way to publish their metadata. The service providers send OAI-PMH requests to data providers and harvest their metadata that will serve as the basis for the development of more advanced services. The interaction between these two entities is depicted in the Fig. 1. As one can observe, a service provider that wants to harvest metadata sends a HTTP request to a data provider, which, according to the request, responds with a XML message.

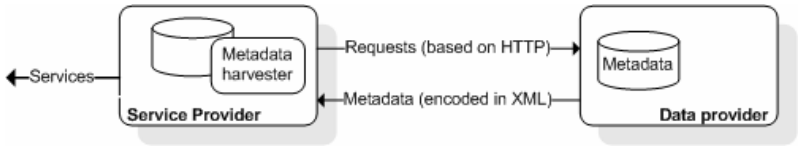

Fig. 2 - Interaction between OAI-PMH entities

For data providers to be able to publish their metadata through OAI-PMH, they must implement six types of requests (called the verbs in this context):

- GetRecord - This verb is used to retrieve an individual metadata record from a repository. Required arguments specify the identifier of the item from which the record is requested and the format of the metadata that should be included in the record [10].
- Identify - This verb is used to retrieve information about a repository. Some of the information returned is required as part of the OAI-PMH. Repositories may also employ the Identify verb to return additional descriptive information [10].
- ListRecords - This verb is used to harvest records from a repository. Optional arguments permit selective harvesting of records based on set membership and/or datestamp [10].
- ListIdentifiers - This verb is an abbreviated form of ListRecords, retrieving only headers rather than records. Optional arguments permit selective harvesting of headers based on set membership and/or datestamp [10].
- ListMetadataFormats - This verb is used to retrieve the metadata formats available from a repository. An optional argument restricts the request to the formats available for a specific item [10].
- Listsets - This verb is used to retrieve the set structure of a repository, useful for selective harvesting [10].

Fig. 3 shows an request of an request that list the metadata formats that can be disseminated from the repository http://www.perseus.tufts.edu/cgi-bin/pdataprov for the item with unique identifier oai:perseus.tufts.edu:Perseus:text:1999.02.0119.

The response to this request (Fig. 4) shows that 3 metadata formats are supported for the given identifier: oai_dc, olac and perseus. For each of the formats, the location of an XML Schema describing the format, as well as the XML Namespace URI is given.

```
http://www.perseus.tufts.edu/cgi-bin/pdataprov?
       verb=ListMetadataFormats&identifier=oai:perseus.tufts.edu:Perseus:text:1999.02.0119
```

Fig. 3 – OAI-PMH request with the verb *ListMetadataFormats*

```
<?xml version="1.0" encoding="UTF-8"?>
<OAI-PMH xmlns="http://www.openarchives.org/OAI/2.0/"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.openarchives.org/OAI/2.0/
         http://www.openarchives.org/OAI/2.0/OAI-PMH.xsd">
  <responseDate>2002-02-08T14:27:19Z</responseDate>
  <request verb="ListMetadataFormats"
    identifier="oai:perseus.tufts.edu:Perseus:text:1999.02.0119">
    http://www.perseus.tufts.edu/cgi-bin/pdataprov</request>
  <ListMetadataFormats>
   <metadataFormat>
     <metadataPrefix>oai_dc</metadataPrefix>
     <schema>http://www.openarchives.org/OAI/2.0/oai_dc.xsd
       </schema>
     <metadataNamespace>http://www.openarchives.org/OAI/2.0/oai_dc/
       </metadataNamespace>
   </metadataFormat>
   <metadataFormat>
     <metadataPrefix>olac</metadataPrefix>
     <schema>http://www.language-archives.org/OLAC/olac-0.2.xsd</schema>
     <metadataNamespace>http://www.language-archives.org/OLAC/0.2/
      </metadataNamespace>
   </metadataFormat>
   <metadataFormat>
     <metadataPrefix>perseus</metadataPrefix>
     <schema>http://www.perseus.tufts.edu/persmeta.xsd</schema>
     <metadataNamespace>http://www.perseus.tufts.edu/persmeta.dtd
       </metadataNamespace>
   </metadataFormat>
  </ListMetadataFormats>
</OAI-PMH>
```

Fig. 4 – OAI-PMH response

4 Architecture and system operation

In this section, we describe the architecture and operational characteristics of a system that uses the OAI-PMH to offer two important services: centralization of metadata and interoperability between repositories.

Metadata centralization

Fig. 2 shows a simplified diagram of the system's architecture that uses the OAI-PMH to harvest metadata from several EAD repositories [1-3].

The protocol provides, as shown in the diagram, two main types of participants: the data providers and the service providers. In this particular example, the data providers are the digital repositories that hold the archival metadata. To ensure interoperability, the data providers must provide their metadata according to common descriptive metadata standard. In this case, this should be the EAD. The service provider offers builds additional added-value services from the metadata harvested and stored in its central repository (CR).

The harvesting task is performed by the Metadata harvesting module by sending OAI-PMH requests to data providers, which according to the type of request, will receive appropriate XML responses. Some of those will deliver the EAD metadata that is hosted in the data provider.

The metadata harvested by the previous module is transformed and adapted as necessary to fit the central repository (CR). This process is carried out by the "XML EAD to CR" component.

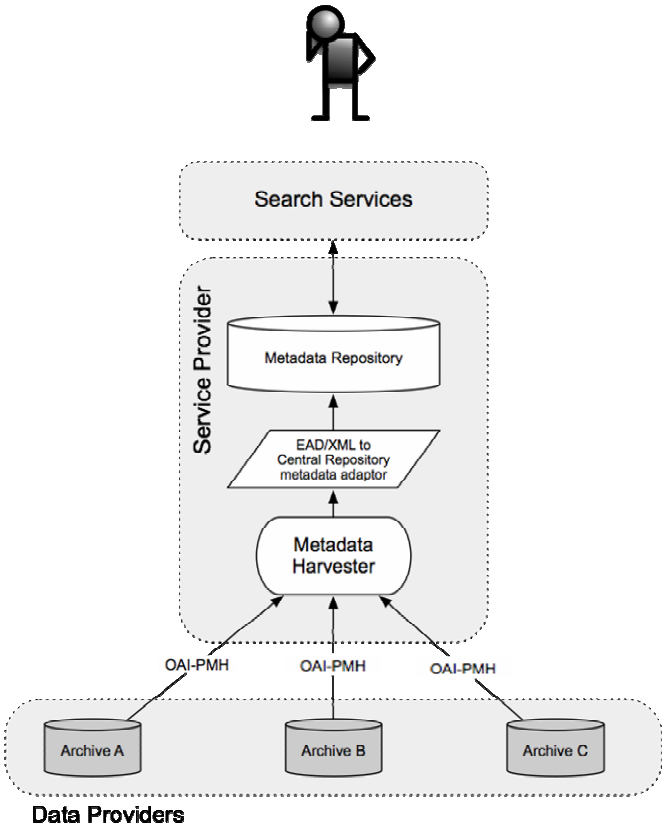

Fig. 5 - Overview of the system's architecture

EAD represents hierarchically the overall records that compose a collection. A single collection may range from a few dozen nodes to a staggering size of millions. This being said, a question may be raised: If the atomic unit in an EAD file is the collection (with all its complexity and size), how does one harvest this type metadata if only one of its nodes gets updated? Well, this problem can be addressed in three different ways:

1. **Harvest the complete collection that holds the updated record.**

This solution is simple to implement, since the answer to the request consists in sending the corresponding updated collection. The service provider must only integrate the new version of the collection in its central repository by simply replacing the old one by the new collection. Although easy to implement, it is an inefficient solution, because a simple change or insertion of a new record in a previously

harvested collection will trigger a subsequent harvest of the complete set of nodes that compose the collection. This strategy is very inefficient at the bandwidth usage level.

2. **Harvest the whole branch that contains the updated record.**

This method triggers a transfer of data much lower that previously described. However, the operation of extracting the nodes from data providers and the integration of these records with the service providers is much more complex. The extraction involves selecting the nodes all the way up the branch of the collection's tree. Consequently, the integration will be achieved by the replacement of old records by the received records on the corresponding collection in the central repository.

3. **Harvest only the updated record.**

This is certainly the most efficient approach, as only the new or changed records are harvested independently of this position in the collection structure. The problem in this approach is that an EAD file is not valid if whole collection is not present. Using this strategy implies that the nodes are identified uniquely at the national level as so to guarantee that they are integrated in the right collection in the correct position. Analysing EAD one can verify the existence of a CountryCode (code of the country) and a RepositoryCode (code of the repository) elements, which compose the complete reference of a record. This way, the existence of these fields in the complete reference of a record and the unique references inside a given repository, guarantee the uniqueness references in the repositories universe that publish EAD. So, the task of harvesting only a record (or a set of records), independently of the hierarchical organization can be possible, as long as we assume to be exchanging incomplete EADs that are not valid from the EAD Schema perspective.

Interoperability between repositories

To increase interoperability among repositories, these should disseminate their metadata in formats other than EAD. For example, it is common practice for the library community to disseminate its records in Dublin Core (DC) [11] independently of the metadata schemas that are being used in their information systems. Having a dissemination port for Dublin Core enables the EAD repositories to be compatible to an assortment of pre-existing service providers, e.g. OAIster, ROAR and Google Scholar.

Because the metadata schemas used in the archival repositories depicted in this paper are based in EAD [2], one must implement a crosswalk from EAD to DC. This issue has already been addressed by Prom and Habing in [12].

5 Conclusions and Future Work

In this paper was described the simple architecture of an information system capable of implementing the concept of a National Federation of Archives that is based on EAD that uses the OAI-PMH for exchanging metadata.

The EAD structure, due to its hierarchical nature and overwhelming flexibility makes the use of OAI-PMH a non-straightforward process. The paper identifies the causes that make this type metadata be so hard to harvest and synchronize. To address this issue, we have identified and described three possible solutions, that differ both in complexity and efficiency. Instead of exchanging full blown EAD collections we propose that solely the updated nodes are harvested and integrated in their corresponding collections. However, one should note that the description of a record outside of the context of a collection may not be sufficient to fully understand it, as the ascending nodes of description are required to make it complete and structured. However, for the sole purpose of transferring data and incrementally updating a central repository this solution seems highly appropriate.

As future work we will address in more detail the EAD to Dublin Core crosswalk strategy so that the national repositories may also disseminate their metadata and integrate with existing service providers.

It may also be interesting, on the service provider side, to develop modules that transform metadata in formats other than EAD so that one can provide services for other types of repositories.

REFERENCES

[1] M. Ferreira and J. C. Ramalho, "DigitArq - Creating and Managing a Digital Archive," presented at ICCC/IFIP International Conference on Electronic Publishing, Brasília, Brazil, 2004.

[2] M. Ferreira and J. C. Ramalho, "DigitArq: Creating a Historical Digital Archive," presented at 5ª Conferência da Associação Portuguesa de Sistemas de Informação, Lisboa, 2004.

[3] J. C. Ramalho, M. Ferreira, L. Ferros, M. J. P. Lima and A. Sousa, "Digitarq 2 - Nova plataforma aplicacional para gestão de Arquivos Definitivos," presented at 2nd International Conference on Enterprise Archives (2ª Conferência Internacional de Arquivos Empresariais), Seixal, Portugal, 2006.

[4] Google, "Google Scholar." [Online]. Available: http://scholar.google.com.

[5] University of Michigan, "OAIster." [Online]. Available: http://www.oaister.org/.

[6] University of Southampton, "Registry of Open Access Repositories (ROAR)." [Online]. Available: http://roar.eprints.org/.

[7] Library of Congress, "EAD - Encoded Archival Description," in Library of Congress, 1998. [Online]. Available: http://www.loc.gov/ead/. [Accessed 2004]

[8] International Council on Archives, "ISAD(G): General International Standard Archival Description, Second edition," International Council on Archives 0-9696035-6-8, 1999.

[9] Open Archives Initiative, "The Open Archives Initiative Protocol for Metadata Harvesting." [Online]. Available: http://www.openarchives.org/pmh/.

[10] Open Archives Initiative, "The Open Archives Initiative Protocol for Metadata Harvesting Version 2.0." [Online]. Available: http://www.openarchives.org/OAI/2.0/openarchivesprotocol.htm.
[11] Dublin Core Metadata Initiative, "Dublin Core Metadata Initiative." [Online]. Available: http://dublincore.org/.
[12] C. J. Prom and T. G. Habing, "Using the open archives initiative protocols with EAD " presented at 2nd ACM/IEEE-CS joint conference on Digital libraries, Portland, Oregon, USA 2002.

RADX - Rapid development of web applications in XML

José Paulo Leal and Jorge Braz Gonçalves

DCC-FC, University of Porto
R. Campo Alegre, 823 – 4150 – 180 Porto, Portugal
zp@dcc.fc.up.pt, jgoncalves@ipg.pt

Abstract. This article presents an on-going project whose goal is the fast development of web applications based on the RAD model. The system resulting from this project - RADX - generates web applications for XML data management. RADX consists of two main components: an application engine to run web applications based on XML documents that it is configured using XSLT transformations; a meta-application for generating and managing applications that run with the application engine. The main feature of the meta-application is the ability to generate XSLT configurations from 2nd order transformations, applied to data document type definitions in XML Schema. These configurations can be changed, allowing customization of the application. RADX intends to be a system rapid development of small web application and prototypes of larger systems.

Keywords: XML, RAD, web applications, framework, prototyping.

1 Motivation

The goal of this project is the implementation of a system for rapid and easy development of web applications based on XML [1] documents. These data management applications, with a simple and intuitive interface, are intended for small data sets and/or applications prototypes. XML is increasingly used to transfer and archive data since it enables the interoperability between different platforms and facilitates its future use.

By using XML for data representation, RADX enables the use of XML technologies in all web application layers, avoiding the need for successive format conversions. Web applications are usually composed of three layers, each with its own data model: the presentation layer based on HTML or XHTML trees, the logical layer based on object graphs, and the data layer based on tables of a relational databases management system. The use of different formats in each layer requires several data conversions that have a significant cost.

As RADX uses XML both in the data layer and the presentation layer, we decided to explore the use of XML also in the logic layer using XSLT [2] transformations on XML documents.

2 An example

This section describes, through an example, the development and use of a web application with RADX. This system has two main components: the *Application Manager* and the *Application Engine*. The former is a meta-application that allows the creation, management, and elimination of applications that run in the latter.

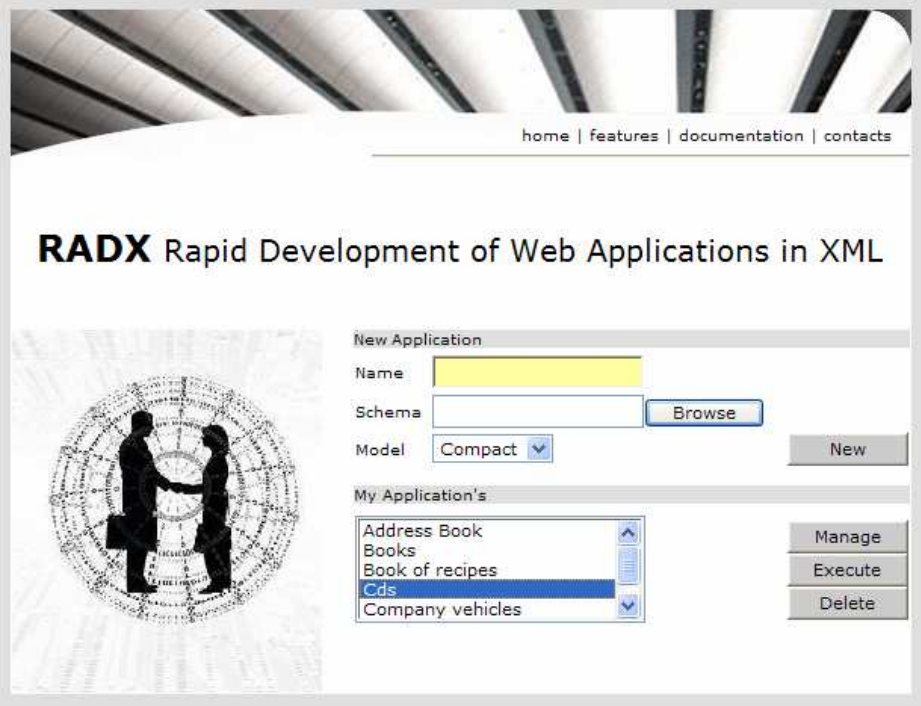

Fig. 1. A screen shot of the Application Manager

The initial screen of the Application Manager is shown on Fig. 1. One of its main features is the ability to generate an application from a XML Schema [3] document. This document describes the structure of the application's data. Usually, this document will be produced in an IDE with specific support for this standard, such as XML Spy, Oxygen or Eclipse.

The upper part of the Application Manager's initial screen is a form for generating a new application. It requires entering the application's name, the location of the XML Schema document and choosing a model for the application's GUI. As part of

the creation of a new application, several second order XSLT transformations are executed, producing documents of various types. These documents will be needed by the Application Engine to run this application.

Generated applications are listed on the form on the lower part of the screen (Fig. 1). The selected application can be managed, executed or eliminated. The manage button gives access to a form for editing individual documents generated for an application. The delete button eliminates the application from the system. The execute button launches the selected application on the Application Engine.

Let us assume we want to generate an application to manage a collection of music CDs. Using a specialized XML Schema editor we have already produced a document type similar to the presented in Fig. 2:

```
- <schema targetNamespace="http://www.example.org/cds" elementFormDefault="qualified">
  - <element name="cds">
    - <complexType>
      - <sequence>
        - <element name="cd" maxOccurs="unbounded" minOccurs="0">
          - <complexType>
            - <sequence>
                <element name="title"/>
                <element name="interpreter"/>
                <element name="year"/>
              </sequence>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </schema>
```

Fig. 2. An XML Schema for a CD management application

Using the Application Manager we create a new application named *cds* with the file containing the document on Fig. 2 and the compact GUI model. In the resulting application the user will have access to a range of options: search CDs, create a new CD entry, edit or delete existing CDs and navigate trough them.

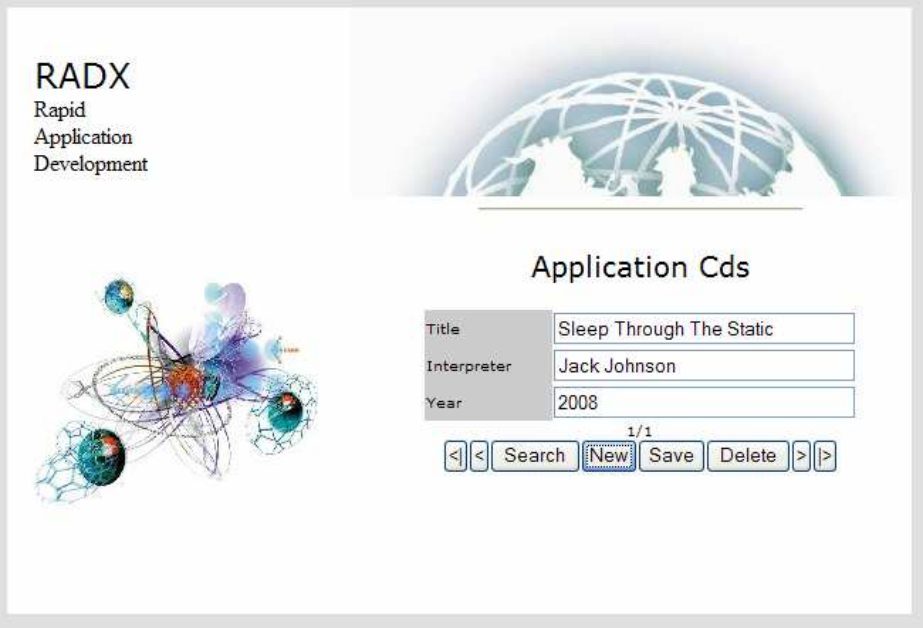

Fig. 3. Application Engine screen shot

The GUI shown on Fig. 3. depends both on the data structure, given by the XML Schema document, and on the selected GUI model. The currently available options are the compact model, used in the example, and the expanded model. The compact model is based on a single form used for all operations: create, modify, search, display, navigate and remove elements. The expanded model has other views for some operations. For instance, the result of a search operation is a list containing the selected elements. Other GUI models are planned for future versions of the Application Manager.

3 Related work

James Martin presented, in 1991, a model of software development known as RAD (Rapid Application Development) with the publication of a book [4] with the same name. This model aims to shorten the software development cycle, producing faster results and reducing costs without losing quality. It also intends to address to the problem of excessive project duration felt in the standard development methodologies commonly used before the 90, when the time to conclude many projects affected their viability.

The RAD model became popular and has been used on tools targeted to different database management systems and/or programming languages. Some of these RAD tools are used for development of web applications, as is the case of Omnis [5], Intraweb [6], RAD-Studio [7], Delphi-for–PHP [8], WebSnap [9], TurboGears [10].

However, most of these tools need a significant amount of programming to produce a working application. The goal of RADX is the creation of a working application without any programming. To be sure, in order to customize an application in RADX the programmer may need to edit some configuration files but a working application is created immediately after defining its data schema.

The architecture of RADX is based on the standard MVC (Model-View-Controller) architectural pattern. This pattern is normally used in programs with graphical interaction with the user [11]. This standard was proposed by Trygve Reenskaug in 1978 as a design solution for Smalltalk [12]. Its main purpose is to serve as a mediator between the human mental model and the digital model that actually operates in the computer, facilitating the control of large and complex data structures [13]. In this pattern the model represents the knowledge of the program, the view is a representation of the model in which are highlighted some of their attributes and controller serves as an intermediary between the user and the system, acting on the model and providing views of the model in accordance with the user requests.

The MVC pattern is typically used in the design of graphical applications implemented in object oriented languages. The participants in this pattern are classes of objects. Nevertheless, the MVC concepts of Model, View and Controller can also be used to structure a graphical application without being used in its design. These concepts have been successfully used to separate and structure configuration files in highly configurable web applications [15].

4 Architecture

As mentioned before, RADX integrates two distinct components: an application engine for running web applications operation based on XML, a meta-application for managing applications running on the application engine. This section describes the architecture of each component.

4.1 Application engine

The application engine's architecture is based on the MVC pattern, a pattern often used in programs with interaction with the user. As outlined in the section 1, we pretend to use XML as the data format, and XSLT transformations as the corner stones of RADX development. Therefore, we try to base each of the participants in the MVC standard - Model, View and Controller in the processing of XML documents.

The model of the RADX applications – the set of its features - is the management of data persisted in XML documents. Therefore, the model can be encoded as a transformation on the data, generating an updated XML document as a result.

The graphical interface of RADX's applications consists of HTML pages that allow the user to view or interact with the data. These HTML pages are the views and are obtained by processing the XML data.

It should be noted that the changes implemented in either the model or view need a set of variables associated with the interaction state as parameters of the

transformation. For example, the navigation in a set implies that the present view only shows the element (the equivalent of a record) currently selected. These XSLT transformation parameters are specific to each user and change during his interaction with the application.

The controller, as the name suggests, is responsible for controlling the other constituents of the application, namely the model and view. As explained earlier, these two participants in the MVC pattern are XSLT transformations controlled by a set of parameters that constitute the state. Thus, in order to encode all logic as XSLT files, the model is implemented as a transformation that produces a XML document representing the state.

The application engine is a framework for running web applications. It has three extension points (usually called "hot spots") for each web applications it runs. Those extension points are XSLT transformations and each corresponds to one of the participants of the MVC model.

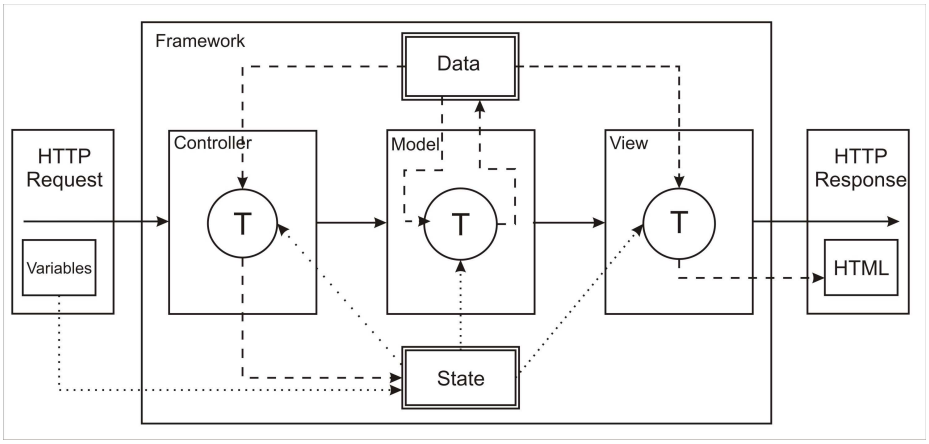

Fig. 4. Processing an HTTP request in the RADX application engine

To better explain this architecture we present in Fig. 1. the processing of an HTTP request received by the RADX application engine. When an HTTP request is received from a web browser it triggers three transformations in sequence, producing an HTML document that is sent back to the browser in the HTTP response. In the diagram we denote execution flow by solid arrows. Transformations are represented by circles with a “T” label, connected by dashed arrows to their input and output documents and by dotted arrows to their parameters.

Two DOM objects – Data and State - have a central role in this process and are both represented by double-line rectangles in the diagram. The former reflects the application data as persisted in a XML document file; there is a Data object for each application managed by the application engine. The latter is the state of the interaction with each user; there is a State object for each active user session of the application engine. Both these objects are used in all three transformations: the Data object is the transformation data source and the State object contains the transformation parameters.

In the beginning of this process the variables of the HTTP request are copied to the State object for the current user. The first transformation, representing the controller, uses Data as the original document and changes the State. The second transformation, representing the Model, transforms the Data into itself. The third transformation, representing the View, transforms Data into HTML and sends it in the HTTP response.

4.2 Application Manager

The implementation of the application manager was based on "Model 2" that is the MVC applied to Java web applications. Therefore, the controller is composed by a servlet that receives HTTP requests and according to these requests acts on the model and the view. The Model is a set of Java beans that processes instructions from the servlet. The View is composed of JSPs that uses the model data to produce an HTML output.

The application manager's function is to manage the applications supported by the application engine. To create the new applications it generates XSLT configuration for the application engine using 2nd order transformations applied to XML Schema type definitions.

5 Implementation

The RADX system was implemented as two independent Java web applications: the application engine and the application manager. They were both implemented using the Tomcat servlet container. In this section we highlight the main issues encountered in the development of these two components.

5.1 Application engine

The application engine has a rather simple design: it has a single servlet that acts as front controller to all applications requests; this servlet instances the Application class for each web application it manages.

The main function of the servlet is to apply the three XSLT transformations associated with the model, view and controller, as explained in the previous section. It also manages the users' state using the standard session mechanism provided by the servlet container.

Each Application instance contains a DOM object and a collection of transformations. The main method of this class invokes a transformation on the data object that is outputted to different objects according to its type: model transformations are copied to the data object itself and serialized in the file system; controller transformations change the DOM object representing the state; view transformations produces HTML that is outputted to the HTTP response channel.

When the Application class is instanced the corresponding data file is loaded to its DOM object as well as all its transformations. It should be noted that some HTTP requests just change the current view and do not activate model transformations. The data object is serialized to its data file only when it actually changes. On the other hand the Application class is thread-safe to ensure data integrity in concurrent operations.

5.2 Application manager

The application manager is a standard model 2 Java web [15] application that allows the creation, customization and deletion of web applications supported by the application engine.

The main feature of the application manager is the creation of a new application when a XML Schema is uploaded. Several second order XSLT transformations are applied to this document in order to create the XSLT files that configure the hot spots of the application engine, as well as the initial XML data file.

The main issue in these transformations is the identification of definitions of elements containing sets of elements of the same type, which would correspond to entities in a relational model. For this purpose we use the XML schema sequence indicator.

Element types in XML Schema can be either named or anonymous. To simplify the detection of type definition with sequence indicators we start by normalizing XML Schema documents. As would be expected, this normalization is also an XSLT transformation that unfolds all named type reference into a XML Schema with only anonymous type definitions.

The same procedure for identifying repeated elements is used for generating each of the three XSLT transformations from the normalized schema. For each set of repeated elements a corresponding set of templates is produced in the target transformation.

Using the application manager the programmer can edit the individual transformations to customize its application. In most cases the XSLT encoding the view will be the first candidate for customization in order to change the application graphical appearance. The controller transformation will need to be edited only to add or remove access to model operations. We expect the model transformation to be the one to require less customization.

6 Conclusion

This article presents RADX, a system for development of web applications prototypes based on XML documents in a rapid and easy way.

The fact that in RADX data persistence and graphical interfaces are both based on XML, led us to use XML transformations in the implementation of the logic layer. The architecture of the Application Engine of RADX is based on applying three successive transformations corresponding to the constituents of the MVC

architectural pattern. We also highlight the main issues encountered during the development of the RADX system.

As it is an ongoing project is not yet possible to make an assessment of the RADX system efficiency. In any case, since it is a system designed for prototyping, its main advantage is the ease of applications creation and adequacy of the user's needs.

References

1. Extensible Markup Language (XML), http://www.w3.org/XML/
2. XSL Transformations (XSLT), http://www.w3.org/TR/xslt/
3. XML Schema Definitions (XSD), http://www.w3.org/XML/Schema
4. Martin, J: Rapid Application Development. Macmillan Coll Div, New York, (1991)
5. Omnis Studio - http://www.omnis.net;
6. Intraweb - http://www.atozedsoftware.com/intraWeb;
7. RAD-Studio - http://www.codegear.com/products/radstudio;
8. Delphi-for - PHP - http://www.codegear.com/products/delphi/php;
9. WebSnap - http://dn.codegear.com/article/27404;
10. TurboGears - http://www.turbogears.org.nyud.net/;
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, (1994)
12. Reenskaug, T.: Models-Views-Controllers, Xerox PARC technical note, http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html, (1979)
13. Reenskaug, T.: The Model-View-Controller (MVC) - Its Past and Present, http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html, (2003)
14. Leal, J., Domingues, M., Rapid development of web interfaces to heterogeneous systems, SOFSEM 2007: Current Trends in Theory and Practice of Computer, pp 716-725, Lecture Notes in Computer Science, Springer-Verlag.
15. Singh, T., Sterns, B., Johnson, M., et al.: Designing Enterprise Applications with the J2EE Platform, Addison-Wesley, (2002)

CGI::Auto – Automatic Web-Service Creation

Davide Sousa, Alberto Simões, and José João Almeida

Departamento de Informática
Universidade do Minho
kevorkyan@gmail.com,{ambs,jj}@di.uminho.pt

Abstract. The creation of a CGI or a WebService as an interface for a command line tool is not as unusual as it may seem. It is extremely usual and useful.
There are applications developed as command line tools that can be useful for different purposes, and different kind of users. Some of these users might not be able to run these tools directly. For instance, it is not easy to install a bunch of Perl modules to have a small tool working. For these situations, it is easier to make the tool available in the Web or as a WebService.
The problem with making the tool available in these fashions, is that programmers tend to rewrite the tools to incorporate the CGI or XML specific layers.
We defend that these CGI or WebService interfaces should use the already available command line tool, without any change. This interface should be able to read a simple textual specification of how the command line tool works, and buid the CGI or XML specific layers automatically.
The `CGI::Auto` module aims this purpose: to encapsulate command line tools in a CGI layer based on a textual specification, transforming the command line tool in a web application.

1 Introduction

When solving a problem in a quick way, programmers tend to write small command line tools. These commands (or scrips) can do a lot of interesting things, but their dissemination is not easy. People who want to run them have to install (and probably compile) the command.

The process of making these tools available to other users can involve two approaches:

- to prepare a standalone package, that can be downloaded over the Internet for local installation;
- to prepare a WebService [1] or an interactive Web Application for final-user usage.

While the first approach is preferred for most cases, it is not the easier way for all kind of users. If the tool was developed using, for instance, a scripting language, the final-user will need to install the scripting language interpreter to run it. If

the programmer wants to distribute a binary it is needed to compile the tool for each architecture and operating system. Other option might be the distribution of the source-code. That is handy if the final-user is also a programmer, or else, she will be lost when compiling the tool.

When the command line tool is to be used by a non technical user, most programmers tend to choose the second approach to release their tools. To prepare a Web Application is quite easier, as the tool will run in the server computer, and thus, no cross compiling is necessary.

The problem with this approach is that while it is not difficult to develop a user-friendly web interface, it involves the modification of the original tool, adding a Web layer.

We might argue that this process is performed just once. But it is not true. When adding a new feature to the command line tool we will want to reflect it in the Web application. In the same way, when changing the Web application, we will want to reflect the changes in the command line tool.

We defend that the best approach is to develop a Web layer that uses the command line tool directly. The programmer can change the command line tool, and the new features will be available for the final-user automatically, with no need to re-design the Web Application.

For this purpose we developed a Perl module, named `CGI::Auto`. This module provides means to write a CGI[1] [3] application that behaves accordingly with a textual specification of how the command line tool is executed. `CGI::Auto` will not re-implement command line tools: it will run them directly on the host operating system shell.

This module uses a textual specification that describes how to run a specific command (or generically any pipeline of commands), and generates the Web application on the fly. The textual specification just describes the accepted flags and option for the command line tool, together with some strings to make the Web application more user-friendly.

`CGI::Auto` will be introduced, showing how to build a Web interface for two well known Unix command line tools: word count (`wc`) and pattern searching (`grep`).

2 `CGI::Auto` Approach

Before developing `CGI::Auto`, we needed to study how command line tools work. The main requirement for `CGI::Auto` is that it should be general enough to produce interfaces for all kind of command line tools. Command line tools can interact with the user in many different ways [2]:

- accept input through standard input;
- print errors to a standard error file handle;
- print results to the standard output;
- create files;

[1] Future work intends to incorporate a WebService layer as option.

- read files;
- accept optional flags or required ones;
- accept optional or required arguments;
- can be compound using a pipeline;

All these kind of operations need to be treated by `CGI::Auto`, and need to be defined in some way in the specification file, in a simple an concise description.

The `CGI::Auto` CGI have just one command: the invocation of the `CGI::Auto` main method with the textual specification of how the Web Application should interact with the command line tool.

The CGI will act as an abstraction layer between the command line tool and the Web Application. It will show a simple form to the user, where the command line tool available options are shown as common HTML widgets: radio buttons, combo boxes, text fields and text areas. When the user fills in the form and submits it, a command line pattern is filled with the user information and run on the host operating system. The output will then be presented to the user.

2.1 Example 1 — Word Count

A command line tool works with a number of specific parameters, and usually operates on one or more files. As a first example, consider the word count command (`wc`) which, among other things, allows the user to count the number of lines and the number of words in a given file. Its main usage is:

```
wc -l -w file
```

Basically, it accepts two switches that specify what we want to count (lines or words) and a filename. The two switches are shown to the user as HTML radio button, where the user can select one, both or none.

Regarding the file to process, we need to make it available in the host operating system. We can supply it to the Web application using three methods: to upload the file using a common CGI upload field, to copy and paste the content of the file into a text area, or to specify an URL where the file should be downloaded.

The specification to run this command under `CGI::Auto` is as simple as:

```
my %wc= (
  description => "Counting lines and words in a file.",
  command => "wc [%words%] [%lines%] [%file%]",
  args => {
    words => {type => "flag",value => "-w",name => "Count words"},
    lines => {type => "flag",value => "-l",name => "Count lines"},
    file  => {type => "upload",name => "File to process..."}
        }
);
```

The first argument on this specification is a string that describes the web application. Follows the command line pattern with placeholders, delimited by `[%`

and `%]`. These placeholders are then explained, one by one, in the arguments list.

These arguments can have different types. On this example we have flags (for command line switches), and upload fields (for supplying files to the command line). Flags have a value (the value that is replaced in the command line pattern if the user select it), and a name, that is shown to the user in the Web application. File uploads just have a name, with a simple description of what will be done with that file.

Figure 1 shows the CGI created with this configuration. The area **A** corresponds to the user interface with the command line tool. After activating the desired parameters and uploading the file to process, the command result is presented (area **B**).

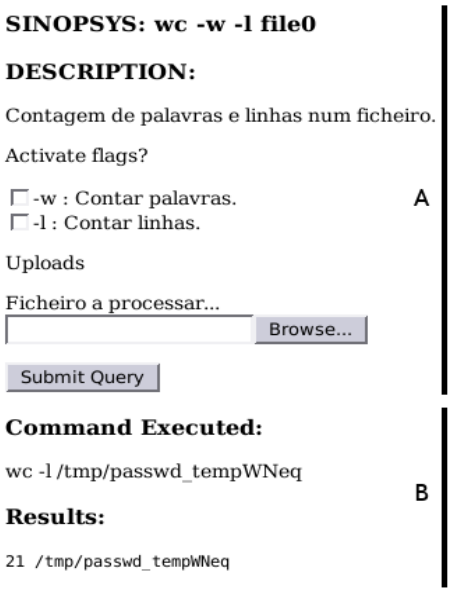

Fig. 1. Word-Count (wc) with a web interface.

This interface is not the best for all kind of operations, but is fully configurable by the `CGI::Auto` specification.

2.2 Example 2 — Pattern Search

The second example includes a third type of parameter: textual expressions to be replaced directly in the command line pattern.

To demonstrate this kind of parameter we will build an interface to the `grep` unix tool. Also, to explain how we can supply a pipeline of tools, we will tail the

result of the grep, showing just the final n results (being n a value specified by the user).

```
my %grep= (
  command => "grep [%regexp%] [%file%] | tail -n [%nl%]",
  description => "Pattern Search.",
  args => {
    regexp => {type => "textfield",name => "Pattern to Search"},
    nl     => {type => "textfield",name => "Maximum number of results"},
    file   => {type => "upload"   ,name => "File to process.."}
  }
);
```

The specification is straightforward. The command is just another command line pattern. It is written as if it was executed directly on your shell. Regarding arguments, we have the third type: text fields. These arguments are fields where the user can write anything, and their content will be replaced in the command line pattern. Figure 2 shows the Web interface generated for this description.

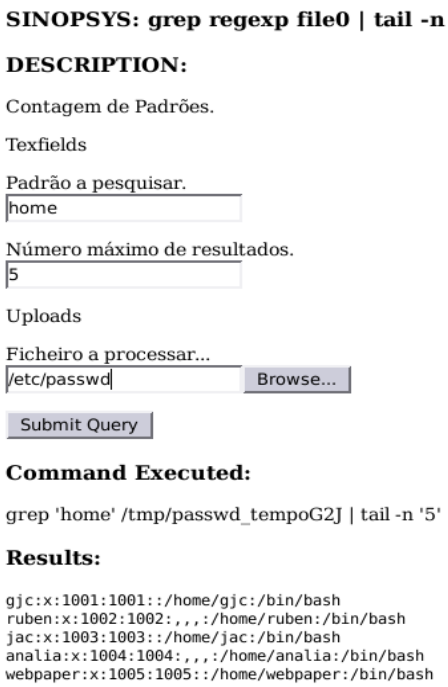

Fig. 2. Pattern Search (grep) with a web interface.

3 Conclusion and Future work

While this work is still in progress, it shown that the approach is viable. It is quite easy to specify any command line behavior using a command line pattern and a set of arguments description.

There are a lot of new features that are being currently implemented. They include more flexibility in the user interface (for instance, to generate mime-type rich documents, like PDF), incremental pipeline execution, and batch jobs. This last feature is very important as some tools will make the web-server time-out before a result is shown. For these kind of tools we are developing a deferred approach. The user will fill in the form and supply an email. Later, when the job is finished, the user will receive an email with the job results (or for some jobs with big results, with a specific URL where the result can be downloaded).

At the moment we know that the interpolation of user input directly in the command line patter is a big source of exploits. At the moment we want to prepare the basic module, and later include a security layer to filter and validate user input.

Currently, the integration with WebServices is not yet a reality. While its implementation is not complicated, it requires the development of a WebService client. Without this client we can not validate it. With web applications this kind of validation is quite easier.

References

1. Carlos Jorge Lopes and José Carlos Ramalho. *Web Services — Aplicações Distribuídas sobre Protocolos Internet.* FCA, 2005.
2. Eric Steven Raymond. *The Art Of Unix Programming.* Addison-Wesley, 2003.
3. L.D. Stein. *Official guide to programming with CGI. pm.* Wiley New York, 1998.

XML-based Extraction of Terminological Information from Corpora

Ana Belén Crespo Bastos, Xosé María Gómez Clemente,
Xavier Gómez Guinovart, and Susana López Fernández

Grupo TALG
Tecnoloxías e Aplicacións da Lingua Galega
Universidade de Vigo
{acrespo,xgomez,xgg,susanalopez}@uvigo.es
http://sli.uvigo.es

Abstract. In this paper, we present a methodology for the extraction of terminological information from textual corpora, showing the processes we follow for identification of term candidates in corpora, and for recognition in textual data of term definitions and conceptual relations. Both the textual corpora that are used as the source for terminological information, as well as the terminological database we build from this information, are stored and maintained by linguists in XML format, and converted to MySQL format for consultation through a PHP-based web application.

Key words: Natural language processing, textual corpora, terminological databases, ontologies, information extraction

1 Introduction

In this paper[1] we present a methodology developed at the University of Vigo by the TALG Research Group ("Galician Language Technology and Applications") for the extraction of terminological information from textual corpora leading to the creation of linguistic resources in the field of terminology for Galician. We will explain the main characteristics of the CLUVI corpus and the CTG corpus which constitute the source of this work, the process followed for the preparation of the Terminological Databank of the University of Vigo (TUVI), as well as the results obtained so far and the tasks we are undertaking and the ones we have in prospect.

The Linguistic Corpus of the University of Vigo (CLUVI) is an open collection of textual corpora with translations in specific areas of the contemporary

[1] This work has been funded by the Ministerio de Educación y Ciencia and the Fondo Europeo de Desenvolvemento Rexional (FEDER) within the project "Diseño e implementación de un servidor de recursos integrados para el desarrollo de tecnologías de la lengua gallega (RILG)" (HUM2006-11125-C02-01/FILO), a coordinated project between the University of Vigo (TALG Research Group) and the University of Santiago de Compostela (Instituto da Lingua Galega).

Galician language, accessible in the web since September 2003 at `http://sli.uvigo.es/CLUVI`. With a current total length exceeding the 20 million words, the CLUVI comprises six main parallel corpora belonging to four specialized registers (from fiction, computing, popular science and legal-administrative fields) and five different language combinations with Galician (Galician-Spanish bilingual translation, English-Galician bilingual translation, French-Galician bilingual translation, English-Galician-French-Spanish tetralingual translation and Spanish-Galician-Catalan-Basque tetralingual translation) [4]. The format chosen for storing the aligned parallel texts is an adaptation of the TMX format (Translation Memory eXchange), as this is the XML encoding standard for translation memories and parallel corpora, regardless of the application used [9]. A translation memory is a database which holds the original and translated version for each of the sentences translated in the framework of a computer-aided translation system, with the aim to reuse translations by the program. With some differences, an aligned parallel corpus is equivalent to a translation memory and, in practice, there is a considerable number of TMX-encoded aligned parallel corpora, with the added advantage that these corpora can be used as translation memories for feeding computer-aided translation programs [10].

The Galician Technical Corpus (CTG), available since 2006 for free consultation at `http://sli.uvigo.es/CTG`, is an open monolingual corpus of contemporary specialized Galician, in the fields of law, computing, economics, environmental science, sociology and medicine, with a current extension of 12 million words. The CTG is stored in the XML format, annotated with bibliographic and thematic information, and segmented into sentences. The web application developed in PHP for the searching and browsing of the CTG permits to query words or groups of words, use wildcards looking for complex patterns (regular expressions), and specify the subset of the corpus to which you want to limit the search. At present, the CTG is being annotated with information about the lemma and part-of-speech of words.

The Terminological Databank of the University of Vigo (TUVI) is a terminological database based on the monolingual and parallel specialty texts collected in the corpora of the University of Vigo, namely in the Linguistic Corpus of the University of Vigo (CLUVI) and in the Galician Technical Corpus (CTG). This terminological database is freely accessible on the web at `http://sli.uvigo.es/TUVI`, and currently has 5,625 terms documented in the CLUVI and CTG corpora belonging to the areas of law (1,411 bilingual and monolingual entries), sociology (954 tetralingual and monolingual entries), economy (1,163 monolingual entries) and ecology (1,324 monolingual entries). All terms in the TUVI are documented in the corpora, the terminological inventories in the fields of computer science and medicine being in progress.

In the TUVI, terminological information is structured around concepts. Each TUVI record includes all the information relating to a concept expressed with a Galician term which can be recorded also with variants, both intralinguistic (synonymic terms, spelling variants, or dialectal variants) and interlinguistic (translations or, more properly, equivalences). The information collected for each

variant (including the common or unmarked variant) includes the lemma of the term, its grammatical category, its definition, and a context of usage documented in the corpus. Registers or concepts in the database are grouped according to their thematic area within a branch of a hierarchical thematic tree of the matter. Also, the concepts in the database form a navigateable lexical-semantic network where conceptual nodes interact with each other according to the semantic relations (antonymy, hiperonymy, holonymy, etc.) among them.

Both the textual corpora that are used as the source for terminological information, as well as the terminological database we build from this information, are stored and maintained by linguists in XML format, and converted to MySQL format for consultation through a PHP-based web application, which allows for significantly expedite processing. XML to MySQL conversion is done in two ways, depending on the original XML document. For these textual corpora, which have a relatively simple structure, we created an ad hoc database with two related tables, one for the source texts and another for the sentences, and the data is imported from two delimited texts generated through XSL. For the termbase, which has a complex structure branching at various levels, we use Altova XMLSpy in order to export XML as delimited text. Altova XMLSpy converts XML input in ten interrelated tables which are imported from a MySQL database created from the output of XMLSpy converter. Terminological information in the TUVI database is structured according to the following DTD:

<!ELEMENT dic (cc+)> <!−−a dictionary is a set of concepts−−>
<!ELEMENT cc (ic, rs*, def*, ct+, lg+)>
<!ELEMENT ic (♯ PCDATA)> <!−−ic: concept index−−>
<!ELEMENT rs (♯ PCDATA)> <!−−rs: semantic relations−−>
<!ATTLIST rs <!−−tipo-rs: set of semantic relations−−>
tipo-rs (hipo | hiper | ant | mero | holo | eant | epost | tant | tsim | tpost
| axente | prod | caus | efec | instr | fin) ♯ REQUIRED >
<!ELEMENT def (texto_def, fonte_def?)> <!−−def: definition−−>
<!ATTLIST def xml:lang (gl | es | en | fr | pt) ♯ REQUIRED >
<!ELEMENT texto_def (♯ PCDATA)>
<!ELEMENT fonte_def (♯ PCDATA)>
<!ELEMENT ct (♯ PCDATA)> <!−−ct: thematic field−−>
<!ATTLIST ct st CDATA ♯ REQUIRED > <!−−st: standard for classification−−>
<!ELEMENT lg (var+)> <!−−lg: language-specific information−−>
<!ATTLIST lg
xml:lang (gl | es | en | fr | pt) ♯ REQUIRED >
<!ELEMENT var (lema, pms?, cat, ex*, frec+)>
<!ATTLIST var <!−−var: linguistic variant−−>
tipo (com | orto | morf | sigla | acro) ♯ REQUIRED
<!ELEMENT lema (♯ PCDATA)>
<!ELEMENT cat EMPTY> <!−−cat: grammatical category−−>
<!ATTLIST cat
valor (m | f | s | com | adx | lconx | lprep | ladx | lnom | lvb | ladv |

ladvlat | vt | vi | mpl | fpl | spl | compl) ♯ REQUIRED >
<!ELEMENT pms (♯ PCDATA)> <!−−morphosyntactic pattern−−>
<!ELEMENT ex (texto_ex, fonte_ex)> <!−−ex: term in context−−>
<!ELEMENT texto_ex (♯ PCDATA)>
<!ELEMENT fonte_ex (obra, num?)>
<!ELEMENT num (♯ PCDATA)>
<!ELEMENT obra (♯ PCDATA)>
<!ELEMENT frec (fab, vcorpus, palcorpus)> <!−−frec: term relative frequency in the corpus−−>
<!ELEMENT fab (♯ PCDATA)> <!−−fab: absolute frequency−−>
<!ELEMENT vcorpus (♯ PCDATA)> <!−−vcorpus: corpus version−−>
<!ELEMENT palcorpus (♯ PCDATA)> <!−−palcorpus: corpus size−−>

2 Extraction of term candidates

Next, we will explain our methodology for the extraction of term candidates from corpora, focusing on the AUGA corpus, a subset of CTG corpus devoted to language of ecology and environmental sciences, consisting of 2,349,362 words of journalistic, legislative, academic and informative texts. The texts in the AUGA corpus, as a whole, are about different themes on relations between human beings and nature, including the study of environmental problems and models for sustainable development.

2.1 Wordgrams-based extraction

First, we extract the most frequent words of the corpus, as well as the most frequent sequences of n-words (wordgrams), taking into account sequences until 4 words (bigrams, trigrams and tetragrams). Below we include some examples of terms identified in this way, with the frequency with which they occur in the corpus analyzed:

- *ambiental* (5,216 times)
- *ambiente* (3,807 times)
- *especies* (2,464 times)
- *contaminación* (1,658 times)
- *impacto ambiental* (903 times)
- *educación ambiental* (527 times)
- *augas residuais* (516 times)
- *avaliación ambiental* (508 times)
- *residuos perigosos* (499 times)
- *xestión de residuos* (455 times)
- *calidade do aire* (274 times)
- *plan de xestión* (223 times)
- *organismos modificados xeneticamente* (214 times)
- *autorización ambiental integrada* (213 times)

- *avaliación de impacto ambiental* (247 times)
- *gases de efecto invernadoiro* (220 times)
- *estudo de impacto ambiental* (166 times)
- *declaración de impacto ambiental* (160 times)

2.2 Extracting low frequency terms

The wordgrams-based terminology extraction is completed with the consultation of authoritative literature on the subject, which allows us, among other things, to identify key terms for the domain that have a low frequency in the corpus. In the case of environmental terms, the reference work for the Galician language was the Léxico do medio (`http://www.linmiter.net/lexique/_index.html`), elaborated by the União Latina (`http://www.unilat.org`) in the European project "Linmiter" (`http://www.linmiter.net`), a project created as a tool to support the terminology of minority Latin languages. Thus, we have completed our wordgrams-based inventory with the terms of the Léxico do medio which are equally documented in our corpus but which were not included in our initial list. These are new terms which, despite having a low presence and frequency in the body, are important terms in the field.

2.3 Extracting morphosyntactic patterns

As we said earlier, the CTG corpus is being tagged with information about lemma and part-of-speech of words. The tagset used in CTG is based on the tagset proposed by the Eagles group [5] for the annotation of morphosyntactic lexicons and corpora for all European languages. Here follows, by way of example, a fragment extracted from the CTG corpus in its PoS-tagged version and in its untagged version:[2]

> *<s>Galicia é a primeira Comunidade Autónoma pesqueira do Estado español, o sector pesqueiro representa o 8% do PIB e o 5% da poboación activa, estas cifras a pesar de estar en consonancia coa importancia do litoral a nivel mundial, o 40% da poboación do mundo vive nas zonas costeiras, presenta unhas cifras moi por enriba de calquera dos outros países comunitarios.</s>*

> *<s><t w="Galicia" c="NP00000">Galicia</t> <t w="ser" c="VIP3-S00">é</t> <t w="o" c= "AFS">a</t> <t w="primeiro" c="NO0-FS">primeira</t> <t w="Comunidade" c="NCFS000">Comunida-de</t> <t w="Autónomo" c="A0FS0">Autónoma</t> <t w="pes-queira" c="A0FS0">pesqueira</t> <t w="de" c ="SPS00">do</t> <t w="o" c="AMS">∼</t> <t w ="Estado" c="NCMS000">Esta-do</t> <t w="español" c="A0MS0">español</t> [...] </s>*

[2] Textual segment included in the CTG corpus belonging to the doctoral dissertation of Alfredo López Fernández, *Estatus dos pequenos cetáceos da plataforma de Galicia* (University of Santiago de Compostela, 2003), directed by Angel Guerra Sierra and Graham J. Pierce.

From this tagged corpus we can retrieve new term candidates based on the morphosyntactic patterns most frequently found in the terminological database built until that moment. Thus, we first determine which are the most common tag combinations in the terminological inventory done so far, and afterwards we observe the sequences of tokens in the corpus that correspond to these patterns. For example, this is the list of morphosyntactic patterns most frequently found in the inventory of 1,444 terms extracted from the AUGA corpus on environmental sciences (we indicate the number of times the term occurs in the list with the specific morphosyntactic pattern):

- Singular feminine noun (216 times)
- Singular masculine noun (209 times)
- Singular feminine noun + singular feminine adjective (157 times)
- Singular masculine noun + singular masculine adjective (145 times)
- Singular feminine noun + singular common adjective (98 times)
- Singular masculine noun + singular common adjective (97 times)
- Singular masculine noun + preposition + singular feminine noun (66 times)
- Singular masculine adjective (52 times)
- Singular feminine noun + preposition + singular feminine noun (41 times)
- Singular common adjective (34 times)
- Singular masculine noun + preposition + singular masculine noun (33 times)
- Verb (21 times)

Comparing the results of the most common morphosyntactic patterns for environmental terms with the patterns identified in the list of 1,768 legal terms extracted from GALEX subcorpus of legal texts (2,516,846 words from CTG), it can be seen that the frequency of patterns is similar. Thus, the most popular patterns are, on the one hand, nouns and noun+adjective combinations and, on the other hand, noun+preposition+noun combinations.

We apply these results to the search for term candidates in the tagged section of the CTG corpus, grouping subcategories such as gender and number of nouns, adjectives and articles, and discarding monocategory patterns. Thus, we obtain a comprehensive list of term candidates, with their frequency in the corpus, from which we extract the valid terms after a thorough review. The resulting list of candidates contains combinations from 2 to 6 elements, for example:

- noun + adjective: *diversidade biolóxica, catástrofe ecolóxica, auditoría ambiental, política forestal, cambio climático, augas residuais*
- noun + preposition + noun: *planta de tratamento, capa de ozono, dióxido de carbono*
- noun + preposition + article + noun: *calidade da auga*
- noun + preposition + article + noun + adjective: *avaliación do impacto ambiental*
- noun + adjective + adjective: *recursos naturais renovables, autorización ambiental integrada*
- noun + preposition + noun + conjunction + noun: *centro de recollida e descontaminación*

– noun + preposition + article + noun + preposition + noun: *saturación do proceso de cambio*

We are currently working on the process of filtering the data automatically extracted from the corpus to obtain term candidates based on morphosyntactic patterns. For the moment, in this process of filtering we have applied two complementary approaches: human testing of data, and verification of the terms in a corpus other than the corpus which is used as source of the data.

Of course, the most reliable approach is the human revision of data by Galician terminologists and specialists in the field. However, due to limited resources and the difficulty of finding experts willing to collaborate, we have to opt for other less efficient methods.

A variant of the second approach is to verify the presence of the term candidates in other subcorpora of CTG different from the subcorpus used as source. For example, during the identification of environmental terms extracted from the AUGA corpus, if a term candidate does not appear or has a very limited presence in other subcorpora from other thematic areas, we believe it has a good chance of being a significant and necessary word in the environmental sciences. In the opposite case, the term candidate increases the likelihood of not being a specific term of its field, because it would also have visibility in other fields like law, sociology, medicine, economics, etc.

Another different variant of the second approach is to search the term candidates in the internet. If its presence rate is not very high, it will be an index of the specificity of the term and, therefore, an index of its terminological relevance.

3 Extraction of semantic relations and definitions

The work on corpora being done by our research group allows us, as we said, both to extract lexical units with specialized value (terms) and their terminographic treatment which focuses on the description of the name and concept. With respect to that issue, the work undertaken so far has focused on two aspects: the identification of conceptual relations, and the linguistic expression of concepts (definitions). These two aspects are the subject of much attention by applied and theoretical terminology, because they point very clearly to the true role of scientific terms in texts: transmission of knowledge.

The TUVI terminological database was created with an onomasiological approach where concepts are “the door” to enter the term. This approach leads directly to focus our interest in the description of the conceptual relations and definitions, without forgetting the necessary adscription of terms to a specific branch of the conceptual tree.

Apart from the traditional and necessary identification of semantic relations and the development of definitions (through the use of thesaurus and ontologies, the systematic search in specialized dictionaries, and the consultation with specialists), we began the work of the automation process, focused on the search for linguistic and typographical patterns that can be discovered in corpora, both

for semantic relations [3] and definitions [6, 1]. We understand that the semantic relations can be identified by textual segments that function as real anchors, which are also segments that lead to retrieve textual information relevant to the semantic explicitation of a term, and that when an author of a text defines a term, she or he does so through definitory contexts, considering as definitory context any textual fragment from a document which provides specialized information useful to define a term [2]. All of this information can be automatically retrieved from a PoS-tagged corpus (in a faster and clearer way) or from an untagged corpus. The following data was extracted from the untagged version of the AUGA corpus on ecology and environmental sciences.

3.1 Methodology for semantic information extraction

For the identification of semantic relations, we draw on [3], because in her paper she describes the general framework of conceptual relations that we use in our analysis and she also presents textual markers that identify them in a Catalan corpus (textual markers adapted and supplemented by us for Galician). For definitory contexts we draw on the work done in the Corpógrafo [8, 7], Alarcón [1] and especially in the classic work of Pearson [6], which explains that when an author, in a given text, wants to define a term, she or he may resort to typographical elements to highlight this term, and to the definition and definitory patterns to relate the term to its definition. In our research we also believe it is interesting to take advantage of any relevant information which, even without being a definition, can be related to semantic aspects of the term.

3.2 Patterns for semantic relations

In a pattern [X p Y] for the automatic extraction of semantic relations, both X and Y are (inflected) terms well defined within a specific domain (and documented in our terminological database), and "p" is a linguistic pattern that can be formed by verbs, verbal phrases, connectors and typographical elements. Currently, the search pattern is based on [X p] to seek any textual segment including Y. Here are a sample of the the linguistics patterns "p" we use to search for semantic relations in the corpus expressed as regular expressions:

- Resemblance:
  - Partial resemblance: *(é parecid[oa]* a | *son parecid[oa]s* a *)*
  - Antonymy: *((é|son) o contrario de* | *é contrario* a | *se opón* a | *oponse* a | *se opoñen* a | *opóñense* a | *distínguen?se de* | *diferéncian?se de* | *se distinguen? de* | *se diferencian? de)*
- Inclusion:
  - Hyponym-hyperonym: *((é|son) un tipo de* | *é (un|unha)* | *considéran?se* | *se consideran?)*
  - Hyperonym-hyponym: *(agrupa* a | *como:* | *tal como:?* | *como [oa]s?)*
- Sequentiality:

  - General space: *(aparecen? en | ocorren? en | realízan?se en | se realizan? en | están? situad[oa]s? en | orixínan?se en | se orixinan? en | (ten|teñen) lugar | localízan?se en | se localizan? en | d[áa]n?se en | se d[áa]n? en | atópan?se en | se atopan? en | (están? |)presentes? en)*
  - Front/posterior space: *(están? (situad[oa]s? |)(antes de|despois de|detrás de|diante de))*
  - Previous/simultaneous/posterior time: *(seguen? | iníciase en | prodúcese antes de | prodúcese despois de | prodúcese durante | se inicia en | se produce antes de | se produce despois de | se produce durante| ten lugar antes de)*
- Instrumentality: *(serven? (de|para|como) | úsan?se (de|en|para|como) | se usan? (de|en|para|como) | emprégan?se (de|en|para|como) | se empregan? (de|en|para|como) | utilízan?se (en|para|como) se utilizan? (en|para|como) | son empregad[oa]s (en|para|como) | son utilizad[oa]s (en|para|como) | é empregad[oa] (en|para|como) | é utilizad[oa] (en|para|como) | realízan?se (con|mediante|por medio de))*
- Causality: *(orixinan? | causan? | é causa de | son causa| é a causa de | son a causa de | provocan? | contribúen? a | d[aá]n? lugar a | implican? | producen? | son provocad[oa]s por | é provocad[oa] por)*
- Meronymy: *(están? compost[oa]s? (por|de) | constan? de | están? constituíd[oa]s? por | abranguen? | engloban? | están? formad[oa]s? por)*

We can see some examples of these semantic relations identified in the AUGA corpus:

- Resemblance:
  - *Quercus suber* [...] O seu aspecto é parecido ó da *aciñeira*, aínda que se diferencia dela pola súa grosa e esponxosa casca de máis de 15 cm de grosor, chamada cortiza, e polas súas follas menos espiñentas cá desta.
- Sequentiality:
  - Ata a actualidade a *avaliación da calidade do aire* realízase puntualmente nos *lugares de medición*, sen que exista un coñecemento preciso da representatividade territorial das medicións obtidas.
- Causality:
  - Nos solos non agrícolas, a *acidificación* dá lugar á *perda de vitalidade* das plantas producindo a perda e deterioración de follas e en último caso a morte das especies vexetais acompañada de cambios nos organismos do solo, ao favorecer a proliferación de especies acidófilas.
- Meronymy:
  - O *biogás* está constituído na súa meirande parte por *dióxido de carbono* e *metano*, ademais tamén posúe pequenas cantidades de hidróxeno e sulfuro de hidróxeno.

3.3 Patterns for definitions

In a pattern [X = Y] for the extraction of term definitions, "X" is a term from the database, "=" is a definitory pattern based on verbs, linguistic or metalinguistic

phrases (including reformulative markers) and typographical elements, and Y is the definition or the relevant syntactic elements that can lead to the creation of a definition. With regard to Y, it must be clear that it can also be a term that is the superordinate in the sort of classic definition based on the gender and the difference [X = Y [specific semantic features]]. Currently, the search pattern is based on [X =] to seek any textual segment including Y. Here are a sample of the patterns "p" we use to search for term definitions in the corpus expressed as regular expressions:

- Verbs: *(é | son | (concíbe|enténde|considéra)n?se | se (concibe|entende|considera)n? | poden? (concibir|entender|considerar)se | se poden? (concibir|entender|considerar) | póden?se (concibir|entender|considerar) | poden? ser (concibid|entendid|considerad)[oa]s?)*
- Reformulative markers: *(, isto é|, é dicir)*
- Linguistic expressions: *(,? como:? |,? tal como)*
- Typographical elements: *(:)* // colon sign

Here follow some examples of definitory contexts identified in the AUGA corpus:

- *A acuicultura* defínese como *o conxunto de actividades encamiñadas ao cultivo de especies acuáticas.*
- *Aire ambiente*: *o aire troposférico e exterior.*
- No Regulamento (CE) nº 2792/1999 do Consello, de 17 de decembro de 1999, polo que se definen as modalidades e condicións das intervencións coa finalidade estrutural no sector da pesca, queda recollida a definición de *acuicultura* como: *a cría ou cultivo de organismos acuáticos con técnicas encamiñadas a aumentar, por encima das capacidades naturais do medio, a produción dos organismos en cuestión; estes serán, ao longo de toda a fase de cría ou de cultivo e ata o momento da súa recollida, propiedade dunha persoa física ou xurídica.*
- *O dióxido de carbono* é *o principal responsable da contribución humana ao efecto invernadoiro a través do uso de combustibles fósiles.* [semantic features]
- *A agricultura ecolóxica* é *unha manifestación da recente preocupación da poboación polo medio natural e o consumo de produtos saudables.* [semantic features]
- *A avaliación de impacto ambiental* é *un proceso de análise mediante o que se integra o medio ambiente e o proxecto deseñado, oferecendo unha serie de vantaxes a ambos aínda que en moitas ocasións estas só son evidentes a longo prazo e que poden permitir aforros nos investimentos e os custos das obras, deseños mais aperfeizoados e integrados no entorno e maior aceitación social dos mesmos.*

3.4 Results and further work

Below we include some results on the accuracy of the patterns used in the extraction of definitions, where "DCIP" stands for "number of different definitory con-

texts identified by the pattern"; "precise" means "number of definitory contexts identified by the pattern that presents a definition"; "relevant" means "number of definitory contexts identified by the pattern that presents an information relevant to a possible definition"; and "irrelevant", "error in the identification of the definitory context".

term	**DCIP**	**precise**	**relevant**	**irrelevant**
acuicultura	6	3	3	0
aeroxerador	12	0	1	11
agricultura ecolóxica	4	0	4	0
aire	15	0	5	10
aire ambiente	4	2	0	2
aleta dorsal	4	0	4	0
ambiente	13	0	7	6
amianto	1	0	1	0
amoniaco	6	0	6	0
amonio	2	0	1	1
avaliación de impacto ambiental	6	1	1	4
dióxido de carbono	5	0	3	2
emisión	7	3	1	3
medio acuático	1	0	1	0
medio natural	8	0	5	3

In short, in 94 DCIP, there are 9 (9.5%) precise definitions, 43 (45.7%) contexts with relevant information, and 42 (44.6%) contexts with irrelevant information. After reviewing the data, we can conclude that:

a) the typographical pattern (:) produces an especially large amount of noise due to the peculiarities of use of this punctuation mark. However, it also identifies precise specific information.

b) The number of precise definitions is low (9.5%, which leads us to redefine and complete the patterns used in extraction.

c) Nevertheless, the relevant information (semantic features that contribute to the understanding of the concept and allow the drafting of definitions) found with the patterns used is high (45.7%), which shows that the automatic extraction is fully justified.

d) As a whole, the percentage of the retrieved semantic information is higher than the noise.

From now on, we will work on a very important aspect in the identification of semantic relations and definitions: the elimination of the "noise" that occurs when linguistic patterns are applied to a corpus. In this regard,

a) we must develop rules for exceptions that can rely on the introduction of linguistic elements that deny the validity of a pattern. The presence of adverbs like "non", "nunca" or phrases like "en ningún caso", will serve to adjust the searches.

b) For the extraction of semantic relations, we must create a specific subcorpus in which the two terms necessary for the establishment of the semantic

relations (X and Y) are clearly identified. Currently, the search pattern is based on X to seek any textual segment (including Y). If we limit the contexts to those where X and Y occur, the noise level will necessarily lessen.

4 Conclusions

The CLUVI corpus and the CTG corpus allow retrieval of linguistic information that facilitates studies on pragmatic aspects of the Galician language. With regard to the terminological treatment of units drawn from the CTG, it must be said that the TUVI terminological database allows a full approximation to the term (usage, denominative information and conceptual information). Automated extraction greatly facilitates identification of term candidates, of conceptual relations and of definitions in large amounts of text, and helps the eventual manual extraction. For the moment, the patterns that retrieve conceptual information (as seen for definitions) reflect a fairly high level of irrelevant information, but they are still very useful to describe the concept.

References

1. Alarcón, R.: Extracción automática de contextos definitorios en corpus anotados. Seminaris de l'IUULATERM, Universitat Pompeu Fabra, Barcelona (2006). <http://www.iula.upf.edu/materials/060526alarcon.pdf>
2. Alarcón, R., Sierra, G.: Reglas léxico-metalingüísticas para la extracción automática de contextos definitorios. Encuentro Nacional de Computación (ENC 2006), San Luis Potosí, México (2006). <http://ccc.inaoep.mx/~tec_lenguaje06/articulos/TLH06-paper3.pdf>
3. Feliu, J.: Relacions conceptuals i terminologia: anàlisi i proposta de detecció semiautomàtica. Ph.D. thesis, Universitat Pompeu Fabra, Barcelona (2004)
4. Gómez Guinovart, X., Sacau Fontenla, E.: Parallel corpora for the Galician language: building and processing of the CLUVI (Linguistic Corpus of the University of Vigo. In: Lino, T. et al. (eds.), Proceedings of the 4th International Conference on Language Resources and Evaluation, LREC 2004, pp. 1179-1182. Lisboa (2004)
5. Leech, G., Wilson. A.: Recommendations for the Morphosyntactic Annotation of Corpora. EAGLES Guidelines (1996). <http://www.ilc.cnr.it/EAGLES96/annotate/annotate.html>
6. Pearson, J.: Terms in Context. John Benjamins, Amsterdam (1998)
7. Pinto, A.S.: Neurodemo: um exemplo de extracção semi-automática de definições e relações semânticas usando o Corpógrafo, Linguateca (2006). <http://poloclup.linguateca.pt/Neurodemo.htm>
8. Pinto, A.S., Oliveira, D.: Extracção de definições no Corpógrafo, Linguateca (2004). <http://www.linguateca.pt/documentos/OliveiraPintoOut2004.pdf>
9. Savourel, Y. (ed.): TMX 1.4b Specification. Localisation Industry Standards Association (2005). <http://www.lisa.org/standards/tmx/specification.html>
10. Simões, A., Dias de Almeida, J.J., Gómez Guinovart, X.: Memórias de Tradução Distribuídas. In: Ramalho, J.C., Simões, A. (eds.), XATA2004 - XML, Aplicações e Tecnologias Associadas, pp. 59-68. Universidade do Porto, Porto (2004).

A Toolkit for an Oral History Digital Archive *

Silvestre Lacerda[1]
lacerda@iantt.pt
Norberto Lopes[2], Nelma Moreira[2], Rogério Reis[2]
{nml,nam,rvr}@ncc.up.pt

[1] Direção-Geral de Arquivos/ Arquivo Nacional da Torre do Tombo
[2] DCC-FC& LIACC, Universidade do Porto

Abstract. In this work we propose an XML based toolkit for the construction of an oral history digital archive that allows the filing, classification and annotation of multimedia resources associated to a corpus of interviews. We describe the general organization of the archive and focus on content creation tools. In particular, we present a document editor for the classification and annotation of interview transcriptions that allows the definition of category hierarchies.

Keywords: XML languages, XML editors, oral history archives, multimodal resources, *metadata*, corpus

1 Introduction

In this work we propose an XML based toolkit for the construction of an oral history digital archive that allows the filing, classification and annotation of text and multimedia resources associated to a corpus of interviews.

The two basic objects used in this archive are interviews and persons. Each interview (to a person) can have several audio and video files, photos and other images attached, and several derived text documents that result from different analysis of the interview's data.

To ease the access, classification and search of the information, all text based documents should be in a semi-structured (XML) format and all multimedia documents should have *metadata* information associated. To help the construction of these documents, in particular by non computer specialists, dedicated software applications must be built. We present a document editor for the classification and annotation of interview transcriptions that allows the definition of category hierarchies.

This paper is organized as follows. In the next section we present the motivation and the scope of this work. Section 3 describes the general organization for the interview's archive. Section 4 presents a document editor for classification and annotation. Some related work is discussed in Section 5 and Section 6 concludes with some ongoing and future work.

* Work partially funded by FCT through the grant MTCIO (POCTI/CED/60786/2004) and POSI Program.

2 Motivation

This work is part of the research work developed by the *Documentation and Information Center on Working Class and Popular Movement of Porto* (**CDI**) of Universidade Popular do Porto [16] . **CDI**'s goals are the preservation of the social, cultural and political memory, and oral and social history of Porto, during the 20th century, as well as its diffusion. Available since 2001, **CDI**'s Web site contains information about several workers' organizations, including documental inventories, archival descriptions and digitalized documents; abstracts of workers' interviews and a chronology of workers' related events during the 20th century. Search in the site can be text based or using a special controlled vocabulary.

In what the oral history is concerning, **CDI** has already collected a corpus of about one hundred interviews with workers of different professions and with different social experiences. All biographical narratives were recorded in audio and video, and a transcription of the interview was produced. A small abstract of each narrative is already available in **CDI**'s Web site.

The quantity, diversity and quality of the collected information by the **CDI** inspires its study in a multidisciplinary approach. The research team of this project involves different social sciences domains (Linguistics, Education Sciences, History, Information Sciences), and Computer Science.

To ease the access, the search and the multi-disciplinary analysis of the biographical narratives, new software applications are being developed. In Silvestre *et. al* [9] we presented the main aims for an oral history archive and described some tools already implemented. We briefly review some of those tools in the next section. We would like to emphasize our commitment in using and developing *free software* tools and documents with open specifications, as the only way to ensure timeless accessibility, portability and easy updating. The choice of `XML` for the format of text documents illustrates well our options.

3 A Digital Interviews Archive

As we pointed out in the introduction, the two basic objects used in an oral history archive are interviews and persons. Each interview can have several associated resources. Here we will consider the following:

- audio and video recordings
- photos, digitalized documents, images
- text documents:
  - transcriptions
  - abstracts
  - classifications associated with structural content analysis
  - other documents

All documents must have *metadata* information associated. For the analysis and research the documents should have *annotations* that associate a concept to a

text segment. A text segment can have multiple annotations. For a more efficient search, annotations should use controlled vocabularies, and specially *thesauri*. The *thesauri* will allow a more fine-grained classification of documents that can be of value for social scientists research.

In Silvestre *et. al* we presented a set of `XML` based software tools for the production, annotation and search of multimedia documents. In particular we specified `XML` languages for : *metadata* based on the *Dublin Core* standard [7]; *thesauri* based on the `Zthes` standard[19]; annotations based on a specific controlled vocabulary; transcriptions of interviews based on the Transcriber schema [12, 1, 2]; and, for small abstracts of the interviews contents organized in sections, called *stories*. We developed special purpose document editors for *metadata* and abstracts. Both for the *thesauri* and the abstracts an Web site was implemented that allows browsing and search. The `XML` schema used to define each language is `Relax NG` [17]. Besides its expressive power it has a very elegant (compact) syntax and a well-defined semantics based on regular tree languages that allows efficient and clear implementations.

The annotations language was easily extended to deal with other vocabularies. In the next section we will describe an extension of the abstract's editor to a new editor that allow general classification of transcriptions by the definition of arbitrary category hierarchies.

3.1 Organizing the Resources Associated with an Interview

To organize the resources associated with an interview we used an approach similar to the *IMDI* standard developed by ISLE (International Standard for Language Engineering) Meta-data Initiative [6, 3] for linguistic multimedia resources.

In the *IMDI* standard the main concept is *session* which *bundles all information about the circumstances and conditions of a linguistic event.* Although some of this information can be retrieved from the resources' *metadata*, other should be duplicated in each resource in order to be possible to group and collect it. For example, every document should have information about which interview it refers to and that would not be easy to infer without the notion of *interview.* We consider the main *metadata* elements proposed by *IMDI* and we specified the `session` `XML` element which the `Relax NG` schema is in Figure 1.

Note that we restricted the content model of many elements, as many linguistic specifications were not meaningful to our goals. Almost all elements are self-explanatory and similar to the ones defined in the *Dublin Core* standard. In the `content` element, the attribute `genre` describes the discourse type of the session contents. In our *corpus* the genre will be in general `interview`. The attribute `task` can be used to describe the topic of the session (for instance, *biographical narrative*).

The most relevant element for us is `person`. In Figure 2, the `Relax NG` schema for that element is given.

```
start = session
session = element session {
      attribute name {text},
      attribute title {text}?,
      attribute description {text}?,
      date+, place, project+, content,
      contributor*, resource*
}
place = element place {text}
project = element project {
   attribute name {text},
   attribute title {text}?,
   attribute description {text}?
   attribute person-id {text}?,
   contact?,
}
content = element content {
   attribute genre {text},
   attribute task {text}?,
   attribute description {text}?,
   subject*, language*
}
subject = element subject {text}
language = element language {text}
contributor = element contributor {
  attribute person-id  {text},
  attribute resource {text},
  attribute role {text}
}
resource = element resource {
         attribute type {text},
         attribute link {text},
         attribute access {"private" | "owner" | "public"}
}
```

Fig. 1. `Relax NG` schema for interviews.

Note again that, in contrast with the *IMDI* schema, the element `contributor` in a *session* has an attribute that refers to *person* and is not itself a `person` element. The value of the `role` attribute of the `contributor` element belongs to a controlled vocabulary that includes names as *interviewer*, *interviewee*, *transcriptor*, etc..

3.2 Building an Archive

An interviews' archive is a set of sessions, a set of persons and a set of associated resources. The organization of an archive can be specified as a tree of directories in a file system. The basic structure is presented in Figure 3.

```
start = person
person = element person {
   attribute id {text},
   attribute name {text},
   attribute fullname {text}?,
   attribute gender { "male" | "female" }?,
   birth? & situation? & language? & contact?,
   (photo | place | education | activity | organization | observation)*
   }
photo = element photo {text}
birth = element birth {date? & place?}
place = element place {text}
language = element language {text}
education = element education {text}
activity = element activity {
      attribute type {text}
      text }
situation = element situation {attribute type {text}},
   element session {attribute id {text}}*}
institution = element institution {text}
observation = element observation {text}
}
```

Fig. 2. `Relax NG` schema for persons.

The file `persons.xml` in the *directory* `_persondb` contains information about which persons are in the archive, each one described using a `person` element. In the same way, the file `sessions.xml` contains information about each session in the archive. These files will be manageable even if their size will reach some megabytes. This conclusion is based of some performance tests with more than 50000 records (sessions or persons). In this way we can profit from `XPath` facilities for querying and presenting the archive information.

There must exist a central archive. Each user can have a local archive which imports/exports information from and to the central archive. When a user exports a new version of an existing document a new revision is added, and the original document will not be modified.

4 Interviews Classification

For the data analysis of the biographical narratives corpus, the text annotations provide a useful indexation of the data, but is not enough for a structural content analysis. Some topics or categories can be defined globally driven by the research goals, but each narrative motivates the introduction of new categories. Those categories are then hierarchically organized into trees and several text segments can be associated to each category.

```
_archive/
      /_persondb/
                persons.xml
                /_photo
                  photo1.png
                  photo2.png
                    .
                    .
                    .
      /_sessiondb/
                 sessions.xml
                 E01/
                    trans1.xml
                    class1.xml
                    audio1.xml
                    video1.xml
                 E02/
                   .
                   .
                   .
```

Fig. 3. Archive basic structure.

This content analysis can produce one or several classifications of the original transcription. Each classification can be used for several transcriptions and each transcription can be associated with several classifications.

A classification is then a forest (set of trees) where each node is labelled by a category and may have several text segments associated to. Currently, we only accept the association of a transcript to a classification. Figure 4 presents the `Relax NG` schema of a classification. The element `node` corresponds to a category. The element `content` corresponds to a text segment of a transcription and can be identified by a `start char` and an `end char`. This text segment (contained in the `value` element) cannot be modified, although it can have independent annotations in the classification and in the transcription. The attribute `link` can be used for identifying the transcription that segment of text belongs to (in the case that multiple transcripts are allowed).

In Figure 5 we present the new `XML` specification for annotations and for vocabularies (that are not a *thesaurus*).

In future implementations, each `content` should also be marked with insertions, deletions or substitutions. These would be useful for a printed version of a classification where some minor modifications of the original transcript can be allowed (for instance, for omitting punctuation or a private part of the discourse).

```
include "metadata.rnc"
start = classification
classification = element classification {
   metadata,
   name,
   description?,
   nodes?
}
nodes = element nodes { node+ }
node = element node {
   ref?,
   name,
   description?,
   comment?,
   contents?,
   nodes?
}
contents = element contents { content+ }
content = element content {
   attribute startchar { text },
   attribute endchar { text },
   attribute  link  { text }?,
   ref?,
   description?,
   comment?,
   value
}
name = element name { text }
description = element description { text }
comment = element comment { text }
ref = attribute ref { text }
value = element value { text }
```

Fig. 4. The `Relax NG` schema for classifications.

4.1 The Classification Editor

The interviews' transcriptions are `XML` documents with a specification based on the Transcriber schema [12, 1, 2], but supporting *metadata* and annotations. Its `Relax NG` schema is presented in Figure 7. For now we do not have a transcriber application, but only some converters from several formats into our transcription format.

For the classification of transcriptions we developed an editor that allows the dynamic definition of categories trees and builts new documents based on the information in the transcriptions.

Figure 6 presents a screen-shot of the editor. The editor main frame is divided into two panels. In the left panel, an hierarchy of categories can be built. A new node can be created and attached anywhere. A node has a `name` (the category), a

```
annotations = element annotations {
   attribute vocabulary {text},
   attribute type {text},
   attribute marks? { text },
   annotation+
}
annotation = element annotation {
   attribute startchar { text },
   attribute endchar { text },
   attribute type { text },
   attribute normtext { text },
   text
}
```

```
vocabulary = element vocabulary {
      attribute name { text },
      attribute date { text },
      attribute link { text}
      description?,
      entry+
}
description= element description {
   attribute lang {text}
   text
}
entry= element entry {
   attribute value {text}
   text
}
```

Fig. 5. The `Relax NG` schemas for annotations and vocabularies, respectively.

`description` and a `comment`. There may exists several top nodes, so several trees are allowed. It is also possible to change a node's place or to edit its attributes.

The right panel, has two windows: the lower one for a transcription and the upper one, will have the contents associated with a category. In the transcription, the different speakers are identified by a number. The identity of each speaker can be checked by selecting an option in the `View` menu.

To associate a text segment to a category, choose the category and just select the text segment from the transcription. Then, the text segment will appear in the upper window, and a small description will be also attached below it's category (in the left side). In Figure 6 the highlighted text is a description of the content which is shown in the upper window on the right, and which category is *Primaria*.

When a category is selected all its contents are listed in the upper window on the right and each text segment is highlighted in the transcription (lower window on the right).

The options in the `Edit` menu allow the edition of *metadata* and of some preferences.

The options in the `File` menu allow to open/create an archive and choose a transcription (we plan to allow choosing more than one transcription simultaneously). It is possible then to create a new classification or open an existing one. A classification can be saved or exported to an `HTML` file. The category hierarchy can be saved independently (what we call a `schema`) in order to reuse it for other classification (and possibly for other transcriptions).

4.2 Implementation

The main programming language of this project is `Python` [18].

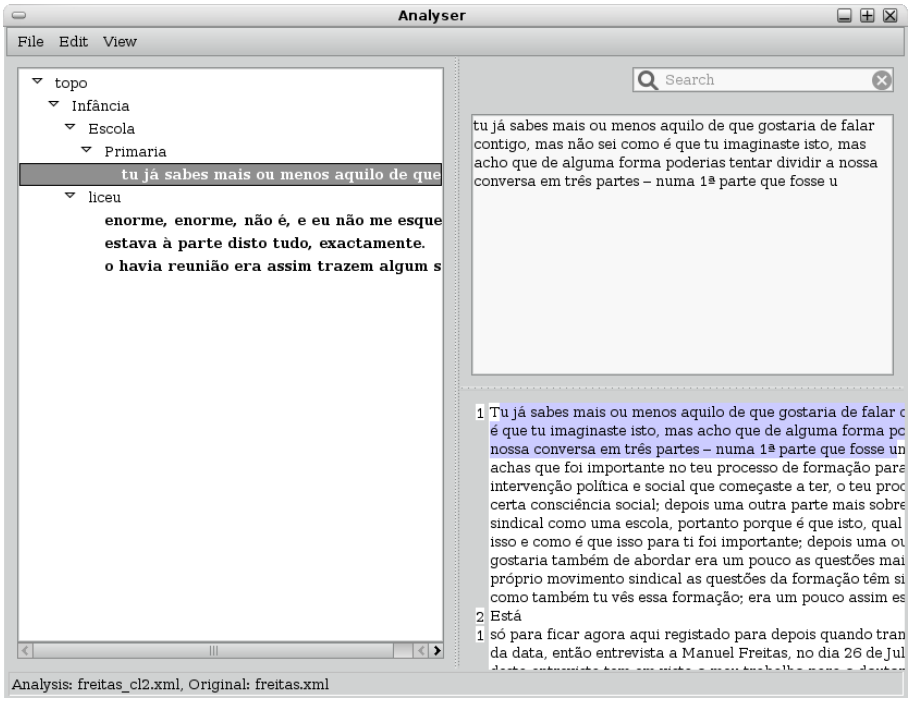

Fig. 6. The classification editor.

The graphical interfaces are implemented with the `wxWidgets API` [15]. Text editor objects are based in the `wxStyledTextCrl` class, that implements the editing component `Scintilla` [5]. The category tree graphical interface uses the `CustomTreeCtrl` class.

`XML` processing uses the `lxml` [11] library that implements the `elementtree API` [10], but with greater support for `XSLT` and `XPath`. This `API` is `Python` based and much faster and less space consuming than `DOM` or `SAX Python` implementations.

For the classification editor, the classification's tree structure is built using `Python` classes and it is saved as a document in the classification `XML` language.

5 Related Qualitative Data Analysis tools

There are several software tools available for the qualitative data analysis of a set of documents, that can be used for the structural content analysis of our corpus of interviews. From this set, we can refer the *Ethnograph* [14], the `Atlas.ti` project [4, 13], and the *Nvivo 7* [8]. All those tools are commercial proprietary software products with very expensive licenses. Only the *Atlas.ti* project can use `XML` files.

```
start = transcript
transcript = element transcript {metadata, actor+, section*, annotations*}
section = element section {
      attribute title { text },
      attribute desc { text },
      attribute starttime { text }?,
      attribute endtime { text }?,
      conversation+
   }
conversation = element conversation {
      attribute title { text },
      attribute actor-id { text },
      attribute starttime { text }?,
      attribute endtime { text }?,
      (comment | event | sync | text)+
   }
comment = element Comment {
   attribute desc {text}
}
sync = element sync {
   attribute time {text}
}
event = element event {
   attribute desc {text}
   attribute type {text}
   attribute extend {text}
}
```

Fig. 7. `Relax NG` schema for transcriptions.

The *Nvivo 7* is the most popular in the academic world. Its previous versions were not very user friendly and it was difficult to be used by a non computer specialist. *Nvivo 7* allows to create a project with several documents (in the RTF format). Arbitrary (free) nodes can be created and text segments associated to those nodes. The nodes can then be organized in a tree. Search facilities includes search for nodes or text, and proximity search of two items.

In comparison, our approach includes the basic features of this system (for now, for a single document) with the advantages of being free software and of using semi-structured documents with open formats.

6 Conclusion

In this paper we described ongoing work towards the construction of an oral history digital archive based on a corpus of interviews. We also described an editor that aims to help social scientists in the corpus structural content analysis. The current version of the editor works only with one interview (transcription)

at a time. This is not a major drawnback as in most of the cases each interview has its own set of category hierarchies. Although it is still in development, it is already being used by some project team members. We are now extending the editor to allow the classification of multiple interviews simultaneously and improving its search facilities. For that it is essential to have a precise notion of an *archive*. In this paper, we also described the basic structure for building an oral history archive. This organization is also essential for an improved search in the corpus. We plan to implement search mechanisms that will allow

- querying the archive using controlled vocabularies and in special *thesauri*, or more complex ontologies;
- more complex queries, by the development of a specific query language fitted to the archive contents and simple to use;
- save the query results in a file for future use.

In what dissemination is concerned we note that provision must be made in order to restrict the access to several documents of the archive although general sessions information and documents *metadata* should be in general accessible.

7 Acknowledgements

We thank several members of Universidade Popular do Porto and team members of the project *Memories of Work: Construction Processes of a Worker's Identity* for their comments and suggestions. Specially we would like to thank Cristina Nogueira and Teresa Medina for many discussions about the classification editor. Thanks are also due to the referees whose comments and suggestions allowed to improve a first version of this paper.

References

1. Claude Barras, Edouard Geoffrois, Zhibiao Wu, and Mark Liberman. Transcriber: a free tool for segmenting, labeling and transcribing speech. In *First International Conference on Language Resources and Evaluation (LREC)*, pages 1373–1376, 1998.
2. Claude Barras, Edouard Geoffrois, Zhibiao Wu, and Mark Liberman. Transcriber: development and use of a tool for assisting speech corpora production. *Speech Communication*, 33(1–2):5–2, 2001.
3. D. G. Broeder, H. Brugman, A. Russel, and P. Wittenburg. A browsable corpus: accessing linguistic resources the easy way. In *LREC 2000 Workshop*, Athens, 2000.
4. ATLAS.ti Scientific Software Development. ATLAS.ti. `http://www.atlasti.com/`, Date of Access: October 2007.
5. Scintilla Project Group. Scintilla. `http://www.scintilla.org/`, Date of Access:2007.
6. IMDI Team. IMDI Metadata Elements for Session Descriptions. Technical report, MPI Nijmegen, October 2003.

7. The Dublin Core Metadata Initiative. Dublin Core Metadata Terms. `http://dublincore.org/`, Date of Access: October 2007.
8. QSR International. Nvivo. `http://www.qsrinternational.com/`, Date of Access: November 2007.
9. Silvestre Lacerda, Norberto Lopes, Nelma Moreira, and Rogério Reis. Ferramentas para a construção de arquivos digitais de história oral. In Luís Carriço José Carlos Ramalho, João Correia Lopes, editor, *Actas XATA 2007, XML: aplicações e tecnologias associadas*, pages 139–150. Universidade de Lisboa, Fevereiro 2007.
10. Fredrik Lundh. ElementTree API. `http://effbot.org/zone/element-index.html`, Date of Access:2007.
11. lxml development team. lxml. `http://codespeak.net/lxml/`, Date of Access:2007.
12. Sylvain Galliano Mathieu Manta, Fabien Antoine and Claude Barras. Transcriber. `http://trans.sourceforge.net/`, Date of Access: October 2007.
13. Thomas Muhr. Increasing the reusability of qualitative data with xml. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research*, Date of Access: October 2007.
14. Qualis Research. The Ethnograph. `http://www.QualisResearch.com`, Date of Access: October 2007.
15. Julian Smart, Robert Roebling, Vadim Zeitlin, and Robin Dunn. *wxWidgets 2.6.3: A portable C++ and Python GUI toolkit*.
16. Universidade Popular do Porto. Centro de Documentação e Informação sobre o Movimento Operário e Popular do Porto. `http://cdi.upp.pt/`.
17. Eric van der Vlist. *RELAX NG*. O'Reilly, 2003.
18. Guido van Rossum. *Python Library Reference*, 2.4.2 edition, 2005.
19. The Zthes working group. The Zthes specifications for thesaurus representation, access and navigation. `http://zthes.z3950.org/`, Date of Access: October 2007.

Navegante – An Intrusive Browsing Framework

Nuno Carvalho, José João Almeida, and Alberto Manuel Simões

Departamento de Informática
Universidade do Minho
smash@cpan.org,{jj|ambs}@di.uminho.pt

Abstract. Navegante is a generic framework to build superior order proxies for intrusive browsing. This framework provides the means for developing tools that behave as proxies, but perform some processing task on the content that is being browsed. Parallel to this content processing, applications can also run other user-defined functions with different purposes and interfaces, but we'll explain those later. Currently, Navegante only builds applications that run as CGIs, but this is intended to change in a near future. Applications are built writing programs in Navegante's Domain Specific Language (DSL).
Navegante is a work in progress. This article aims to describe the current state of development. What applications can be built and how. Also, we identify some implementation problems, and briefly discuss some future improvements. Finally, we try to illustrate most of the concepts described using a couple of case studies.

1 Introduction

A proxy[6] is, typically, a service that handles requests for a set of clients that are using other services. Proxies are traditionally used for caching or authorization purposes. On most of these cases the content served is kept unchanged. On some cases the response can be blocked or redirected. Some set of tools can be designed as proxies, but that in some way change the content in the response message. By doing this they become intrusive on the content, but they still apply the proxy concept – they indirectly serve information provided by other services. On-line spell checking tools, language detection, accessibility improvers[5] are some first glance examples of intrusive proxies. One way of seeing these tools is as a generic processor (or a higher order function [4]) that given a processing function, analysis the originally served content. So, this processor works as a proxy but it also applies this processing function to the content before delivering it to the final client.

Navegante [1] is a framework that can be used to build such applications. Applications that work as proxies for the Internet but that perform some additional kind of task on the browsed content. These applications are capable of giving feedback to the user about any processed data at any given time and in different ways. Applications are also capable of keeping state information over sequential requests and, have the ability to keep state information associated with browsed content.

This framework consists in a set of tools that are developed in Perl, taking advantage of well known Perl modules available on the Comprehensive Perl Archive Network (CPAN). For content processing we chose to use XML::DT [2], because it can be used to process XML or HTML exactly in the same way. Which means that we can, without much effort, and without changing our defined functions migrate our applications to feed on XML source files as well as HTML. We used the *Parser::YAPP*[3] Module to build the language parsers.

In the following sections, we expect to objectively cover the necessary topics to give the reader a basic understanding of the framework, and it's various components. We also illustrate the process of building applications and highlight the advantages of using the framework instead of building them from scratch.

2 The Proxy Pattern

In a client-server paradigm[8] the typical message flow can be described in three simple steps:

Step 1: The client sends a message to the server specifying the request.
Step 2: The server processes the request and sends a response message to the client.
Step 3: The client receives the response message from the server.

The diagram in Fig. 1 helps to illustrate this simple communication flow. There are no middle-mans in this architecture, and the message is delivered to the client exactly as it was sent by the server, content wise at least.

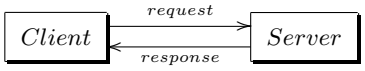

Fig. 1. Simple client-server paradigm.

For our intrusive proxies we add a new layer to the previous architecture. We add a new step in communications on both incoming and outgoing flows so we can perform some task after the server sends the response but before the client receives it. Ideally, we would like to add this extra step in the flow from the server to the client only, but this is not possible since there is no way we can tell the server that that specific response should go into our application before being forwarded to the client, without heavy changes on the client side. The goal here is to be intrusive but we don't want to tamper with every application, we want to keep things working the way they are. So, we upgrade our earlier description to a five step communication architecture:

Step 1: The client sends a message to the proxy specifying the request.

Step 2: The proxy forwards the message to the server unchanged.
Step 3: The server processes the request and sends a response message to the proxy.
Step 4: The proxy catches the response message, processes the content, and forwards the processed content to the client.
Step 5: The client receives the response message from the proxy.

This way we can expect a response from the server, in a given context, and perform some defined transformation on the content before delivering the processed message to the client. The client can be aware of this transformation, or not. The diagram in Fig. 2 helps to illustrate this new approach.

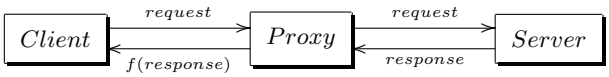

Fig. 2. Simple client-server paradigm with a middle-man.

The current implementation of Navegante only allows the generation of applications as CGIs[7]. So, our final application will be a form where we start by making the initial request (supplying an URL to Navegante's application) and then browse the web as we normally would, but being aware that every request is being processed by our intrusive proxy. At any time, we can tell Navegante to call a previously defined function to display some kind of data compilation or result (to show a state that is maintained while browsing the web).

3 Navegante's Approach

To build an application using Navegante's framework we need to feed the parser a Navegante program file. This file is to be parsed by Navegante, that will use a Perl skeleton file to create an application. This flow is illustrated in Fig. 3.

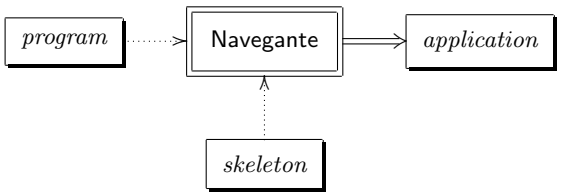

Fig. 3. Navegante's application building flow.

Currently, the only skeleton defined builds a CGI. Therefore, we can only build applications that behave as CGIs. Meaning that they work as standard

HTML pages, and are used as an entry point for intrusive browsing. Once you start browsing using the CGI, the application is already running. Every time a request is made, some processing function defined previously processes the content before delivering it to the client.

Another task being executed, and maybe a not so obvious one, is that every time content is processed, all links are analyzed, so that links to other pages are rewritten. Thus, when these links are followed, we are actually following them using our application, so its resulting content will also be processed. Using this method it is possible to have intrusive browsing without having to change anything in the user or client behavior, since all browsing is done exactly as it would be normally done. There are still some issues with the address rewriting engine, but we will discuss them later.

The application also adds a banner on top of every processed page. This banner provides some interesting features and is illustrated in Fig. 4.

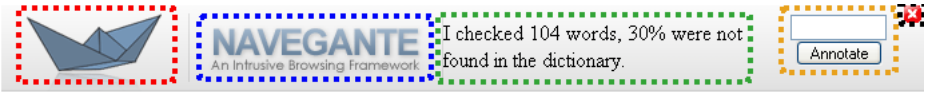

Fig. 4. Sample application banner.

The red outline box is a link to the application's *feedback* function, the function behavior is defined by the application. This is an optional feature, if the application doesn't define a *feedback* function this image it's simply not a link. The blue outline box is just Navegante's logo with a link to Navegante's homepage. The green outline box is the direct result of the function defined as *liveFeedback* that can exist in the application. This function can be used to deliver any kind of data regarding the processed content, and is also optional. The orange outline box is a form, defined in the application, that can be used to annotate pages. This simply stores in the application's state a correspondence between the URL and the user data inserted using the form. Finally, in the black outline box, is illustrated a quit button. This is a last call made to the application that executes a final function when defined by the application. This can be used to provide data summaries of present any other output to the user related to a browsing session.

Another feature that should be pointed out here, is that data can be preserved between requests (data can be made persistent). This means that we can build applications that use data from previous requests and can forward that data along to the following requests. Giving applications a sense of state even that it does not normally exists in protocols like HTTP. This persistent data abstraction is given using *cookies*, but other storage methods can be implemented.

As stated before, because we are using XML::DT, we can take advantage of this module features, and rework our application to process XML structured files.

Without having to change anything on our defined functions because basically, we are using the same processing engine.

3.1 Navegante Programs

Navegante programs are written in plain text files and have a well defined structure. The file should be divided in two major sections, as shown in Fig. 5. These sections are divided by the ## symbols, which has a special meaning for the parser.

```
1 {DSL definitions}
2 ##
3 {generic definitions}
```

Fig. 5. Navegante program structure.

These distinct sections should be used in different ways and with different purposes:

- the first section is used to specify application parameters using a well defined DSL, detailed in the next section.
- the second section is used to define any kind of functions needed by our application in Perl syntax. Everything in this section is almost directly inserted in our final application.

Ideally, it should be possible to write the second section of a Navegante program in any language that can be processed in the environment on which the application is running. Currently Navegante is not *other languages aware*, so this section of the program needs to be written in Perl.

3.2 Navegante's Domain Specific Language

As stated before, Navegante programs are written using a DSL. Most applications parameters are defined using this language. Although in a developing stage, it already can be used to describe many applications behavior. Be aware that most of these statements are CGI specific, because we are mainly building CGIs, but can be used on other contexts as well.

Here is a set of example statements that can be used:

proc: the name of the processing function. It defines the function that is to be called over the content that is being processed;

feedback: the name of the function that is called when feedback is requested, feedback is requested by following the link illustrated in the previous section;

livefeedback: the name of the function that is called every time the banner is generated, as illustrated in Fig. 4;

cginame: the final application CGI name;
formtitle: the title of the newly generated form;
desc: the name of the function that prints a description about the application;
init: the name of the function that is first called when the application starts (mainly used to initialize the application state).
annotate: the name of the function that processes the data submitted using the banner form, and if needed changes the application state;
iform: describe the fields that are to be rendered in the banner form;
quit: defines the function that is going to be used when the application's quit button is clicked.

This directives are used to define the behavior of our application in such a way that Navegante's parser can understand.

3.3 Navegante's Parser

Navegante's parser is currently the main tool in the framework. This tool is responsible for building an intrusive application given a program as described earlier.

The parser is written using *Parse::YAPP*. The parser uses a skeleton definition that can be seen as an application template. That is, the skeleton is defined in such a way that the parser's main job is to contextualize the skeleton using the program's DSL section. The grammar used is actually very simple and is described in Fig. 6.

```
1  Start -> actions fop
2  fop -> '##'
3  actions -> actions 'init' arg
4           | actions 'desc' arg
5           | actions 'proc' arg
6           ...
7           |
```

Fig. 6. DSL grammar.

We have a starting rule that derives in a list of actions and in a terminating symbol. The list of actions is our language specific statements, some of which were described earlier in this section. These statements will precisely describe how the skeleton is converted to a new application. The terminating symbol is simply the `##`, which splits the main sections of our program file.

The second section of our program is directly injected in the application, in order to have access to all the needed functions and variables that will be used during content analysis and/or transformation. Because of this, this section needs to be written in Perl. An idea of improvement would be to allow the use of other languages in this section, because this code is out of the framework scope.

Although it still needs to interact with the framework itself, to access state data for instance.

Currently, included in the parser is also the application skeleton. The application skeleton is nothing more than the final application template and a set of auxiliary functions used by it. The parser uses this template to build the final application.

4 Case Studies

This section describes how to build a couple of very simple applications, using Navegante, for illustration purposes. Keep in mind that the code described in this section most of the times is simplified, and/or not complete, and may actually, by itself, not be enough to implement the application's described behavior.

4.1 *reverse*

The first application we are going to build does not make use of state or feedback related concepts. It simply reverses the content being browsed. To create this application using Navegante, we first need to write a program file describing our application behavior. This example application can be written as show in Fig. 7.

```
1  cginame(./reverse)
2  formtitle(Read Everything Backwards)
3  proc(reverseFunction)
4  desc(reverseDesc)
5  init(reverseInit)
6  ##
7  sub reverseDesc {
8      return "This reverses every sentence in content.";
9  }
10 sub reverseFunction {
11     my $item = shift;
12     reverse($item);
13 }
```

Fig. 7. *reverse* program.

We can clearly see the symbol **##** in line six splitting the file in our two major sections. From line one to line five we are setting a wide range of values for our application using the DSL. We can also see that most of these values are actually callback functions that are going to be later defined in the program. Those are the same exact functions what we are defining from line seven to line thirteen. Our goal is to reverse the content being browsed, so let's take a closer look at line three, where we are using the DSL specific statement *proc* to define the function that will be used to process content, this function name is *reverseFuncion*. We

are defining this function behavior later in our program in line ten. Our function is receiving the content being browsed and reversing it.

After writing the program we can build our application. For this task we only need to run Navegante as shown in Fig. 8.

```
1 $ navegante examples/reverse
```

Fig. 8. Run Navegante.

After running this step a new file called *reverse* is created, as defined in Fig. 7, line one. This new file behaves as a traditional CGI, so we will copy it to a web server, and call it using a browser. The resulting page is illustrated in Fig. 9. Also, we can positively establish that our programs options were used, namely the CGI name, and the description function output.

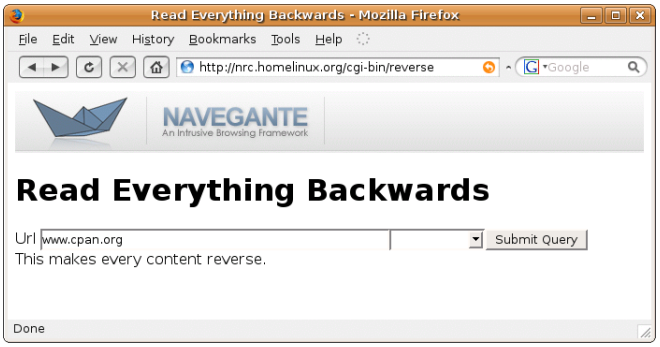

Fig. 9. *reverse* application form.

We can now use this form to start browsing pages, having the content being processed by our application, to be more specific, by the *reverseFunction* in our example.

We can see in Fig. 10 an example of a CPAN visit, where everything that is content was *reversed* by our intrusive application. Note that we are only analyzing content itself, and so the HTML code is kept unchanged, thus pages are correctly rendered by the browser. This is of course the expected behavior, we would not like to have all our HTML or CSS code reversed.

Also note, on both Fig. 9 and Fig. 10, the banner being inserted on the top of the page. Since we didn't define any of the *feedbackLive*, *iform* or *quit* functions for our application, nothing much interesting there.

Fig. 10. *reverse* application being used.

Plus, remember that our processor is rewriting links to make followed content be processed by our application. So, if we follow any link on the page we will browsed the referenced page after being processed by our *reverse* application, and thus all content will be reversed.

4.2 *htspell*

Let's now take a glimpse at a simplified version of an application called *htspell*. This application checks for existing words in a dictionary. In our example we are using the GNU Aspell spell checker to check words, in parallel to this we are also keeping track of not found words in our application's state. We keep being intrusive and changing the content being browsed, by underlining words we can't found in the dictionary. We are using the *feedback* function to summarize words that aren't found for the user and the *feedbackLive* function to present some data to the user about the processed content, namely the number of processed words and the percentage of words not found. We can see some important pieces of this application being defined in Fig. 11.

Now let's give our *htspell* application a try. We visit a simple test case that prints a well know sentence with a couple of misspelled words. We can see the resulting page in Fig. 12. Notice the banner at the top, we can see the output of our *feedbackLive* function printing the processed number of words.

Our template uses a set of auxiliary functions to store state information in browser's *cookies*. Applications can take advantage of this feature by using the *estado* special hash. That's exactly what we are doing in line fifteen in Fig 11. We are keeping track of not found words in this special variable *estado*. We can then

```
1  (...)
2  proc(htspellFunction)
3  feedback(htspellFeedback)
4  livefeedback(htspellLive)
5  ##
6  (...)
7  sub htspellFunction {
8  (...)
9    foreach (@words) {
10     if ($speller->check($_)) {
11       $found++ and push @new, $_;
12     }
13     else {
14       $not_found++ and push @new, "<u>$_</u>";
15       $estado{$_}++
16     }
17   }
18 (...)
19 sub htspellFeedback {
20     h3("Words not found in the dictionary:").
21     small(ul(li([map{"$_ - $estado{$_}"}
22         sort {$estado{$b} <=> $estado{$a}} keys %estado])));
23 }
24 sub htspellLive {
25   "I checked $processed words, ".int($not_found/$processed*100).
26   "% were not found in the dictionary.";
27 }
28 (...)
```

Fig. 11. *htspell* program.

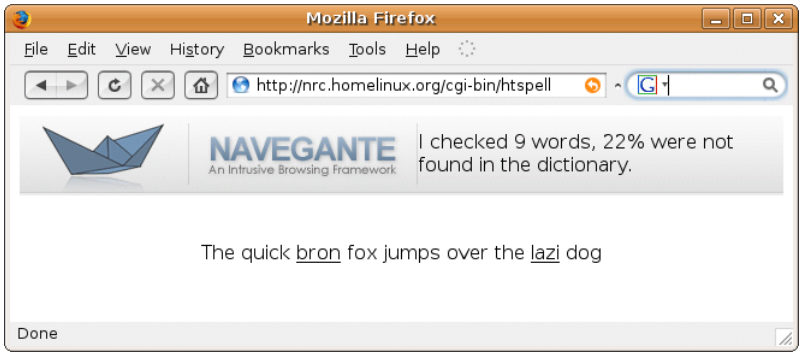

Fig. 12. *htspell* application.

later print this data to the user, using the function *htspellFeedback* as defined in line nineteen. We can verify this by following the *feedback* link, the small sailing boat, and verify the list of occurrences of not found words. This is illustrated in Fig. 13. Remember that our processing functions are nothing more than Perl code, so you can do whatever Perl allows you to do in your application, which is pretty much about everything.

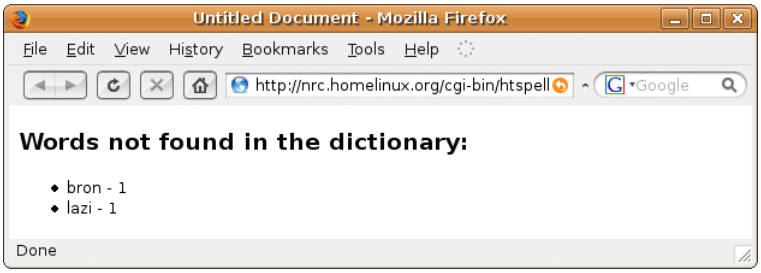

Fig. 13. *htspell* application *feedback* function.

5 Future Work

At the moment Navegante has some limitations:

- the address rewriting engine needs to be javascript aware. Nowadays many links are built using javascript in many obscure ways. This is often hard to detect, and rewriting those specific links can be almost impossible. But some improvements can be made, on this particular area, in order to lessen applications misbehavior.
- design and implement more skeleton templates, to be able to build different types of applications, and not only CGIs. A standalone service, that works like a traditional proxy, would be a nice addition.
- after adding new skeleton files, Navegante's DSL should be refactored, to allow the use of other contexts and options.
- allow the use of other programming languages in Navegante programs besides Perl. Make it possible for applications to fire up other interpreters or compilers to build or run the foreign code, in runtime or buildtime, depending on the language being used.
- implement other options to store data between requests. Currently, data can only be stored in browser's *cookies*.

6 Conclusion

Navegante is a framework aimed to craft applications that behave as proxies, but with an extra processing layer capability. This extra layer, allows to do any kind of transformation, on the fly, over the browsed content before being delivered to the client. We describe this approach, from the point of view of the application, as intrusive browsing. Intrusive browsing can be a very solid solution for many problems, since this approach can easily do complex processing, without adding much entropy to an already working scenario. Another advantage of this approach, in many situations, is that there is no need to change already working solutions to add some extra functionality.

Parallel to content processing, applications can be deployed with a diversified set of functions that can perform many tasks using different interfaces. These tasks can present new output by using the banner provided built-in with every application. Or, use the same banner to ask for user input, and merge it with state information. There's also the feedback option, that can process state information to display any kind of data at any given time. The subtlety of quitting the application, is also built-in in your application if you need it, it's just a way to finish the browsing session if the application needs to deliver any final conclusion regarding state or content information.

There are already some applications built based on this approach, which let us defend the usability and utility of this framework. These applications include:

- an online spell checker [9] that underlines each unknown word, and accumulates the list of unknown words for later reference;
- a terminology collector, that extracts from visited web pages words that are not common, and thus, are probably terminology terms;

Most of the times, this kind of applications are not easy or quick to implement, but with the right tools to aid development, complex tasks can be solved very easily. Also, some edge cases and common issues are already handled by some feature in the framework, like HTTP flow control, address rewriting, or being able to store state information. Navegante is still a work in progress but has already demonstrated to be a valuable tool to develop this specific gender of applications. Also, it is a very clean way to deploy applications that rely on other services, and in many cases even without those services being aware that their content is being used to feed other applications. In sum, there are unarguable facts that intrusive browsing is the natural solution for many complicated problems, and Navegante will be the natural way to implement them.

References

1. José João Almeida and Alberto Simões. Navegante: um proxy de ordem superior para navegação intrusiva. In José Carlos Ramalho, Alberto Simões, and João Correia Lopes, editors, *XATA 2006, Aplicações e Tecnologias Associadas*, pages 376–377, Portalegre, Fev. 2006. ESTGP. poster.
2. José João Almeida and Alberto Manuel Simões. Xml::dt. http://search.cpan.org/perldoc?XML::DT.
3. Francois Desarmenien. Parse::yapp. http://search.cpan.org/perldoc?Parse::Yapp.
4. Mark Jason Dominus. *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann, 2005.
5. António R. Fernandes, Alexandre Carvalho, J. João, and Alberto Simões. Transcoding for Web Accessibility for the Blind: Semantics from Structure. In *ElPub 2006 — Digital Spectrum: Integrating Technology and Culture*, Bansko, Bulgaria, June 2006.
6. Hans Rohnert. The proxy design pattern revisited. pages 105–118, 1996.
7. L.D. Stein. *Official guide to programming with CGI. pm*. Wiley New York, 1998.
8. W.R. Stevens, B. Fenner, and A.M. Rudoff. *UNIX Network Programming, Vol. 1*. Pearson Education, 2003.
9. Rui Vilela. Webjspell, an online morphological analyser and spell checker. In *Procesamiento del Lenguaje Natural 39 - SEPLN*, pages 291–292, 2007.

XCentric: Constraint based XML Processing

Jorge Coelho[1] and Mário Florido[2]

[1] Instituto Superior de Engenharia do Porto & LIACC
Porto, Portugal
[2] University of Porto, DCC-FC & LIACC
Porto, Portugal
{jcoelho,amf}@ncc.up.pt

Abstract. Here we present the logic-programming language XCentric, discuss design issues, and show its adequacy for XML processing. Distinctive features of XCentric are a powerful unification algorithm for terms with functors of arbitrary arity (which correspond closely to XML documents) and a rich type language that uses operators such as repetition (*), alternation, etc, as types allowing a compact representation of terms with functors with an arbitrary number of arguments (closely related to standard type languages for XML). This new form of unification together with an appropriate use of types yields a substantial degree of flexibility in programming.

1 Introduction

XML is a powerful format for tree-structured data. A need for programming language support for XML processing led to the definition of XML programming languages, such as XSLT [28], XDuce [14], CDuce [1], Xtatic [29] and Xcerpt [2].

In this paper we present XCentric, a logic programming (LP) language based on the *unification of terms with flexible arity function symbols* extended with a new *type system* for dealing with sequences and new features for searching sequences inside trees at arbitrary depth.

The main features of XCentric rely on the use of:

Regular expression types: regular expression types give us a compact representation of sequences of arguments of functors with flexible arity. They also add extra expressiveness to the unification process. Let us present an illustrating example.

The following declaration introduces regular expression types describing terms in a simple bibliographic database:

```
:-type bib ---> bib(book+).
:-type book ---> book(author+, name).
:-type author ---> author(string).
:-type name ---> name(string).
```

Type expressions of the form $f(...)$ classify tree nodes with the label f (XML structures of the form $< f > ... < /f >$). Type expressions of the form $t*$ denote

a sequence of arbitrary many ts, and $t+$ denotes a sequence of arbitrary many ts with at least one t. Thus terms with type *bib* have *bib* as functor and their arguments are a sequence of one or more books. Terms with type *book* have *book* as functor and their arguments are a sequence consisting of one or more authors followed by the name of the book.

The next type describes arbitrary sequences of authors with at least two authors:

```
:-type type_a ---> (author(string),author(string)+).
```

A new form of unification: in the previous example, to get the names of all the books with two or more authors in XCentric, one just needs the following query ($= * =$ stands for unification of terms with flexible arity functors and $t :: \tau$ means that term t has type τ):

```
?-bib(_,book(X::type_a,name(N)),_)=*=BibDoc::bib.
```

This unifies two terms typed by *bib*. The type declaration in the first argument is not needed because it can be easily reconstructed from the term. In this case we bind the variable N to the content of the name element of the first book element with at least two authors. Note how the type *type_a* in the first argument of the unification is used to jump over an arbitrary number of arguments and extract the name of the first book with at least two authors. All the results can then be obtained, one by one, by backtracking. This goes far beyond standard Prolog unification.

Sequence variables and unification of terms with functors of flexible arity gives XCentric the power of partially specifying terms in breadth (i.e. within the subterms of the same term). In XCentric one can also partially specify a term in depth using the *deep* predicate. A query term of the form $deep(t_1, t_2)$ matches all subterms of t_2 that match the term t_1. Consider the following example (in XCentric we can explicitly refer to sequences of terms $t_1, ..., t_n$ as $< t_1, ..., t_n >$): suppose we want to find sequences of two authors in a document to which the variable *XML* is bound. We can use the query:

```
?-deep(<author(A1),author(A2)>,XML).
```

The names of the two authors will bind variables A_1 and A_2, and all solutions can be found by backtracking.

The main contributions of XCentric to the logic programming paradigm, are to give programmers a tool that makes it much easier and more declarative to process XML, and to show the impact of a new form of unification (*typed unification of terms with functors of flexible arity*) in programming. Note that subjects such as databases, data-mining and Web programming, are quite relevant application areas of logic programming, and in these areas XML is becoming more and more a standard data format for information exchange.

In this paper we show the previous claim about XCentric, showing its contribution with respect to:

Prior non-LP XML processing work: mainstream languages for XML processing such as XSLT ([28]), XDuce ([14]), CDuce ([1]) and Xtatic ([29]) rely on the notion of trees with an arbitrary number of leaf nodes to abstract XML documents. However these languages are based on functional programming and thus their key feature is pattern matching, not unification. Regular expression patterns are often ambiguous and thus functional XML processing languages presume a fixed matching strategy for making the matching deterministic. In contrast, XCentric just leaves ambiguous matching non-deterministic and examines every possible matching case by backtracking. This makes it possible to write complicated XML processing tasks in a quite declarative way. Although pattern matching is desirable in many applications of XML processing, there are situations where the use of unification is a gain. It is known that unification may even improve efficiency in some cases (by careful use of the logical variable) and it is on the basis of a truly relational programming, improving declarativeness and modularity in many cases. In this paper we show examples where these situations also arise in XML processing. This, and the use of unification of terms with functors of flexible arity, show that there are aspects of XML processing in XCentric that do not have counterparts in other approaches to XML processing.

Prior LP work: some Prolog systems have libraries [23, 22] to deal with XML. These libraries translate XML documents to a list of Prolog terms and use standard Prolog for processing. XCentric uses recent results of unification theory (*unification of terms with functors of flexible arity* [17, 7]) and novel type languages based on regular types [8, 6], as the theoretical basis of a logic programming language for XML processing where sequences of terms are first class objects of the language. This leads to a much more declarative way of processing XML when compared to the standard Prolog libraries for XML processing. Xcerpt ([2]) is a logic programming query language for XML which also used terms with flexible arity function symbols as an abstraction of XML documents. It used a special notion of term (called *query terms*) as patterns specifying selection of XML terms much like Prolog goal atoms. The underlying mechanism of the query process was *simulation unification* ([3]), used for solving inequations of the form $q \leq t$ where q is a query term and t a term representing XML data. Concepts behind Xcerpt are more directed to query languages and technically quite different from the typed unification of terms with functors of flexible arity used in XCentric.

Regular expression matching was also used in [18, 19] to extend context sequence matching with context and sequence variables. This work dealt with matching, not unification, and it was not integrated in a programming language. Unification of terms with functors of flexible arity generalizes previous work on word unification ([15]), equations over lists of atoms with a concatenation operator ([10]) and equations over free semigroups ([21]), by enabling the use of as many flexible arity symbols as we wish and of arbitrarily nested terms. Recently, in [16], a specific language was defined to denote relations between XML documents. This

language used its own new syntax, based on mainstream functional languages for XML processing, and its definition stresses the usefulness of an approach based on the truly relational programming paradigm: logic programming.

Monadic Datalog has also been successfully applied to XML querying in the absence of data values [13].

Note that the work described in this paper was first presented in a previous paper from the authors ([5]).

In the rest of the paper we assume that the reader is familiar with logic programming [20] and XML [26]. We start in section 2 by presenting some examples of the language. In section 3 we present sequence variables and flexible arity functions and in section 4 we present sequence types. In section 5 we explain the role of types in the unification process, and finally in section 6 we conclude and outline the future work.

2 XCentric by example

Here we present several simple examples to familiarize the reader with the language before presenting the details.

In XCentric, an XML document is translated to a term with flexible arity function symbol. This term has a main functor (the root tag) and zero or more arguments. Although our actual implementation translates attributes to a list of pairs, since attributes do not play a relevant role in this work we will omit them in the examples, for the sake of simplicity.

Example 1. Consider the simple XML file:

```
<addressbook>
  <record>
   <name>John</name>
   <address>New York</address>
   <email>john.ny@mail.com</email>
  </record>
  ...
</addressbook>
```

Its corresponding term is:

```
addressbook(record(name('John'),address('New_York'),
                                email('john.ny@mail.com')),...)
```

Suppose that the previous XML file is valid with respect to the following DTD:

```
<!ELEMENT addressbook (record*)>
<!ELEMENT record (name,address,phone?,email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

This DTD can be described in XCentric by the type rule:

```
:-type type_addr ---> addressbook(record(name(string),address(string),
                                         phone(string)?,email(string))*)
```

From now on, whenever a variable is presented without any type information it is implicitly associated with the universal type *any* (which types any term). Through the following examples we will use the built-in predicates *xml2pro* and

pro2xml which respectively convert XML files into terms and vice-versa. We will also use the predicate *newdoc(Root,Args, Doc)* where *Doc* is a term with functor *Root* and arguments *Args*.

Example 2. Suppose we have an XML document with a catalog of books like the following one:

```
<catalog>
  ...
   <book number="500">
     <author>Simon Thompson</author>
     <name>
     Haskell: The Craft of Functional Programming (2nd Edition)
     </name>
     <price>41</price>
     <year>1999</year>
   </book>
    ...
</catalog>
```

To get all the books with two or more authors using SWI-Prolog [23] (which has a quite good library for processing XML in Prolog) we need the following code:

```
pbib([element(_,_,L)]):-
                    pbib2(L).
pbib2([]).

pbib2([element('book',_,Cont)|Books]):-
                    authors(Cont),!,pbib2(Books).
pbib2([_|Books]):-
                    pbib2(Books).

authors([element('author',_,_),element('author',_,_)|R]):-
                    write_name(R).

write_name([element('name',_,[N])]):-
                    write(N),nl.
write_name([_|R]):-
                    write_name(R).
```

To do the same in XCentric, assuming that the XML file is translated to a term binding variable *BibDoc*, the following query is enough:

```
catalog(_,book(name(N),author(_),author(_),_),_) =*= BibDoc.
```

All the solutions can then be obtained, one by one, by backtracking. The simplicity and declarativeness of the second solution speaks by itself when compared to the first one.

If we want to verify the document consistency with respect to a given DTD, we just have to replace in the previous query, the variable *BibDoc*, by the typed variable $BibDoc :: tc$, where type tc is the translation of the DTD to its corresponding XCentric type.

2.1 Incomplete terms in depth

XCentric also provides predicates that allow the programmer to find a sequence of elements at arbitrary depth, to search for the *nth* occurrence of a sequence of elements and to count the number of occurrences of a sequence. The predicates are *deep/2*, *deepp/3* and *deepc/3*, respectively.

Example 3. This example is based on a medical report using the HL7 Patient Record Architecture and inspired by the XQuery use cases available at [27]. Given *report1.xml* (figure 1), find what happened between the first *incision* and the second *incision* and write the result in a file named critical.xml:

```
<report>
  <section>
    <section_title>Procedure</section_title>
     <section_content>
      The patient was taken to the operating room where she was placed...
      <anesthesia>induced under general anesthesia.</anesthesia>
     <prep>
       <action>A Foley catheter was placed to decompress </action>...
     </prep>
     <incision>
       A curvilinear incision was made <geography> in the midline
       immediately infraumbilical </geography> and the subcutaneous
       tissue was divided <instrument> using electrocautery.
       </instrument>
     </incision>
     The fascia was identified and <action> #2 0 Maxon stay sutures were
     placed on each side of the... </action>
     <incision>
       The fascia was divided using <instrument> electrocautery
       </instrument> and the peritoneum was entered. </incision>
     <observation> The small bowel was identified. </observation>...
    </section_content>
  </section>
</report>
```

Fig. 1. report1.xml

```
translate:-
    xml2pro('report1.xml',Rep),
    deep(<incision(_),Critical,incision(_)>,Rep),
    newdoc(critical_sequence,Critical,FL),
    pro2xml(FL,'critical.xml').
```

The result is:

```
<critical_sequence>
 The fascia was identified and <action> #2 0 Maxon stay
 sutures were placed on each side of the...</action>
</critical_sequence>
```

2.2 Unification and XML Processing

Mainstream XML processing languages rely on pattern matching. In our approach we also use unification for XML processing. In this section we present some examples where unification has advantages over pattern matching.

Example 4. In functional based languages for XML processing transformations are unidirectional, enabling the use of a description of the structure of an XML document and the use of pattern matching to extract some of its subparts.

Being a relational language, XCentric can easily describe the structures of two documents and relate their subparts. For example, we can write the following simple predicate in XCentric for converting between fragments of two kinds of address books.

```
translate(<person(name(N),C1)>,<card(person-name(N),C2)>):-
                                        address_content(C1,C2).

address_content(<>,<>).

address_content(<A1,address(A),A2>,<L1,location(A),L2>):-
                              address_content(<A1,A2>,<L1,L2>).
```

This relates a *person* element and a *card* element, where the pattern of the first argument of *translate* requires the *person* to contain a *name* element followed by

a sequence of *address* elements, and the second argument describes the structure of the *card* containing a *person-name* element follower by a sequence o *location* elements. Variable N, which appears in both arguments, specifies that its corresponding subparts, the contents of *name* and *person-name*, are the same. Predicate *address_content* relates *address* in a person element with *location* in the corresponding *card* element. This is trivially expressed in an unification-based relational language such as XCentric, but impossible to express in a functional (thus unidirectional) language based on pattern matching. Note that variables occurring in sequences, are interpreted in the domain of sequences, thus, in this program, unification is not Prolog unification, but the non-standard unification of XCentric. Also note the gain in modularity: this predicate can be used in three different ways. 1) to transform an XML document with the format specified by the first argument of *translate* into the format specified by its second argument, 2) to do the opposite transformation, or 3) to guarantee that two different documents in the two different formats are related in the way specified by the predicate (corresponding, respectively, to call it with the first, second or both arguments ground). In a functional-based language these three different behaviors have to be implemented by three different functions.

Example 5. Suppose we have an XML document that represents an article entry:

```
<text>
Mainstream languages for <b>XML</b> processing such as XSLT <ref>W3C
</ref>, XDuce <ref>Hosoya </ref>, CDuce <ref>Frish Casagna and Benzaken
</ref> and Xtatic <ref>Pierce</ref> rely on the notion of trees with an
arbitrary number of leaf nodes to abstract <b>XML</b> documents.
</text>
```

This document has references like **<ref>W3C</ref>** which appear in a simple bibliography database, where each **ref** element has a corresponding **author**:

```
<bibliography>
 <bib>
   <author>Coelho and Florido</author>
   <name>Type-based XML Processing in Logic Programming</name>
 </bib>
 <bib>
   <author>Hosoya</author>
   <name>XDuce: A Typed XML processing language</name>
 </bib>
...
</bibliography>
```

The idea is the following:

1. Create a new bibliography document only with references occurring in the article but ordered by author name.
2. Create a new article were each reference is replaced by a number corresponding to the author order in the bibliography.

As result we want the following:

```
<text>
Mainstream languages for <b>XML</b> processing such as XSLT <i>4</i>,
XDuce <i>2</i>, CDuce <i>1</i> and Xtatic <i>3</i> rely on the notion
of trees with an arbitrary number of leaf nodes to abstract <b>XML</b>
 documents.
</text>
```

and the bibliography file as:

```
<bibliography>
 <bib>
  <index> 1 </index>
  <author>Frish Casagna and Benzaken</author>
  <name>CDuce an XML-centric general-purpose language</name>
 </bib>
...
</bibliography>
```

To achieve this result using a similar method but based in pattern matching approach, note that, as we only know all the references after processing the entire document, we must process the document, retrieve all the references found, order the references and then process the document a second time to replace the references with the corresponding indexes. Using unification it is possible to process the document only once: the references are replaced by free variables which are associated with the corresponding references (by means of an association list) creating an intermediate document which is not a ground term. We can then order the association list by author and bind the corresponding variables with the correct index. The document now becomes a ground term (by the use of unification) which is the desired output. Note that we only processed the document once (the complete implementation can be found at http://www.ncc.up.pt/xcentric/).

3 Sequence Variables and Flexible Arity Functions

Here we briefly review the notions of sequence, sequence variable and functor with flexible arity. A detailed description of this subject and of (untyped) unification of flexible arity terms can be found in [7]. We extend the domain of discourse of Prolog (trees over uninterpreted functors) with finite sequences of trees.

Definition 1. *A* sequence $\tilde{t}$*, is defined as follows:* ϵ *is the empty sequence and* $t_1, \tilde{t}$ *is a sequence if* t_1 *is a term and* $\tilde{t}$ *is a sequence.*

We now proceed with the syntactic formalization, by extending the standard notion of Prolog term with flexible arity function symbols and sequence variables.

Consider an alphabet consisting of the following sets: the set of standard variables, the set of sequence variables, the set of constants, the set of fixed arity function symbols and the set of flexible arity function symbols.

Definition 2. *The set of terms over the previous alphabet is the smallest set that satisfies the following conditions:*

1. *Constants, standard variables and sequence variables are terms.*
2. *If* f *is a flexible arity function symbol and* $t_1, \ldots, t_n$ $(n \geq 0)$ *are terms, then* $f(t_1, \ldots, t_n)$ *is a term.*
3. *If* f *is a fixed arity function symbol with arity* n*,* $n \geq 0$ *and* $t_1, \ldots, t_n$ *are terms such that for all* $1 \leq i \leq n$*,* t_i *does not contain sequence variables as subterms, then* $f(t_1, \ldots, t_n)$ *is a term.*

Remark 1. To avoid further formality, we assume that the domain of interpretation of variables is predetermined by the context where they occur. Variables occurring in a constraint of the form $t_1 = * = t_2$ are interpreted in the domain of sequences of trees, otherwise they are standard Prolog variables.

3.1 Sequences

We use a special kind of terms, here called *sequence terms*, for implementing sequences.

Definition 3. *A* sequence term*, $\bar{t}$ is defined as follows:*

- ϵ *is a* sequence term *that represents the empty sequence.*
- $seq(t, \bar{s})$ *is a* sequence term *if t is a term and $\bar{s}$ is a* sequence term*.*

Definition 4. *A sequence term in* normal form *is defined as:*

- ϵ *is in normal form*
- $seq(t_1, t_2)$ *is in normal form if t_1 is not of the form $seq(t_3, t_4)$ and t_2 is in normal form.*

Sequence terms in normal form are the internal representation of sequences. For example, $seq(a, seq(b, \epsilon))$ represents sequence a,b. Note that for simplification purposes we drop the *seq* operators for sequences of just one element.

4 Types

In this section we present the type language starting with a description of *Regular Types* [11] and then their extension to type sequences of terms.

4.1 Regular Types

Definition 5. *Assuming an infinite set of* type symbols*, a* type term *is defined as follows:*

1. *A constant symbol (we use $a, b, c, etc.$) is a type term.*
2. *A type symbol (we use $\alpha, \beta, etc.$) is a type term.*
3. *If f is a flexible arity function symbol and each τ_i is a type term, $f(\tau_1, ..., \tau_n)$ is a type term.*

Definition 6. *A* type rule *is an expression of the form $\alpha \rightarrow \Upsilon$ where α is a type symbol and Υ is a finite set of type terms.*

Sets of type rules correspond to *regular term grammars* [25].

Example 6. Let α and β be type symbols, $\alpha \rightarrow \{a, b\}$ and $\beta \rightarrow \{nil, tree(\beta, \alpha, \beta)\}$ are type rules.

Definition 7. *A type symbol α is defined by a set of type rules T if there exists a type rule $\alpha \rightarrow \Upsilon \in T$.*

Regular types are the class of types that can be defined by finite sets of type rules. In XCentric a type rule $\alpha \rightarrow \{\tau_1, \ldots, \tau_n\}$ is represented by the declaration:

$$\text{:-type } \alpha \text{ - - -> } \tau_1; \ldots; \tau_n.$$

It is well known that regular types can be associated with unary logic programs (see [31, 30]). For every type symbol α, there is a predicate definition α, such that $\alpha(t)$ is true if and only if t is a term with type α (note that we are using the type symbol as the predicate symbol).

4.2 Regular Expression Types

We now define regular expression types, which describe sequences of values: a* (sequence of zero or more a's), a+ (sequence of one or more a's), a? (zero or one a), a|b (a or b) and a,b (a followed by b). We translate regular expression types to our internal sequence notation:

$$\begin{array}{l} a* \Rightarrow \alpha_* \rightarrow \{\epsilon, seq(a, \alpha_*)\} \\ a+ \Rightarrow \alpha_+ \rightarrow \{a, seq(a, \alpha_+)\} \\ a? \Rightarrow \alpha_? \rightarrow \{\epsilon,\ seq(a, \epsilon)\} \\ a|b \Rightarrow \alpha_| \rightarrow \{a, b\} \\ a, b \Rightarrow \alpha_{seq} \rightarrow \{seq(a, seq(b, \epsilon))\} \end{array}$$

Note that DTDs (Document Type Definition) [26] can be trivially translated to regular expression types. XCentric also has some *XML Schema* [24] support, basic types like string, integer, boolean and float, bounding the occurrences of sequences and orderless sequences are supported.

5 Types in the unification process

In this section we explain the role of types in the unification process.

Definition 8. *A* type declaration *for a term t with respect to a set of type rules T is a pair $t :: \alpha$ where α is a type symbol defined in T.*

Example 7. Consider the equation $a(X, b, Y) :: \alpha_a = * = a(a, b, b, b) :: \mu$, where α_a is defined by the type rules:

$$\begin{array}{l} \alpha_a \longrightarrow \{a(\alpha_x, b, \alpha_y)\} \\ \alpha_x \longrightarrow \{\mu\} \\ \alpha_y \longrightarrow \{b, (b, \alpha_y)\} \end{array}$$

then this unification gives only two results:

1. $X = a$ and $Y = b, b$
2. $X = a, b$ and $Y = b$

Note that without the types the solution $X = a, b, b$ and $Y = \epsilon$ would also be valid.

Implementation: An equation of the form $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ is translated to the following query:

$$? \text{ - } t_1 = * = t_2, \alpha_1(t_1), \alpha_2(t_2).$$

and the respective predicate definitions for α_1 and α_2 (as described in section 4). Correctness of $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ comes for free from the correctness of the untyped version of $= * =$ (presented in [7]) and noticing that if $? - t_1 = * = t_2, \alpha_1(t_1), \alpha_2(t_2)$ succeeds then $t_1\theta \in [\alpha_1]_T \cap [\alpha_2]_T$, where θ is the substitution resulting from $t_1 = * = t_2$.

6 Conclusions and Future Work

XCentric is an extension of Prolog with a richer form of unification and regular types, designed specifically for XML processing in logic programming. It enables a highly declarative style of XML-processing and it is based on a sound foundation of a very small core of well studied key features, such as *unification of terms with flexible arity* [7, 17] and *regular types for logic programming* [31, 12]. Also note that XCentric is now being used successfully in practice in some areas, such as website auditing and verification [9, 4]. Ongoing work is being done to improve efficiency. We have benchmarks indicating that, compared with pattern matching, XCentric is rather inefficient when applied to large files (more than 15 KB). This is, somehow, expected, since pattern matching itself is more efficient than unification, and pattern-matching based languages, such as XDuce or CDuce, are compiled, highly optimized languages. Future and ongoing work on this matter, includes the use of tabling in the unification algorithm and extending the WAM with new instructions for dealing directly with sequences. We are also interested on applying XCentric to other application areas besides XML. Bioinformatics, relying intensively on the notion of sequence, is a natural candidate for new applications.

References

1. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN Int. Conference on Functional Programming*, Uppsala, Sweden, 2003.
2. F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *2nd Annual International Workshop Web and Databases*, volume 2593 of *LNCS*, 2002.
3. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *International Conference on Logic Programming (ICLP)*, volume 2401 of *LNCS*, 2002.
4. Jorge Coelho and Mário Florido. Type-based static and dynamic website verification (to appear). In *ICIW'07*. IEEE Computer Society, 2007.
5. Jorge Coelho and Mario Florido. Xcentric: Logic programming for xml processing. In *9th ACM International Workshop on Web Information and Data Management*. ACM Press, 2007.
6. Jorge Coelho and Mário Florido. Type-based XML Processing in Logic Programming. In *Practical Aspects of Declarative Languages*, volume 2562 of *LNCS*, 2003.
7. Jorge Coelho and Mário Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Ontologies, Databases and Applications of SEmantics (ODBASE)*, volume 3291 of *LNCS*. Springer Verlag, 2004.
8. Jorge Coelho and Mário Florido. Unification with flexible arity symbols: a typed approach. In *Informal proceedings of the 20th International Workshop on Unification (UNIF'06)*, Seattle, USA, 2006.
9. Jorge Coelho and Mário Florido. VeriFLog: Constraint Logic Programming Applied to Verification of Website Content. In *Int. Workshop XML Research and Applications (XRA'06)*, volume 3842 of *LNCS*. Springer-Verlag, 2006.

10. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
11. P. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.
12. Mário Florido and Luís Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.
13. Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.
14. Haruo Hosoya and Benjamin Pierce. XDuce: A typed XML processing language. In *Third Int. Workshop on the Web and Databases*, volume 1997 of *LNCS*, 2000.
15. J. Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.
16. Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In John H. Reppy and Julia L. Lawall, editors, *ICFP*, pages 201–214. ACM, 2006.
17. Temur Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Joint AISC'2002 - Calculemus'2002 conference*, LNAI, 2002.
18. Temur Kutsia. Context sequence matching for xml. In *Proceedings of the 1th Int. Workshop on Automated Specification and Verification of Web Sites*, 2005.
19. Temur Kutsia and Mircea Marin. Can context sequence matching be used for querying xml? In Laurent Vigneron, editor, *Proceedings of the 19th Int. Workshop on Unification (UNIF'05)*, pages 77–92, Nara, Japan, 22 April 2005.
20. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
21. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik USSR*, 103:147–236, 1977.
22. Pillow: Programming in (Constraint) Logic Languages on the Web. http://clip.dia.fi.upm.es/Software/pillow/pillow.html, 2001.
23. SWI Prolog. http://www.swi-prolog.org/.
24. XML Schema. http://www.w3.org/XML/Schema/, 2000.
25. J.W. Thatcher. *Tree automata: An informal survey*. Prentice-Hall, 1973.
26. Extensible Markup Language (XML). http://www.w3.org/XML/, 2003.
27. XQuery Use Cases. http://www.w3.org/TR/xquery-use-cases/, 2005.
28. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/, 1999.
29. Xtatic. http://www.cis.upenn.edu/~bcpierce/xtatic/, 2004.
30. E. Yardeni and E. Shapiro. A type system for logic programs. In *The Journal of Logic Programming*, 1990.
31. Justin Zobel. Derivation of polymorphic types for prolog programs. In *Proc. of the 1987 International Conference on Logic Programming*. MIT Press, 1987.

Using OWL to specify and build different views over the Emigration Museum resources

Flávio Xavier Ferreira and Pedro Rangel Henriques

Departamento de Informática, CCTC
Universidade do Minho
{flavioxavier,prh}@di.uminho.pt

Abstract. This paper discusses the approach used to create the exhibition rooms of the virtual (Web-based) section of Portuguese Emigration Museum (Museu da Emigração e das Comunidades, MEC) founded by the Cultural Department (Casa da Cultura) of Fafe's Town Hall (Câmara Municipal de Fafe). The museum's assets are made up of documents (paper or digital format) of more than 8 kinds, ranging from passport records to photos/cards or building-drawings. Each room is no more than a view over the information contained in those single or interrelated resources. The information exhibited in each room is described by an ontology, written in OWL. That ontology is used to specify the information that is extracted from the resources and to provide a semantic-network navigator to the museum visitor. That approach can be automatised to allow a very systematic way to deal with the huge and rich museum assets; we will also discuss some technical details concerning its complete implementation in the near future.

1 Introduction

Fafe, as many other Portuguese towns and villages, mainly at the north, has a huge cultural heritage characterising the social phenomena of emigration (especially to Brazil) along the nineteenth and first half of twentieth centuries.

In this context Miguel Monteiro[1], supported by the staff of Fafe's Town Hall (via Cultural Department), started some years ago collecting information from passports' governmental records into a database. But soon from this project arise the the idea to gather all sorts of documentation and create a web-based virtual museum that makes easily accessible this rich cultural heritage to emigrants and theirs descendants as well as to all those researching in that area, and of course the general public.

The Museum was born in 2001 with the designation of Museu da Emigração e das Comunidades (hereafter referred as MEC). Its material is inherited mainly from official documents or personal writings reporting on the departure, travel, and stay abroad, but there are also a large number of assets bearing witness

[1] An History professor, responsible for the MEC creation, and its current director.

to the less usual phenomena of *emigrants' return* – besides the documents, a large set of buildings (private or public, professional or philanthropic, and other non-physical evidences) left by the emigrants around the country can be also considered assets. The MEC is structured upon six Rooms (see `http:///www.museu-emigrantes.org/museu.htm`), but at the moment these rooms are handmade, difficult to maintain and to add new information, and most important, they are lacking a systematic way to for information acquisition, treatment and exhibition; inconsistencies are evident from room to room but even inside the same room.

Some years ago the MEC, started a collaboration with University of Minho Language Specification and Processing Group (GEPL), to develop a project aiming at systematic approach for the acquisition, archiving, treatment and exploration of the Museum's documental resources. In our perspective, each room is seen just as *a specific view over a common information repository*. The repository should be a digital archive (in database format or as a collection of XML files) of all the information resources referred above as museum's assets. Each *view* (the knowledge enclosed in the respective room) can be specified by an *ontology*, as traditionally done by philosophers to organise the discourse over a certain closed-world. The extraction process[2] can be automatise by resorting to a standard notation for the ontology description, and moreover, the browsers that will implement the user-interface in each room (as a semantic network navigator) can also be automatically built.

In the past, that approach was realised using Topic Maps[13], at the moment we are experimenting with OWL. This paper describes our more recent research work exploring RDF/OWL.

So, we start, in section 2, with a catalogue of the MEC information resources, which constitute the Museum's assets; as the basis for all the exhibition rooms, it gives the motivation to our work and proposal. In section 3 we introduce the ontology concept and present the main standards for ontology description; Topic Maps are just briefly referred as they where explored in previous work; RDF/OWL is studied in detail because it is going to be the main focus of this work. Section 4, not the biggest but the central one, presents a detailed discussion of our methodological approach, briefly introduced above as *the use of ontologies to specify and construct each museum's room*. Section 5 is concerned with MusVis, the extraction and navigation system we are developing; its architecture is defined and its technical implementation is briefly referred. The paper ends at section 6 with the traditional remarks and future work.

2 Emigration Museum and its Information Resources

The MEC is a web-museum (although it also has physical headquarters and some exhibitions), that gathers knowledge, and resources about the Portuguese emigration.

[2] The task of building up the ontology from the information resources data

The MEC wants to discover and show the effects of mixing people and cultures, in the social, cultural and economical history of Portugal. It focus, mainly on the past Portuguese emigration to Africa and the more recent emigration to Brazil (19th and 1st half of 20th century) and to Europe (2nd half of the 20th century), but it is by no means restricted to them [2].

The MEC assets are vast and multifaceted, this is supported by the fact that emigration documents and objects come from the most diversified sources, ranging from official government records to old newspapers and photo albums. The document types are themselves heterogeneous (from official travel reports to local stories). Some documents where converted to an electronic format (plain ASCII text, Ms-Word, Ms-Excel, Ms-Access, HTML, etc.), but many others are, still, in paper format stored in Archives and Libraries.

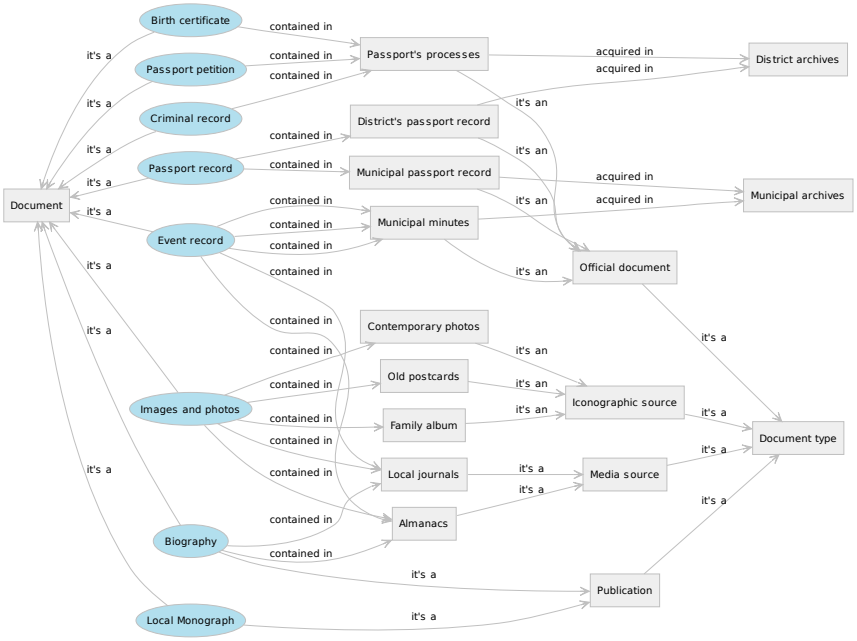

Fig. 1. Information sources and resources semantic web

This tremendous amount of resources and their variety gives the MEC an enormous potentiality as a museum, but at the same time it is very difficult to organise and display all this information in a straightforward way. To overcame this problem the sources of information (the so-called museum's assets) were catalogued. The *conceptual map* (a *graph of concepts*) in Figure 1 shows the organisation of the sources and the types of the documents (*ellipses* denote doc-

ument types). Based on that classification, we have defined (using XSD) an XML format to enable the encoding of those documents to a structured (annotated) digital format, adequate for archiving and subsequent processing (this will be addressed in section 5). We are also developing an editor to assist the acquisition phase and the creation of the XML files.

3 Ontologies and their Notation

An ontology is originally a philosophic concept, concerned with the study of being or existence and forms the basic subject matter of metaphysics. In computer science an ontology represents a set of concepts and there relations in a given domain; it can also be used to infer knowledge and information about the domain's objects[9].

Technically speaking an ontology is defined by a set of classes, individuals, attributes, and relations. *Classes* or *concepts*, are abstract sets of objects, that can contain other individuals and other classes (subclasses). *Individuals* or *instances* are the actual objects we want to represent, they can be people, animals, numbers, web pages, etc.. The ontology objects can be described using *attributes*, each attribute is a name/value pair. *Relations* are connections between the ontology objects, they allow the representation of concepts and the creation of associations within the ontology objects. In the example seen in Figure 2, we

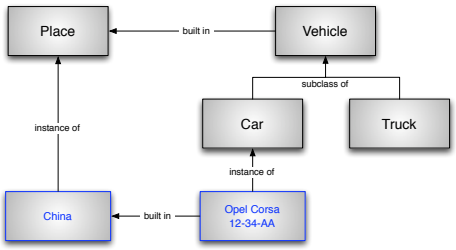

Fig. 2. An ontology example

can identify the classes `Vehicle`, `Car`, `Truck` and `Place`; and the individuals `China` and `Opel Corsa`; the licence plate `12-31-AA` can be seen as an attribute; and there are also the relations `built-in`, `subclass-of` and `instance-of`.

To use ontologies in the MEC we still need a specific notation to write them. There are several ontology description languages available[6], but our attention (in the next subsections) goes to: the International Organization for Standardization (ISO) standard Topic Maps (TM)[11]; and the World Wide Web Consortium (W3C) standards Resource Description Framework (RDF), RDF Schema (RDFS) and Web Ontology Language (OWL)[3].

[3] All three are part of the W3C semantic web effort[5].

3.1 Topic Maps

Topic Maps can be seen as a way to share and represent knowledge, with focus on information retrieval; this notation was created to allow knowledge description that are equally processable by machines and humans[12]. TM is a standard (ISO 13250:2003) for the representation and interchange of knowledge, with an emphasis on the findability of information, it provides a standardised notation for interchangeably representing information about the structure of information resources used to define topics, and the relations between topics[11]. In a topic map, information is represented using topics, associations and occurrences:

topic is the basic element of a topic map, representing some subject (ex. persons, contries, organisations, software modules, etc.);
association connects two or more topics, definning a semantic relationship between the themes represented by those topics;
occurrence represents a relationship between topics and information resources relevant to them.

There are several notations for TM. The most usual is the standard XML-based interchange syntax called XML Topic Maps (XTM).

TM lacks a schema language, that defines the topics structure and constraints. One of the possible solutions is the ISO standard Topic Maps Constraint Language (TMCL)[1]; this proposition is still underdevelopment. XTche language, developed by Giovani Librelotto[12], is a concrete proposed intended to comply with the TMCL requirements.

Topic Maps proved in practise to be natural and easy to use, allowing the effective construction and the handling of semantic networks. However, the scientific comunity is nowadays more inclined to use the W3C standards mainly on account of RDF.

3.2 RDF and OWL

OWL was chosen to represent the ontologies for the MEC, but OWL is not a standalone technology, it benefits and uses many RDF and RDFS constructs, and is considered an extention of these languages. As such, we will take a deeper look into those three W3C recommendations.

RDF language was design as metadata model, but is largely used as a general method for modeling information. RDF metadata model is based upon the idea of making statements about resources in the form of subject-predicate-object expressions, called triples in the RDF terminology.

A RDF resource is any data or information source we want to describe. It can be anything from physical objects to web resources (ex. a city, a database, a web page, etc.). A resource is allways represented using a Universal Resource Identifier (URI) (the URI does not need to be on a web accessible path, nor has any normalisation rules; it will just suffice that its meaning is known by the reading application).

The subject, is the resource we are describing. The predicate or property denotes a characteristic and expresses a relation between the subject and the object, it is represented by a RDF resource. The value or object is represented either by a literal string or a resource[12]. As an example, the sentence "Ana lives in Portugal" in RDF, is the triple "Ana" (subject), "lives in" (predicate) and "Portugal" (object). This example is shown in Figure 3 using RDF/XML[4].

```
1  <rdf:RDF
2          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3          xmlns:pro="http://people.org/predicates#">
4
5          <rdf:Description rdf:about="http://people.org/Ana">
6                  <pro:lives_in>
7              <rdf:Description rdf:about="http://countries.org/Portugal"/>
8          </pro:lives_in>
9          </rdf:Description>
10 </rdf:RDF>
```

Fig. 3. RDF/XML example

RDFS is a RDF extension, that allows the definition of classes of resources, restrictions and properties over RDF, in a way that establishes the application vocabulary. Figure 4 illustrates this new abstraction layer over RDF.

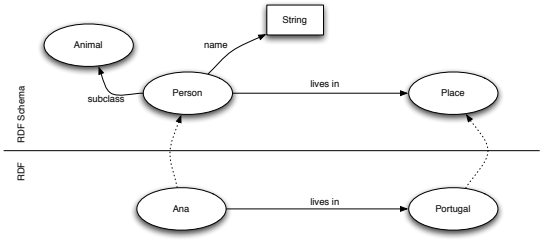

Fig. 4. A RDF statement and its corresponding RDF Schema

RDFS adds some constructs to the RDF language like, `rdfs:Class`, `rdfs:subClassOf`, `rdf:Property`, they allow the creation of a class hierarchy. Figure 5 shows the RDFS description[4] for the sample sentence used above (Figure 3).

RDFS has some limitations as a standalone ontology language because: there is no distinction between the language constructs and the ontology vocabulary; it does not allow to define class and property restrictions; it is too week to describe resources in detail[3,5,12].

[4] This example is in the RDFS abbreviated format, the extended format its much similar to the notation of RDF, both formats have however the same meaning

```
1  <rdf:RDF
2      xmlns:rdf= "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
4      xml:base=  "http://people.org/rdf#">
5
6      <rdfs:Class rdf:ID="Animal" />
7      <rdfs:Class rdf:ID="Person">
8          <rdfs:subClassOf rdf:resource="#Animal"/>
9      </rdfs:Class>
10     <rdfs:Class rdf:ID="Place" />
11
12     <rdf:Property rdf:ID="lives_in">
13         <rdfs:range rdf:resource="#Place"/>
14         <rdfs:domain rdf:resource="#Person"/>
15     </rdf:Property>
16     <rdf:Property ID="name">
17         <rdfs:range rdf:resource="rdfs:Literal"/>
18         <rdfs:domain rdf:resource="#Person"/>
19     </rdf:Property>
20
21         <Person rdf:ID="Ana">
22         <name>Ana</name>
23         <lives_in>
24             <Place rdf:ID="Portugal"/>
25         </lives_in>
26         </Person>
27 </rdf:RDF>
```

Fig. 5. An RDFS example coded in RDF/XML

The OWL was built on top of RDF and RDFS as a language for the represention of web ontologies[3]. This language was designed to be used by applications that process the information content instead of just presenting it to humans[10,12]. An OWL ontology includes class descriptions, along with there associated properties and instances, as well as related restrictions.

An OWL file, as seen in Figure 6, is opened by a namespaces declarations, followed by the `owl:Ontology` element; this element contains the ontology URI (line 10), generic information's about the ontology (`rdfs:comment` - line 11), version control (`owl:priorVersion`) and imported ontologies (`owl:imports` - line 12).

OWL uses the constructs `owl:Class` (lines 15 and 16) and `rdfs:subClassOf` (line 17) to represent classes and subclasses . Ontology relations are defined in OWL by the `owl:ObjectProperty` element (lines 20 and 23). Ontology attributes are defined by `owl:DatatypeProperty` (line 28); this element relates a OWL class to an XML Schema (XSD) datatype. In OWL, the individuals are created using the classes identifiers (lines 35 and 37).

The code presented in Figure 6 is a straitforward example, but there are some things to notice:

- the usage of the elements `rdfs:domain` (lines 22 and 30) and `rdfs:range` (lines 21 and 29) within `owl:ObjectProperty` to define the domain and range of a property;

```
1  <rdf:RDF
2      xmlns:="http://people.org/owl#"
3      xmlns:per="http://people.org/owl#"
4      xmlns:pla="http://places.org/places#"
5      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6      xmlns:owl="http://www.w3.org/2002/07/owl#"
7      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
8      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
9
10   <owl:Ontology rdf:about="http://people.org/owl">
11      <rdfs:comment>This document contains the class definition of people and
            animals, there properties and some instances.</rdfs:comment>
12      <owl:imports rdf:resource="http://places.org/places"/>
13   </owl:Ontology>
14
15   <owl:Class rdf:ID="Animal"/>
16   <owl:Class rdf:ID="Person">
17      <owl:subClassOf rdf:resource="#Animal"/>
18   </owl:Class>
19
20   <owl:ObjectProperty rdf:ID="lives_in">
21      <rdfs:range rdf:resource=pla:#Place"/>
22      <rdfs:domain rdf:resource="#Person"/>
23   </owl:ObjectProperty>
24   <owl:ObjectProperty rdf:ID="address_of">
25      <owl:inverseOf rdf:resource="#lives_in"/>
26   </owl:ObjectProperty>
27
28   <owl:DatatypeProperty rdf:ID="name">
29      <rdfs:range rdf:resource="xsd:#string"/>
30      <rdfs:domain>
31            <owl:Class rdf:about="#Person"/>
32      </rdfs:domain>
33   </owl:DatatypeProperty>
34
35   <Person rdf:ID="Ana">
36      <lives_in>
37         <pla:Place rdf:ID="Portugal"/>
38      </lives_in>
39      <name>Ana</name>
40   </Person>
41 </rdf:RDF>
```

Fig. 6. An OWL example, using RDF/XML syntax

- the element `owl:inverseOf` (line 25) allows the definition of inverse properties;
- the declaration of the individual `Portugal` (line 37) is inside another individual `Ana` (line 35), witch is completely valid and serves to show the OWL syntax freedom;
- the individual `Portugal` (line 37) does not belong to a local class (`Place`), in fact this individual belongs to the ontology `http://places.org/places`, this happens because in OWL it is valid to extend existing ontologies in other files.

OWL also offers various other elements such as `owl:Restriction`, `owl:cardinality`, `owl:intersectionOf` and `owl:disjointWith`, they allow restrictions, cardinality, class intersection and disjoint classes respectively. They, along with

others improve the task of ontology modeling. Regarding properties there are also the elements `owl:TransitiveProperty`, `owl:SymmetricProperty`, `owl:FunctionalProperty` and `owl:FunctionalInverseProperty` that are specialised types of `owl:ObjectProperty`.

It can also be noticed that in Figure 6 the `address_of` property is not present in the individual `Portugal`, but it could be inferred from its inverse (the `lives_in` property) by using a OWL reasoner. A reasoner is a tool that produces valid logical conclusions about an ontology.

OWL has three sub-languages, OWL Lite, OWL DL, OWL Full. We used OWL DL in our project. OWL lite is the most simple as it uses only a subset of OWL constructs, therefore is very simple to implement. OWL Full is the full language without any semantic restrictions, its very close to RDFS. OWL DL is an intermediate solution, that is very expressive, but also maintains a computable completeness and decidability; it includes all the language constructs but they can only be used under certain conditions[10].

4 Using Ontologies to Create Museum Rooms

The MEC needs a simple and organised way to show its assets to the public. For that purpose, we created theme oriented museum exhibition-rooms, or as we call them, *views*. Those views are described in a rigorous way by means of semantic networks, this is, concept maps. This is a new approach, that uses related information gathered from the various information sources rather than just showing each one of them (Figure 7). This approach allows the user to browse in an interactive and differentiated way through the information, and also allows to create more than one perspective over the same information. Views are

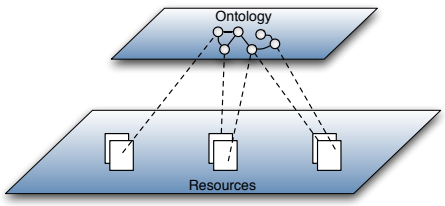

Fig. 7. The ontology and resources planes interaction

represented by ontologies. So a OWL ontology[5] is defined for each view created. Each view is intended to focus in a particular aspect or theme, for instance:

– Emigrants by date: view that shows, for an given time interval, all the known emigrants and associated data.

[5] For each view there is only one ontology, however many OWL files per view can exist (the definition file, and many files with individuals).

– Event Surroundings: taking as input an event in the life of a given emigrant (ex. departure), this view will show information about the physical and social surrounding environment at the epoch and place where the event occurred.
– Emigrant's Places (V1): view that reports on the different places of emigration cycle (birth, departure, arrival, etc.).

We will now take a closer look at this last view and its specification.

V1 shows the main events of the emigrant's life and their location, along with images of the events and places. That information is retrieved from passport petitions, passport records, birth certificates, criminal records, events records, and images, postal cards and photos, as can be seen in Figure 1.

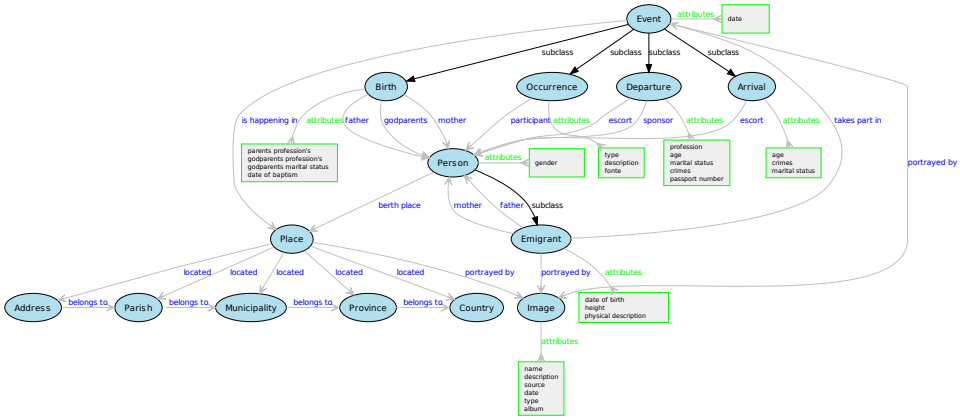

Fig. 8. V1 ontology

Figure 8 shows the ontology specification, i.e, its classes, its properties[6] and its attributes. In a global analysis of the ontology, it is clear that the `Emigrant` class is the centrepiece, followed by the `Event`, `Place` and `Image` classes. This four classes represent V1 main idea: *the emigrant, his life events, where those events happened, and the images of the emigrant, the events and places.*

5 MusVis - an ontology navigation system for the museum visitors

In this section, a short description of our ontology navigation system (MusVis) is presented. MusVis is more than a browser, will be a modular software system

[6] Inverse properties are not represented in the image

that allows one to gather MEC basic information from the various sources, build an ontology with that information, and create an web-based navigator over the ontology. MusVis architecture is presented in Figure 9; it is made up of four modules, each with a specific data transformation task. Now we will take a closer look into each module. As seen in section 2, the MEC assets are multifaceted so

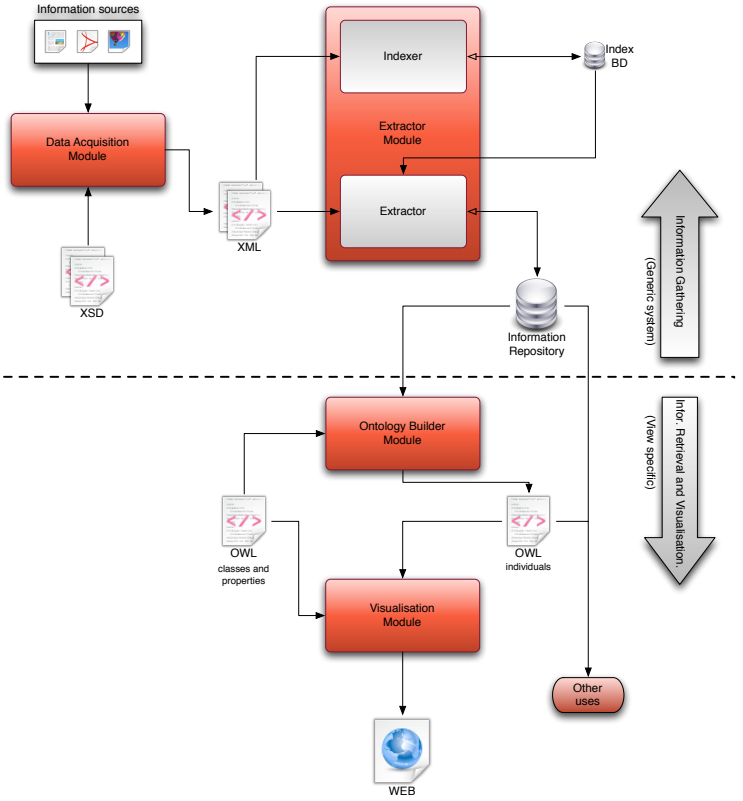

Fig. 9. The MusVis architecture

a common digital file format for the various documents presented in Figure 1 was needed—XML markup system was the principle adopted. As already told, an XML-Scheme was defined for each type of document, setting up the XML tags that can be used each time a new document has to be added to the Museum's digital repository. The Data Acquisition module assists in this task; it is similar to a graphical text editor, and allows an historian to translate and markup documents. The markup is done by means of colours and symbols, always hiding the XML backend from the historian. ThisThese module is based upon previous work developed inside our research group [8,7].

The Extractor module is responsible for indexing and organising in a common repository all the information obtained from the XML files created in the previous module. The data model chosen for the information repository allows a straightforward implementation of the next module, the Ontology Builder, as an information retrieval system.

The Ontology Builder module is responsible, as said above, for the creation of MEC ontologies that support each museum's room. This module is view specific, which means that needs a different instantiation (although with a similar behaviour) for each view. Taking into account the ontology definition (classes, properties, etc.) and the view specific input, it extracts all the information relevant to set up that view, and creates the ontology individuals with the information repository data. These last three modules update the ontologies with new data, every time a XML file is added to the system; this allows the new data visualisation by the last module, without any configuration, since it retrieves the information from the ontologies dynamically.

Fig. 10. A MusVis screenshot

Finally, the Visualisation module traverses the semantic network (corresponding to the ontology), and exhibits its content (each ontology component) to the user. It accomplishes this by using the ontology data and a set of standard and custom (one for each ontology) presentation rules to generate a set of dynamic web pages (JSP) for each ontology. Those pages are entity-oriented, and centred on each individual attributes and connections. Figure 10, a screenshot of V1, illustrates such a webpage; it is precisely the webpage generated by MusVis from the V1 ontology seen on figure 8.

The system modules are being implemented using Java. JAXP framework is used for XML processing (Saxon implementation for XPath); and the Jena OWL

framework with Pellet as a reasoner, for RDF/OWL processing. Sqlite is used for the database. Working environments are: Protégé for OWL editing; XMLSpy to create XSD and XML documents; and Eclipse for Java programming.

6 Conclusion

Along this paper the idea that exhibition rooms of the virtual Emigration Museum, are no more then structured views over the the museum digital archive of documents (its assets) was defended. Although it was assumed along the paper that the repository is made up from XML documents, if a subset of them is supported in databases or whatever digital format, the approach can be the same. OWL documents can be represented in it's XML standard notation or we can use a database[7].

That perspective allows to systematically extract the information from data sources and automatically build the OWL ontology that formally describes the meaning of each view (by other words, the content of each room). An OWL navigator, general purpose or a specific one, can then be used to implement the visitor interface.

Future work goes in two directions. On one hand, more implementation work should be done to finish the first and last modules, the Data Acquisition and the Visualiser. Additionally, some extra analysis must be made to improve the automatisation degree. After that, MusVis can be deployed and real tests carried on to measure its performance, and effective impact and usability. On the other hand, other views should be specified, in order to build the respective ontologies. At this moment (without a full implementation) a OWL vs TM, realistic comparison can't be achieved.

References

1. Document Description and Processing Languages. `http://www.isotopicmaps.org/tmcl/`.
2. Museu da Emigração - O que somos? `http://www.museu-emigrantes.org/ficha_tecnica.htm`.
3. OWL Web Ontology Language Guide. `http://www.w3.org/TR/owl-guide/`.
4. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts/`.
5. W3C Semantic Web Activity. `http://www.w3.org/2001/sw/`.
6. O. Corcho and A. Gomez-Perez. A Roadmap to Ontology Specification Languages. *Knowledge Engineering and Knowledge Management: Methods, Models, and Tools: 12th International Conference, Ekaw 2000 Juan-Les-Pins, France, October 2-6, 2000 Proceedings*, 2000.
7. Flávio Ferreira, Hugo Pacheco, and José Vilas Boas. Pda's no levantamento de informação em arquivos históricos. Technical report, Universidade do Minho, 2007.

[7] With Jena we can access a OWL in a database with the same API as a OWL serialised as RDF/XML

8. Rafael Félix. Sistemas de Digitalização e Anotação de Documentos. Technical report, Departamento de Informática, Universidade do Minho, 2002. Relatório de Projecto (Opção III).
9. T.R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
10. I. Horrocks and P.F. Patel-Schneider. Reducing OWL entailment to description logic satisfiability. *Proc. of the 2003 International Semantic Web Conference (ISWC 2003)*, pages 17–29, 2003.
11. ISO/IEC. *ISO/IEC JTC1/SC34: Topic Maps Constraint Language* .
12. Giovani Rubert Librelotto. *[Topic maps: da sintaxe à semântica.* PhD thesis.
13. G.R. Librelotto, J.C. Ramalho, and P.R. Henriques. Topic maps aplicados ao sistema de informação do Museu da Emigração. 2006.

A SPARQL Query Engine over Web Ontologies using Contextual Logic Programming

Nuno Lopes and Salvador Abreu

Universidade de Évora

Abstract. Querying is one of the key aspects in the Semantic Web. SPARQL, a W3C recommendation, attempts to become the standard Web query language. The XPTO system is capable of representing and querying ontologies described in the OWL language using Contextual Logic Programming. Here is presented a component of the XPTO system that enables answering SPARQL queries.

1 Introduction

The XPTO[1] system, enables accessing OWL [DSB+04] (Web Ontology Language) ontologies from within a Contextual Logic Programming environment, namely GNU Prolog/CX. It also allows to integrate these ontologies in the running program enabling using them as a part of the computation. The XPTO system is further detailed in [FLA07,LFA07].

Here is described a component of the system that enables it to answer queries formulated using the SPARQL query language and thus presenting the possibility of making the system visible to the World Wide Web though a Web Service.

2 SPARQL Query Engine

The presented component is dedicated to SPARQL query resolution: it allows for the possibility of querying the internal representation of the ontology using the SPARQL query language. It is split into 3 parts: the parser, the query resolution and the returning of the results as XML. The implemented SPARQL parser follows the specifications of the language defined in [PS06] and the results are returned in XML as specified in [BB06].

The SPARQL query is parsed to produce a GNU Prolog/CX context representing the query that is then activated to calculate the output and display the resulting XML.

[1] XPTO is a recursive acronym that stands for *XPTO Prolog Translation of Ontologies*.

2.1 Representation of a SPARQL query

The query representation process consists of a SPARQL parser that converts a query defined in the SPARQL syntax [PS06] into a GNU Prolog/CX context. This context represents the entire query and can be used to return the results. The execution of the generated context, triggered by a default message, that will bind the variables present in the query and show the results.

Element representation The representation of query elements, such as SPARQL variables and resources, is presented next.

Variables The SPARQL variables are represented as Prolog variables. Thus, once the result is calculated, the query resolution system simply binds the corresponding variable to return the results.

There are some other structures needed to display the results: it is necessary to store the name of the variable in the SPARQL query in order to return it in the results. To achieve this, all the variables in the SPARQL query are stored in a list that will be the argument of the unit `vars/1`. The elements of this list are in the format `SparqlVariableName = PrologVariable`. `SparqlVariableName` corresponds to the name of the variable in the SPARQL query and `PrologVariable` is the Prolog variable assigned to represent it. `PrologVariable` will start unbound and, as the context is resolved, will be instantiated with the solutions it may have. SPARQL variables appear in the generated context for the query using the `PrologVariable` representation, enabling a simple access to the value of the variable or direct instantiation of an unbound variable. This representation can be seen in the GNU Prolog/CX context shown in Figure 2.

Resources Resources are represented using Prolog terms or atoms. If the resource is an absolute IRI (delimited by `'<'` and `'>'`) it is represented as an atom containing the entire IRI. If it corresponds to a prefixed name (a prefix label and a local part separated by a colon `':'`) it is represented as Prolog compound term of arity 2 with the functor ':'. The arguments of the term are the prefix name and the local part respectively. If the prefix name is empty the atom `''` will be used to represent it.

Query representation A SPARQL query is represented as GNU Prolog/CX context whose structure is similar to the structure of the query. The elements of the query can be clearly identified in the representation: `select`, `where` as well as the *Modifiers* (if there are any present in the query).

The example query presented in Figure 1 is a `select` query containing two basic graph patterns with a shared variable: `?t` and the context produced by the parser is shown in Figure 2.

A context is represented by a Prolog list containing unit names. The first element of the list will be the unit that first tries to evaluate the goal upon execution. The individuals and property values are gathered from the units in

```
1 SELECT
2        ?flavor ?color
3 WHERE {
4    ?t :hasFlavor    ?flavor  .
5    ?t :hasColor      ?color  .
6 }
```

Fig. 1. Query example (simple select)

```
1 [ where([set([
2              triple(A,hasFlavor,B),
3              triple(A,hasColor,C) ])
4       ]),
5  select([flavor=B,color=C]),
6  vars([flavor=B,color=C,t=A]) ]
```

Fig. 2. Generated context (partial) for the query in Figure 1

a higher position in the context. This way in the final positions of the list are found the units `select/1` (in the case of a select query) and `vars/1`. These units contain in their arguments a list of variables and will allow any unit in the context to access either all the variables in the context or the selected variables.

2.2 SPARQL resolution system

The core unit in the query resolution process is the `triple/3` unit, which is responsible for instantiating the variables in the query by accessing the data.

This unit can be redefined in order to access data available from different sources. It generates one query to the XPTO system for each property that appears in the SPARQL query. The pattern in line 2 of Figure 2 (page 3) will generate the following query:

```
/> property(hasFlavor,F) :> item(I).
```

The argument of the `item/1` goal will be instantiated with the name of the individual. The arguments of the unit `property/2` are the name of the property being queried and the value of that property for the returned individual. Using the `property` unit to query the internal representation has the advantage of being able to perform the query using a Prolog variable in the position of the property name, thus enabling to return all the properties of the individual or querying the property name based on the property value.

3 Conclusion

The developed component acts as a translator: mapping SPARQL queries to a representation of the query that is based on CxLP units. In this representation each operator and each part of the SPARQL query corresponds to a unit occurring in the context and the complete query is represented by a context that combines the available units.

There are still further improvements necessary such as:

Complete the SPARQL support: Currently not all of the SPARQL constructors are implemented

Adopt the latest SPARQL specifications: The SPARQL system was developed against the specifications of 6 April 2006 in which SPARQL was considered W3C Candidate Recommendation.

References

[BB06] D. Beckett and J. Broekstra. SPARQL Query Results XML Format. W3C recommendation, W3C, April 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-XMLres-20060406/.

[DSB+04] M. Dean, G. Schreiber, S. Bechhofer, Frank van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, Feb 2004. http://www.w3.org/TR/owl-ref/.

[FLA07] Cláudio Fernandes, Nuno Lopes, and Salvador Abreu. On querying ontologies with contextual logic programming. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWL: Experiences and Directions 2007*, volume 258 of *CEUR Workshop Proceedings ISSN 1613-0073*, June 2007.

[LFA07] Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Contextual logic programming for ontology representation and querying. In Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus, editors, *2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services*, September 2007.

[PS06] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, W3C, 2006. Available at: http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.

Generating Semantic Networks to the PubMed

Giovani Rubert Librelotto[1], Mirkos Ortiz Martins[1], Henrique Tamiosso Machado[1], Juliana Kaizer Vizzotto[1], José Carlos Ramalho[2], and Pedro Rangel Henriques[2]

[1] UNIFRA, Centro Universitário Franciscano, Santa Maria - RS, 97010-032, Brasil
{librelotto, mirkos, htmachado, juvizzotto}@gmail.com
[2] Universidade do Minho, Departamento de Informática
4710-057, Braga, Portugal
{jcr, prh}@di.uminho.pt

Abstract. This paper presents a topic map approach to PubMed in order to create a knowledge representation for this information system. PubMed is a free search engine that gives very full coverage of the related biomedical sciences. With more than 17 millions of citations since 1865, PubMed users have several problems to find the papers desired. So, it is necessary to organize these concepts in a semantic network. To achieve this objective, we use the Metamorphosis system, choosing the keywords from MeSH ontology. This way, we obtain an ontological index for PubMed, making easier to find specific papers.

1 Introduction

Daily, a lot of data is stored into PubMed system. There is a problem that organization requires an integrated view of their heterogeneous information systems. In this situation, there is a need for an approach that extracts the information from their data sources and fuses it in a semantically network. Usually this is achieved either by extracting data and loading it into a central repository that does the integration before analysis, or by merging the information extracted separately from each resource into a central knowledge base.
Topic maps are an ISO standard for the representation and interchange of knowledge, with an emphasis on the findability of information. A topic map can represent information using topics (representing any concept), associations (which represent the relationships between them), and occurrences (which represent relationships between topics and information resources relevant to them). They are thus similar to semantic networks and both concept and mind maps in many respects. According to Topic Map Data Model (TMDM) [GM05], Topic Maps are abstract structures that can encode knowledge and connect this encoded knowledge to relevant information resources. In order to cope with a broad range of scenarios, a topic is a very wide concept. This makes Topic Maps a convenient model for knowledge representation.
This paper described the integration of data from PubMed information system using the ontology paradigm, in order to generate an homogeneous view of those

resources. PubMed is introduced in section 2. This proposal uses an environment, called Metamorphosis (section 3), for the automatic construction of Topic Maps with data extracted from the various data sources, and a semantic browser to navigate among the information resources – it is described in section 4. The section 5) presents the concluding remarks.

2 PubMed

PubMed [oM07] is a free search engine that provides very full coverage of the related biomedical sciences, such as biochemistry and cell biology. It also offers access to the MEDLINE database [oM06] with citations and abstracts of biomedical research articles.

The PubMed core subject is medicine and its related fields. It is offered by the United States National Library of Medicine as part of the Entrez information retrieval system. The inclusion of an article in PubMed does not endorse the article's contents, as other indexes. Nevertheless, many PubMed citations contain links to full text articles which are freely available, often in the PubMed Central digital library.

MEDLINE database covers over 4.900 journals published around the world primarily from 1966 to the present and is composed of more than 17 millions of citations. Information about the journals indexed in PubMed is found in its Journals Database, searchable by subject or journal title, Title Abbreviation, the NLM ID (NLM's unique journal identifier), the ISO abbreviation, and both the print and electronic International Standard Serial Numbers (pISSN and eISSN). The database includes all journals in all Entrez databases. A PubMed entry includes among other information the following details: *PubMed identifier*, *Authors' name*, *Title*, *Journal*, *Publication date*, *Language*, and *Mesh terms*.

The PubMed database consists of three tiers of software. At the bottom is a database management system (DBMS) that manages a collection of facts. At the top is the web browser that transmits requests for data to the database and renders the responses as web pages. In the middle is a software layer that mediates between the DBMS and the web browser to turn data requests into database queries, and to transform the query responses into hypertext mark-up language (HTML).

The PubMed data structure is composed of citations metadata. Each citation has the same structure. The main part of its schema can be formalized by the following context free grammar:

```
MedlineCitation     ==> PMID, DateCreated, DateCompleted, Article,
                        MedlineJournalInfo, ChemicalList,
                        CitationSubset, MeshHeadingList
Article             ==> Journal, ArticleTitle, Pagination,
                        Abstract, Affiliation, AuthorList,
                        Language, PublicationTypeList
Journal             ==> ISSN, JournalIssue, Title
JournalIssue        ==> Volume, Issue, PubDate
PubDate             ==> Year, Month, Day, Hour?, Minute?, Second?
MedlineJournalInfo  ==> Country, MedlineTA, NlmUniqueID
ChemicalList        ==> Chemical+
```

```
Chemical            ==> RegistryNumber, NameOfSubstance
MeshHeadingList     ==> MeshHeading+
MeshHeading         ==> DescriptorName, QualifierName?
AuthorList          ==> Author+
Author              ==> LastName, ForeName, Initials
PublicationTypeList ==> PublicationType+
```

PubMed files are intended for automatic processing and therefore available in XML format. Each set of 30.000 PubMed citations is stored as an XML instance defined by a DTD. Notice that the context free grammar above was obtained direct and systematically from the PubMed DTD.
For these reasons, it was defined an XML Schema to PubMed files. The view of this structure is shown in figure 1.

3 Metamorphosis

The main idea behind Metamorphosis is close the gap between Topic Map technology and its users. Metamorphosis is being developed to become a Topic Map workbench easy to use and accessible to a common user (we are not there yet). Figure 2 shows the usage scenario proposed in this paper. It illustrates some of the interaction between the system components, information resources and users.

1. Metamorphosis Repository (MMRep) is the central component that takes care of Topic Map storage and management. All the other components interact with MMRep.
2. Topic Map Discovery (TMDiscovery) is a Topic Map driven browser that allows users to navigate inside the Topic Maps stored in MMRep.
3. Topic Map Extractor (Oveia) automates the task of Topic Map harvesting; it enables the user to specify the extraction task and generates a Topic Map in XTM syntax that can be uploaded into MMRep. Oveia implements some extraction mechanisms with which is possible to populate an ontology.
4. Information resources that we want to access.
5. Web interface driven by a topic map stored in MMRep that provides access to information resources.

Metamorphosis can be used to prototype web interfaces or to expose information systems on the web. To do this the user only needs to specify a topic map for each view he wants. Information integration is accomplished by concept integration in the topic map: to integrate two information systems we need to specify the two sets of concepts in the same topic map and specify the associations that will materialize that integration.
In the next sections we are going to discuss the main components of this workbench prototype: Metamorphosis Repository, Topic Map Discovery, Oveia and XTche.
This way, Metamorphosis let us achieve the semantic interoperability among heterogeneous information systems because the relevant data, according to the desired information specified through an ontology, is extracted and stored in a

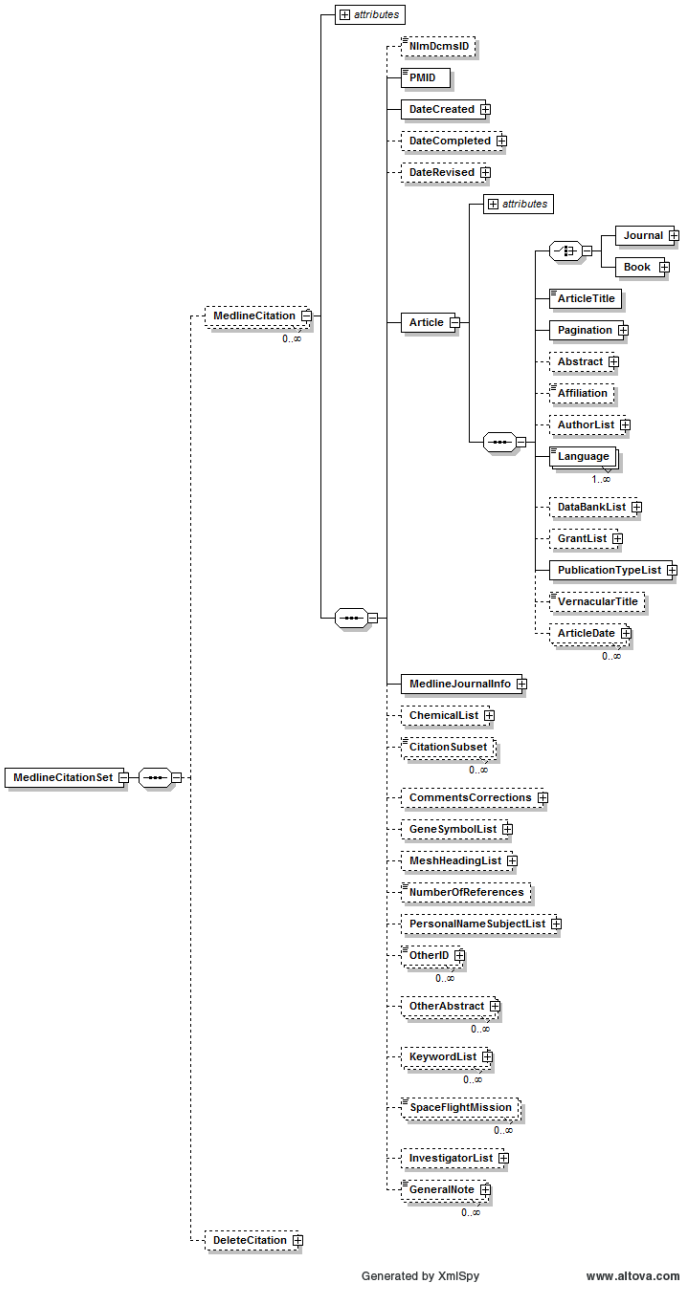

Fig. 1. PubMed's XML Schema

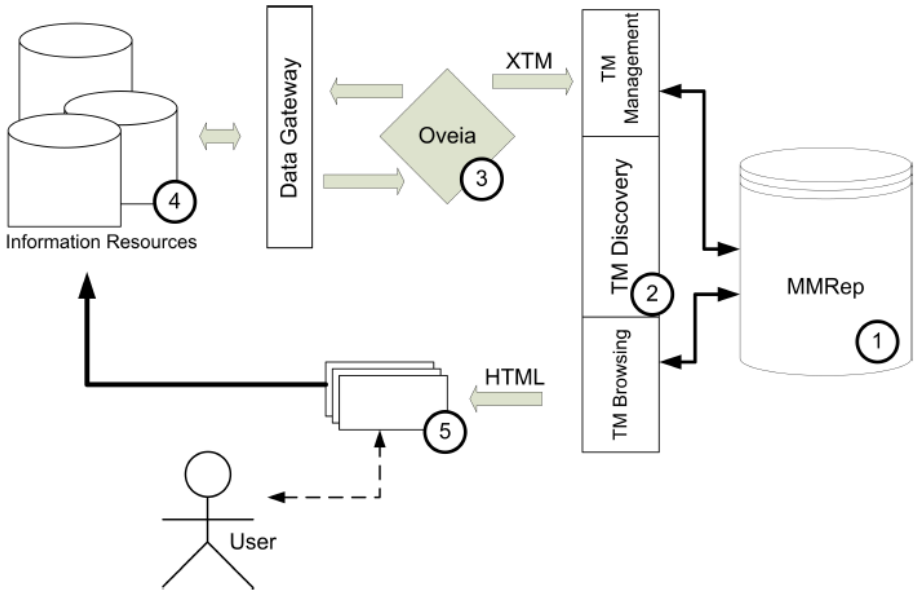

Fig. 2. Metamorphosis Functional Diagram

topic map. The environment validates this generated topic map against a set of rules defined in a constraint language. That topic map provides information fragments (the data itself) linked by specific relations to concepts at different levels of abstraction. Note that not all data items need to be extracted from the sources to the Topic Map. We only extract the necessary metadata to build the intended ontology. This ontology will have links to enable a browser to access all data items.
Thus the navigation over the topic map is led by a semantic network and provides an homogeneous view over the resources – this justifies our decision of call it semantic interoperability.

4 Topic Maps applied to PubMed

In order to obtain a semantic network from PubMed data, we divided this task in a few parts, as shown Figure 3.
In the first one, we created a relational database to store all contents of XML data obtained from PubMed data source. This database is generated according to the PubMed DTD using the Exult tool. An SQL script processes the result database to remove the redundant data and to erase several tables unnecessary. The final PubMed local database has XX tables.
To extract data from this database we use Metamorphosis [LRH06]. Metamorphosis has mechanisms to query the PubMed local database (Oveia) according to an ontology specification (XS4TM). Besides, there is a Web interface to make

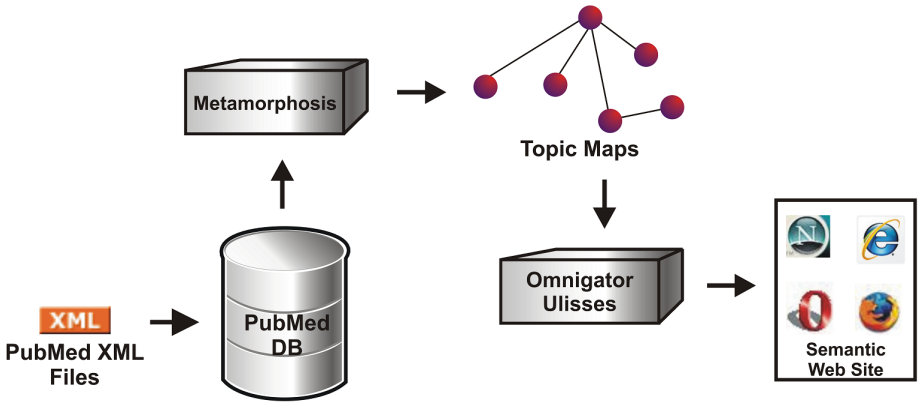

Fig. 3. The system's architecture

a query over the database. This interface has a text field to the user puts his query. After the query submission, Metamorphosis processes this string finding MeSH terms that describes the desired publications. These terms are structured in a RDF file [vAMMS07].
Using these MeSH terms as keywords, Metamorphosis searches articles that match with the user's query. This search processes includes several fields, like article's title, abstract, keywords, chemical substances, and MeSH terms. When an article satisfies the query, it will be mapped to a topic, as well its main fields, creating associations between them.
When the system receives a request, the required data will be collected from the selected databases at runtime. Then it will be further processed and converted into semantically relevant data by Metamorphosis. The resulting data has the standard XTM format. So, one of the advantages of this approach is that no new database will be created and no redundant data will be produced.
After end of the process, Metamorphosis has all topics and associations stored in its repository. The generated XTM documents can be then processed and displayed to the user by the presentation tier. This way, any topic maps navigator tool is able to browse the semantic network composed by these concepts. For instance, Ulisses [LRH04] allows the topic maps navigation over Metamorphosis' repository and XTM files (in last case, it is also possible to use Ontopia Omnigator [Ont02]). Information is interconnected within a huge knowledge network navigable in any direction.

4.1 Defining the Topic Maps concepts to PubMed citations

In order to define the topic map extraction from PubMed instances, the first task is to specify the main concepts (topic types). This way, the topic types in this domain are:

Article : each article is stored in a tag called $< MedlineCitation >$;

Author : the article authors are declared in $< Author >$;
Keyword : the keywords are MeSH terms. They are defined in $< MeshHeading >$;
Publication year : this metadata is in $//PubDate/Year$ path;
Journal : all journals are found in $< Journal >$ tag;
Language : the paper's language is define in $< Language >$;
Chemical substances : all chemical items cited in each paper are referenced in $< Chemical >$;

After the topics choice, the next step is the topic characteristics definition. Below we have the main ones:

Article : PMID (PubMed identifier), title, pagination, abstract, DOI, ...;
Author : initials, last name, middle name, and first name;
Keyword : descriptor and qualifier terms;
Journal : ISSN, title, abbreviation, volume, issue, and publication date;
Chemical substances : register number and substance name;

At this moment, all topics and its characteristics are defined. The final topic map definition step is the specification of association type. The main association types and some roles are described below:

- Author writes article;
- Keyword describes article;
- Article was published in an year;
- Article is published in a journal;
- Article is written in a language;
- Article refers to chemical substances;
- Author publishes in an year;
- Author writes paper in a language;
- Journal refers to the keywords;

Looking at a TM we can think of it as having two distinct parts: an ontology and an object catalog. The ontology is defined by what we have been designating as topic type, association type, and association role. The catalog is composed by a set of information objects that are present in information resources (one object can have multiples occurrences in the information resource) and that are linked to the ontology.
The PubMed's topic map ontology defined above (topic types, roles, and association types) and the topic characteristics are mapped to an XS4TM specification as can be seen in next subsection.
The XS4TM specification describing the PubMed scenario was defined in a XS4TM Web editor. Figure 4 shows a view of this specification, which defines seven topic types, nine association types, and eighteen role types.
On the left side, XS4TM presents the XML tree extracted from PubMed's XML Schema. The topic types from this case study are shown in the center window. To create a new topic type, the user just needs to make a simple drag and drop

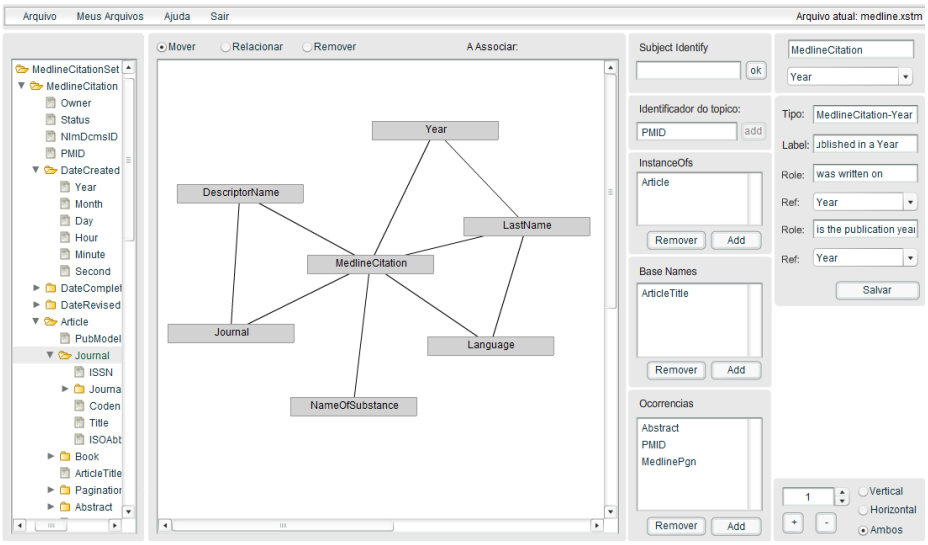

Fig. 4. PubMed's XS4TM Specification

from the XML tree. The topic characteristics are defined in the first column and the association characteristics are defined in the last column.
With the complete XS4TM specification, Oveia[3] can processes it. Its behavior can be described in four steps: (1) reads the XS4TM specification, (2) extracts the topics and associations from the query result set, (3) creates the topic map, and (4) stores it in the repository.

4.2 Browsing the topic map

When it will be browsing the semantic network obtained from PubMed local database, Ulisses gives the user an interface to navigate inside any of the stored topic maps. It allows the following interfaces:

Topic Maps : is the browser entry point and shows a list of all stored topic maps.
Ontology Index : gives you a structured view of a topic map showing the abstract concepts: topic types, association types, occurrence types, and association role types.
Individuals Index : lists all non-type topics in alphabetical order.
Full Index : lists all named topics.
Topic View : lists a subset of the available information about a topic; for the moment: the basenames, its type, all the associations it participates in together with the other members and their roles, internal occurrences and external occurrences.

[3] Oveia is a Metamorphosis' module

Association View : lists the names associated with the association and all its descendants.

Figure 5 a view to the topic of type article called Mycobacterium leprae anddemyelination. This page display every topic characteristics and its associations in a Web way, as well in a graph view.

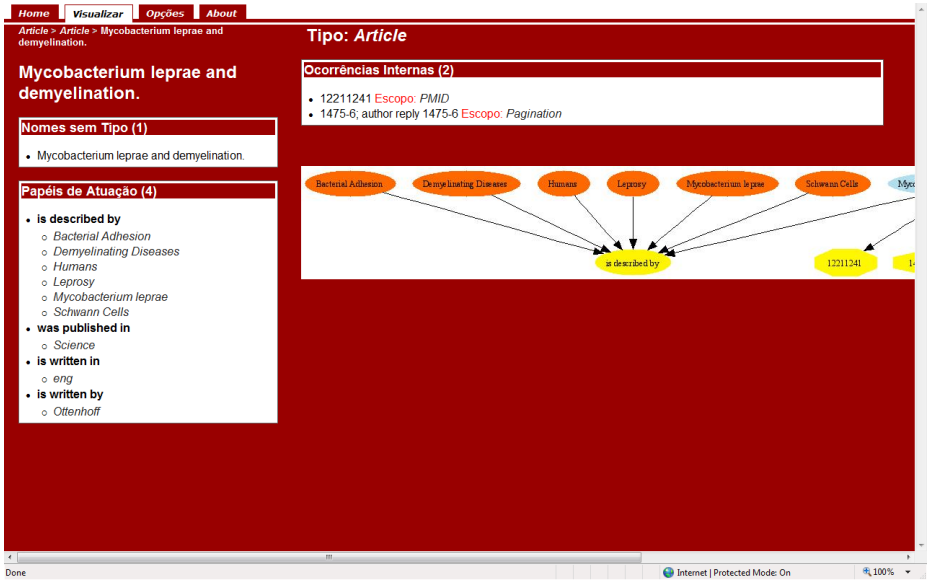

Fig. 5. Ulisses topic view

Creating a virtual map of the information enables us to keep the information systems in their original form, without changes. It is also possible to create as many virtual maps as the user wants generating multiple semantic views for the same sources.

5 Conclusion

This paper described the integration of data from PubMed information system using the ontology paradigm, in order to generate an homogeneous view of this resources. PubMed is a searchable compendium of biological literature that is maintained by the National Center for Biotechnology Information (NCBI).
The proposal uses Metamorphosis for the automatic construction of Topic Maps with data extracted from the various data sources, and a semantic browser to navigate among the information resources.
Topic Maps are a good solution to organize concepts, and the relationships between those concepts, because they follow a standard notation – ISO/IEC 13250 – for interchangeable knowledge representation.

In this paper we claimed that the semantic integration of PubMed documents is possible to achieve with Metamorphosis. In order to achieve this we proposed the following methodology:

1. Look at the information resources and decide how your conceptual view should look like;
2. Choose what information bits must be extracted in order to produce that conceptual view;
3. Specify the extraction task using Oveia;
4. Upload the generated Topic Map into MMRep;
5. Browse it with TMDiscovery and use this interface to access the information resources.

With this methodology the original information resources are kept unchanged and we can have as many different interfaces to access it as we want. We just have to create/generate/specify a Topic Map for each one.
As a future work we aim the integration of Topic Maps and MeSH headings minimizing *false hits* and saving time in the searches. Another project is to identify other useful – but frequently overlooked – features of the PubMed database.

References

[GM05] Lars Marius Garshol and Graham Moore. Topic Maps – Data Model. In *ISO/IEC JTC 1/SC34*. `http://www.isotopicmaps.org/sam/sam-model/`, January 2005.

[LRH04] Giovani Rubert Librelotto, José Carlos Ramalho, and Pedro Rangel Henriques. Ulisses: Um Navegador Conceptual para Topic Maps. In *XXXI Conferencia Latinoamericana de Informática*, pages 783–794, 2004.

[LRH06] Giovani Rubert Librelotto, José Carlos Ramalho, and Pedro Rangel Henriques. Metamorphosis - A Topic Maps Based Environment to Handle Heterogeneous Information Resources. In *Lecture Notes in Computer Science*, volume 3873, pages 14–25. Springer-Verlag GmbH, 2006.

[oM06] U.S. National Library of Medicine. MEDLINE – Fact Sheet. http://www.nlm.nih.gov/pubs/factsheets/medline.html, 2006.

[oM07] U.S. National Library of Medicine. PubMed. http://www.ncbi.nlm.nih.gov/sites/entrez?db=PubMed, 2007.

[Ont02] Ontopia. The Ontopia Omnigator, 2002. `http://www.ontopia.net/omnigator/`.

[vAMMS07] Mark van Assem, Véronique Malaisé, Alistair Miles, and Guus Schreiber. A Method to Convert Thesauri to SKOS. http://thesauri.cs.vu.nl/eswc06/, 2007.

SPARQL Back-end for Contextual Logic Agents

Cláudio Fernandes and Salvador Abreu

Universidade de Évora

Abstract. XPTO is a contextual logic system that can represent and query OWL ontologies from a contextual logic programming point of view. This paper presents a prototype of a SPARQL component for that system which is capable of mapping Prolog/CX to SPARQL queries.

1 Introduction

We present a back-end that aims to transparently merge the reasoning of the XPTO [1] [FLA07] [LFA07] internal knowledge base with external OWL [BvHH+05] ontologies, more exactly its Lite and DL sub languages, available from third parties, by means of the SPARQL [PS06] query language. To achieve this, we developed a system that provides functions for communicating with Web SPARQL agents for ontology querying purposes. It provides the system with the ability to pass a SPARQL query to an arbitrary SPARQL Web agent and get the solution, encapsulating the results as bindings for logic variables.

The presented back-end grants XPTO with capabilities for writing Prolog/CX [AD03] programs to reason simultaneously over local and external Web ontologies.

2 Mapping Prolog to SPARQL Queries

Although it can be viewed as a single independent component, the back-end purpose is to allow the XPTO-using programmer to query external and internal ontologies using the same query syntax and declarative context mechanics as the XPTO internal system. This will allow to transparently query internal and external ontologies and merge their results in the same program.

To achieve this level of functionality, we developed a Prolog/CX to SPARQL engine that satisfies the following requirements:

- Translate a partially bound Prolog/CX goal into SPARQL;
- Send the SPARQL query to the specified Semantic Web SPARQL service;
- Fetch the XML result file, parse it and return the solutions as Prolog variable bindings using the Prolog/CX backtrack mechanism to iterate over sets of answers;

A SPARQL query in the back-end environment is a Prolog/CX context execution. Figure 1 illustrates a definition of a back-end query.

[1] XPTO is a recursive acronym that stands for XPTO Prolog Translation for Ontologies.

```
QUERY := sparql(URI) /> P1 ... Pn :> ITEM

     URI  := URL
     P    := property(VALUE) or where(PROP, VALUE)
     ITEM := item(INDIVIDUAL)
```

Fig. 1. Back-End Query Definition

On the left side of the defined operator `'/>'` is specified the external agent and on the right side are the goals and query restrictions. The right side of the operator encodes the query that must be mapped to SPARQL. We translate that information into RDF triples, much in the same way a database is translated into triples, i.e, for each of the `n` stated properties about an individual, the back-end must translate it to `(n-1)` triples. The triples are extracted by the union of each **property** term of the right side and the **item** term, which represents the subject of the triple.

3 Examples and Query solutions

We now present an example. We will use the *XAK* - XML Army knife [Dod06] SPARQL service which implements the SPARQL Protocol for RDF and provides a SPARQL query engine for RDF data available on the Internet.

The *Wine* OWL DL ontology [2] is a sample ontology used in the OWL specification documents and will serve as the use case ontology in this paper. Among others, the `IceWine` class present in the ontology defines two properties: `hasBody` and `hasColor`.

Figure 2 shows an example of a back-end query that asks *XAK* to search the *Wine* ontology for all the individuals that have both of these properties.

```
1 ?- sparql('http://xmlarmyknife.org/api/rdf/sparql/') />
2    hasBody(A) :> hasColor(B) :> item(IND).
```

Fig. 2. Back-end Prolog/CX query to *XAK*

The Prolog/CX query in Figure 2 has no ground Prolog atoms besides the **url** that identifies *XAK*. It includes two specified properties, thus originating two RDF triples, one for each property. Figure 3 shows the correspondent SPARQL generated code.

[2] The ontology is accessible in http://www.w3.org/TR/owl-guide/wine.rdf

```
SELECT ?id ?hasColor ?hasBody
WHERE {
?id :hasColor ?hasColor.
?id :hasBody ?hasBody.
}
```

Fig. 3. Generated SPARQL for the query in Figure 2

After the SPARQL generation, the code is sent to *XAK*. In order to successfully communicate with it, the back-end must first encode the query as specified in the SPARQL Protocol for RDF [Cla06] and establish the values of a few parameters like the default graph to be queried. (Figure 4 shows the generated string that is sent over to *XAK*).

```
GET http://xmlarmyknife.org/api/rdf/sparql/query?default-graph-uri
=http://www.w3.org/2001/sw/WebOnt/guide-src/wine.owl&query=
PREFIX+:+<http://www.w3.org/2001/sw/WebOnt/guide-src/wine%23>
+select+?id+?hasColor+?hasBody+where+{?id+:hasColor+?hasColor+.+
?id+:hasBody+?hasBody}
```

Fig. 4. Back-end encoded query example

If a successful query response code is returned, a file with the solutions is received. This file is in the SPARQL Query Results XML Format [BB06] and includes one solution. This XML file is then parsed and the solution values are returned as bindings for Prolog variables as illustrated by the last lines in Figure 5.

```
?- sparql('http://xmlarmyknife.org/api/rdf/sparql/') />
   hasBody(A) :> hasColor(B) :> item(IND).

A ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#Medium'
B ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#White'
IND ='http://www.w3.org/2001/sw/WebOnt/guide-src/wine#SelaksIceWine' ? ;
(4 ms) no
```

Fig. 5. Prolog/CX query to *XAK* and the returned solution

The solution presents only one individual, **SelaksIceWine**, and the values **Medium** and **White** for properties *hasBody* and *hasColor* respectively. This means the whole ontology only has one individual that has those two properties defined.

4 Initial Assessment and Conclusions

The component presented in this paper is still work in progress. With the current capabilities, one can use the expressiveness of Logic Programming to perform basic queries to an ontology via a third party SPARQL Web Service. These capabilities can then be combined with other Prolog/CX data access forms for reasoning over different data repositories. For example, an application can indifferently use local data provided by the XPTO engine, external data through the SPARQL back-end and data residing in a relational data base accessed using ISCO [AN06].

Although no proper benchmarks were defined yet, the experimental work revealed no particular performance issues on the back-end side, which means that practically only the *XAK* connection will introduce some latencies. Note, however, that the generation of SPARQL is currently done in a *per-query* basis. One important feature to be implemented as future work is to allow the generation of SPARQL code for a composite (e.g. conjunction) of Prolog/CX queries.

References

[AD03] Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.

[AN06] Salvador Abreu and Vítor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Masanobu Umeda and Armin Wolf, editors, *Declarative Programming for Knowledge Management*, volume 4369 of *LNCS*, Fukuoka, Japan, 2006. Springer.

[BB06] Dave Beckett and Jeen Broekstra. SPARQL Query Results XML Format. Candidate recommendation, World Wide Web Consortium, 25 December 2006. http://www.w3.org/TR/rdf-sparql-XMLres/.

[BvHH+05] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. Recommendation, World Wide Web Consortium, 19 October 2005. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[Cla06] Kendall Grant Clark. SPARQL Protocol For RDF. Candidate recommendation, World Wide Web Consortium, 6 October 2006. http://www.w3.org/TR/rdf-sparql-protocol/.

[Dod06] Leigh Dodds. XML Army Knife. http://xmlarmyknife.org/api/rdf/sparql/query, 5 December 2006.

[FLA07] Cláudio Fernandes, Nuno Lopes, and Salvador Abreu. On querying ontologies with contextual logic programming. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *OWL: Experiences and Directions 2007*, volume 258 of *CEUR Workshop Proceedings ISSN 1613-0073*, June 2007.

[LFA07] Nuno Lopes, Cláudio Fernandes, and Salvador Abreu. Contextual logic programming for ontology representation and querying. In Axel Polleres, David Pearce, Stijn Heymans, and Edna Ruckhaus, editors, *2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services*, September 2007.

[PS06] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Candidate recommendation, World Wide Web Consortium, 25 July 2006. http://www.w3.org/TR/2006/CR-rdf-sparql-query-20060406/.

A Survey on Workflow Aspects in Content Management Systems

Pedro Pico, Alberto Rodrigues da Silva
pedro.coelho.pico@gmail.com, alberto.silva@acm.org

Instituto Superior Técnico / Universidade Técnica de Lisboa

Abstract. Content Management Systems (CMS) are software platforms that strongly contribute to make organizations more agile, flexible and dynamic concerning the management of their contents: business-oriented structured and no-structured information. A CMS's extra feature is Workflow support since it can allow task automation, ultimately increasing organizations productivity. While there are different kinds of Workflow platforms, this paper concentrates mostly in Content Management Workflow, analyzing key features like: Workflow definition, representation, instance management, content mapping and third-party application communication. Finally, it is also analyzed and discussed the workflow support in existing CMS such as Alfresco, Typo3, OpenADMS and Vignette.

Keywords: Content Management System, CMS, Workflow, Workflow Management System, Business Process.

1. Introduction

Workflow is an important concept and technology that is relevant within Software Engineering as well as Organizational Engineering. Nowadays, there are a relevant number of organizations that are increasingly embracing it. Due to the richness and abstraction of the concept, Workflow will only be mentioned in the Software Industry endeavor, meaning that it will only be applied to Software Application issues. Bearing that in mind, the Workflow Management Coalition, WFMC http://www.wfmc.org is a reference organization responsible for the definition of standard specifications regarding Workflow. The WFMC defines Workflow as "the computerized facilitation or automation of a business process, in whole or part" [1]. On the other hand, Marshak defines Workflow as "The automation of the processes we use every day to make our business through. A Workflow Application does automatically the sequence of actions, activities and tasks to run a process, including all the routing within the stages of each instance of a process, as well as the tools to manage the process itself" [2].

Automation and business processes are concepts that are mentioned on both definitions, leading to a first definition of *Workflow* as a "business process automation".

WFMC still introduces the concept of a *Workflow Management System*, as being a "system that completely defines, manages and executes the workflow through the execution of software whose order of execution is driven by a computer representation of the workflow logic" [1].

These definitions indicate the Workflow logic has to be represented in a formal language, so that running software may be able to build, manage and execute that workflow logic.

Workflow technologies may be applied with several purposes and application contexts, such as (1) system's integration support, (2) user interface and (3) content workflow in CMS (Content Management Systems).

Workflow support in systems integration. Information system's high-level abstractions and business service's interactions may be seen as a Workflow system [13]. Currently, Service Oriented Architecture (SOA) [14] is an example of this approach, in which technology is only a tool for orchestration of business processes and services. According to the SOA approach there should be a Service repository which agglomerates services – which according to the OASIS [32] organisation is defined as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.". These services, in their turn, communicate among them and with front and back-end applications, through a communication channel. To accomplish this, each service application must expose a programming or computing interface. Workflow plays the role of orchestrating the interactions amongst all these services. Microsoft Biztalk [17] and BEA WebLogic [31] are the leading application servers that support SOA in the industry.

Workflow support in User Interface Software Applications. The Workflow concept can also be applied to the end-user interface definitions for software applications. The links between interaction spaces (e.g. web page, window screen) are events that may be triggered by human or third-party applications. For example, the submission of a web form is a human interaction, while the presentation of a RSS (Really Simple Syndication) feed is a third-party application interaction.

To make the most out of this vision of software design and implementation, some companies have been publishing Interface Workflow Managers. This functionality is usually associated with the software developing process since the tools that support these features are usually within the Integrated Developing Environments, IDE, of the existing platforms. With Microsoft Visual Studio[18] the users have the ability to visually edit and see their application's interfaces as a tree of nodes, in which each node is an interaction space and the connection among nodes define the hierarchy of the nodes in the tree. On the other hand, Sun released in 2006 JAVA Web Studio Creator[19], supporting the same functionality for J2EE applications.

Workflow support in CMSs. Content management environments are also a relevant application for Workflows. These systems evolved from meta-applications and frameworks which were used to produce other applications. The urge for Documental Management Applications left only one step further what would later be known as Content Management Applications. The difference between these two is the object that is managed, while the former manage documents, the latter manage contents, which is an abstraction in which documents can be included.

In these environments a Workflow can be seen as the set of stages that content may assume since its creation until it is made available. Consider the following example: (1) a document is created and submitted to an application; (2) then it is approved in chain by a set of users; (3) until it becomes visible for all convenient users.

In this example the stage transitions are mainly triggered by human interaction. Users trigger stage transitions so that contents may evolve through the hierarchy structure of the organization, being progressively approved by users with more and more responsibilities. This type of Workflow systems has to make sure that if content is not approved by a user, it must return to its previous stage. Stage transitions also have to be able to trigger automatic actions. For instance, sending e-mails so users may be notified of pending decisions they have to make. Stage transitions may not only be triggered by human interaction, but also by third-party applications.

This paper analyzes and discusses the Workflow support that is required in enterprise applications, designed on the top of CMS platforms. In particular it identifies (1) Workflow concepts as well as their common use cases; (2) the way Workflows can be implemented; and (3) functionalities available for their end-users.

After analyzing and comparing relevant CMS, we propose a generic reference model, based on which we discuss them in what concerns their Workflow's features.

2. Content Workflow in CMS

This section describes the Workflow mechanisms supported by Content Management Systems: its background, its elements and its functionalities.

2.1. Technological Support Aspects

Content Management Systems (CMS)[20] promote the separation between contents and services. The latter are responsible for content's presentation, manipulation and access, while the former are the artefacts that are passed throughout services. Pictures, texts, links, news, videos, and documents are all examples of contents.

The main goal of Workflows in CMS is to provide a path for contents since their creation, until they are made available for other users to see – often defined as publication. The path can be described by a set of consecutive evaluations, usually referred as stages, in which users, defined by a specific business role may, or may not, approve contents. For example, in a newspaper, each article, after written by a journalist, must be reviewed by the journalist's supervisor. If the article is approved, it will go on to the next evaluation, and an upper supervisor will have to evaluate the article; otherwise it will return to the previous stage, and the journalist will have to rewrite the article. The evaluations will go on in chain until eventually some supervisor will approve the article's publication. A Workflow in a CMS is exactly the chain of evaluations that content undergoes since its creation until its publication. It is now assumed a Workflow has several stages. While each stage is associated with one role – set of users – who are responsible for evaluating the content – either approving it, or declining it. When a supervisor declines content, the content will return to the

previous stage so it can be reviewed or rewritten again. If, on the other hand, the content is approved it will move on to the next stage. If it achieves the last stage, the content will be published. While in the physical world articles are printed in paper, in the digital world contents may either be created or replace existing versions of the content. As an example it can be considered any page available in the *wikipedia* web site, *http://wikipedia.org*. In this web site each page may be replaced by a new version, as well as new pages may be created. Pages are the contents in this example.

The multiplicity of instances of the same content that have to co-exist lead to the need of content versioning. This happens since at least two versions of the same content will be needed. One that is published, and thereby the one all users can see, and another one which is evaluated by supervisors, and thereby available for supervisors to see. The former will be referred as the published version, while the latter will be referred as the draft version. This can lead us to the conclusion that content is characterized by its *version*.

Finally, there is also another aspect to be added, the possible existence of predefined actions that may occur every time a stage is achieved or departed from. A simple notification to the author every time the content he submitted is approved by a supervisor is an example of what a predefined action can be.

From the above observations it is concluded that in the CMS domain a Workflow has several stages. While each stage has one responsible *role* and a set of predefined actions that may be triggered either when the content arrives or leaves the stage. Finally the content, which is the artefact that goes through the stages of the Workflow, has to be identified by a version, since in the most simplistic scenario at least two instances of the same content will have to co-exist.

2.2. Content Types

In general terms, content management workflows may manipulate two types of contents: unitary and aggregators. The former are the basic cells manipulated by the CMS, the ones which are processed as a single unit and thereby elementary operations are made upon them. An image, a defined piece of html code, or a file, are examples of unitary contents. On the other hand, aggregator contents are sets of other contents,. The aggregators provide the “glue”, which connect its sub-contents. Examples of this type of contents are a list of links, a list of documents, or a custom content which aggregates one image, one link and one text.

The existence of aggregators brings up another issue: the hierarchy which is formed from multi-level aggregator contents. This issue will have an important impact in the Workflow implementation within the CMS, because, like it is previously stated, contents are the artefacts which go through stages of a Workflow. So far content was always assumed to be unitary, but if it is a set of contents, in a several level hierarchy, a much more careful approach has to be done.

In order to escape from the abstraction of this issue, a scenario will be drawn to materialize it, making it easier to understand.

In a web CMS context it may be considered the following hierarchy of contents: the web site, the page, and the unitary content, as illustrated in Figure 1. Each of these types of Contents has their own characteristics as described below.

The **web site** is the highest granularity content and it aggregates page contents. This type of content must be used in a limited way since a Web site should be stable and any content under it should be updated without having to update the whole Web site. Nevertheless the Website may be submitted to a Workflow when it is created, so that the process may be monitorized by the organization's website administrators.

The **web page**, or page for short, aggregate unitary contents. In spite of not being the most used content type within the Workflow mechanism, it is fairly more used than the portal, since creation, removal, edition and configuration of its attributes are all operations that are made in the portal life cycle. It is also important to make a distinction from Static to Dynamic pages because of the way their content is processed. The former are built in compile time and do not require any level of interaction, while the latter have contents that are created in *runtime* and that may require some level of interaction. Therefore static pages may be seen a unitary content which allows to edited a huge portion of html code; while dynamic pages can be seen as an aggregator content, since they provide several outputs to the user according to what was given as an input.

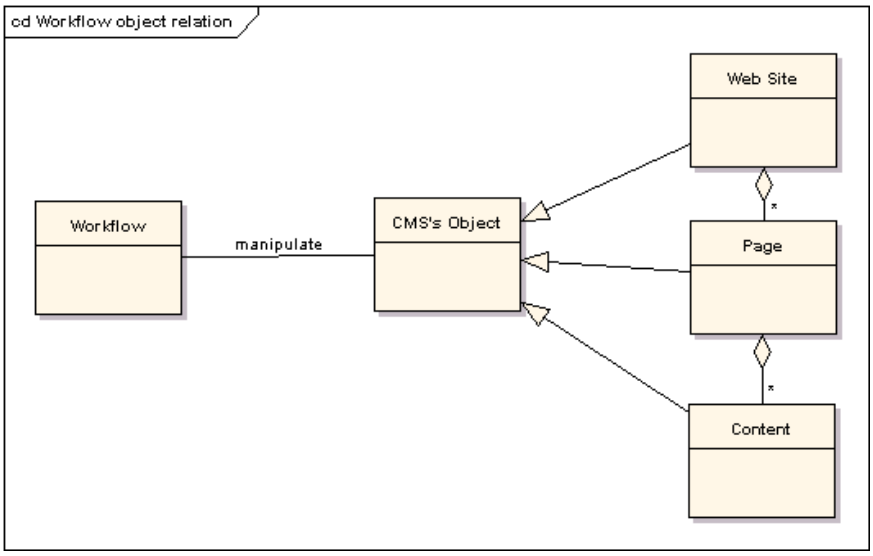

Figure 1: Workflow content relation

At last, the **unitary content** is the lowest granularity content in this type of Workflow. This content have its own attributes, some of them are even common to all contents – such as the name or page in which it is included. The attributes which vary from content to content are the ones that identify the content itself (e.g. an image has a filename, width and height, or a byte array; while an html element does only have the html source code; in its turn a link is formed by the text and the url). The unitary

content is the most often processed content type by the Workflow Manager because of its independence from content to content and from the pages in which they are included.

Considering the above scenario it is clear that content has to be able to follow a Workflow despite its type. It is also clear that content representation is definitely an important issue that a concrete implementation of Workflow in a CMS has to address since contents may vary on their building blocks, assuming complex hierarchies. Finally it can be stated that despite content representation is a custom problem each CMS has to address (and that is not the subject of this paper), all types of contents which desirably will be able to follow Workflows in that CMS have to be understood by its Workflow engine as contents, meaning they have to share the concept of content which is accepted by the Workflows of that CMS.

2.3. Workflow Elements

This section lists and summarizes the concepts explored so far, as well as defines other relevant elements. Figure 2 relates these concepts, forming a reference model, with concrete CMS Workflow elements.

Workflow. A Workflow has several stages. The concrete number of stages should be set when a Workflow definition is created. A CMS should allow managers to create Workflow definitions, giving them the opportunity to choose then the number of stages that that particular Workflow definition should have. In other words the number of stages of a Workflow should be dynamic since it allows different number of stages for different Workflow definitions. One tricky way of achieving the dynamic number of stages is to not determine it when the Workflow definition is created. In stead, each stage is responsible to determine the next stage, also determining when the Workflow should end. This type of workflow definition will be referred as not having a defined number of stages as in opposite of static and dynamic number of stages. The drawback of such solution is that the path between stages is not memorized and if the same stages are always used, they have to be defined every time a Workflow instance executes. Finally a Workflow definition with a static number of stages has the drawback of not allowing two different Workflow instances to run with different number of stages.

Stage. Every stage, as previously stated, has a supervisor role, who can determine if the content is accepted or declined. If the former is picked the next stage is reached, while the previous stage is reached if the latter is chosen. There are two special stages: the initial and the final. The initial stage is the one that starts the workflow, and the final is the one that defines when the workflow execution comes to an end. Every stage may have a set of associated operations that can fall into two categories: entry and exit operations. The former are executed when the content gets to a stage, while the latter are executed when the content leaves the stage.

Stage transitions. Empirically a transition is defined as a set of three elements: event, condition and operation. The event is external to the transition and when it happens it triggers the condition to be tested. The condition is the heart of the transition since it determines if the transition is executed. The operations are usually

performed if the condition is met. However events can also trigger Operations to be performed. Having this definition in mind, in CMS Workflows the events are content creation and edition, content approval (or disapproval), or messages from third-party applications. As to the conditions they consist of checking if the input given by the supervisor was an approving or disapproving instruction. When content is approved the condition is met and for example an operation of notifying the next supervisors may be executed.

Users. Users are responsible for approving or declining contents and so, firing events. Since content management systems deal with several users, they usually group users into roles, in which one role can have one or more users.

Content. The content, not regarding its type, is the object that runs through the Workflow. The content is needed to have a version so several instances of one content may co-exist, in order to supervisors and regular users may see different versions of the content.

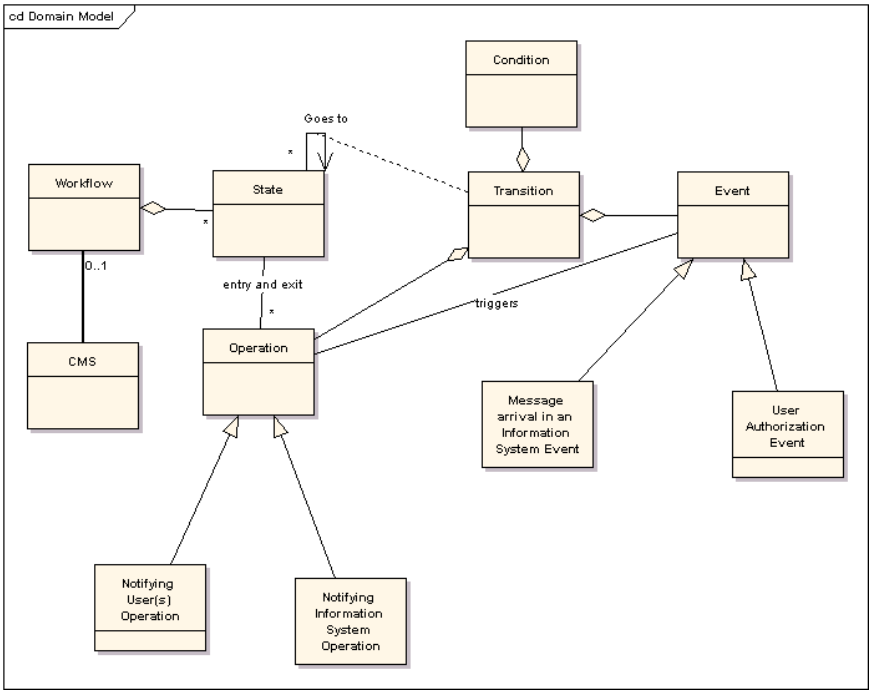

Figure 2: CMS's Workflow Support – Domain Model

2.4. Functionalities

Figure 3 presents the use case model that reveals Workflow's main functionalities in a CMS. The actors present in the diagram are: (1) Registered User (URegistered), (2) Workflow Manager (UWorkflowManager) and (3) External Information Systems (IS-External). The UWorkflowManager can manipulate workflows, associate them to contents, as well as manage workflow instances. URegistered edit contents, receive notifications, authorize contents and monitor workflow instances. Finally, IS-External can notify and be notified by the Workflow Management System.

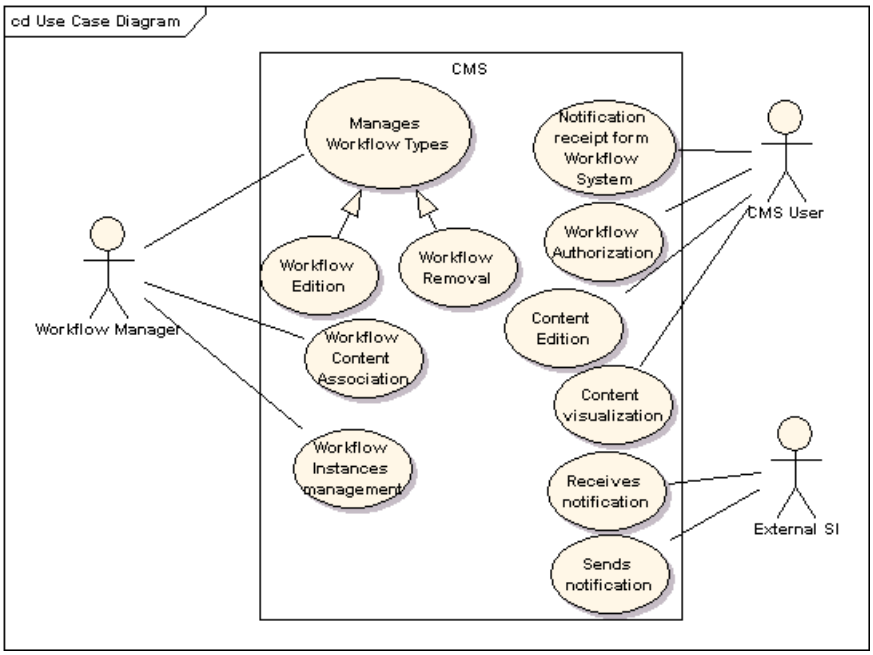

Figure 3: CMS's Workflow Support – Use Case Model

We set the focus to the main functionalities a Workflow Manager must have in order to support Content Workflow. The functionalities to be mention are only the ones that directly relate to the Workflow Engine.

Workflow definition: When performing it the number of stages should be set, as well as the responsible role for each stage.

Association between Workflow and content: With this functionality the user assigns a content to the Workflow.

Workflow edition: Existing Workflow definitions may be able to change.

However one should be very careful on how to handle existing executing instances of Workflows in order to avoid the loss of their contents.

Workflow deletion: it should be possible to delete a Workflow definition. The same situation happens as when Workflow definitions are edited, so the user must explicitly choose what to do to existing workflow instances.

Workflow Instance Management: Allowing administrators to stop workflow instances that didn't come to an end and that may be in a dead lock or starvation situation.

3. CMS with Content Workflow Support

Bearing in mind the concepts and functionalities discussed in section 2, four CMS were chosen to be analyzed and discussed and, consequently, to give us a better understanding of the problems in consideration. The criteria for choosing these CMS systems are the following: (1) provide Workflow functionalities; 2) distinct among them in which refers to being open-source or commercial; 3) distinct among them in which concerns their main purpose;

It should be stressed that it was surprisingly hard to find CMSs supporting Workflow technology.

3.1. Alfresco

Alfresco [22] is an open-source CMS focused on documental management. Its highlight features are the version control, role support, content transformation, search engine and navigation either in the file system or via Web.

The contents which are handled by Workflows are exclusively unitary contents, meaning that it is not possible to apply workflows on aggregation contents. There may be three predefined stages: draft, review and published. The supported actions are authorization (approval or denial) by users and code actions (javascript). The latter are executed as soon as a stage is reached. There are also discussion forums, associated to each Workflow, so that the stakeholders may exchange opinions about the evolution of the given content. The notification system is based on the e-mail as well as through a module of pending actions that each user has access to, which shows the actions the user may execute. There is an administration console in which it is possible to see the stage, each Workflow is currently at. It is also possible to cancel the workflow in this administration console.

3.2. Typo 3

Typo3 [23] is an open source PHP- based CMS. For Typo 3, a Workflow consists of a name, a user, and a deadline. There is the possibility of notifying users when the workflow starts. The only contents supported are also unitary contents. The user who is responsible for one stage determines the next transition – not defined number of stages - , whether accepting or declining the received content. The user may also schedule the deadline of the next stage. When the final stage is reached and the user

associated with that stage accepts the changes, the workflow ends, and the content is published.

Contents in intermediate stages are saved as drafts, while published contents are saved as final content, so there are always two versions of a determinate content: draft and final.

This Workflow model allows the existence of a variable number of stages. Despite the stages are built in execution time, while contents go through the Workflow. The disadvantage of this model is that equal Workflows have to be created every time there is a need of a new instance. In fact this means there is no workflow definition operation.

Bottom line is that this is a very simple and straightforward Workflow Managing System, which provides the content flexibility to the evolution within the Workflow. However it reveals a lack of automation, concerning stage definition, since stages are defined by its instances at runtime.

3.3. Altimate OpenEDMS

Altimate OpenEDMS [24] is a CMS that allows Workflow definition and management.

The Workflow definition process is supported by an activity diagram visual editor, which enables the user to create stages and transitions. Each stage has a responsible role, automatic actions and destination stages. When the Workflow definition is completed there is a validation in which the system determines if the Workflow is, or is not, valid. Afterwards the Workflow is saved.

The notification system may be done in two ways, e-mail or private message (system internal messages), while it is possible any kind of combination of these types of notification.

Terminated and pending Workflows are possible to inspect, as well as to start a new Workflow. To perform this last one the user has to name the Workflow, insert a comment, describing it, and a starting date. When a workflow finishes it is possible to distribute its content to selected users.

The most relevant feature of this System is the Workflow definition process, since it is completely visual and user friendly.

3.4. Vignette CMPortalSolution

Vignette's [26] CMS is a commercial product and a reference among its peers in the industry.

Workflow is a central concept within this CMS, and it uses both unitary and aggregation contents. Workflow definition is achieved visually, via Microsoft Visio, and stage transitions may be triggered whether by user authorization or external IS actions. After edited, contents have to be approved, so they may be published. The instance management is done in a console that allows its edition and cancellation.

3.5. Comparative Analysis

The study of the referenced CMS leads to the conclusion that only one out of the four actually allows the use of aggregation contents on Workflow. Also, out of the four CMS, only Alfresco has a static number of stages. The Workflow definition process is different among the CMS. Typo3 is the only one that does not allow determining the number of stages at the Workflow definition, since the Workflow is defined in run time by the user that is responsible for the current stage. V7CMS is the only that allows external IS interaction, while all of them allow User interaction. Notifications and deadlines are important concepts, although not supported by all the CMS.

Concerning the supported functionalities, the most important are the Workflow definition, the content, Workflow association and Workflow deletion and management.

	Alfresco	Typo 3	Altimate OpenADMS	Vignette
Concepts				
Number of Stages	Static (3)	Not defined	Dynamic	Dynamic
Supported Contents	Unitary Contents	Unitary Contents	Unitary Contents	Unitary & Aggregation Contents
User interactions	Yes	Yes	Yes	Yes
IS interactions	No	No	No	Yes
User Notification Operations	Yes (E-mail)	Yes (E-mail)	Yes (E-mail, internal messaging system)	Yes (E-mail)
WF Supported Operations				
Definition	No	No	Yes (Visual Interface)	Yes (Microsoft Visio)
Association	Yes	Yes	Yes	Yes
Edition	No	No	Yes	Yes
Removal	Yes	Yes	Yes	Yes

4. Conclusions

The adoption of Workflow technology into contemporary CMSs would allow the automation of content's production as well as its better integration according the business interests. This will also lead to an optimization of the organisation's business processes.

Concerning CMS there are actually very few that support generically Workflow. Those which do, only have partial support with still limited and inflexible features.

The generic reference model introduced in section 2, allow us to analyze any of the given Workflow mechanisms, in section 3, defining additional concepts that may be useful for future releases.

Aggregation contents have little support for Workflows, since only one out of the four CMS supported it, which may lead to the conclusion that aggregation contents are not as relevant as contents from the Workflow perspective. Nevertheless, given the early stage that these classes of systems are, it may be plausible that such contents would be better supported in the future.

Despite the lack of Workflow support by CMS, there are many requests by the industry so that a standard solution of Workflow management for all content types may be supported by these classes of systems. This can be a reasonable indicator that this is an important topic for the software industry, which should emerge in the years to come.

References

1. David Hollingsworth, The Workflow Management Coalition Specification reference model, 1995.
2. Ronni T. Marshak, Workflow: applying automation to group processes, Groupware: technology and applications, Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1995.
3. Tim Bray, Jean Paoli, Eve Maler, François Yergeau Extensible Markup Language (XML) 1.0, 4th Edition, 2006 .
4. Dr. Michael Kay, Building Workflow Applications with XML and XQuery available at http://www.stylusstudio.com/xml/Workflow.html# , last visited in 17-01-2007.
5. Robert Shapiro, Mike Marin, Roberta Norin, Workflow Management Coalition. XML Process Definition Language, 03-10-2005 available at http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-10-03.pdf, last visited in 21-07-2007.
6. Keith D. Swenson, Sameer Pradhan, Mike D. Gilger Wf-XML 2.0., Draft 08-10-2004, available at http://www.wfmc.org/standards/docs/WfXML20-200410c.pdf, last visited in 21-01-2007.
7. Keith D Swenson, ASAP/Wf-XML 2.0 Cookbook—Updated in Workflow Handbook, Layna Fischer, 2005.
8. Nilo Mitra, *SOAP Version 1.2 Part 0: Primer*. W3C Recommendation 24-06-2003. Available at: http://www.w3.org/TR/2003/REC-soap12-part0-20030624/, last visited in 21-01-2007.
9. Unified Modeling Language Resource Page 1997-2007, available at http://www.uml.org/, last visited in 11-12-2006.
10. Marlon Dumas, Arthur H. M. ter Hosfstede, UML Activity Diagrams as a Workflow Specification Language, 2001.
11. Windows Workflow Foundation, 2006, available at http://wf.netfx3.com/, last visited in 04-12-2006.
12. Paul C. Zikopoulos , A Beginner's Recipe for a Service-Oriented Architecture, available at http://www.db2mag.com/story/showArticle.jhtml?articleID=193200616, last visited in 12-12-2006.
13. Michael Rowell, Understanding EAI: Enterprise Application Integration, Sams, Indianapolis, Indiana, 08-2001.
14. Dirk Krafzig, Karl Banke, Dirk Slama,. Enterprise SOA, Prentice Hall, 2005.
15. SAP Network: Standards and Enterprise SOA, available at https://www.sdn.sap.com/irj/sdn/developerareas/esa/standards, last visited in 12-12-2006.

16. Balwinder Sodhi, Implement a customizable ESB with JAVA, 08-08-2005, available at http://www.javaworld.com/javaworld/jw-08-2005/jw-0808-esb.html, last visited in 12-12-2006.
17. What is Biztalk Server 2006?, 07-11-2005, available at http://www.microsoft.com/biztalk/evaluation/what-is-biztalk-server.mspx, last visited in 21-12-2006.
18. Dino Esposito, Building Web Solutions with ASP.NET and ADO.NET, Microsoft Press, 2002.
19. Gail Anderson, Paul Anderson, Java Studio Creator Field Guide, Sun Microsystems Press, 2006.
20. James Robertson, How to evaluate a content management system, KM Column 2002.
21. How to Choose a Content Management System, WebSideStory WHITE PAPER, 2005.
22. Alfresco CMS, available at http://www.alfresco.com/, last visited in 29-01-2007.
23. Typo3 CMS,, available at http://typo3.com/, last visited in 29-01-2007.
24. Altimate OpenADMS CMS, documentação available at http://www.altimate.ca/Workflowdemo.html, last visited in 21-01-2007.
25. Adxstudio CMS, documentation available at http://www.adxstudio.com/cms-product/features/Workflow, last visited in 21-01-2007.
26. Vignette Solutions, available at http://www.vignette.com/, last visited in 29-01-2007.
27. Daniel Rubio, Biztalk Server: Microsoft‘s SOA building block, 24-01-2006, available at http://searchwebservices.techtarget.com/tip/1,289483,sid26_gci1161311,00.html, last visited in 12-12-2006.
28. Wil van der Aalst e Kees van Hee , Workflow Management, MIT Press, 2002.
29. Edward A. Stohr, J. Leon Zhao Workflow Automation: Overview and Research Issues, Springer Netherlands, 2001.
30. Openflow: Open source Workflow management system, available at http://www.openflow.it/EN/index_html, last visited in 18-12-2006.
31. BEA WebLogic, available at http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/, last visited at 12-09-2007.
32. Organization for the Advancement of Structured Information Standards, OASIS, available at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm, last visited at 20-01-2008.

Exploring and Visualizing the "alma" of XML Documents

Daniela da Cruz[1], Pedro Rangel Henriques[1], and Maria João Varanda[2]

[1] Universidade do Minho
Departamento de Informática, Campus de Gualtar
Braga, Portugal
[2] Instituto Politécnico de Bragança
Campus de Santa Apolónia
Bragança, Portugal

Abstract. In this paper we introduce eXVisXML, a visual tool to explore documents annotated with the mark-up language XML, in order to easily perform over them tasks as *knowledge extraction* or *document engineering*.
eXVisXML was designed mainly for two kind of users. Those who want to analyze an annotated document to explore the information contained—for them a visual inspection tool can be of great help, and a slicing functionality can be an effective complement.
The other target group is composed by document engineers who might be interested in assessing the quality of the annotation created. This can be achieved through the measurements of some parameters that will allow to compare the elements and attributes of the DTD/Schema against those effectively used in the document instances.
Both functionalities and the way they were delineated and implemented will be discussed along the paper.

1 Introduction

Our recent research on program comprehension using slicing and visual inspection, as well as the work on grammar metrics led us to investigate how those approaches could be adapted to the field of document engineering. As a consequence we have conceived a tool, denominated eXVisXML, to aid in the inspection and analysis of XML documents.

By analogy with another tool (ALMA) we have developed in the past for program visualization and comprehension, we say that eXVisXML allows us to capture the "*alma*" (in English, the "*soul*") of structured documents, i.e., the intrinsic characteristics of XML documents. eXVisXML allows us to visualize the structure of the document (the hierarchy of XML elements), and provides a set of quality metrics, which enable us to reason out the document properties.

On one hand, our tool shows, in a graphical form, the document tree with the content associated to the leaves, providing means to navigate over it; moreover

it displays, in a tabular form, all the element occurrences associated with the respective attribute/value pairs. Using forward slicing techniques, eXVisXML allows the user to select parts of the document to focus his analysis just on some aspect; namely one can regenerate the original document restricted to some elements. These features are aimed at the comprehension of the document and its exploration (in the sense of knowledge extraction). Inspired in ALMA system, this functionality is displayed in two windows, one for the tree, and the other for the table of elements. We argue that the graphical representation of the abstract syntax tree complemented by the table of elements provides an easy to read and effective way to grasp the sense of the document.

On the other hand, eXVisXML allows the document engineer to assess the quality of his annotation schema (the DTD/XML-Schema he has designed) when applied to real cases. eXVisXML computes automatically a set of syntactic and semantic parameters (according to the standard metrics for XML documents) and shows them in a separate window. Those parameters are evaluated over the actual document and the respective schema in order to be possible, for instance, to compare the total number of elements available against the actual number of different elements used.

Before introducing our tool, eXVisXML, in section 5—describing its architecture and discussing the implementation strategies—we will write about the visualization of XML documents (section 2) and related work, i.e., other tools also developed with a purpose similar to eXVisXML; then we discuss, in section 3, the concept of document slicing and how it can complement the visualization and navigation, making easier the comprehension of the document; at last, we dedicate section 4 to the introduction of metrics to assess XML documents. The paper ends in section 6 with concluding remarks

2 XML Documents Visualization

The ability to retrieve information from plain documents, in a simple and efficient way, is one of the objectives that has motivated the search for markup languages. Concerning machine manipulation, the annotation systems like XML, so far developed, were completely successful; XSL and other production-systems can easily extract information from annotated documents and transform them. However for human beings, this task is not as easy as desirable, mainly if the annotation is complex or the document too big.

To help in finding the document fragments corresponding to some kind of element/attribute, or even located in some sub-document, document engineers developed specific query languages. In the last few years, appeared among many other, XPath [CD99,OMFB02] and XQuery [Cha02] languages specially designed to query collections of XML data. XPath or XQuery stand for XML like SQL for databases, making possible to find and extract elements and attributes from structured documents.

Moreover, the research for tools to visualize XML documents, is not a new issue. People recognized a long time ago that the existence of visual editors was crucial to create or read structured documents.

Nowadays there are many tools which merge the XPath querying facilities with the visualization of XML documents. Some of this tools are:

- XPath Analyzer by Altova [Alt07];
- XPath Visualizer [Top07];
- XPath Viewer by Microsoft [Mic07];
- XPath Query Editor by Stylus Studio [Stu07];

Although these tools offer a (textual) hierarchical view with highlighted syntax, as illustrated in figure 1, and make easier the manipulation of documents, allowing to expand and collapse sets of elements, they are not always powerful enough for the exploration of the document's constituents (elements and attributes) and the relationships among them.

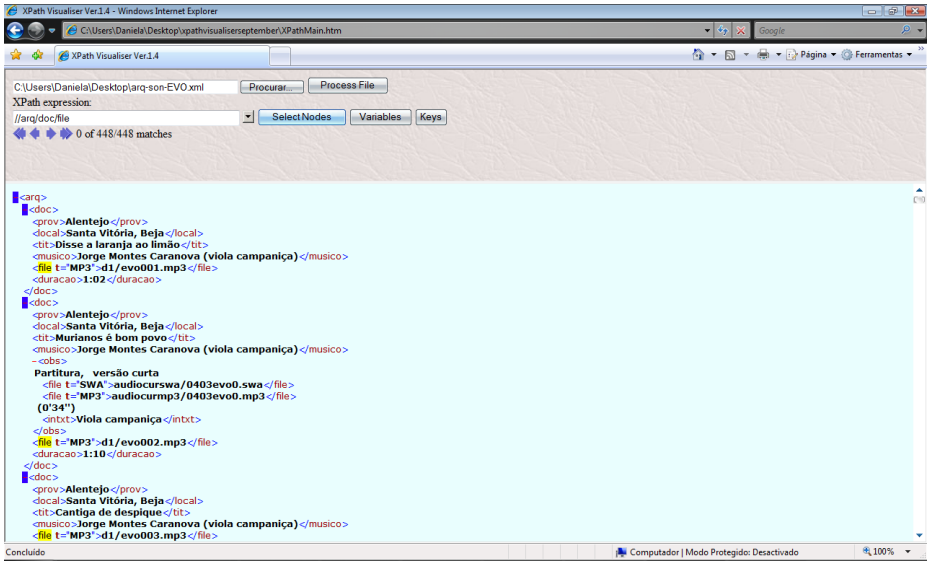

Fig. 1. Visualization of a XML document and selection of `file` nodes

The tool closest to our proposal is XML Schema Designer [Mic08]; however, that tool just deals with XML schemas. XML Designer provides a visual representation of the elements, attributes, types, and so on, that make up XML schemas. With XML Designer we can: construct new or modify existing XML schemas; create and edit relationships between tables; create and edit keys.

Actually, the kind of visualization that we propose is similar to the one provided by XML Schema Designer, but also applicable to XML documents. This is, we

propose a graphical representation of the internal abstract tree associated with the XML document, where intermediate nodes are XML elements and the text fragments (`#PCDATA`) are the leaves.

Edges describe the direct inclusion of document parts. So, we can distinguish two kinds of nodes: *text nodes* and *structure nodes*. The labels of *structure nodes* correspond to XML element types and *text nodes* (always leaves) are labeled with `#PCDATA` components (the actual text of the document). The visual representation used to show this information is lighter than the usual XML tag representation. It is well known the advantage of the use of graphical features to expose and explain structural and behavioral information.

3 XML Documents Slicing

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion C constitute the *program slice with respect to criterion C*. The task of computing program slices is called *program slicing* [Tip95].

As referred in [Sil05], the slicing technique can also be applied to XML documents. Essentially, given an XML document, it is produced a new XML document (a slice) that contains the relevant information in the original XML document according to some criterion (the *slicing criterion*). Furthermore, it is also possible to slice a DTD, where the output is a new DTD such that the computed slice is valid according to the original DTD.

This technique was implemented in a Haskell prototype tool called XMLSlicer [Sil06], using the HaXML library [Mer01]. In this approach, XML documents and DTD's are seen as trees; and the slicing criterion consist of a set of nodes in the tree. In both types of slicing—DTD slicing and XML slicing—given a set of elements, it will be extracted those elements which are strictly necessary to maintain the tree structure, i.e., all the elements that are in the path from the root to any of the elements in the slicing criterion. The difference between them is that while a slicing criterion in a DTD selects a type of elements, a slicing criterion in an XML document can select only some particular instances of this type.

Both slicing techniques produce valid XML and DTD slices with respect to the slicing criterion, if both the original are valid.

As a conclusion, we can say that this slicing technique can be seen as an easier way to query an XML document, simpler than an XPath/XQuery statement; it does not require to write the complete path to locate some information (or elements) in document.

4 XML Documents Metrics

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality.

In the last years, a wide set of software metrics was defined and can be classified as follow: product metrics (to evaluate a software product); process metrics (to evaluate the design process); and resources metrics (to appraise the required resources).

In the field of XML, the quality assessment is also relevant because the approach followed by engineers, or end-users, to design the annotation-schema (the type of a family of documents), or even to markup existing texts, is many times improvised and naif. Concepts like *well-formedness* or *validity* are not sufficient to appraise XML documents; they are only prerequisites to achieve quality.

Some of the software metrics (briefly referred above) have been adopted to measure the quality of XML documents [KSH02], being applied both to DTDs and XML-schemas (XSDs).

A tool dealing with XSD metrics is XsdMetz [Vis06,LKR05]. The tool was implemented in the functional programming language Haskell, using functional graph representations and algorithms. The tool is related with SdfMetz, which computes metrics on SDF grammar representations [AV05]. XsdMetz tool exports successor graphs in dot format so that they could be drawn by GraphViz [KN02].

However, in this paper we will only focus on the metrics defined over DTDs.

As a consequence of that research effort, a set of XML metrics was defined—*size, structure complexity, structure depth, fan-in and fan-out, instability, tree impurity*. Below and after our own contribution (*attributes per element, non-used components and text length*), we introduce them, as they form the basis of the quality measurement that will be implemented by the proposed tool.

Before presenting those metrics, we should define the notion of a *sucessor graph* (SG), now applied to DTDs [Vis06,LKR05], in order to measure the dependence between components. Given a DTD, we say that a new *component* (in this case, an *element* or an *attribute*) is an *immediate successor* of the *element under definition*, i.e., the component in the context of which the new one appears; then, we introduce an arrow (an oriented edge) from the element to the component. Based on this relation, the result is a graph representation of the structure of the XSD/DTD.

Size

$Size(DTD) = n_{EL} + n_A$

where n_{EL} — number of elements in the DTD, and n_A — number of attributes in the DTD.

Given a DTD, its *size* (i.e. the value for this metric) is the total number of nodes in the SG, i.e., the number of DTD components.

Structure complexity

To determine the complexity of a DTD, the McCabe metrics, developed to evaluate the control flow of software, was adopted. There exist slight variations of McCabe Complexity measure (MCC), but in essence MCC counts the number of linearly independent paths through the control flow graph of a program module.

MCC for grammars may simply count all *decisions* in a grammar, this is, operators for *alternative*, *optional* and *iteration*. Because DTDs are equivalent to context-free grammars, *Lammel et al*, in [LKR05], argue that in the same way, the MCC for DTDs correspond to the addition of edges to SG if quantifiers $+$ and $*$ occur and if mixed content elements (but not #PCDATA) exist.

So, the formula to measure the complexity of a DTD is:

$Compl(DTD) = e - n + 1 + n_{IDREF}$,

where e is the number of edges in the SG, n is the number of nodes in the SG and n_{IDREF} is the number of *IDREF* attributes. Note that actually the number of references to other identifiers increases the complexity.

In fact, if the DTD corresponds to a pure tree (which always has n nodes and $n-1$ edges) without internal references, then we get as structural complexity the value $Compl(DTD) = 0$. On the other side, every recursion, all iterators $+$ and $*$, and all $IDREF$ attributes increase the complexity.

Structure Depth

This metric, which computes the depth of the SG, also provides information about the complexity of the schema.

To compute the depth of the SG, we have to eliminate recursion, otherwise the result would be infinite. Then, the depth of each node is computed as follows:

$$Depth(n) = \begin{cases} 0 & n \text{ is leaf} \\ max(Depth(n_i)) + 1 \; for \; each \; n_i \text{ (child node of n)} \end{cases}$$

According to [KSH02], an SG with a depth much higher than seven is complex and reveals a bad DTD design.

Fan-in and Fan-out

These two new metrics are defined as follows:

$Fan-in(n) = \#\{n_i | n_i \text{ is parent node of } n\}$.

$Fan-in$ gives the number of incoming edges in the node.

$Fan-out(n) = \#\{n_i | n_i$ is child node of $n\}$.
$Fan-out$ gives the number of outgoing edges in the node.
Both metrics are directly applicable to the nodes of SG. For the graph as a whole, the average and the maximum values for those parameters can be useful to spot unusual nodes, which can be inspected to detect the anomaly and fix the problem. Elements with a high *Fan-in/Fan-out* value are more complex than other elements with a lower value.

Instability

Based on *Fan-in/Fan-out* metrics, a measure related with the *instability* of a node can be computed as follows:

$Instability(SG) = \frac{Fan-out}{Fan-in+Fan-out} \times 100\%$
A node with a low instability allows us to conclude that it is less dependent of other nodes, while many nodes are depend on it. This is, *instability* can be interpreted as resistance to change, hence a node with low instability corresponds to a situation where changes that occur over the node will affect relatively many other nodes.

Tree Impurity

$TI(SG) = \frac{n*(e-n+1)}{(n-1)*(n-2)} * 100\%$
where n is the number of nodes in the SG and e is the number of edges.
This metric is clearly inspired in Fenton's impurity concept used in the context of software or grammar quality assessment.
A tree impurity of 0% means that a graph is a tree and a tree impurity of 100% means that it is a fully connected graph.

Now we introduce the set of complementary new metrics, which we have defined.

Attributes per Element

To complement the *Size metric*, we define
$AttrsEle(DTD) = \frac{\sum n_A}{n_{EL}}$
where n_{EL} — is the number of the elements in the DTD, and n_A — is the number of attributes.
This metric allows us to figure out the average number of attributes defined per element in the DTD.
A similar metric could be defined over the XML document.
$AttrsEle(XML) = \frac{\sum n_{Au}}{n_{ELu}}$

where n_{ELu} — is the number of the elements used in the document, and n_{Au} — is the number of attributes actually used.

This metric, applied directly to the XML document, allows us to figure out the average number of attributes actually used per effective elements present in the XML document.

Non-used Components

In order to detect the non-used components (elements and attributes) in an actual XML document, we define:

$NonAttr(XML) = Attr(DTD) - Attr(XML)$

$NonElem(XML) = Elem(DTD) - Elem(XML)$

if $Attr(DTD)$ represents the set of attributes defined in the DTD, and $Attr(XML)$ represents the set of actual attributes (the attributes used in the XML document instance), then $NonAttr(XML)$ is the set of non-used attributes.

The set of non-used elements, $NonElem(XML)$, is defined precisely in the same way; once again, it gives an idea of the elements in the DTD that are not used in XML instances (it is similar to the notion of *dead-code* in a class—this is, *a set of methods that are never called*).

Then we define two metrics:

$NAttr(XML) = \#NonAttr(XML)$

$NElem(XML) = \#NonElem(XML)$

that measure the size (number of elements) of those two sets.

Text Length

$$TxtLen(XML) = \frac{\sum length(PCDATA)}{n_{PCDATA}}$$

where, length(PCDATA) computes the total length of the document's text (the sum of the length of all text fragments, i.e., text associated with element tags, or untagged text), and n_{PCDATA} is the number of text fragments (the number of $PCDATA$ leaves that appear in the XML document tree).

In a similar way,

$$AttTxtLen(XML) = \frac{\sum length(AttPCDATA)}{n_{Au}}$$

measures the average attribute text length.

Usually, the choice between the use of an *element* or an *attribute*, in a XML document type, is an ambiguous matter; in practice, some document engineers consider some particularities as elements, while others consider them as attributes. That metrics is precisely useful to study that phenomena; in fact, when we write a XML instance that duality/ambiguity becomes clear. We have the perception that an attribute should be used when its content is not too large, while an element should be used when we do not know how much large will be its content.

Over XML-schemas, the metrics applied are similar to the referred above, but with a slight difference: usually, the successor graph is built in the same way but the set of nodes that are strongly connected are grouped into the same node (a *module*). However, as said previously, we will not consider them in this paper; to learn more about the common metrics defined over XML-schemas, we suggest the reading of [Vis06].

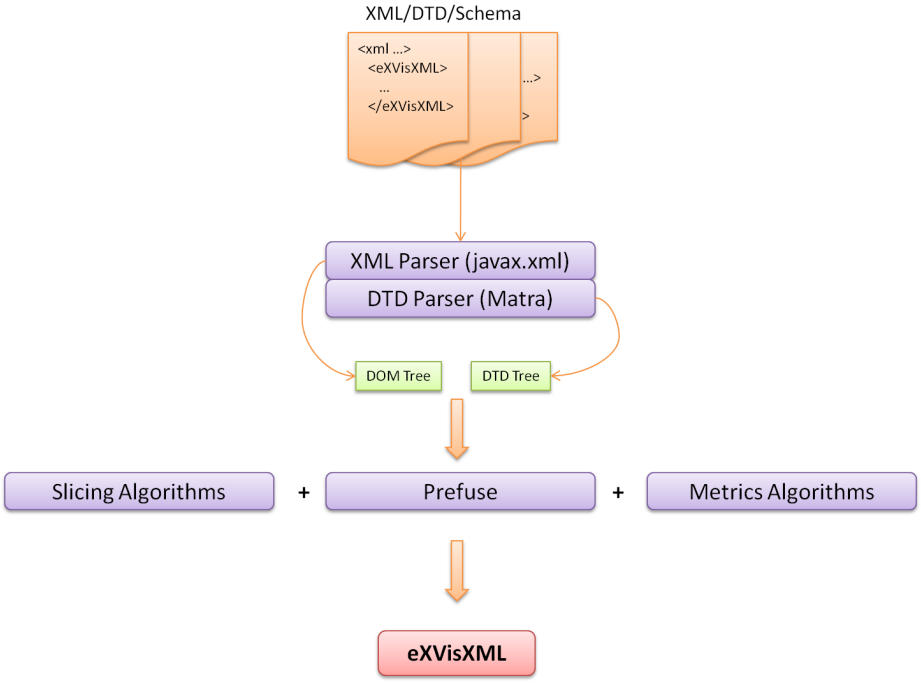

Fig. 2. Architecture of eXVisXML

5 eXVisXML, XML Document Visualization and Exploration

In this section, we concretize the ideas introduced along the previous sections, concerned with visualization, slicing and measuring of XML and DTD documents, discussing how they will be fully implemented in the proposed tool eXVisXML.

Nowadays, the development of a tool requires that the implementor searches for existing programming resources (libraries, design-patterns or program-templates, frameworks, generators, etc.), which can be used in his specific project.

So, in order to get advantage from other tools to build up our own, Java language and the Eclipse platform will be used as the programming environment.

The input for our tool are the the DTD and the XML document. From these 2 documents we can extract all the information needed. The information extraction process will be done by parsing the documents.

In this context, we will use the Java API for XML Processing (JAXP), which can be applied to parse and transform XML documents independent of a particular XML processing implementation. This API provides 2 basic interfaces: the DOM interface and the SAX interface.

The main difference between them arises from the fact that DOM interface builds a complete in-memory representation of the XML document, while SAX interface does not create an in-memory representation; instead, SAX parser informs clients of the XML document structure by invoking callbacks, and so it is faster and uses less memory.

For the purpose of our tool, we chose DOM because we need all the information stored in memory for visualization, slicing and measurement. DOM parser is called through the DocumentBuilder class, which creates an org.w3c.dom.Document instance, an abstract tree representing the structure of the XML Document.

Concerning the parsing of DTDs, a search over the Web led us to a tool called Matra, a Java-based XML/DTD parser utility, that parses the DTD and builds up a tree representation. Other tools of this kind were found and studied; although Matra proved to be the best, concerning its final representation.

Figure 2 depicts the architecture of eXVisXML, summarizing the technical decisions described above.

We discuss in the following subsections how to visualize, slice and measure the input documents. Those features will be illustrated by means of an working example (see appendix A for the XML document and its DTD)—*an excerpt of the well-known screenplay by William Shakespeare, The Romeo and Juliet Love Story [New01], (RJls)*—previewing the output that it will produce. Moreover, this will give a flavor of eXVisXML behavior.

5.1 Visualization

The role of the visualization technology, in fields like program comprehension and software engineering, is strongly recognized by the computer science community as a very fruitful one. The use of software visualization features allows us to get a high quantity of information in a faster way. Graphical representations have a positive impact in learning process because it engages the users in a more efficient comprehension process.

There are several kinds of views that can be produced: they can show *operational data* or *behavioral data* (more abstract view); they can be *static* or *dynamic*; they can be more *structural* or they can be more *quantitative* (based on metrics or other kind of statistical information).

These graphical or iconic representations must be carefully chosen because they usually depend on the problem domain. In our case, we want to visualize XML declarations or documents. Since structure/content visualization is used as a vehicle to make easier the comprehension of a document, it is necessary to care about the choice of visual paradigms/styles that will be used.

Taking this fact into account and inspired in ALMA, our visualization tool for program animation, the eXVisXML interface for the visual inspection of XML documents will be divided into 3 main parts:

- one window that displays the source document;
- one window exhibiting the tree associated with the source document — both tree representations, the graphical one (see Fig. 3 and Fig. 4) and the hierarchical textual view (like the one in Fig. 1), will be available;
- one window to show the Attribute Table (AT), formally a map: $Name \times Value$ — for each *element* selected over the tree, the AT shows the set of attributes of that element and the actual value of each one.

5.2 Slicing

According to a *slicing criterion* given by the end-user, the tool shall be able to select and highlight the path from the root until the node satisfying the criterion. If the *slicing criterion* matches an attribute, not only the node where the attribute appears will be highlighted, but also the corresponding line in the Attribute Table.

Considering again the working example, RJIs document globally shown in Fig. 3, suppose that "*Greg*" was chosen as the *slicing criterion* value in order to find all the screenplay components where actor *Gregory* appears.

The slicing algorithm, included in our tool, will traverse the tree looking for all matches of "*Greg*" with the value of each attribute and each leaf (#PCDATA value). The result of this *slicing operation* will be the enhancement of each path from the root of the document tree until each node where a match happened, as can be seen in Fig. 5.

As an additional feature, eXVisXML can generate a new XML document including only the components along the pathes highlighted in the previous *slicing operation*; the result is shown in Fig. 6. Notice that this new XML is also valid according to the submitted DTD, hence the structure of the XML document is not changed.

5.3 Metrics

Applying to the working example (the RJIs screenplay in appendix A), the set of metrics defined in section 4, we obtain the measures listed below. To evaluate part of those parameter values it was necessary to build first the *sucessor graph* for the given DTD. Fig. 7 sketches that SG.

- $Size(DTD) = 25 + 2 = 27$
- $Compl(DTD) = e - n + 1 + n_{IDREF} = 37 - 25 + 1 + 0 = 13$
-

$$Depth(n) = \begin{cases} 0 & n \text{ is leaf} \\ max(Depth(n_i)) + 1 & for\ each\ n_i \text{ each child node of n} \end{cases}$$

Taking n as the root, the Depth of the SG is 7, since the deepest branch has depth 6.

For the Fan-in and Fan-out metrics, let us consider the node *scene*.

- $Fan - in(n) = \#\{n_i | n_i \text{ is parent node of } n\} = 3$.
- $Fan - out(n) = \#\{n_i | n_i \text{ is child node of } n\} = 6$.

Considering the node *title*, the value of Fan-in and Fan-out metrics are 6 and 0, respectively.

Using the metrics above, the result of Instability metric is computed:

- $Instability(scene) = \frac{Fan-out}{Fan-in+Fan-out} \times 100\% = 3,3\%$
  $Instability(title) = 0\%$
- $TI(SG) = \frac{n*(e-n+1)}{(n-1)*(n-2)} * 100\% = \frac{24*(33-24+1)}{23*21} * 100\% = 58,9\%$
- $AttrsEle(DTD) = \frac{\sum n_A}{n_{EL}} = \frac{2}{25} = 0,08$
- $AttrsEle(XML) = \frac{\sum n_{Au}}{n_{ELu}} = \frac{2}{74} = 0,027$
- $NonAttr(XML) = Attr(DTD) - Attr(XML) = 0$
- $NonElem(XML) = Elem(DTD) - Elem(XML) = 25 - 24 = 1$
  The element stagedir under the context of the element line is never used.
- $TxtLen(XML) = \frac{\sum length(PCDATA)}{n_{PCDATA}} = \frac{2135}{57} = 37,46$
- $AttTxtLen(XML) = \frac{\sum length(AttPCDATA)}{n_{Au}} = \frac{2}{2} = 1$

When the user selects the appropriate option from eXVisXML menu, our tool will compute automatically the metrics above, and open a new window to display the values obtained.

6 Conclusion

Along the paper, we defend the idea that an useful tool to explore XML documents can be setup merging principles from similar areas (like software and grammar engineering, comprehension and quality assessment), as well as resorting to technological solutions already implemented.

As a proof of concept, we conceived and partially built eXVisXML, as proposed in section 5.

Basically we reuse visualization principles (section 2), slicing techniques (section 3), and software/grammar metrics (section 4), aiming at an exploration environment that allows us to comprehend by visual inspection the structure and contents of XML documents, and provides quantitative information to reason about the quality of the mark-up schema as well as the annotation itself.

The complete implementation of eXVisXML is the task we are working on, at moment. This is crucial to test the tool and prove our ideas, as well as to carry out performance, and usability measurements. After that we will apply the tool to a vast suite of test-cases in order check the set of metrics here proposed; maybe some of them are useless, and some others are missing. Of course, that test suite will be useful to tune the visualization, as well as to verify the effective importance of the slicing functionality for document understanding and re-engineering.

References

[Alt07] Altova. Xmlspy. http://www.altova.com/products/xmlspy, 2007.

[AV05] Tiago Alves and Joost Visser. Metrication of sdf grammars. Research report, Departamento de Informática, Universidade do Minho, Maio 2005.

[CD99] James Clark and Steve DeRose. Xml path language (xpath) version 1.0. Technical report, World Wide Web Consortium, 1999.

[Cha02] D. Chamberlin. Xquery: An xml query language. *IBM Syst. J.*, 41(4):597–615, 2002.

[KN02] Eleftherios Koutsofios and Stephen North. *Drawing graphs with* dot, 2002.

[KSH02] Meike Klettke, Lars Schneider, and Andreas Heuer. Metrics for xml document collections. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, pages 15–28, London, UK, 2002. Springer-Verlag.

[LKR05] R. Lämmel, Stan Kitsis, and D. Remy. Analysis of XML schema usage. In *Conference Proceedings XML 2005*, Novembro 2005.

[Mer01] David Mertz. Transcending the limits of DOM, sax, and xslt: The haxml functional programming model for xml. *IBM developerWorks (XML Matters column)*, October 2001.

[Mic07] Microsoft. Xpath viewer. http://msdn2.microsoft.com/en-us/library/aa302300.aspx, 2007.

[Mic08] Microsoft. Xml schema designer. http://msdn2.microsoft.com/en-us/library/ms171943(VS.80).aspx, 2008.

[New01] Greg Newby. Xml and project gutenberg. http://www.ils.unc.edu/ bluec/gutenbergDTD/, 2001.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward, 2002.

[Sil05] Josep Silva. Slicing xml documents. In *WWV*, pages 121–125, 2005.

[Sil06] Josep Silva. Xmlslicer. http://www.dsic.upv.es/ jsilva/xml/, 2006.

[Stu07] Stylus Studio. Xpath query editor. http://www.stylusstudio.com/xpath_evaluator.html#, 2007.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[Top07] Xpath visualizer. http://www.topxml.com/xpathvisualizer/, 2007.

[Vis06] Joost Visser. Structure metrics for xml schema. In *XATA - XML: Aplicações e Tecnologias Associadas, Portalegre - Portugal*, Fev 2006.

A The Romeo and Juliet love story

In this appendix we list the two documents—a DTD and an XML text—used as the eXVisXML input for the working example run along the subsections of section 5.

A.1 The Screenplay DTD

The DTD specified to define an XML dialect to mark-up Shakespear screenplays.

```
<!ELEMENT guttext (markupmeta, play, endgutmeta)>
<!ELEMENT markupmeta (title, gutdate, textnum, para, gutfilename)>
<!ELEMENT play (frontmatter, playbody)>
<!ELEMENT endgutmeta (#PCDATA)>
<!ELEMENT stagedir (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT gutdate (#PCDATA)>
<!ELEMENT textnum (#PCDATA)>
<!ELEMENT para (#PCDATA)>
<!ELEMENT gutfilename (#PCDATA)>
<!ELEMENT pgroup (#PCDATA | title | persona)*>
<!ELEMENT persona (#PCDATA)>
<!ELEMENT frontmatter (titlepage, personae)>
<!ELEMENT titlepage (pubinfo, title, author)>
<!ELEMENT personae (title, pgroup+)>
<!ELEMENT playbody (scene, act+)>
<!ELEMENT scene (scndesc | title | stagedir | speech | note)+>
<!ATTLIST scene
    id NMTOKEN #IMPLIED
>
<!ELEMENT act (title?, scene+)>
<!ATTLIST act
    id NMTOKEN #REQUIRED
>
<!ELEMENT scndesc (#PCDATA)>
<!ELEMENT speech (speaker | line | stagedir)*>
<!ELEMENT note (#PCDATA)>
<!ELEMENT speaker (#PCDATA)>
<!ELEMENT line (#PCDATA | stagedir)*>
<!ELEMENT pubinfo (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

A.2 The XML document

An excerpt from RJIs screenplay by William Shakespear, annotated according to the mark-up language defined by the DTD in previous subsection.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<guttext>
    <markupmeta>
        <title>The Complete Works of William Shakespeare
The Tragedy of Romeo and Juliet</title>
        <gutdate>November, 1997</gutdate>
        <textnum>1112</textnum>
        <para>The Library of the Future Complete Works of William Shakespeare
Library of the Future is a TradeMark (TM) of World Library Inc.</para>
        <gutfilename>
******This file should be named 1ws1610.txt or 1ws1610.zip*****
Corrected EDITIONS of our etexts get a new NUMBER, 1ws1611.txt
VERSIONS based on separate sources get new NUMBER, 2ws1610.txt
</gutfilename>
    </markupmeta>
    <play>
        <frontmatter>
            <titlepage>
                <pubinfo>1595</pubinfo>
                <title>THE TRAGEDY OF ROMEO AND JULIET</title>
                <author>by William Shakespeare</author>
            </titlepage>
            <personae>
                <title>Dramatis Personae</title>
                <pgroup>
                    <title>Chorus</title>
                    <persona>Escalus, Prince of Verona.</persona>
                    <persona>Paris, a young Count, kinsman to the Prince.</persona>
                    <persona>Capulet, heads of two houses at variance with each other.</persona>
                    <persona>An old Man, of the Capulet family.</persona>
                    <persona>Romeo, son to Montague.</persona>
                    <persona>Abram, servant to Montague.</persona>
                    <persona>Sampson, servant to Capulet.</persona>
                    <persona>Gregory, servant to Capulet.</persona>
                    <persona>Lady Montague, wife to Montague.</persona>
                    <persona>Lady Capulet, wife to Capulet.</persona>
                    <persona>Juliet, daughter to Capulet.</persona>
                </pgroup>
                <pgroup>Citizens of Verona; Gentlemen and Gentlewomen of both houses;
     Maskers, Torchbearers, Pages, Guards, Watchmen, Servants, and
     Attendants.
   </pgroup>
            </personae>
        </frontmatter>
        <playbody>
            <scene>
                <scndesc>SCENE.--Verona; Mantua.</scndesc>
                <title>THE PROLOGUE</title>
                <stagedir>Enter Chorus.</stagedir>
                <speech>
                    <speaker>Chor.</speaker>
```

```
                    <line>Two households, both alike in dignity,</line>
                    <line>In fair Verona, where we lay our scene,</line>
                    <line>From ancient grudge break to new mutiny,</line>
                    <line>Where civil blood makes civil hands unclean.</line>
                    <line>From forth the fatal loins of these two foes</line>
                    <line>A pair of star-cross'd lovers take their life;</line>
                    <line>Whose misadventur'd piteous overthrows</line>
                    <line>Doth with their death bury their parents' strife.</line>
                    <line>The fearful passage of their death-mark'd love,</line>
                    <line>And the continuance of their parents' rage,</line>
                    <line>Which, but their children's end, naught could remove,</line>
                    <line>Is now the two hours' traffic of our stage;</line>
                    <line>The which if you with patient ears attend,</line>
                    <line>What here shall miss, our toil shall strive to mend.</line>
                    <stagedir>[Exit.]</stagedir>
                </speech>
            </scene>
            <act id="1">
                <scene id="1">
                    <scndesc>Scene I.  Verona. A public place.</scndesc>
                    <stagedir>Enter Sampson and Gregory (with swords and bucklers) of the house
of Capulet.</stagedir>
                    <speech>
                        <speaker>Samp.</speaker>
                        <line> Gregory, on my word, we'll not carry coals.</line>
                    </speech>
                    <speech>
                        <speaker>Greg.</speaker>
                        <line> No, for then we should be colliers.</line>
                    </speech>
                    <speech>
                        <speaker>Samp.</speaker>
                        <line> I mean, an we be in choler, we'll draw.</line>
                    </speech>
                    <speech>
                        <speaker>Greg.</speaker>
                        <line> Ay, while you live, draw your neck out of collar.</line>
                    </speech>
                    <speech>
                        <speaker>Samp.</speaker>
                        <line> I strike quickly, being moved.</line>
                    </speech>
                    <speech>
                        <speaker>Greg.</speaker>
                        <line> But thou art not quickly moved to strike.</line>
                    </speech>
                    <note>THE END</note>
                </scene>
            </act>
        </playbody>
```

```
    </play>
    <endgutmeta>
End of this Etext of The Complete Works of William Shakespeare
The Tragedy of Romeo and Juliet
</endgutmeta>
</guttext>
```

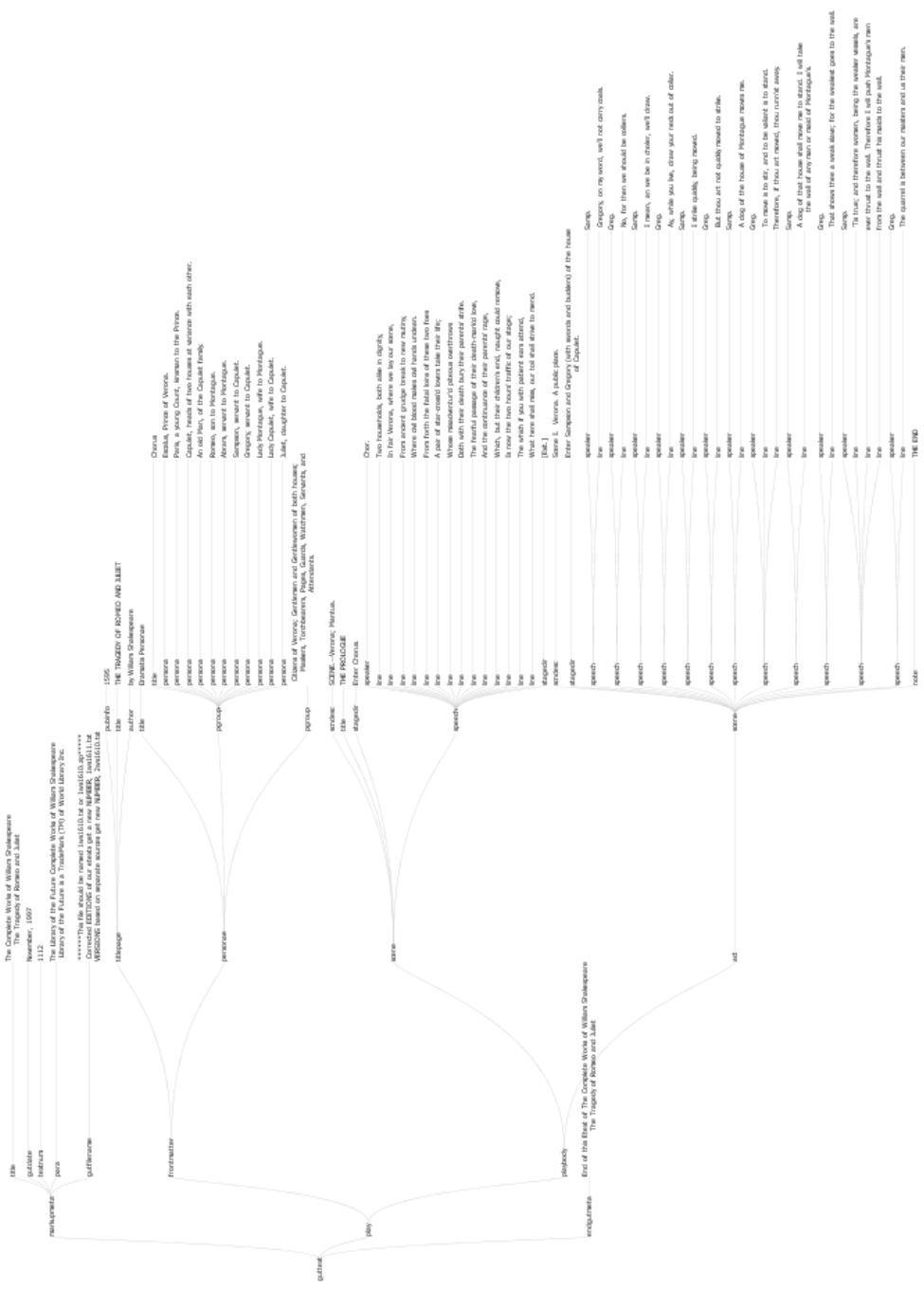

Fig. 3. Tree representation for RJls Doc. — global view

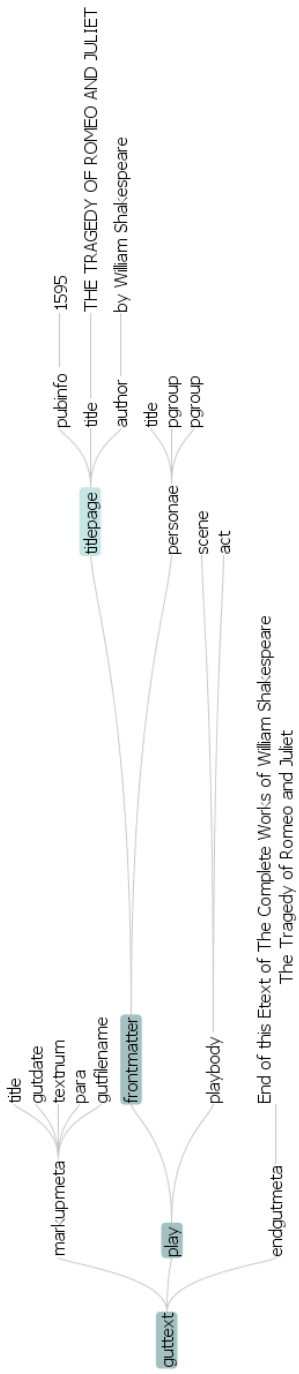

Fig. 4. Tree representation for RJIs Doc. — partial view (some nodes collapsed)

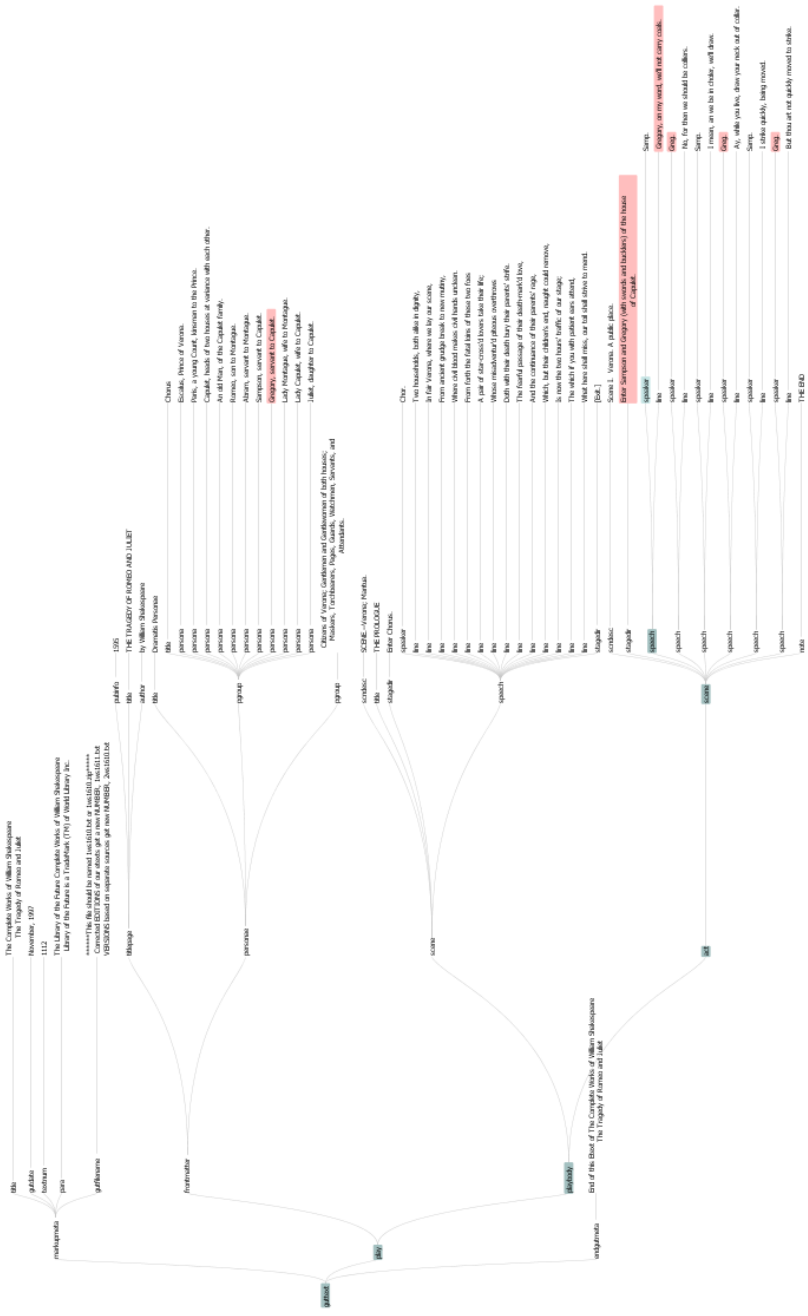

Fig. 5. Result of the *slicing criterion* "Greg"

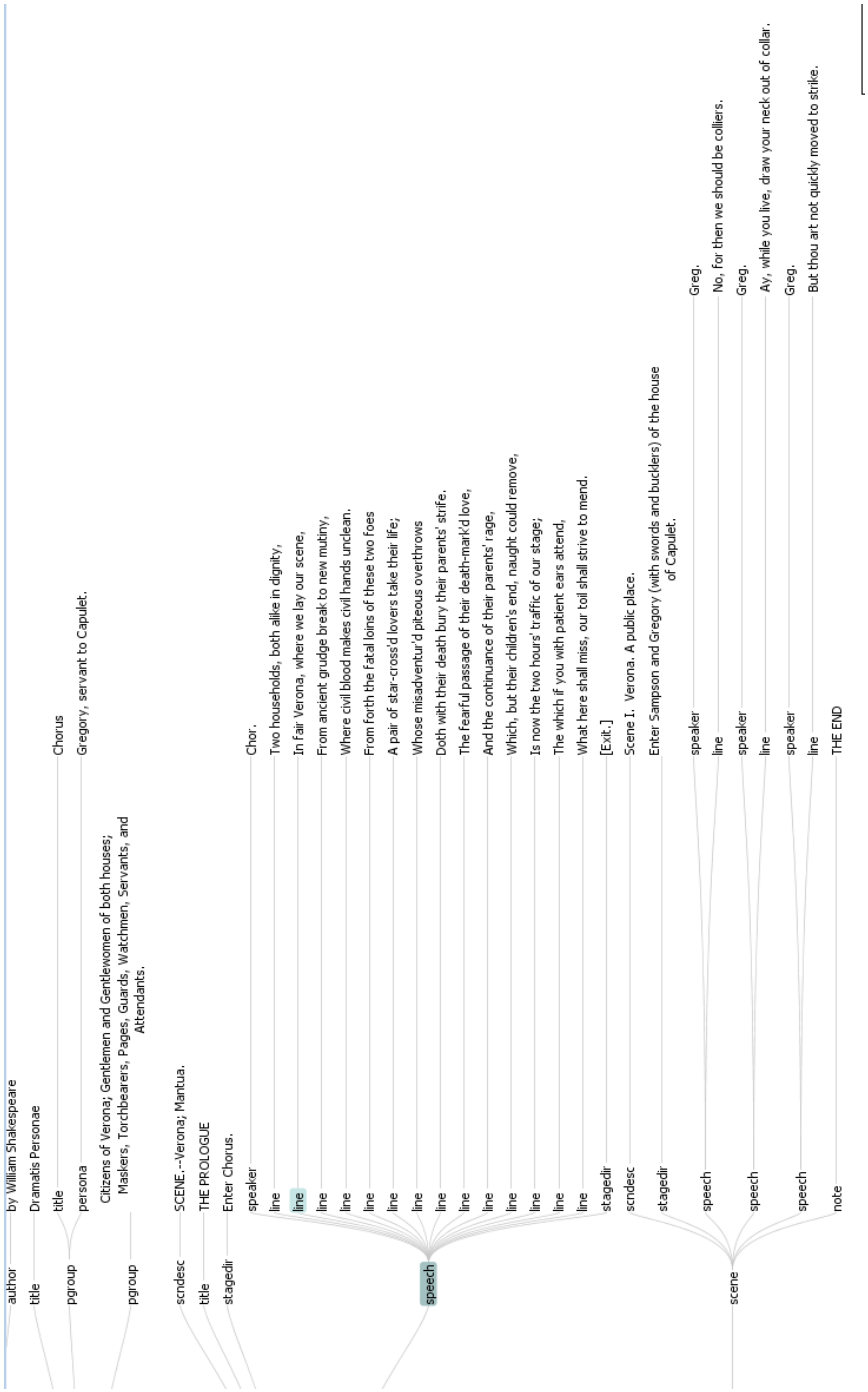

Fig. 6. New XML generated according to the *slicing criterion*

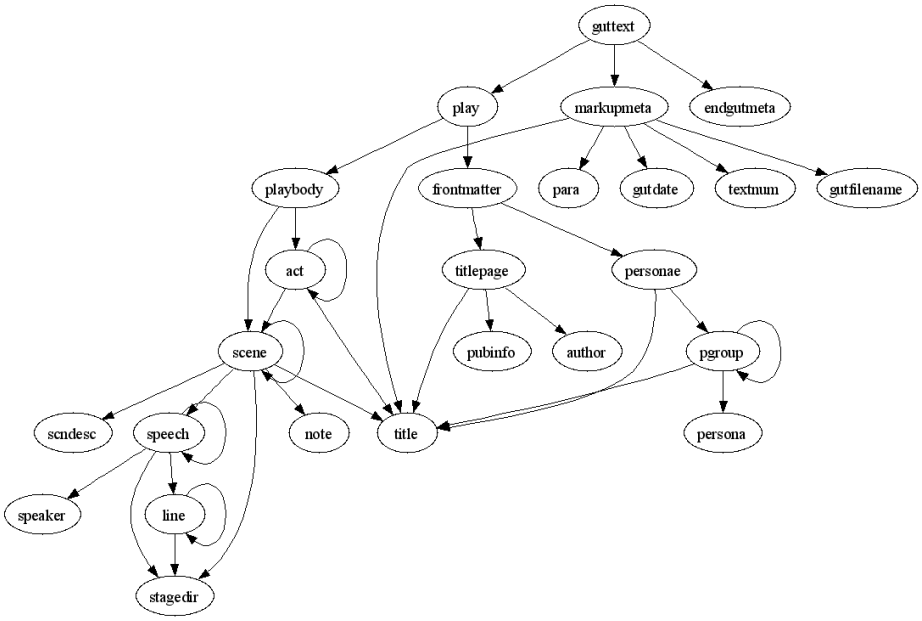

Fig. 7. Sucessor Graph for RJIs DTD

X-Spread - A software modeling approach of schema evolution propagation to XML documents

Vincent Nelson Kellers da Silveira and Renata de Matos Galante

Instituto de Informática - Universidade Federal do Rio Grande do Sul
{vincent, galante}@inf.ufrgs.br

Abstract. This paper presents X-Spread, a software modeling approach to propagation of XML schemata evolution to XML documents referring the modified schemata. This mechanism focuses on modifications of XML schemata on different distribution scenarios considering the possible distribution scenarios of schemata and documents as well, allowing for software engineers to focus on application design and not in the issue of XML documents adaptation. This mechanism is composed by change detection algorithms, responsible by the detection of differences between schemata, by storage of the different schema versions, by a revalidation process of existing documents, considering only modifications performed on parts of the schemata and by an adaptation process of documents considered invalid due to schema modifications, with the mechanism execution as a whole demanding no participation of the end user. This mechanism addresses semistructured artifacts distribution scenarios usually not addressed by papers about propagation of XML schema modifications to documents.

Key words:XML, schema, evolution, adaptation

1 Introduction

The disjoint nature of XML documents and schemata, associated to the potential distributed nature of semistructured database enabled applications are factors that increase the complexity of XML schema evolution process, adding to the software modeling complexity of current large software systems. Given that modifications performed on schemata are not automatically reflected on documents, that may imply on invalidation of documents until the documents or schemata themselves are fixed.

Applications based on exchange of schemata and XML messages are examples of applications that may be subject to a schema evolution process. A distributed book database, updated and queried over a network could use the schema depicted on Figure 1a. In a new version of the application responsible for the distributed data management such schema could be changed in order

to address new attributes not modeled by the first application implementation, with the addition of elements to the database, as depicted on Figure 1b.

Considering the distributed nature of this database, the entirety of documents stored by the database may not be accessible on a given time instant, since a given network node may not be operational. Other than the physical distribution of documents referring to a schema, the schema itself may be found in only one network node or replicated on different nodes, referred over the network by applications performing document validations. An alternative scenario for schema distribution is observed when a given schema version is encapsulated in an application performing user-based generation of new database records.

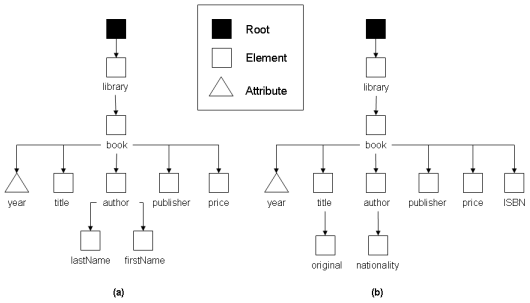

Fig. 1. XML schema evolution

Usually the impact of schemata evolution on XML documents is not addressed by recently proposed mechanisms of schema evolution such as [1], [2] and [3]. When this feature is specifically addressed, some XML artifacts usage scenarios were not considered, for instance, XML documents automatically generated by applications or documents found on unreachable network nodes at the moment of schema evolution [4], what renders these solutions as incomplete.

This work aims to provide a mechanism that stands as a common platform for detection, storage, document revalidation and propagation of modifications performed on schemata to XML documents referring to schemata subjected to an evolution process, considering the different distribution scenarios of the involved artifacts, with minimization of user input during the documents adaptation process.

The contribution of this work is based on the approach of different distribution scenarios of semistructured artifacts usually not addressed by studies on propagation of modifications on XML schemata to documents. Such scenarios encompass XML documents found on unreachable network nodes at the moment the schema is modified or documents that can be deemed as invalid when sent to other network nodes.

The proposed mechanism stands as a common platform for XML documents adaptation leveraged by a whole set of different semistructured data enabled applications, which allows engineers to focus on the design of these application

themselves, leaving the XML documents adaptation to be handled by the proposed mechanism.

The rest of the paper is structured as follows: Section 2 briefly exposes the proposed mechanism to later describe the differences detection algorithm applied to XML documents, the logical data model adopted for storage of different schema versions and associated information, the approaches to document revalidation after schema modification and the adaptation process of invalid XML documents. Section 3 discusses related work and Section 4 concludes the paper with final remarks and comments on future work.

2 X-Spread Overview

The mechanism proposed in this paper is based on four main components:

- Diff - Difference detection between different XML schema versions.
- Store - Storage of XML schema versions and related information.
- Revalidation - Revalidation of XML documents referring to modified schemata.
- Adaptation - Adaptation of XML documents considered invalid after schema modifications.

Figure 2 depicts the X-Spread components and the relationship between them.

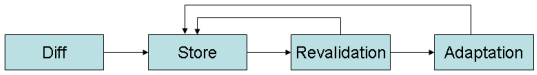

Fig. 2. X-Spread components

The first X-Spread component, Diff, performs difference detection on versions of a given XML schema. Differences found between versions of a schema are named deltas, and form the scripts on which transformation from the original schema version to newer versions is based on.

Diff's generated deltas and XML schema versions themselves are stored by Store component, in order to build the historical record on schema evolution. Since XML documents referring to considerably older versions of the schema may become part of the database at any given moment, the history of modifications performed on a schema must be ensured to exist, in case these documents have to be adapted to newer versions of the schema.

Component Revalidation takes as input a set of XML documents, a set of schema versions and a set of deltas referring to these versions, and performs validations on portions of the XML documents, restricting the validations to portions potentially affected by the schema evolution process.

Component Adaptation receives as input a set of XML documents identified as invalid by the Revalidation component, a set of deltas and a XML schema,

and proceeds with the documents adaptation. The component task is to make a set of XML documents valid again with respect to the schema specification. In the following sections we describe each X-Spread component in details.

2.1 Diff - Difference Detection Algorithms for XML Schemata

The Diff component is based on difference detection algorithms for XML documents. This component receives as input two XML documents and it generates the specification of the differences found as output. This component works with configurations set by the X-Spread administrator, allowing the specification of XML schema locations, whose evolution process must be observed.

As described on [5], many factors must be taken into account when studying XML documents comparison algorithms, for instance, their time and space complexity, output quality, the number of supported operations and distinction between ordered and unordered documents.

Even though usually schemata do not constitute large documents, the time complexity of some algorithms for XML documents comparison may generate results on unacceptable time frames on scenarios where the schemata must be compared with high performance, such as applications in a web environment.

Since the algorithm output will lead to the generation of documents adaptation scripts, the output quality of the algorithm is quite relevant. Thus, the algorithm must generate only correct, relevant and preferably compact outputs, with the concatenation of several primitive operations for representation of complex update operations. Outputs featuring these characteristics will cause improved performance when the adaptation scripts are applied to invalid XML documents.

Existence of an open source implementation of the difference detection algorithm for XML documents is relevant for this work given that definition and implementation of a whole new diff algorithm is not on the scope of this work.

Considering the information gathered on differences detection algorithms for XML documents, the performance and the output quality of algorithms with an available open source implementation, the chosen algorithm was XyDiff[1] [6].

X-Spread mechanism abstracts the implementation of schemata comparison algorithm, allowing for future algorithm substitution, in case a new, better suited algorithm is developed to this task, with no impact on other X-Spread components and on the stored set of schema versions and deltas.

2.2 Store - Schema and Deltas Storage

Store component is responsible by storage of XML schema versions and Diff component outputs. Store, as pictured in Figure 3, has three different modules: DeltaPointer, responsible by translation of deltas generated by the Diff component, in order to guarantee an homogeneous delta storage even if the difference detection algorithm is changed; Persist, responsible by delta and schema versions storage itself and Retrieve, responsible by queries to stored artifacts.

[1] Open source implementation available at http://potiron.loria.fr/projects/jxydiff

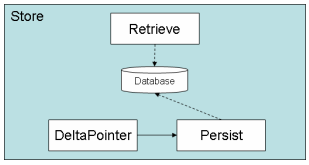

Fig. 3. Store component modules

DeltaPointer module is responsible by translation of Diff output and generation of XPath references, in order to ensure the storage of deltas generated by Diff component in a standard format, indifferently of the adopted difference detection algorithm.

Most of the XML schema modifications detected by the Diff component will lead to the generation of two XPath references, each associated to an operation identifier. Operations such as element or attribute deletion will lead to the generation of a single XPath reference.

Persist module is responsible by storage of DeltaPointer output and the input schemata submitted to the component. These artifacts persistence is needed since they may be queried in order to start the adaptation process of an invalid XML document generated by an application based on an older version of a schema.

In order to perform schemata and deltas persistence, the set of approaches proposed for the storage of XML documents addressing time [7] [8] and version aspects [9] [10] [11], or approaches using the parametric data model are not suitable, since they do not hold to one premise used on the Persist module regarding XML schemata: all stored versions must be valid. Inclusion of new attributes on XML schema artifacts will lead the artifact to an invalid state.

For instance, versioning and temporality related attributes are not part of the original format definition [12], and modeling of such attributes on XML schema would demand development and usage of new schema parsers and document validators, which should be attached to these schemata in all environments using them.

Discarded the inclusion of new XML schema elements and attributes, XML schema versioning attributes were discarded for versioning control as well. Version attribute defined in [12] has no associated semantics, which renders the attribute ignored by XML documents validators.

Usage of different namespaces for each different schema version was also discarded since it implies on explicit modification of XML documents referring to the schema, what breaks the second premise used on the Store component: XML documents must not be changed due to X-Spread's schemata storage model.

With that in mind, the logical data model depicted on Figure 4 was developed for schema versions and deltas storage. Physical implementation of this data model can be done through a relational database or even through storage of schema versions and deltas on the file system, with storage of additional information, such as validity dates, version identifiers, references to other versions

and identification of operations in a XML document developed specifically for this purpose.

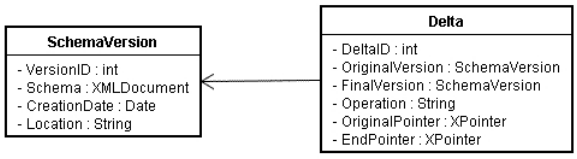

Fig. 4. Store component modules

Retrieve module is responsible for supplying querying tools to deltas and schema versions persisted by Store component. Independently of the DBMS or the storage device used for persistence of XML artifacts, queries performed against this repository will always be performed through the same interfaces.

2.3 Revalidation - XML Documents Revalidation

The Revalidation component is responsible by validation of a set of XML documents, finding out, after the schema evolution process, whether the documents are still valid.

This component has as premise the high performance, given that the number of documents to revalidate is unknown, and ability to abstract the XML documents location, applying the same algorithms both to documents stored at the file system where the schema is found and to documents exchanged over a network.

The Revalidation component has settings specified by the X-Spread administrator, allowing for specification of locations both on local and remote file systems whose contents are subject to revalidation when a XML schema modification is recorded by Store component. Changing these settings will lead to the start of a document revalidation process, considering that the newly included locations on the component settings may contain invalid XML documents.

Input given to the Revalidation component is based on a set of deltas generated by the Diff component, a set of XML documents and a set of XML schema versions. Based on the set of deltas, a minimum set of XML documents sections is defined, where this minimum set of sections corresponds only to parts of the XML schema affected by modifications. The minimum set of sections is used in a revalidation process quite similar to that described in [13].

This revalidation takes into account only sections of the XML documents in order to improve the revalidation process performance, given that a full revalidation of the XML documents could imply in low performance of the X-Spread mechanism.

Several approaches to XML documents incremental validation were recently described. However, some of them, such as [14] are not suitable to some scenarios addressed by the X-Spread mechanism, since they demand a pre-processing phase of the XML documents.

On X-Spread in particular, the full set of XML documents referencing a given schema is not known by the time the schema is modified. Given that the documents may be located at other network nodes, with their inclusion on the set of documents to revalidate only after the schema evolution has already taken place, or the documents may be automatically generated by user applications that necessarily implies no pre-processing can be performed.

Validation of XML documents generated by applications is achieved through the specification of communication ports that must be watched by a proxy and reverse proxy tool like WebScarab[2].

This tool must have not only the ability to intercept requests and responses exchanged over a network, where these requests or responses are originated from or sent to a node hosting X-Spread, but also it must have abilities to manipulate these requests contents and execute the very same validations performed by X-Spread on XML documents located on a file system.

The distinct feature of the revalidation process applied to XML documents exchanged over a network and on the scenario of inclusion of different and new file system locations whose contents must be revalidated on the components configurations is based on the fact that one of the parameters accepted by the Revalidation component, the schema version referred to by the XML documents, can be unknown: prior to identifying the document validity with respect to newer schema versions, the application has to identify to what schema version the XML document is referring to, in order to start the adaptation process in case the document is deemed as invalid.

A first approach to schema identification on XML messages is based on schema extraction from the message, followed by comparison of the extracted schema with versions recorded by the Store component.

A second approach is based on validation of the document with respect to the current version of the schema and, if the document turns out to be invalid, a validation of the document with respect to previous schema versions follows, until a schema where the document is deemed as valid is found.

A third approach is based on finding out if the deltas generated by the Diff component have any reference to the element set that caused the invalidation of the document with respect to the current schema version. In case these deltas are found, the document will be adapted by applying on the document a chain of reverse deltas, until the document is in a state where it can be validated with respect to the schema version that features modifications on elements that first caused the document invalidation. In case the document is once again deemed as invalid, the process will be repeated until the whole chain of schema deltas is applied to the document or until the applications finds out that the elements causing the document invalidation are subject of no delta, what would halt the

[2] Implementation available at http://www.owasp.org/software/webscarab.html

revalidation and adaptation process of the document. In case the schema version referred to by the document is found, the remaining processes can take place.

In order to avoid full execution of the schema detection process whenever a XML message with no schema associated is exchanged over a network, a cache can be created, associating a given network address or file system location to a given schema version, improving the performance of the schema detection process on future validations originated from or executed on a file of a cached location.

Considering the designed functionality of the Revalidation component, a possible execution scenario is based on unfeasibility of identification of the schema used by a XML document, what can happen when the Diff component was not able to identify partial versions of a XML schema, what means that no delta and versions storage process was executed. When this scenario is detected, the Revalidation component will sign to X-Spread's administrator such occurrence and the revalidation and adaptation process of the document will be halted.

XML documents stored on a file system or exchanged over a network and deemed as invalid with respect to a schema by the end of the revalidation process, along with the referred schema itself and a set of deltas will be taken as inputs by the Adaptation component, which will have the document contents modified, so it can regain validity with respect to a schema. The Adaptation component is described in details in the next section.

2.4 Adaptation - Invalid XML Documents Adaptation

The Adaptation component is responsible by adaptation of XML documents deemed as invalid by the Revalidation component due to an evolution process that took place on a XML schema. This component takes as inputs a set of XML documents to adapt, a set of deltas and a XML schema referred to by the document.

The XML document adaptation takes place based on operations described by deltas generated by the Diff component and taken as inputs, thus, these deltas are associated to the schema version taken as a parameter on this component as well. The Adaptation component will issue a query to the Store component, in order to identify whether the given schema has child versions. When child versions are found, successive deltas will be applied to the XML document, until the current schema version deltas are reached and applied.

The difference detection algorithms subject to usage by the Diff component can generate many XML document modification operations other than the basic inclusion and deletion of elements and attributes, such as move and copy of complex elements. Even though not all operations are used on a given moment by the algorithm applied by the Diff component, the Adaptation component must have the ability to understand and deal with all update operations that can be generated by the Diff component.

The move and copy of elements in a XML schema has trivial impact and implementation on XML documents taken as input by the Adaptation component. The adaptation process basically consists on moving an element inside a XML document, or copying an element from one location to another.

Deletion of attributes and elements also has a trivial implementation, based on removal of the corresponding artifact from the XML document. The exception to this trivial implementation is verified when exclusion of control artifacts of the XML Schema takes place, such as exclusion of use, minOccurs and maxOccurs attributes.

In case of exclusion of XML Schema control attributes, the semantics of each attribute must be subject to analysis in order to identify what is the impact on XML documents. Control attributes semantics must also be taken into account by the Revalidation component when validation of sections of documents takes place.

The inclusion operation has some particular and non trivial cases: inclusion of XML Schema control attributes and elements, inclusion of application attributes, inclusion of optional elements and inclusion of required elements. Among these cases, inclusion of optional elements only has trivial implementation on the documents adaptation process, which demands no adaptation action at all. Inclusion of XML schema control elements must be subject of detailed analysis, in order to identify what is the impact on XML documents.

A possible approach to the problem of specification of new values for new elements and attributes of a XML schema is based on end user input. The application user could supply the missing values during the adaptation process of the documents.

Even though this approach will generate valid documents whose content is valid from an application standpoint, some scenarios such as those where XML messages are adapted and exchanged over a network, may prove this approach as unfit due to many aspects. On these scenarios, where the user input is needed on the adaptation process of XML messages exchanged by distributed applications, a request timeout can occur, other than the need of X-Spread installation not only on the network node where the application is executed, but also on client machines, or at least, the modification of client source code would be needed, in order to enable them to interpret some kind of messaging from X-Spread, which would sign the need for user input on the adaptation process of XML messages.

A possible approach to this problem is creation of new structurally valid elements, even though potentially invalid from an application standpoint, given that the values used on attributes and on primitive data types would be default values of each data type, i.e., an empty array of characters, the zero value for integers, and so on.

In order to adopt an homogeneous approach to all possible XML documents adaptation scenarios, both when documents are located at a file system reachable by X-Spread and when XML messages are exchanged over a network, the mechanism can use settings specified by the X-Spread administrator, where a default value is associated to operations described by deltas persisted by the Store component. The mechanism would first perform a detailed analysis of the stored deltas, finding out those that require user input, present them to the X-Spread and ask for specification of values that should be applied to documents whenever these deltas are invoked by the documents adaptation process.

These approaches to the document adaptation process can be configured by the X-Spread administrator in order to achieve the best results considering the different usage scenarios addressed by the mechanism.

3 Related Work

Researches on semistructured database evolution usually address the document evolution, in other words, only part of the database evolution process is modeled. Documents evolution usually is addressed by implementation of temporality [7] [8], versioning [10] [11] or by a mix of these concepts [15].

Schema evolution is addressed in [1] however the impact of schema evolution on existing XML documents is not taken into account. On the other hand, [4] describes a document evolution and adaptation proposal, with an implementation description on [3]. However, this proposal is based on particular schema update tools, what interferes on one of the main XML format features, which is the ease of update of each artifact, which can be performed with simple tools such as text editors with no outstanding features.

Other than that, the proposal described in [4] may demand user input during documents adaptation to a new schema version. This may render this proposal as unfit for scenarios where XML documents are automatically generated by distributed applications.

A DTD-based evolution mechanism is described on [2], where schema evolution is triggered by patterns detected on documents through usage of data mining techniques. However, when a modification on a schema is performed due to patterns discovered on a given number of documents, documents referring to the schema remain unchanged even after schema evolution. Considering that the schema evolution is not propagated to the XML documents and that patterns that led to the schema evolution may not be found on the entirety of XML documents that form the semistructured database, some formerly valid documents can be deemed as invalid after the schema evolution.

A new approach to XML documents adaptation, based on object oriented databases is described in [16]. In this approach DTDs are converted to user data types and the XML documents are inserted into the database. Modifications to the DTD schema are mapped to database modification operations. These database modifications are subject to validations that may accept or reject the modification if the database consistency is not guaranteed after the modification execution.

Another approach where an object oriented database is used to control XML schemata evolution is described in [17]. In this approach, XML documents are loaded into the database only after the DTD is registered into the database management system. Valid schema modifications supported by XEM are automatically propagated into XML documents, while modifications that lead the database into an inconsistent state are rejected.

As these approaches are very similar to traditional databases approach to schema evolution, these mechanisms may not be suitable for all semistructured

data enabled applications. For instance, applications that use XML message to communicate over a network may not count on this mechanism due to performance issues or plain unfeasibility of the attachment of an object oriented database to the current system implementation.

Evolution of semistructured schemata and documents usually is addressed on the literature as different problems, when in fact they are strongly linked. The proposed mechanism aims to combine different techniques in order to approach in an homogeneous and as non-intrusive manner as possible from the end user and semistructured database enabled application standpoints, all the phases that take place during the evolution process of a XML schema.

The unique feature of this work is the approach to different semistructured database distribution scenarios, featuring flexibility on the revalidation and adaptation processes of documents spread over different parts of local and remote file systems and of XML messages generated by applications commnicating over a network.

4 Concluding Remarks

This paper presented X-Spread, a mechanism to propagate XML schema modifications to documents referring to modified schemata. X-Spread has as premises the high performance, ability to abstract the XML documents location subject to revalidation and adaptation and ability to adapt documents without input from the application user nor from X-Spread's administrator.

Acting as an observant system, as described in [18], the X-Spread architecture offers choice flexibility on the algorithm responsible by difference detection between schema versions and on schema versions and deltas physical storage.

Featuring observation of modifications executed on XML schemata, semantic analysis of the executed modifications, abstraction of XML documents location during revalidation and adaptation processes, the X-Spread differences in comparison with existing work such as [1], [4], [2] and [3] are based on the combination of a wide array of different techniques and the approach of different phases of the schema evolution process.

X-Spread takes into account parts of the process usually addressed separately in the literature, allowing for a schema modification followed by propagation of this modification to documents referring to the schema without use of specific tools. With that feature, X-Spread holds on to one of the features of the XML format, which is the ease of artifacts update.

As of right now, the mechanism components are defined and these definitions are subject to improvements, in order to encompass a greater set of possible evolution scenarios of a XML schema. In parallel, after implementation of the Diff component, a study on XML diff algorithms performance and quality is being executed, in order to validate XyDiff choice.

References

1. Coox, S.V.: Axiomatization of the evolution of xml database schema. Program. Comput. Softw. **29**(3) (2003) 140–146
2. Bertino, E., Guerrini, G., Mesiti, M., Tosetto, L.: Evolving a set of dtds according to a dynamic set of xml documents. In: EDBT '02: Proc. of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers, London, UK, Springer-Verlag (2002) 45–66
3. Mesiti, M., Celle, R., Sorrenti, M.A., Guerrini, G.: X-evolution: A system for xml schema evolution and document adaptation. In Ioannidis, Y.E., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C., eds.: EDBT. Volume 3896 of LNCS., Springer (2006) 1143–1146
4. Guerrini, G., Mesiti, M., Rossi, D.: Impact of xml schema evolution on valid documents. In: WIDM '05: Proc. of the 7th annual ACM Intl. workshop on Web information and data management, New York, NY, USA, ACM Press (2005) 39–44
5. Peters, L.: Change detection in xml trees: a survey (2004)
6. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in xml documents. In: ICDE, IEEE Computer Society (2002) 41–52
7. Amagasa, T., Yoshikawa, M., Uemura, S.: A data model for temporal xml documents. In Ibrahim, M.T., Küng, J., Revell, N., eds.: DEXA. Volume 1873 of LNCS., Springer (2000) 334–344
8. Wang, F., Zaniolo, C.: Temporal queries in xml document archives and web warehouses. In: TIME, IEEE Computer Society (2003) 47–55
9. Wong, R.K., Lam, N.: Managing and querying multi-version xml data with update logging. In: ACM Symposium on Document Engineering, ACM (2002) 74–81
10. Iwaihara, M., Chatvichienchai, S., Anutariya, C., Wuwongse, V.: Relevancy based access control of versioned xml documents. In Ferrari, E., Ahn, G.J., eds.: SACMAT, ACM (2005) 85–94
11. Wuwongse, V., Yoshikawa, M., Amagasa, T.: Temporal versioning of xml documents. In Chen, Z., Chen, H., Miao, Q., Fu, Y., Fox, E.A., Lim, E.P., eds.: ICADL. Volume 3334 of LNCS., Springer (2004) 419–428
12. W3C: W3c xml schema (2004) Available at: <http://www.w3.org/XML/Schema>. Last accessed: September 2006.
13. Raghavachari, M., Shmueli, O.: Efficient schema-based revalidation of xml. In Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E., eds.: EDBT. Volume 2992 of LNCS., Springer (2004) 639–657
14. Barbosa, D., Mendelzon, A.O., Libkin, L., Mignet, L., Arenas, M.: Efficient incremental validation of xml documents. In: ICDE, IEEE Computer Society (2004) 671–682
15. Santos, R.G.: Evolução de documentos xml com tempo e versões. Master's thesis, UFRGS, Porto Alegre (2005)
16. Al-Jadir, L., El-Moukaddem, F.: F2/xml: Managing xml document schema evolution. In: ICEIS (1). (2004) 251–258
17. Su, H., Kramer, D., Chen, L., Claypool, K.T., Rundensteiner, E.A.: Xem: Managing the evolution of xml documents. In Aberer, K., Liu, L., eds.: RIDE-DM, IEEE Computer Society (2001) 103–110
18. Dyreson, C.E.: Observing transaction-time semantics with ttxpath. In: WISE '01: Proc. of the Second Intl. Conf. on Web Information Systems Engineering (WISE'01) Vol.1, Washington, DC, USA, IEEE Computer Society (2001) 193

Author Index: