XATA 2010

ISBN: 978-972-99166-9-4

XML, Associated Technologies and Applications

May 19–20, Vila do Conde

Escola Superior de Estudos Industriais e de Gestão

Instituto Politécnico do Porto

Editors

Alberto Simões
Daniela da Cruz
José Carlos Ramalho

Table of Contents

Editorial

These are the proceedings for the eighth national conference on XML, its Associated Technologies and its Applications (XATA'2010).

The paper selection resulted in 33% of papers accepted as full papers, and 33% of papers accepted as short papers. While these two types of papers were distinguish during the conference, and they had different talk duration, they all had the same limit of 12 pages.

We are happy that the selected papers focus both aspects of the conference: XML technologies, and XML applications.

In the first group we can include the articles on parsing and transformation technologies, like "Processing XML: a rewriting system approach", "Visual Programming of XSLT from examples", "A Refactoring Model for XML Documents", "A Performance-based Approach for Processing Large XML Files in Multicore Machines", "XML to paper publishing with manual intervention" and "Parsing XML Documents in Java using Annotations". XML-core related papers are also available, focusing XML tools testing on "Test::XML::Generator: Generating XML for Unit Testing" and "XML Archive for Testing: a benchmark for GuessXQ".

XML as the base for application development is also present, being discussed on different areas, like "Web Service for Interactive Products and Orders Configuration", "XML Description for Automata Manipulations", "Integration of repositories in Moodle", "XML, Annotations and Database: a Comparative Study of Metadata Definition Strategies for Frameworks", "CardioML: Integrating Personal Cardiac Information for Ubiquous Diagnosis and Analysis", "A Semantic Representation of Users Emotions when Watching Videos" and "Integrating SVG and SMIL in DAISY DTB production to enhance the contents accessibility in the Open Library for Higher Education".

The wide spread of subjects makes us believe that for the time being XML is here to stay what enhances the importance of gathering this community to discuss related science and technology. Small conferences conferences are traversing a bad period. Authors look for impact and numbers and only submit their works to big conferences sponsored by the wright institutions. However the group of people behind this conference still believes that spaces like this should be preserved and maintained.

This 8th gathering marks the beginning of a new cycle. We know who we are, what is our identity and we will keep working to preserve that. We hope the publication containing the works of this year's edition will catch the same attention and interest of the previous editions and above all that this publication helps in some other's work.

Finally, we would like to thank all authors for their work and interest in the conference, and to the scientic committee members for their review work.

Alberto Simões
Daniela da Cruz
José Carlos Ramalho

Steering Committee

Cristina Ribeiro (FEUP and INESC Porto)

Gabriel David (FEUP and INESC Porto)

João Correia Lopes (FEUP and INESC Porto)

José Carlos Ramalho (DI/UM)

José Paulo Leal (FCUP)

Pedro Henriques (DI/UM)

Organizing Committee

Alberto Simões (ESEIG/IPP)

Luís Correia (ESEIG/IPP)

Mário Pinto (ESEIG/IPP)

Ricardo Queirós (ESEIG/IPP)

Scientific Committee

Ademar Aguiar (FEUP and INESC Porto)
Alberto Rodrigues da Silva (IST and INESC-ID)
Alberto Simões (ESEIG/IPP)
Alda Lopes Gançarski (Institut N. des Télécommunications)
Ana Paula Afonso (DI/FC/UL)
Benedita Malheiro (ISEP)
Cristina Ribeiro (FEUP and INESC Porto)
Daniela da Cruz (DI/UM)
Francisco Couto (DI/FC/UL)
Gabriel David (FEUP and INESC Porto)
Giovani Librelotto (UFSM)
João Correia Lopes (FEUP and INESC Porto)
João Moura Pires (FCT/UNL)
José Carlos Ramalho (DI/UM)
José João Almeida (DI/UM)
José Paulo Leal (DCC/FCUP)
Luis Ferreira (EST/IPCA)
Luís Carriço (DI/FC/UL)
Manolo Ramos (UVigo)
Marta Jacinto (ITIJ)
Miguel Ferreira (DSI/UM)
Nuno Horta (IST/IT)
Paulo Marques (DEI/UC)
Pedro Rangel Henriques (DI/UM)
Ricardo Queirós (ESEIG/IPP)
Rui Lopes (DI/FC/UL)
Salvador Abreu (DI/UE)
Stephane Gançarski (LIP6, University P. & M. Curie)
Xavier Gómez Guinovart (UVigo)

Full Papers

A Refactoring Model for XML Documents

Guilherme Salerno, Marcela Pereira, Eduardo Guerra, Clovis Fernandes

Aeronautical Institute of Technology, Praça Marechal Eduardo Gomes, 50
Vila das Acacias - CEP 12.228-900 – São José dos Campos – SP, Brazil
{guilhermesalerno, marcelasobrinho,guerraem}@gmail.com, clovistf@uol.com.br

Abstract. Code refactorings are structural changes in the code without alterations on the external behavior. They are well known, studied and largely used. Many times it's also necessary to change the structure where data is stored. Data refactoring is more complicated because it involves, besides the structural change, migration of the existent data. The existing methods and tools developed for this purpose are focused only on database refactoring, but since XML is turning into the leading model for data formatting in several fields, it is also important to be considered. This work proposes a refactoring method specific for XML, which modifies the base XML Schema and updates the XML document through XSLT transformations. A tool was developed in order to validate the technique applicability. This methodology enables XML documents refactoring minimizing any collateral damage that may affect its consumer applications, since it is possible to obtain the document on their recognized format.

Keywords: XML, refactoring, XML Schema, XSLT

1 Introduction

Derived from the SGML (Standard Generalized Markup Language) format [1], the XML (Extensible Markup Language) was designed to deal with challenges of large-scale electronic publishing and, nowadays, has an increasingly role in data exchange on the web and elsewhere [2]. The idea at first was to create a simplified version of SGML, for web applications, keeping its advantages as flexibility, structure and validation, but being easier to learn, use and implement [3].

XML's flexibility allows design markups for a specific context. For instance, it enables professionals to develop their own field-specific markup languages. Despite of being essentially a plain text document, the data in an XML file is self-describing, because its markups are meaningful. This characteristic makes XML documents easier to be read by humans than other previous data storage file formats. It turns out to be the more obvious choice for data exchange, as it is non-proprietary; easy to read and edit; and patents or any other intellectual restrictions protect it. These can be considered some of the main reasons for its popularity.

Its wide use in web applications; in databases; in communication protocols on Service-Oriented Architecture (SOA) [4], among other uses, are the reason of studies on how to evolve and refactor XML files being an area of great interest. Consistent with this, XML documents are frequently used in applications whose code is

constantly evolving. The increasing usage of agile and evolutionary methods, such as Extreme Programming (XP) [6], Scrum [7], Rational Unified Process (RUP) [8], Agile Unified Process (AUP) [9] and Feature-Driven Development (FDD) [10], makes the refactoring process important for all application artifacts, which includes XML Schemas for files consumed by the software.

Refactoring is a structural change into the software which does not change its external behavior [11]. For instance, a usual refactoring is extracting a method from a piece of code in order to make it easier to be read and modified. This is usually necessary due to design decay resulting from code modification and addition of new functionalities [11].

The coupling between applications and the database schema that supports them frequently demands a more complicated refactoring process: the data refactoring. This necessity can also appear on legacy databases or due requirements change that impacts on the data structure [12]. Data refactoring is a new concept and it is more complicated than source code refactoring, because it deals with structure changing and data migration [9,11].

The main publications and tools related to data refactoring are focused on database refactoring. One example of database refactoring is to move a column from a table to another one that is closely related to it. In one hand, it is easy to notice that databases and XML documents have common characteristics, such as schema definition and a close relation between data and structure. These characteristics allow the using of some concepts created for database refactoring into XML refactoring. On the other hand, the schema for XML is defined decoupled from the documents, in a XML Schema file. Consequently, the XML files can be spread through more than one application that uses them, for instance, to exchange information. Those facts highlight the importance of a technique specific for XML refactoring.

The increasing use of XML documents for data representation, especially for web services [13], also increases the need for this kind of refactoring. However, there is no technique to deal specifically with it, and the available tools to refactor XML documents or XML Schemas do not support automatically linking structure changes with document changes. No method suitable for XML refactoring has been found in the literature review. The tools found either only with change the schema without data migration or just change an isolated XML document ignoring its structure definition.

This work proposes a refactoring method specific for XML which modifies the base XML Schema and enables updates on XML documents through XSLT transformations. Its main goal is to enable the refactorings in the XML Schema to be applied statically or dynamically in XML files, consequently making possible for an application to understand an XML document created accordingly other schema version. The Chrysalis tool was developed to automate the defined process in order to validate the technique applicability.

This text is divided into the following sections: Section 2 defines and explains the proposed method for XML Refactoring; Section 3 shows the created tool, Chrysalis, based on the defined method and analyses it as a refactoring tool; finally, Section 4 concludes the work analyzing the proposed method and its validity.

2 XML Refactoring

XML refactoring is a structural change into the document that preserves the meaning of the stored data; in other words, the information received by a consuming application should not change. Three refactorings were considered useful and were used to exemplify the technique in this work. The first one is the element renaming. It can be useful as a XML document evolves and may be necessary to change the meaning of an element or to give it a more significant name. The second is the attribute renaming, motivated by a similar need. The last one is enclosing a group of elements into a new element, which can be useful when this group can be reused in different contexts or when the information grouped has more meaning together. Table 1 presents an example of this refactoring: the first line shows the schemas and the second the XML document.

Table 1. Example of a grouping elements refactoring, middlename and lastname tags are grouped into surname tag.

Original	Refactored

`<?xml version="1.0" encoding="UTF-8"?>`
`<xs:schema`
`xmlns:xs="http://www.w3.org/2001/XMLSchema">`
`<xs:element name="person">`
`<xs:complexType>`
`<xs:sequence>`
`<xs:element name="name"`
`type="string"/>`
`<xs:element name="middlename"`
`type="string"/>`
`<xs:element name="lastname"`
`type="string"/>`
`</xs:sequence>`
`</xs:complexType>`
`</xs:element>`
`</xs:schema>`
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<xs:schema`
	`xmlns:xs="http://www.w3.org/2001/XMLSchema">`
	`<xs:element name="person">`
	`<xs:complexType>`
	`<xs:sequence>`
	`<xs:element name="name"`
	`type="xs:string" />`
	`<xs:element name="surname"`
	`type="surnameType" />`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:element>`
	`<xs:complexType name="surnameType">`
	`<xs:sequence>`
	`<xs:element name="middlename"`
	`type="xs:string" />`
	`<xs:element name="lastname"`
	`type="xs:string" />`
	`</xs:sequence>`
	`</xs:complexType>`
	`</xs:schema>`

`<?xml version="1.0" encoding="UTF-8"?>`
`<person>`
`<name>José</name>`
`<middlename>Almeida</middlename>`
`<lastname>da Silva</lastname>`
`</person>`
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<person>`
	`<name>José</name>`
	`<surname>`
	`<middlename>Almeida</middlename>`
	`<lastname>da Silva</lastname>`
	`</surname>`
	`</person>`

There are two flows in the XML refactoring method: one involves the schema changing and the other updates the XML document. The first flow starts when the user wants to do some kind of refactoring. The user chooses the refactoring type and provides information required to make it. Based on this information, the schema is changed and the style sheets required to change the XML documents are created and stored. Style sheets representing the reverse transformation (from a newer version of a XML to an older one) are also created and stored. The second flow starts when an application needs to read a XML document defined in a refactored schema different

from the schema version that it natively understands. The generated style sheets are applied into the provided XML resulting in a document on the desired version, newer or older. A representation of the XML Refactoring Technique process is illustrated in Fig. 1.

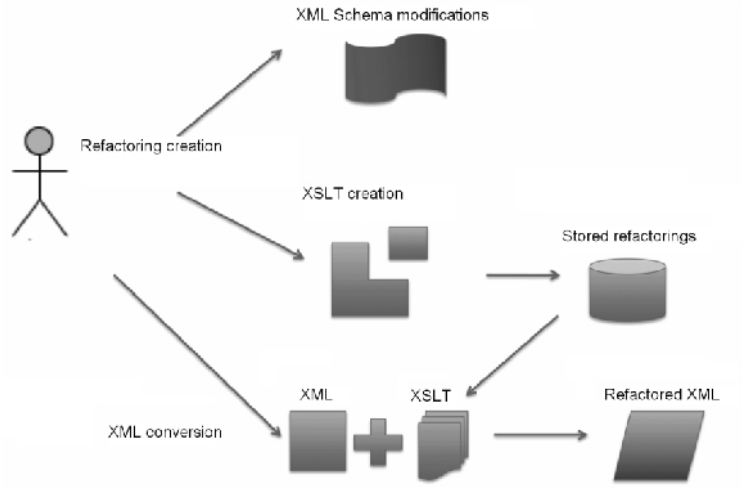


Fig. 1. Graphical representation of the XML Refactoring Technique

2.1 Changing the XML Schema

The proposed XML refactoring model requires documents being refactored to be defined following the XML Schema standard. A refactoring changes in the first place the base schema for those documents. In order to define a new refactoring, it is necessary to determine the parameters required to execute the modifications that affect the schema.

Each refactoring implies a particular set of changes in the XML Schema structure. However, certain consequences or properties are common to several refactorings. An example is the refactoring entity's scope, which notably has influence on the impact over the schema structure. If this entity is local, the refactoring is simpler, once it affects just its declaration. But if the entity is global, changes in all the entities that refer to it might be necessary. Furthermore, if the target of a refactoring is a reference to a global entity, it may be necessary to create a new global entity which should be refactored instead of the original, avoiding impact in other entities that derives from it.

An important aspect that all refactoring must consider is to keep the schema valid. That includes verifying if all entities were created with valid names. The positioning of the entities must also be validated.

2.2 XSLT Transformations

XSLT is a language created to transform an XML document into other documents [14]. The language is, actually, a part of XSL specification, language for style sheets to XML, which besides transforming can be also used to format the document.

For each schema modification resultant from a refactoring, a style sheet is created to adequate any XML document described by the original schema into a transformed document following the new schema description. The style sheet structure of each kind of refactoring is essentially the same. This allows writing a generic model for the XSLT documents. This model is filled based on information got from the changes in the schema document. The information required to fill the model depends on what kind of refactoring is being made. For instance, the path set for the element that must be refactored is important because it permits the XSL to search for this pattern to apply the transformations.

Besides the XSL that applies the changes made in the schema into the XML document, another style sheet to revert the modification is also created. This XSLT file allows an application that is prepared to deal with a previous version of the schema to get a compatible version of an XML document defined in a more recent schema version.

2.3 Version Control

Two aspects are relevant in the refactoring version control: the XSLT files organization and the identification of the XML and XML Schema documents' version. The XSLT files must be stored in a structured fashion in order to be possible to determine which of them belong to each version and the sequence they should be applied. Information about the documents version is also important for their validation and for the correct conversion to a version that the consuming application understands.

The chosen solution is the inclusion of a version attribute in the root element of the XML. This approach forces the XML documents to have an attribute, called *schemaVersion*, indicating their versions, keeping everything else intact. In the first refactoring, the XML Schema is altered in order to add this attribute in all the possible root elements. The attribute is optional since its use is pointless if the element is not the root. The XSLT created forces the first XML refactoring to include this attribute valued '1'. Every XML created for this moment on should have the *schemaVersion* attribute. Additionally, for every version changing is created an XSL to update the *schemaVersion* attribute.

Before migrating data by the conversion of the XML documents, this method requires the target schema version to be closed with the refactoring modifications since the last version finalized. A closed version can be defined as one in which no more refactorings would be made, meaning that it can be used by applications. If you were allowed to apply changes in a closed version, the XML document generated would be obsolete when more refactorings were added, even having a compatible *schemaVersion* attribute with this version.

2.4 Converting XML Files Between Versions

After creating a group of refactorings and closing a version, it is possible to convert XML documents from any other version to this one. In order to do that, the user should provide the document that should be converted and the target version. If the XML doesn't have an *schemaVersion* attribute, the user can provide the version, otherwise it will be considered that the document's version is 0.

From the original and the target version is possible to select which XSLT documents should be used to transform the XML document. These XSLT documents contain all style sheets for direct transformation created between the original and the target version. In case the original version is the oldest, it should include all the reverse ones instead.

The conversion can be made statically, creating a new XML file in the desired version based on an existent one defined in another format. The conversion can also be dynamic, when an application needs to consume the XML document that is in a non-supported version and, thus, the conversion is required. For instance, this last approach is suitable for documents created and consumed at runtime.

2.5 Implemented Refactorings

The refactorings needed by a developer vary significantly, so the number of possible implementations is large. The objective of this work is to propose an XML refactoring method and not to create a refactoring catalog to satisfy all possible situations. In other words, the implemented refactorings were used to exemplify and test the technique. In order to achieve this goal, three XML refactorings were chosen to validate the technique. Those refactorings were considered basic in any refactoring tool, and that is why they were chosen. They were completely defined and automated in the tool described in the next section. Other refactorings could have been implemented, and the flexibility of the tool permits future works to achieve this goal.

The first one is Attribute Renaming. It consists on the attribute's name change on all elements that use it. This is made in order to give a more significant name to the attribute in the document's context. It is important to have such refactoring since the use of an attribute can change as the XML evolves. Changes in the project structure and wrong decisions in the early phases of the data definition may force an attribute renaming as well. In order to perform this refactoring, the attribute and its new name must be informed. With this data, the set of paths to the elements that contain the attribute and its qualified name are obtained from the XML Schema. These information are used to create the XSLT.

Element Renaming is other implemented refactoring, whose objectives are very similar to Attribute Renaming. The information that must be provided is also similar, which in this case are the element being renamed and its new name. This data is also used to obtain the paths and the new element qualified name from the XML Schema.

The last implemented refactoring is Element Grouping. It consists in the creation of an element that contains a set of preexistent elements. This grouping, in general, intends to create a meaningful element to represent the set. For example, if the elements 'street', 'number', and 'city' are recurrent together in more than one place, it is

a good practice to create an element 'address' containing those elements. That can lead to a better presentation and to a more reusable structure. To perform this refactoring it's necessary to inform which elements should be grouped and the name of the new container element. From the XML Schema, it is possible to extract the set of paths where those elements are used, the list of qualified names of the group components and the qualified name of the group. All this information is used to create the XSLT style sheets.

3 Chrysalis Tool

In order to verify the applicability of the proposed method, a tool was developed implementing the refactorings described in subsection 2.5. Chrysalis is an Eclipse plugin that can be installed through the program updates control. It is open source and can be downloaded from the project web site [15]. The next subsections describe Chrysalis more detailed.

3.1 Tool overview

Chrysalis has basically two main functionalities: automated refactoring of XML Schema files and data migration of XML documents accordingly to the modifications made into the schema.

The first functionality is accessed through an Eclipse's XML Schema visual editor. In order to execute refactorings the user should access the context menu of a schema element. The menu item 'XML Refactor' shows the available refactorings for that element, as illustrated in Fig. 2.

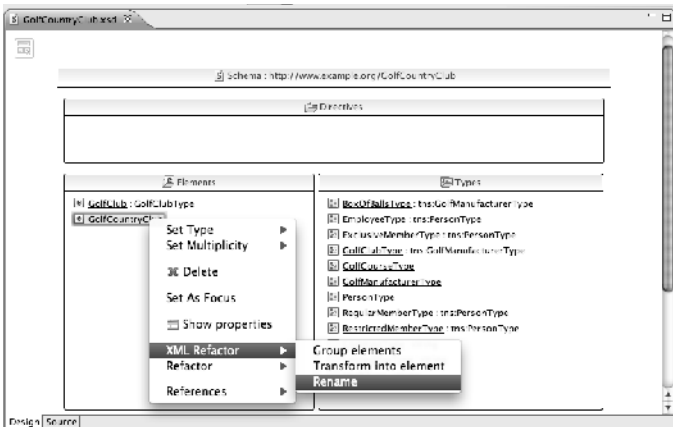


Fig. 2. "XML Refactor" menu

The selection of any refactoring takes to a window that requests the necessary parameters from the user. Finally, it is possible to preview the changes that the

refactoring will generate in the *workspace*, as shown in Fig. 3, and confirm it. The user may see the differences in the refactored XML Schema.

The second functionality changes an XML document's version applying the changes necessary for the refactorings. Additionally, it is available through the context menu associated to the xsd file in the *Package Explorer*.

As it was discussed in session 2.4, it is only possible to apply the changes correspondent to a specific version if it is closed. A version is considered closed if it is more recent than the current version. To close a version, changing its number, the option *Change Version* must be used in the *Context Menu*. This command changes the value of the attribute *schemaVersion* which should be assigned to the refactored XML documents. It also creates a new directory where the working version refactorings should be stored. Besides, with the version closed, it is possible to refactor XML documents to and from this version.

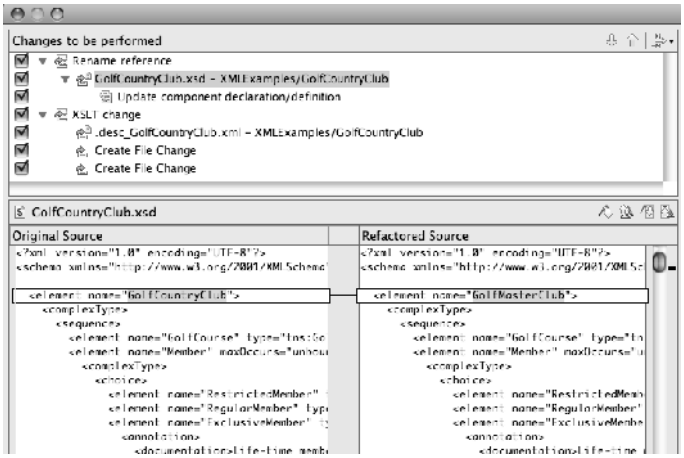


Fig. 3. Preview of an element renaming

In order to apply changes to a document, the option *Apply Changes to XML* must be chosen. Then, a pop-up window is presented for the selection of the XML file to be refactored and the transformation's target version.

In addition to the XML version's change through a graphic interface, it is possible to do it programmatically in Java accessing the *XMLConverter* class. This functionality may be particularly useful if several documents must be updated, or if an application needs to obtain an XML document in certain version dynamically. To use this API, the directory with the generated XSL transformations must be in the application root path.

For each schema, one different instance of the class *XMLConverter* must be created. The constructor must receive an object from the type *java.io.File* which represents the xsd. It is important that the file is the same used during the creation of the refactorings. It is not possible to apply changes from an isolated xsd file, since the XSLT files were created and stored within it.

The target document in which the changes should be made must be passed as an instance of the class *java.io.InputStream* to the *XMLConverter* through the invocation of the *getXML()* method. The desired version for the XML must be passed as well.

The method returns an *InputStream* containing the converted file.

In the case of an XML file without the *schemaVersion* attribute in its root element and not in the original version (version 0), alternatively its version may be passed as a parameter in the method. If it is not passed and the file does not presents *schemaVersion*, the file is considered in version 0.

The example illustrated in Fig. 4 demonstrates how to use the *XMLConverter*. In the second line the XML Schema is passed as parameter to the constructor. In line 4, the method *getXML()* receives the *InputStream* containing the XML being refactored and the value 2 representing the target version. In line 5 the *InputStream* containing the refactored XML can be used.

```
1. File schemaFile = new File("/mySchema.xsd");
2. XMLConverter con = new XMLConverter(schemaFile);
3. InputStream initial = new FileInputStream("/myXML");
4. InputStream final = converter.getXML(original,2);
5. final.read();
```

Fig. 4. Example of using of the XML Converter

3.2 Internal Structure

The tool's architecture was built over the Eclipse's refactoring architecture [16]. Such option was made intending to take advantage from a refactoring system largely used and stable, integrated with a very popular IDE. The Eclipse's refactoring system already provides some functionality to the created tool.

The tool's structure, illustrated in Fig 5, is basically divided in three main modules: the graphic interface; the refactoring mechanism; and the version change. The graphic interface is composed of the Eclipse's XSD Editor and the Wizards, responsible for obtaining the necessary data from the user. The refactoring mechanism has two main components, one responsible for creating the XML Schema changes and other for creating the XSLT files and storing them. The changes application is made by the last component, which can be executed through a graphic interface or programmatically.

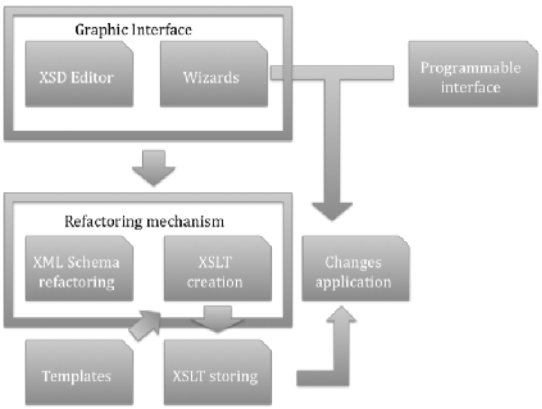


Fig. 5. Chrysalis Architecture representation

3.3 Extensibility

It's possible to extend the tool in two ways: by adding new actions on the implemented refactorings or by creating new ones. In both cases some base classes provided by the Chrysalis tool must be extended.

The motivation for adding actions to the implemented refactorings is that a refactoring process can have several side effects. The architecture adopted from Eclipse enables a structure to deal with this scenario. If someone intends to implement new actions to take place at the same time of an implemented refactoring, a specialization of the class *Participant* must be created. This class is part of the refactoring mechanism, and is the only one that must be extended. A tutorial explaining how to add this kind of functionality is available at [15].

It is also possible to extend the tool to add new refactorings [15]. In order to do that all the base classes from the graphic interface, such as *BaseRefactoringWizard* and *BaseUserInputWizardPage*, and from the refactoring mechanism, such as *BaseXSDParticipant, BaseXSDParticipant* and *BaseProcessor,* must be implemented.

3.4 Tool Evaluation

Chrysalis provides an efficient way to automate the XML refactoring process. It offers a user friendly interface to refactor the XML Schema and to migrate the respective XML documents among the existent versions. It also supplies a programmable interface that can be used directly by applications, which is useful when a lot of documents need to be refactored or when it must be done dynamically.

The project was developed to attend all requirements that [11] described as the criteria to decide whether a refactoring tool is valid. According to this book, the tool should be fast, have support to undo action and be integrated to other tools. Chrysalis took advantage of the Eclipse IDE environment which provides a structure for refactorings with all those characterstics.

This tool is innovative because it puts together changes in the XML Schema and in the XML document, as described before. The existing tools only take care of one of these two parts of the process. For example, the tool OxygenXML [19] offers a set of actions to refactor an XML document, such as creating a new involving element, but it just changes one XML instance and it does support other changes for schema compatibility. Another example is the Eclipse itself, which has a tool for XML Schema refactoring [16]. This tool is limited to renaming and scope change and does not support the data migration of documents that might be affected by these changes. This same kind of refactoring is provided by NetBeans IDE [19] as well. Even through these tools named "refactoring" those functionalities, they are not considered refactorings according to the definition of "data refactoring" provided previously.

The tool was successfully tested in a set of different XML schemas, applying each one of the implemented refactorings. After creating the refactoring in the XML Schema, the transformations were applied into a XML that became valid accordingly to the new XML Schema. It indicates that this technique can be used and automated by a tool. Overall, one can say that Chrysalis accomplishes its main goal: do refactorings into XML documents allowing an application to get the XML in the

desired version, minimizing the need to modify all applications at the same time when refactorings are done. This fact minimizes the impact of changes in the XML Schema in applications that depends on them.

Although, there are still some issues that limit the use of this tool in a real production environment. The following are the ones identified:

- The XML document must be tight coupled to the version control, which can bring problems in cases that a user creates a new document, in a recent version, and forgets to include the schema version.
- The place where the refactoring is applied is strongly coupled to the place where it was created, which means that changing files location can be harmful.
- The XPath weak ability to represent regular expression brings issues to describe some recursive paths, which can generate an infinite set of paths.
- The use of the refactorings in formats that involves more than one XML Schema was not studied and leaved for future works.
- The support of a small set of refactorings limits the tool usage.

Despite the limitations cited above, the developed tool can be extended and evolved to deal with those issues. Consistent with this, Chrysalis can be considered the initial point to a more complete and ready to use in production environment for XML refactoring.

4 Conclusion

System refactoring is undoubtedly a very important theme to software engineering. The need to minimize the impact of an alteration in some part of the system over other ones can be supported by tools that automate the refactoring process [11].

The data used for those applications needs the same attention to be modified. In this context, this work proposes a technique to create changes in an XML Schema providing means for XML documents that are described by them to follow these changes. In order to do that, XSL files that define transformations to implement those changes are created in the process. A version control system was defined to correctly manage and apply the transformations among different versions.

It is important to notice that this methodology can have a positive effect on files that are shared among many applications, which is a common use for XML documents, once an application can retrieve the document in any version regardless the schema version that it is defined. This allows applications that use different schema versions to exchange information among them. As a result, the coupling between the application and the XML document's version that it is consuming is reduced.

The method’s validity was attested by Chrysalis, which is a tool that allows refactorings creation according to the proposed process. It supports the automated refactoring of XML Schema structure and the migration of the respective XML documents, through a user interface or a programmable API. The tool is extensible, enabling the addition of new functionalities and new kinds of refactoring. The tool is fully integrated to Eclipse IDE, which makes it easier to use in existing projects.

The few implemented refactorings intended to validate the viability of the

proposed technique. For the use of this tool in a real development environment, it should be evolved and incremented. For instance, more refactorings should be implemented using the available extension mechanism provided.

References

1. Connoly, D.: Overview of SGML resources. W3 (1995). http://www.w3.org/MarkUp/SGML/
2. Quinn, L.: Extensible Markup Language (XML). W3C (2009). http://people.clarkson.edu/~dhou/courses/EE564-s07/Eclipse-Refactoring.pdf
3. Bosak, J.: XML, Java and the future of the Web. XML.com (1997). http://www.xml.com/pub/a/w3j/s3.bosak.html
4. Erl, Thomas Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River (2005)
5. Booth, D.: Web Services Architecture. W3C (2004). http://www.w3.org/TR/ws-arch/
6. Beck, K.: Extreme Programming Explained: Embrace Change, 2 ed. Addison-Wesley Professional, Boston (2006)
7. Schwaber, K.: Agile Project Management With Scrum, Microsoft Press, Redmond, WA (2004)
8. Kruchten, P.: The rational unified process: an introduction. Addison-Wesley Professional, Boston (2004)
9. Ambler, S.W.: Refactoring for fitness. M-Dev (2002). http://www.ddj.com/windows/184414821?cid=Ambysoft
10. Palmer, S.R., Felsing, M.: A Pratical Guide to Feature-Driven Development. Prentice-Hall, Englewood Cliffs, NJ, USA (2002)
11. Fowler, M., et. al.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co. (1999)
12. Ambler, S.W.: The process of database refactoring. M-Dev (2002). http://www.ddj.com/windows/184414821?cid=Ambysoft
13. Bohannon, P., Freire, J., Roy, P., Simeon, J.: From XML Schema to Relations: A Cost-Based Approach to XML Storage. ICDE (2002).
14. Clark, J.: XSLT Tranformation. W3 (1999) http://www.w3.org/TR/xslt
15. Pereira, M. S., Salerno, G. R.: XML Refactoring repository. http://code.google.com/p/xmlrefactoring/
16. The Eclipse Foundation, http://www.eclipse.org
17. Widmer, T.: Unleashing the Power of Refactoring. IBM Rational Lab, Zurich (2002). http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html
18. Kingler, D.: Creating JFace Wizards. International Business Machines Corp. (2002). http://www.eclipse.org/articles/article.php?file=Article-JFaceWizards/index.html
19. Wheller, S.: <oXygen/> XML Editor 11.0 User Manual. OxygenXML (2002). http://www.oxygenxml.com/doc/ug-oxygen/ln-d4e15.html
20. NetBeans, XML Support in NetBeans IDE. Sun Microsystems (2009), http://xml.netbeans.org

Web Service for Interactive Products and Orders Configuration

António Arrais de Castro[*], Leonilde Varela[1], S. Carmo-Silva[1]

[*,1] Dept. of Production and Systems, University of Minho, Campus de Gualtar, 4710- Braga, Portugal
[*]arraiscastro@gmail.com,[1]{leonilde, scarmo}@dps.uminho.pt

Abstract. Competition in the global economy is intensifying the implementation of the new paradigm of mass customization. This requires a substantial increase in the personalized interaction between client and producer. Agility in the order-production-delivery cycle optimization is a key element for enabling industrial enterprises to meet requirements of this paradigm. The use of agile methodologies requires improving the product specification process and data management. Under today's technology this tends to be predominantly carried out with the aid of the Internet using mostly web services. For a company, this also requires better integration of front-office processes (interaction with the outside) and back office (including production processes). Mass Customization scenarios are characterized by large product variety dependent on specific product requirements set by customers. In this process there is a need to provide the customer with tools for easy product specification, selection and/or configuration. Web-based configurators, can provide an opportunity for both, producer and customer, through an interactive process, for a more formal, faster, effective and better product and orders specification. In this paper, we propose an architecture and describe functionalities of a web service for interactive products and orders configuration. The proposed system can also be a valuable tool for supporting production and delivery.

Keywords: XML, Web Service, Mass Costumization, Product Order Configurator.

1 Introduction

Industrial companies face a continuous challenge of process optimization in order to keep up with growing competition and new demands from customers. Globalization means a large number of business competitors and new challenges that have to be faced in order to maintain market share and improve customer retention. Companies need to implement competitive strategies in order to survive. Achieving cost leadership while maintaining productiveness is a classical key factor for success. Minimizing production costs is not an option but a prerequisite. Simultaneously, minimizing production costs implies minimizing errors and waste.

As a result of this constant need for process optimization several methodologies and techniques focused on production processes and associated back-office systems

have been adopted. These include Six Sigma [1], Work Cells [2] and JIT techniques [3]. However the optimization level achieved at the front-office is much lower, particularly at the customer and partner interaction processes. Front-office activities, like sales and customer support, are no given the required attention by a vast number of companies. By improving front-office activities, these companies can achieve financial and operational benefits by reducing errors, waste and unnecessary manual activities, such as:

- Duplicated efforts
- Incorrect or error-prone information
- Manual process handling
- Bureaucracy, paper based communication
- Rigid processes

The transformation of prospect quotations into customer orders and these into work or production orders and production schedules are frequently associated with manual processes. Additionally, publishing information about company products and associated configurations is often outsourced to external companies responsible for the creation of catalogs and online publishing.

The challenges for process optimization are even higher for large companies. Multinational holdings integrating several small business units, as a result of merging operations, frequently have difficulties integrating front-office processes.

Besides optimizing the integration of front-office and back-office processes, the companies need to adopt a strategy of differentiation. The competitive market dictates that having the lowest prices is no longer enough. The companies have to adapt their products to the specific need of customers. Modern consumers are particularly exigent, demanding higher levels of quality, functionality and correspondence to their specific needs. Traditionally, the inclusion of optional configurations and product variants had more to do with charging “extras” and gaining financial profits than with offering the customer a richer buying experience.

Nowadays there is a growing demand for individualization (a tendency frequently called the “experience economy” [4]. The need to fulfill customers’ individual needs while maintaining prices as low as possible is a permanent challenge for modern enterprises. Companies are forced to create product families with a growing number of variants, adopting a strategy of differentiation based on variety [5].

The concept of Mass Customization was initially defined by Jiao [6] as “producing goods and services to meet individual customers' needs with near mass production efficiency”. Besides offering differentiation opportunities and cost optimization, mass customization allows for a richer interaction with customers, maximizing customer retention. Frank Piller [5] enumerates three distinct and alternative levels of mass customization, connecting them with associated benefits: customized products (differentiation option), efficiency (cost option) and lock-on (relationship option).

A mass customization strategy can only be achieved designing and building modular products, so that each module can be integrated with other modules to build a final product. This generates a larger variety of products to better fit customer preferences. This requires agility in processing information and a high level of integration with existing information systems, particularly at production management

and product conception processes. These requirements imply efficient management of information flows, from customer interaction to production.

Industrial companies can efficiently focus on customers only if they use information intensively [7]. With mass customization principles, the individual wishes and needs of each customer have to be transformed into a unique product specification, according to Piller [5]. The author considers that costs associated with product customization can be determined by quantifying the efforts associated with specifying customers' needs, individual product configuration, transfer of user preferences to the production stages, additional complexity at production management and interactions with suppliers and product distribution.

Companies adopting classical approaches, where products are mass-produced or manufactured to order, need to optimize their information processing strategies in order to evolve to a mass customization model.

Internet technologies offer privileged means for the implementation of interaction channels between client and producers on the mass customization scenario. The advantages can be summarized in two main categories:

a) Improved communication: the client interacts with the producer using a simple and efficient communication channel.

b) Expanded business models.

Using web-based interactive product configurators the customer is able to specify the product that closest matches his/ her individual needs, thus enriching the producers knowledge about market trends.

Outsourcing the time-consuming product configuration process to the customer enables the producer to sell products with reduced financial margins in a way that would be impossible using traditional channels with face-to-face interaction.

2 Literature Review

Several product configuration techniques and methodologies have been proposed during recent years. Some of the techniques have been effectively implemented in information systems operating in industrial sectors.

One of the earliest product configuration systems proposed was XCON [8]. This system uses a rule-based engine that supports product configuration. The system was used by Digital Company to validate and guide the assembly of customer orders. XCON was responsible for the configuration of hardware components (processors, memory, cabinets, power supplies, hard disks, printers, etc.) and software, simultaneously checking marketing restrictions, prerequisites and technical validity.

XCON was fed with a vast set of rules that define how products were structured, their associated restrictions and assembly policies. With this knowledge XCON was able to support interactive selection of generic components, check their completeness, add required parts (if necessary), check software compatibility and prerequisites, check standard compliance, marketing or engineering restrictions and connect to a automate quotation system. It was considered a critical business tool for helping the company to face the continuous challenges imposed by the technological developments and growing technical product complexity.

The main problem with rule based configuration engines is the high cost associated with maintaining their knowledge base [9] and the inclusion of product structure, constraints and policies in a interweaved manner [10].

A natural evolution from rule-base systems are logic-based configurators. These systems use logical models to represent the knowledge needed to solve configuration problems. The PROSE system proposed by McGuinness [11] uses logical operations and semantic rules to represent and manage configuration information. It is based on the CLASSIC language.

An alternative system was proposed by Heinrich and Jungst [12], adopting a resource based paradigm. Components are represented as abstract modules. Starting from the desired functionality, components that partially fulfill its requirements are aggregated. The complete set of components will fulfill the required functionality completely.

Researchers Mittal and Falkenhainer [13] proposed representing a configuration problem using a constraint-based approach. Configurations are obtained from a set of variables, a domain of possible values for each variable and an associated constraint set. This model is known as a CSP – constraint satisfaction problem. Mailharro [14] proposes a CSP approach using an integrated object model, supporting classification methods, inheritance and algorithms to maintain configuration consistency.

Other authors propose alternative approaches. Mittal and Frayman [15] classify the configuration problem as a case based reasoning (CBR) problem. Basically, the technique consists on using the knowledge associated with previous cases, based on similarity criteria [16-18]. A "case" is just a set of existing configurations. When trying to solve a configuration problem, the system tries to find an identical case among the configurations that have been stored, adapting it to the new requirements. CBR configuration is mostly useful when the knowledge is incomplete [19]. Reusing product configuration knowledge and constraints is not supported. This can be a limiting factor when working with a vast and complex product catalog.

The OOKP - "one of a kind" is proposed by other authors as an interesting alternative to the previous methods [20]. Product variations and configuration parameters are modeled using AND-OR trees. Tree nodes contain the parameters associated with the rules. The configuration is built and optimized using a genetic algorithm that considered customer preferences and efficiency parameters. Zhou [21] performed identical work, incorporating configuration restrictions in the genetic algorithm (for example, inclusion and exclusion relations) trying to achieve a higher level of optimization. An identical strategy has been adopted by other researchers [22, 23].

Solving the configuration problem is one part of product configurator's requirements. Another important part is interacting with the customer, particularly when the customer has a reduced knowledge about the configuration details.

Luo [9] proposed the implementation of a matrix mapping technical customer requirements with technical product attributes, thus allowing for automatic selection of the most adequate configurations. Using this matrix the customer can delegate on the system the product selection and configuration process.

To help the customer choose the most adequate product configuration, recommendation systems are frequently used. There are several studies describing theses systems [24-26]. A vast number of internet portals available today help their

customers while choosing or configuring their products using other customers' recommendations. These recommendations are frequently used for cross-selling (namely through Amazon.com and Pixmania.com sites).

A good recommendation system will help hiding technical details from the most inexperienced users, helping them configure their selections. The recommendation process can be implemented with an intelligent software agent that learns from customer product configuration processes and helps them choose the best options.

The majority of recommendation systems are hybrid and combine recommendations based on content (product parameters and customer relationship history) and collaborative methods (analyzing customer and product relations). Collaborative methods are popular since they allow using the knowledge captured during the interaction with other customers in distinct business contexts.

There are several recommendation techniques based on collaborative methods: graph theory [27], linear regression [28], and semantic analysis [29].

Although being very useful, recommendation techniques have several limitations, namely:

- Lack of knowledge about new products.
- Influenced by incorrect or manipulated recommendations published by other customers.

These problems suggest not using the collaboration techniques exclusively. Therefore, a better approach consists on complementing them with content-based recommendation methods.

3 Motivation

Several product configuration techniques and methodologies have been proposed during recent years. The implementation of a web service based product configuration system will allow creation of a central channel for the distribution of information about compound and simple products, supporting customers and business partners with their configuration and product selection needs. Implementing product design by using a set of configurable modules also enables the system to support the implementation of mass customization strategies [30], generating value from an optimized interaction with customers and business partners.

This communication channel can be used to automatically feed distinct point of sale (POS) systems, including: websites, interactive product configurators for physical points of sale, and promotional interactive media (DVDs, CDs, etc.).

Additionally, the system will connect these POS systems with management back-ends, including CRM and production management systems.

Several benefits can be realized in the sales channel, in terms of the following aspects.

- Customer: Interactive product selection and configuration. Better fulfillment of individual needs. Better product information, continuously updated. Better order support.

- Resellers: Configuration opportunities for tailored products. Permanent access to updated product and pricing information. No need to import product configuration

information as it can be dynamically obtained from web services. Better support for electronic interaction, minimizing paper-based processes. Better B2B integration with producer.

- Producer/ Sales: Wider range of product options. Better understanding of customers' needs and preferences. Maximized business opportunities, including support for niche markets.
- Producer/ Order processing: Reduced information about capture errors. Delegation of the configuration process to the system and customer.
- Producer/ Engineering: Greater opportunities for modular design. Maximized component reuse. Easier conversion of sales- orders into production- orders.

Some of these techniques have already been effectively implemented in information systems operating in industrial sectors.

4 Process Agility

The main component of the product configuration system is a web service layer that connects with the company's information systems (data import) and feeds external systems (data export). This layer will expose configuration engine functionality, facilitating the order capture process and its conversion to production orders. It will be a valuable tool to support the implementation of methodologies that require agile product and order information processing.

The system will also help the implementation of agile production methods and approaches, like lean manufacturing. This approach is derived from TPS – Toyota production system, developed by Toyota after World War II. It aims to eliminate waste, targeting for eliminating any activities that do not create value for the end customer. The method has two pillar concepts: automation and JIT – just in time. JIT was introduced by Taiichi Ohno, chief engineer at Toyota in the 1950s, as a method for the fulfillment of customers' needs with a minimum of delays, while reducing in-process inventory. With JIT, inventory is seen as incurring cost and waste, instead of stored value. Nothing should be produced, transported or bought before the exact time. It all comes to having "the right material, at the right time, at the right place, and in the exact amount".

The make-to-order philosophy that JIT implies minimizes risks of having unsold products. Web based product configurators improve agility while capturing sales orders and converting them into production orders, thus helping to implement JIT.

Agile manufacturing was designed to make businesses more flexible, requiring a more dynamic process for production operations, setup and lead-time reduction and a greater efficiency while managing product information. An agile company needs to determine customer needs quickly and continuously repositioning itself against its competitors. This means that it also needs to design products quickly based on customers' individual needs.

The web service based product configuration system will help capture customers' needs and preferences continuously, thus contributing to the agile processes.

5 System Architecture

The product configuration system (WBPC - Web Based Product Configurator) includes two main components:

- Integration module (responsible for integration with existing information systems), and.
- Service module (web service layer that interacts with external systems, client and server applications).

Integration will be built using connection modules called connectors. Several connectors will be available for reading and writing information from/ into external systems. Each read connector is responsible for reading the information available in a backend system and transform this information into XML. The write connectors receive XML data and convert it to a format that is adequate for the associated system. XML Schemas (XSD) will be used to help validate the information and check its completeness and integrity, as illustrated in Figure 1.

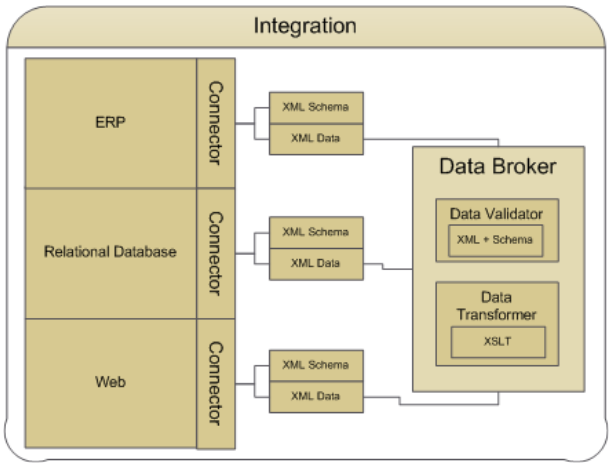


Fig. 1. Integration process.

The data broker component is the central data processing module and controls how data import and export is managed. It includes two sub-components: data validator and data transformer. The data broker is isolated from the details of each external system, accepting only XML information. Data validator maps the information received by the broker to the corresponding XML Schema. Any validation errors cause the rejection of the underlying data.

The data transformer module is responsible for making the imported information compatible with the structure the service broker expects to receive. It uses XSLT StyleSheets (XSL Transformation), applying them to the previously validated XML data, and forwarding the transformation results to the web service layer.

The data broker component can be replaced by a third party middleware solution, like TIBCO (www.tibco.com) or BizTalk Server (www.microsoft.com/biztalk). These solutions will help modeling business rules at the integration level.

The web services layer is responsible for the implementation of functionalities that support product configuration and information publishing, including four main components, as shown in Figure 2.

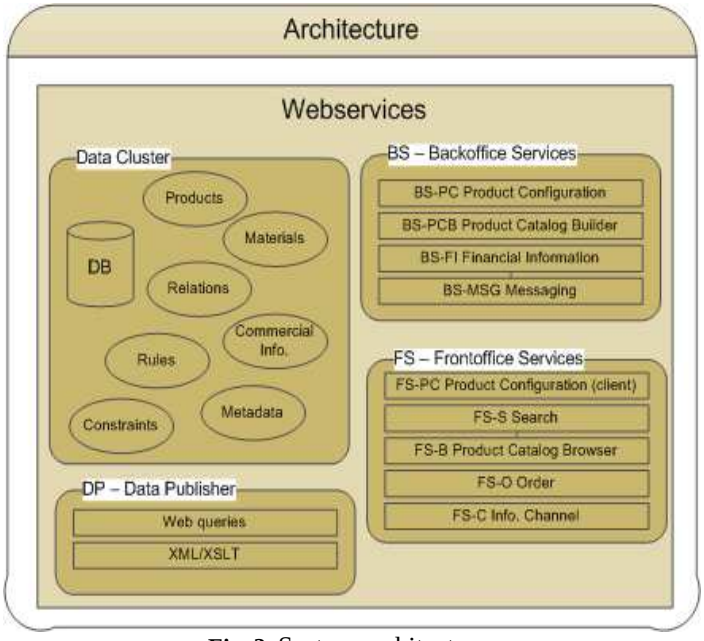


Fig. 2. System architecture.

1. BS – Back-office services, including: BS-PC (product configuration services), BS-PCB (product catalog builder), BS-FI (financial information management) and BS-MSG (messaging engine for notifications and logging).
2. FS – Front-office services, including: FS-PC (product configuration services), FS-S (product search), FS-B (product catalog browser), FS-O (order processing) and FS-C (information channel publishing, i.e., information channels that aggregate one or more product categories).
3. DP – Data publisher, including: web queries (that allow external systems to import information and reporting data, and also scalar or tabular results are available), and XML/XSLT (web queries that return XML results, which supports result transformation using XSLT stylesheets to adequate information for external systems).
4. Data Cluster - Data storage (relational model, with redundancy).

Using XSD (XML Schemas), external systems can validate the data structure that they receive in XML format. This validation would be performed by integration connectors preferably. The following diagram represented in Figure 3 illustrates how an order would be validated before inserting it in the ERP external system.

During product planning the modules and product constraints are defined. The outputs control how products can be built, according to constraints and business policies. A set of pre-approved configurations can be published, although it is not

mandatory. Later the list of configurations can be extended with configurations captured from user interaction.

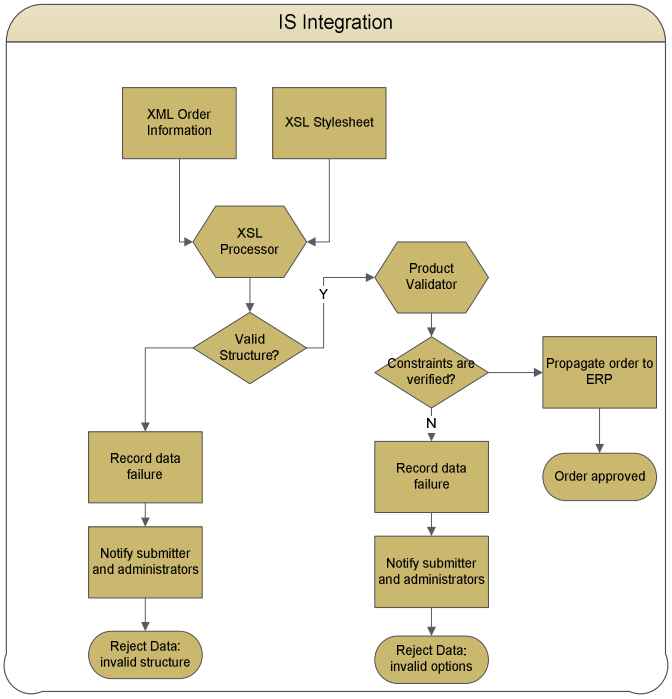


Fig. 3. Integration connectors.

Client applications' only requirement is the ability to call web methods using HTTP and SOAP, and process the returning data (XML formatted).

The diagram, in Figure 4, represents a business scenario where three external applications consume the services published by WBPC:

1. Client back-office application: windows forms application used to administer the system, performing typical activities, by using configuration engine (component management, properties management, constraint definition and management, and typical requirements management, also includes product configuration building and catalog building, configuration of data broker modules, security administration, business information management and reporting.

2. Client Front-office application: client application that the customer or business partner uses to interact with the system. Including, plataforms: web, windows, linux, mobile, etc. Several typical activities for product configuration, order submission, product catalog browse, and product search.

3. External Application: external information system, which uses data publisher to import data into local databases or other systems and typical activities include: import orders into ERP system, product data import into external applications, reporting and statistical processing.

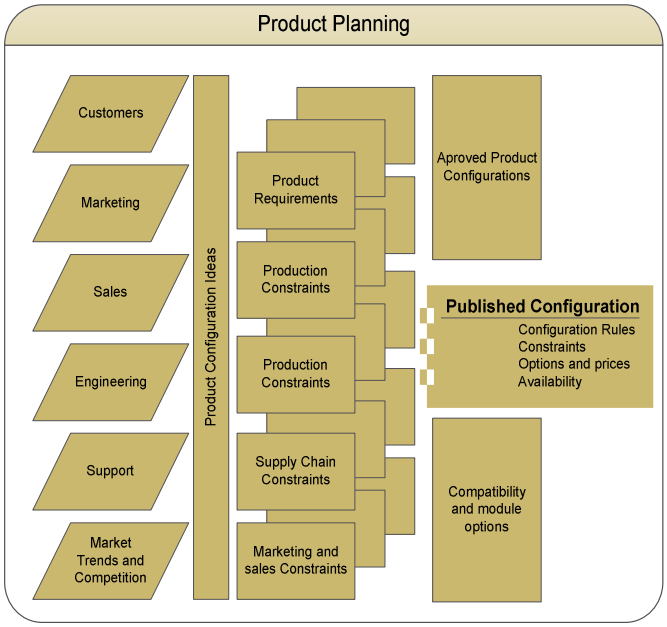


Fig. 4. Business scenario.

6 Conclusion

It is our conviction that the proposed system will be a valuable tool to support the adoption of mass customization with minimum impact on existing information systems. The companies will obviously need to adopt agile production methods in order to quickly respond to customers' orders and configurations.

Promoting increased individualized interaction with customers and business partners will allow companies to obtain higher levels of customer retention.

Integrating web service based product configuration systems with their backend systems, including production and enterprise resource planning, will allow for greater cost savings and more agile processes. This level of integration will contribute to an efficient make-to-order strategy minimizing inventory costs. Adopting integration mechanisms, like a middleware layer, will support a richer level of integration with external information systems, minimizing information processing costs.

An effective mass customization strategy will only be possible if the company builds a successful information flow, covering the configuration process, product knowledge base, marketing and production restrictions and business rules and constraints.

Adopting recommendation techniques will help customers without significant knowledge about product details to express their needs and obtain a configuration that fulfills them. This way personalized interaction and individualization will not be exclusive for effective clients (who have sufficient knowledge about the products) and

business partners, supporting new clients that know a lot about their needs and frequently know very little about product details and associated constraints.

The adoption of web standards widely used and supported will facilitate the evolution of the model and an easier implementation in different scenarios. The impact on existing information systems and IT environment will be minimal if the system architecture includes specialized connectors that exclusively know the details on how the information is stored in associated systems.

A modular approach to the configuration system will support the adoption of different configuration engines, depending on the product complexities and associated constraints.

Using adequate authentication and authorization mechanisms, the producer will support web service consumption from a wide range of heterogeneous clients, supporting the development of multi-channel configuration platforms.

References

1 Horman, P. (2007) "Business process change: a guide for business managers and BPM and six sigma professionals", editor: Morgan Kaufmann, Burlington.
2 Hyer, N., and Wemmerlov, U. (2002) Reorganizing the factory: competing through cellular manufacturing", Productivity Press, NY-USA.
3 Monden, Y. (1983). Toyota Production System, Industrial Engineering and Management Press, ISBN 0-89806-034-6.
4 Pine, J. and Gilmore, J. (1999), "The Experience Economy", Harvard Business School Press, Boston.
5 Piller, F. (2002), "Customer interaction and digitizability - a structural approach to mass customization", Rautenstrauch et al. (ed.): Moving towards mass customization, Springer:Heidelberg/Berlin/New York, p.119-138.
6 Jiao, J. and Helander, M.G. (2006), "Development of an electronic configure-to-order platform for customized product development", Computers in Industry, v.57 n.3 (Abr), p.231-244
7 Blattberg, R.C. and Glazer, R. (1994), "Marketing in the Information Revolution, The Marketing Information Revolution", R.C. Blattberg et al. (eds.), Boston: Harvard Business School Press, p: 9-29.
8 Barker, V.E. and O'Connor D.E., (1989), "Expert systems for configuration at Digital: XCON and beyond." Communications of the ACM, 32(3), p: 298–318.
9 Luo, X and Tu, Y. and Tang, J and Kwong, C. (2008), "Optimizing customer's selection for configurable product in B2C e-commerce application", Computers in Industry, v.59 , n. 8 (Out), p.767-776
10 Sabin, D., and Weigel, R. (1998), "Product configuration frameworks — A survey. IEEE
11 McGuinness, D. and Wright, J. (1998), "An industrial-strength description logic-based configurator platform", IEEE Intelligent Systems 13 (4), p: 69–77.
12 Heinrich, M. and Jungst, E. (1991), "A resource-based paradigm for the configuring of technical systems from modular components", Proceedings of Seventh IEEE Conference Artificial Intelligence Applications, Miami Beach, FL, USA
13 Mittal, S. and Falkenhainer, B. (1991), "Dynamic constraint satisfaction problems", Proceedings of the Eighth National Conference on Artificial Intelligence, Boston, MA, USA
14 Mailharro, D. (1998), "A classification and constraint-based framework for configuration", Artificial Intelligence for Engineering Design, Analysis and Manufacturing 12 (4), p: 383–395.

15 Mittal, S. and Frayman F. (1989), "Towards a generic model of configuration tasks", Proceedings of the 11th international joint conference on artificial intelligence".
16 Tseng, M. and Jiao, J. (2001), "Mass Customization", Handbook of Industrial Engineering, Technology and Operation Management (3rd Ed.)
17 Lee, H.J. and Lee, J.K. (2005), "An effective customization procedure with configurable standard models", Decision Support Systems 2005, 41(1), p: 262–78.
18 Hiramatsu, A. and Naito, A. and Ikkai, Y. and Ohkawa, T. and Komoda, N. (1997), "Case based function tree generator for client-server systems configuration design", Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Orlando, FL, USA
19 Yang, D. and Dong, M. and Miao, E. (2008) , "Development of a product configuration system with an ontology-based approach", Computer-Aided Design, v.40 n.8 (Ago), p.863-878
20 Hong, G. and Hu, L. and Xue, D. and Tu, Y. and Xiong, Y. (2007), "Identification of the optimal product configuration and parameters based on individual customer requirements on performance and costs in one-of-a-kind production", International Journal of Production Research, p:1–30.
21 Zhou, C. and Lin, Z. and Liu C. (2005), "Customer-driven product configuration optimization for assemble-to-order manufacturing enterprises", The International Journal of Advanced Manufacturing Technology.
22 Li, B. and Chen, L. and Huang, Z. and Zhong, Y. (2006), "Product configuration optimization using a multiobjective genetic algorithm, The International Journal of Advanced Manufacturing Technology, 30(1–2):20–9.
23 Yeh, J. and Wu, T. and Chang, J. (2007), "Parallel genetic algorithms for product configuration management on PC cluster systems", The International Journal of Advanced Manufacturing Technology, 31(11–12)
24 Hill, W. and Stead, L. and Rosenstein, M. and Furnas, G. (1995), "Recommending and evaluating choices in a virtual community of use", Proceedings of the 1995 ACM Conference on Factors in Computing Systems, New York, USA
25 Shardanand, U. and Maes, P. (1995), "Social information filtering: algorithms for automating word of mouth", Proceedings of the 1995 Conference on Human Factors in Computing Systems, Denver, CO, USA.
26 Srivastava, J. and Cooley, R. and Deshpande, M. and Tan, P. (2000), "Web usage mining: discovery and applications of usage patterns from Web data", SIGKDD Explorations 1 (2), p: 1–12.
27 Aggarwal, C. and Wolf, J. and Wu, K. and Yu, P. (1999), "Horting hatches an egg: a new graphtheoretic approach to collaborative filtering", Proceedings of the Fifth ACM SIGKDD International Conference, San Diego, CA
28 Sarwar, B. and Karypis, G. e Konstan, J. e Riedl, J. (2001), "Item-based collaborative filtering recommendation algorithms", Proceedings of the 10th International Conference on World Wide Web, Hong Kong.
29 Hofmann, T. (2004), "Latent semantic models for collaborative filtering", ACM Transactions on Information Systems, 22 (1), p:89–115.
30 Tseng, H. and Chang, C. and Chang, S. (2005), "Applying case-based reasoning for product configuration in mass customization environments", Expert Systems with Applications; 29:913–25.

Processing XML: a rewriting system approach

Alberto Simões[1] and José João Almeida[2]

[1] Esc. Sup. de Est. Industriais e de Gestão
Instituto Politécnico do Porto
alberto.simoes@eu.ipp.pt
[2] Departamento de Informática
Universidade do Minho
jj@di.uminho.pt

Abstract. Nowadays XML processing is performed using one of two approaches: using the SAX (Simple API for XML) or using the DOM (Document Object Model). While these two approaches are adequate for most cases there are situations where other approaches can make the solution easier to write, read and, therefore, to maintain.
This document presents a rewriting approach for XML documents processing, focusing the tasks of transforming XML documents (into other XML formats or other textual documents) and the task of rewriting other textual formats into XML dialects.
These approaches were validated with some case studies, ranging from an XML authoring tool to a dictionary publishing mechanism.

1 Introduction

Nowadays we can assert that most XML manipulation is done using the Document Object Model (DOM) approach. The Simple API for XML (SAX) approach is also used and is, usually, the best solution for parsing big documents.

This document presents yet another approach specially tailored to transform XML documents into other but similar XML documents, or to transform any kind of textual documents into structured XML documents. We present a `Text::RewriteRules`, a rewriting system written in Perl. While it can be used to rewrite and produce any kind of textual document, in this article we will focus on rewriting XML documents, and producing XML from different kinds of textual documents.

While there are other rewriting systems (say AWK or XSLT) they lack the flexibility of the Perl programming language and, especially, of Perl regular expressions engine. Also, they do not support both rewriting approaches. Finally, XSLT approach is not to rewrite the XML document looking at its syntax, but looking to its tree, making the parsing indispensable.

The next section focuses on `Text::RewriteRules` engine, explaining the algorithms available and how rules are written. Follows a section on XML generation from two different kinds of textual documents and in section 4 we will discuss how to rewrite XML into other XML formats.

May, 2010 - Page. 27–38 ≻ATA 2010 — ISBN: 978-972-99166-9-4

2 Rewriting Engine

Although the approaches described in this article can be implemented using any rewriting system, given it supports the regular expressions we will describe shortly, for our experiments we are using `Text::RewriteRules`, a Perl module for coding rewriting systems. `Text::RewriteRules` derived from previous work, where a text-to-speech application was built as the composition of different rewriting systems [2].

The basic concept of a rewriting system is simple: given a text T and a sequence of pairs $(pattern, action)$, check if there exists a pattern matching the text. Every time it does, the associated action is executed.

The usual substitution operator/function that is available on most languages can be seen as a rewriting system with just one rule. Every time the pattern matches a portion of text, this text is substituted by a string (or the result of invoking a function on the matched text). If we take a sequence of substitutions and force them to be applied until all substitutions can not be done, we have a simple but complete rewrite system.

While this simple solution can be used for some applications it is not flexible enough. On some situations a simple pattern can not be used to check if a substitution can be performed or not. We might want to check a database table or any other resource to decide if the substitution should or not be performed. Also, this database might have the information about what should be used to substitute the matched text.

For this to be possible we need some more information, other than the *pattern* and the *action*: we need a *condition*. This condition should be able to perform any kind of computation it needs, ranging from querying a database to performing a web-query in a search- engine. Thus, our rules will be defined as triples: $(pattern, action, condition)$.

This simple concept can be expanded with extra functionalities like the ability to execute code in the *action* or activate some *action* before any *pattern* is tested.

`Text::RewriteRules` supports two different working mechanisms:

- **Fixed-Point Approach**: apply rules in the order they appear, exhausting the first ones before trying to apply the following ones. The system iterates until no pattern matches the text;
- **Sliding-Window Approach**: apply rules in the order they appear, forcing that the matching text is at the beginning of the string. As soon as a pattern matches, a cursor is placed right after the match, and the next rule should match in the cursor position. If no rule can be applied the cursor advances (a character or a word accordingly with the user needs). This system iterates until the cursor arrives at the end of the string.

2.1 Fixed-Point Approach

The fixed-point approach is the easier to understand but not necessarily the most useful. Its idea is based on a sequence of rules that are applied by order.

The first rules are applied, and following rules are applied only then there is no previous rule that can be applied. It might happen that a rule changes the document in a way that the previous rules can be applied again. If that happens, they will be applied again. This process ends when there is no rule that can be applied (or if a specific rule forces the system to end).

This algorithm is specially useful when we want to substitute all occurrences of some specific pattern to something completely different. As an example, consider the e-mail anonymization for mailing list public archives:

```
RULES anonymize
\w+(\.\w+)*@\w+\.\w+(\.\w+)*==>[[hidden email]]
ENDRULES
```

This simple rule will substitute all occurrences of the specified pattern[3] by the specified text. As this process intends to substitute all occurrences of e-mails, and the replacement text does not match the e-mail rule, we do not have the problem of the rewriting system entering an endless loop.

2.2 Sliding-Window Approach

The sliding-window approach tries to match patterns right after the position of a specific cursor, and after some portion is rewritten, the cursor is put just after that portion. So, the sliding-window approach will never rewrite text that was already rewritten.

When using this approach the user does not need to position the cursor. It is automatically initialized in the beginning of the text, and is automatically placed after the matching portion every time a substitution is made. If no rule can be applied at the cursor position it is automatically moved ahead one character (or one word).

This approach is especially useful when the rewrite rules output can be matched by some other rule that will mess up the document.

As a concrete example, consider a rewriting system that does brute-force translation (translates each word using a bilingual dictionary). After translating the English word '*date*' to the Portuguese word '*data*' we do not want it to be translated again as if it were an English word.

To solve this problem we might add a mark to each translated word, and removed all of them at the end. Another option is to use a sliding-window approach: we translate the word at the cursor position, and move it to the right of the translated word.

Follows an example of how this works[4]

[3] This is a simple pattern that does not cover all e-mail cases, but good enough for illustrating the DSL.

[4] While the underscore character is being used as the cursor in this example, `Text::RewriteRules` uses a special non printable character, not found on normal text.

```
_ latest train
último _ train
último combóio _
```

To write this translator using `Text::RewriteRules` we just need to use a flag when defining the rule, so that `Text::RewriteRules` knows it should use a sliding window.

```
RULES/m translate
(\w+)=e=> $translation{$1} !! exists($translation{$1})
ENDRULES
```

2.3 `Text::RewriteRules` Rules

`Text::RewriteRules` programs can include one or more set of rules. After compilation, each rules set will generate a function that receives a text, rewrites it, and returns the resulting text.

The generated functions can be composed with standard functions, or with other rewriting functions.

There are different kind of rules: simple substitution, substitution with code evaluation, conditional substitutions, and others. In the *fixed-point* behavior, the system will rewrite the document until no rule matches, or a specific rule makes the system exit.

Each rewrite system is enclosed between `RULES` and `ENDRULES` strings, as in the following example:

```
RULES xpto
left hand side pattern ==> right hand side
left hand side pattern =e=> right hand side
left hand side pattern ==> right hand  !!  condition
ENDRULES
```

This block evaluates to a function named `xpto` that is the requested rewrite system. It is possible to pass extra arguments to the RULES block adding them after a slash like:

```
RULES/m xpto
left hand side pattern ==> right hand side
left hand side pattern =e=> right hand side
left hand side pattern ==> right hand  !!  condition
ENDRULES
```

The `/m` argument is used to change the default fixed-point algorithm to the sliding windows one.

2.4 Rule Types

Regarding rules, there are very different kinds. To help understanding examples in the next sections a quick presentation of the more relevant rules follows:

- simple pattern substitution rules are represented by the `==>` arrow. The left side includes a Perl regular expression and the right side includes a string that will be used to replace the match. This string can use Perl variables, both global or captured values. These simple rules can include restrictions of application. These restrictions are added at the end of the rule after `!!`. This condition can use global variables or captured values and is written in common Perl syntax.
- in some situations it is important to be able to evaluate the right side of the rule, calculating the value to be used based on portions of the matched string. Evaluation rules are denoted by a `=e=>` arrow. These rules allow conditions as well.
- there are two special rules to be used just the first time the function is called, or to force the rewrite system to exit. The first is the `=begin=>` rule. It has no left side and the right side is Perl code that changes in some manner the text that will be rewritten. The `=last=>` rule does not have a right hand side. It just exits from the rewrite system as soon as the left hand side regular expression matches.

2.5 Recursive Regular Expressions

What makes `Text::RewriteRules` suitable for rewriting XML, as shown in section 4, is the Perl ability to define recursive regular expressions.

Fortunately, regular expressions are not regular anymore [3]. The old definition of regular expressions and their direct conversion to automata is no longer the rule when talking about scripting languages regular expression engines.

Languages like Perl support extensions that make their *regular expressions* powerful tools, making them comparable to grammars, supporting capturing and referencing, look-ahead, look-behind and recursion:

- regular expressions can define capture zones: pieces of the regular expression that, after matching, will be stored for latter usage;
- regular expressions can define look-ahead or look-behind, making the regular expression to match just if a specific expression is (or not) before or after the matching zone;
- recursion was introduced in the Perl world with Perl 5.10[5] and lets the user to specify a regular expression that depends on itself. As an example, consider the following expression that matches a balanced parenthesis block:[6]

```
my $parens = qr/(\((?:[^()]++|(?-1))*+\))/;
```

[5] About January 2008.

[6] We will not explain the regular expression as that would take too much space. You are invited to read Perl man-page `perlre`.

`Text::RewriteRules` includes a set of engineered (regular) expressions that make the language rewriting task easier:

- `[[:BB:]]`, `[[:PB:]]` and `[[:CBB:]]` match balanced bracketed blocks, balanced parenthesized blocks and balanced curly braces blocks, respectively;
- `[[:XML:]]` and `[[:XML(tag):]]` match well formed XML fragment. The latter forces the root element name.

These expressions do not just match, but also capture. For the balanced pairs it is possible to automatically capture its contents and for the XML fragments it is possible to capture the top-level tag name, as well as the top-level tag contents.

2.6 XML Generation

For simpler XML generation from Perl code we will use `XML::Writer::Simple` [1]. This Perl module allows the usage of a Perl function for each tag in use, making XML elements generation as simple as the invocation of a function. Although this module is not part of the rewriting system it helped reducing the code size and raising code legibility.

3 Rewriting Text into XML

This section presents two different situations where the ability to rewrite textual documents and produce XML was the fastest solution. While the two examples share the approach, they differ on the objective:

- the first case study rewrites a textual format into another TEI (Text Encoding Initiative) format, maintaining the existing structure but also detecting and annotating new information;
- follows the creation of a DSL (Domain Specific Language) [6] for XML authoring. In this case the rewriting system acts like a computer language compiler.

3.1 Annotating a Textual Dictionary

Dicionário-Aberto [5] is a project aiming to transcribe a general language dictionary. The transcribing process was performed by volunteers using a textual syntax, based only in bold and italic mark-up (using asterisks and underscores, respectively), new lines to separate entry senses and empty lines to separate word entries.

This basic syntax was chosen so volunteers (with different degrees of knowledge) could transcribe easily. The drawback is the lack of annotation on the resulting textual document.

One of the best formats to describe general dictionaries using any kind of mark-up is the Text Encoding Initiative XML Schema [7]. As a brief analysis of the schema can show, this format has a rich dictionary structure.

Our challenge was to find a method to transform the few annotated dictionary format into TEI. The solution was to use a rewrite approach, enhancing the textual format step by step.

The rewriting system is too big to be presented completely in this article. It contains more than 40 rules. Briefly, the system acts in this order:

1. different entries are separated using the empty line separator. A start and end tag is glued to the result of processing the contents of that entry. Also, some extra end tags are added, closing the definition and sense;
2. follows a set of rules to detect morphologic properties. These properties are in italic. Therefore, as described earlier, they are bounded in underscores. Unfortunately there is more text in italic. To distinguish them, lists of the used morphologic properties and geographic classifiers were created. These lists were used to rewrite morphologic properties into the corresponding TEI tags;
3. finally there is a bunch of rules to fix the generated XML. For instance, after the morphological information it is needed a definition opening tag. This is performed finding all morphologic information closing tags and checking if they are followed by the definition opening tag.

All this process is performed by rewrite, with regular expressions being matched and rewriting the document contents accordingly. This kind of analysis of the document would be really hard to perform with standard parsing techniques[7].

To help the legibility of the rewriting system the `XML::Writer::Simple` module is used. Instead of defining textually the XML being generated, this module defines automatically functions for each tag. These functions generate the opening and closing tag with the same name (eg. a `gramGrp` function would generate a pair of `gramGrp` opening and closing tags). This module functions also take care of generating empty elements when no content is supplied, and generating correctly tag attributes.

3.2 An XML Authoring Tool

XML was designed to be an exchange format. Its syntax enhances the information structure representation, but it has readability (and human authoring) problems. Manual XML authoring would be easier if we could:

- reduce the size of structural information;
- create abreviation mechanisms for constant parts or parametric macros;
- create include mechanisms;
- support scripting capabilities;

This subsection will discuss how to develop a simple DSL for XML documents authoring.

Although the basic principles are generic, the examples will use HTML dialect, in order to be easier to follow.

[7] Unfortunately and given article page limits no example of rules are shown. The complete conversion script can be downloaded from `https://natura.di.uminho.pt/svn/main/ProjectoDicionario/txt2xml`.

Generic transformations The first type of constructs added to XPL (XML Programming Language, the name of the language we are defining) just change the syntactic sugar for XML tags, from the usual start and end tag to a simple function-oriented syntax.

Basically, instead of writing the full XML tags like

```
<h1>XML programming language</h1>

<ul><li>DSL</li>
    <li>see the <a href='...'> XPL manual</a> </li> </ul>
```

XPL lets the user to write

```
h1{XML programming language}

ul{li{DSL}
   li{see the a{href:{...} XPL manual}}}
```

To implement this syntax using `Text::RewriteRules` we defined the following Perl code:

```
1 my $ID = qr{\w+};                      # Identifier
2
3 while(<>) { print loadit(html($_)) }
4
5 RULES html
6 ($ID)[[:CBB:]]          ==>  <$1>$+{CBB}</$1>
7 <(.*?)>($ID):[[:CBB:]] ==>  <$1 $2='$+{CBB}'>
8 (\\[{}])                =e=> saveit($1)     # protect escaped \{
9 ENDRULES
```

Explanatory notes:

line 3 rewrite the standard input with the `html` rewrite system, and reload the saved portions.
lines 5–9 define the `html` rewrite system (that generates the `html` function).
line 6 expand tags (`a{b}` to `<a>b</b>`);
line 7 treat attributes (transform `<a>at:{b}...` in `<a at='b'>...`);

This rewriting system uses two new functions available in `Text::RewriteRules`:

- `saveit` takes a string and saves it in a symbol table. It returns a special token that will be placed in the text being rewritten. This is specially useful to protect portions of text that would be otherwise rewritten by subsequent rules.
- `loadit` performs the inverse operation, replacing all occurrences of the special tokens by the respective stored string.

Abbreviation mechanisms In order to provide programmability constructs, the rewriting system will use an auxiliary table (named `TAB`) to store user-defined functions.

```
1 RULES html
2 ...
3 ($ID)=[[:CBB:]] =e=> savefunc($1,$+{CBB},1),""
4 ($ID)[[:CBB:]]  =e=> $TAB{$1}->($+{CBB})  !! defined $TAB{$1}
```

Explanatory notes:

line 3 support function definition (`f={...}`).The function `savefunc`, (not presented here) inserts a function definition in the auxiliary symbol table.

line 4 detect function invocations (`f{arg}`) and evaluate them (if the function is defined in the symbol table).

Adding these two rules to the `html` rewrite system it is now possible to define macros like the following ones:

```
r   ={font{color:{red} #1}}                  # red text
q   ={div{class:{boxed} #1}}                 # to use CSS box
jpgi={img{src:{#1.jpg} alt:{image of a #1}}} # jpeg image

q{ p{r{Animals:} jpgi{cat} jpgi{donkey}} }
```

Note that `#1` is the argument for the macro, idea stollen from TEX. After processing this document the resulting XHTML document will look like:

```
<div class='boxed'>
  <p>
      <font color='red'> Animals:</font>
      <img src='cat.jpg' alt='image of a cat'></img>
      <img src='donkey.jpg' alt='image of a donkey'></img></li>
  </p>
</div>
```

Include mechanisms Finally, the language also supports different types of include mechanisms:

- split the code in order to reduce the size of the source document and have document modularity (just like `#include` in the C programming language);
- to separate different concepts (example: store function definitions in a *new notation* block;
- to include verbatim examples of code (like \`verbatiminput` in TEX);

For XPL we defined two include mechanisms: `inc` – include, and `vinc` – verbatim include. `inc` provides modularity support:

```
RULES html
...
inc[[:CBB:]]  =e=> ‘cat $+{CBB}‘          !! -f $+{CBB}
inc[[:CBB:]]  =e=> die("can’t open file") !! not -f $+{CBB}
```

The inclusion is just the substitution of the include command by the file contents. The rewriting system will take care to process the new commands in next iterations.

For including verbatim files we defined a second rewrite system, that will replace this verbatim include command by the file contents. This could not be done at the `html` function level as other rewrite rules would rewrite the file contents. The inclusion also needs to protect special characters for HTML inclusion.

```
1  while(<>){ print loadit(verbatim(html($_))) }
2
3  RULES/m verbatim
4  <vinc>(.*?)</vinc>=e=> bpre( protect( ‘cat $1‘ ))!! -f $1
5  ENDRULES
6
7  RULES/m protect
8  <==>\&lt;
9  >==>\&gt;
10 &==>\&amp;
11 ENDRULES
```

Explanatory notes:

line 4 `protect` is rewriting the contents of the file being included.
lines 8–10 transforms HTML special characters into entities.

Scripting embbeding XPL also supports Perl code embbeding, using the `perl` command. Its implementation is just the code execution, and substitution by the respective result string.

```
RULES html
perl[[:CBB:]]=e=> do($+{CBB}),$@ !! -f $+{CBB}
...
ENDRULES
```

4 Rewriting XML into XML

`Text::RewriteRules` can easily rewrite XML documents both as if they were plain text documents or using the defined recursive regular expressions for XML.

These regular expressions are tailored so that they match well formed XML blocks making it easier to remove or replace complete elements subtrees.

Textual XML rewriting is not necessarily faster or more efficient than DOM oriented processing. It all depends on the document size and the DOM structure. But textual XML rewriting can be easier to understand and can be more powerful on some specific situations. Also, these two approaches are not mutually exclusive. One can process the document using the rewrite approach to detect the relevant elements to process, and use a DOM aware tool to process that XML fragment. This is the approach we will show in the next example.

The example will remove duplicate entries from a TMX (Translation Memory eXchange) file. These files size is, usually, quite large, but the structure is quite simple and repetitive. They consist of a list of translation units: pairs of sentences, in two different languages.

```
RULES/m duplicates
([[:XML(tu):]])==>!!duplicate($1)
ENDRULES

sub duplicate {
  my $tu = shift;
  my $tumd5 = md5(dtstring($tu, -default => $c));
  return 1 if exists $visited{$tumd5};
  $visited{$tumd5}++
  return 0;
}
```

In the example the rewritting system is very simple. It matches XML trees that have as the tag `tu` as root element, and substitutes it by nothing in case of it is duplicate. The duplication mechanism processes the translation unit using a DOM approach (using `XML::DT` module [4]), concatenating the translation unit contents, and calculating its MD5. If it is already visited a true value is returned. If not, that MD5 is saved for future reference.

5 Conclusions

The presented case studies shown that the text rewriting approach is a powerful technique to convert textual document formats. Also, the ability to use a module to generate XML tags easily and the availability of recursive regular expressions matching complete well formed XML structures make `Text::RewriteRules` suitable to generate XML and to rewrite XML documents.

The described tools are being used in production in different projects, from the generation of a 30 MB dictionary XML file from a 13 MB text file, taking about nine minutes. The difference on the file sizes show the high number of generated XML tags.

We also described how this rewriting approach can be used for the creation of domain specific languages, namely for the authoring of XML documents.

Finally, the ability to rewrite XML documents in other formats (or to rewrite contents) make the tool suitable for rewriting XML dialects. This approach does not require the creating of a complex data structure in memory, just loading the document as text, and searching on it for the relevant fragments. These can be extracted and processed by a common DOM-oriented approach. Using a streaming library this means it is possible to process huge files where small chunks need processing without loading the full document to memory.

All the modules and tools described are available from CPAN[8] and can be used free of charge.

References

1. José João Almeida and Alberto Simões. Geração dinâmica de APIs Perl para criação de XML. In José Carlos Ramalho, Alberto Simões, and João Correia Lopes, editors, *XATA 2006 — 4ª Conferência Nacional em XML, Aplicações e Tecnologias Aplicadas*, pages 307–314, Portalegre, February 2006.
2. José João Almeida and Alberto Manuel Simões. Text to speech — *"A rewriting system approach". Procesamiento del Lenguaje Natural*, 27:247–253, 2001.
3. Jeffrey E. F. Friedl. *Mastering Regular Expressions.* O'Reilly & Associates, Sebastopol, California, January 1997.
4. Alberto Simões. Down translating XML. *The Perl Review*, 1(1):6–11, Winter 2004.
5. Alberto Simões and Rita Farinha. Dicionário aberto: Um novo recurso para PLN. *Vice-Versa*, 2010. forthcomming.
6. Arie van Deursen, P. Klint, and J.M.W. Visser. Domain-specific languages. Technical Report SEN-R0032, ISSN 1386-369X, CWI, 2000. an annotated bibligraphy.
7. Edward Vanhoutte. An introduction to the TEI and the TEI Consortium. *Lit Linguist Computing*, 19(1):9–16, April 2004.

[8] Comprehensive Perl Archive Network, available at `http://search.cpan.org/`

Test::XML::Generator
Generating XML for Unit Testing

Alberto Simões

Esc. Sup. de Est. Industriais e de Gestão
Instituto Politécnico do Porto
alberto.simoes@eu.ipp.pt

Abstract. To define a DTD or a Schema is not a trivial task. It can be compared to the task of preparing a data structure or, in some cases, to program that data structure adding some semantic. This makes this task error prone. It is common that a final Schema/DTD supports some special XML structures that should not be considered correct, or that, although these special structures are correct, they are not being correctly managed by the application parsing them.
We defend that the possibility to automatically generate XML documents based on a Schema/DTD can help preventing these situations. The generated documents can be used for unit testing and help tuning the Schema/DTD or fixing application problems. They can also assist on benchmarking issues as sometimes developers does not have access to full featured real world documents.
In this article we discuss a Perl module (`Test::XML::Generator`) that provides different mechanisms for automatically generate XML documents based on a DTD and a set of controlling parameters.

1 Introduction

While not trivial, the programming task is not properly difficult. Most problems are not related to the algorithm (unless you are programming in highly scientific areas) but to details that are not considered during the analysis and development task. This is often an issue of forgetting that real world situations can not be predictable. While common situations are easy to predict, edge situations are not.

When programming XML (eXtensible Mark-up Language) processors this is also true. As when dealing with relational databases where programmers need to take care of the database schema, when dealing with XML documents programmers need to take care of the XML structure, often specified on a DTD (Document Type Definition) or an XML Schema.

The process of understanding one of these documents is not a trivial task. Most DTD document can be easily understandable with some paper work and with a detailed analysis of the document grammar. While XML Schemas are not easy to read and understand directly from the XML syntax, there are a lot of

schema visualizers that makes this task easier. Although these documents can be analyzed easily, some details can and will always slip.

We defend that the ability to automatically generate XML documents covering all the edge cases of a XML DTD or Schema can help on debugging the applications that will deal with their instance documents.

Edge cases can include:

- lists with no elements
  should the DTD use a one-or-more list instead of a none-or-more list?;
- element with multiple optional child where none are present
  this is a relevant situation especially when dealing with DTD definitions as they lack flexibility for defining these cases;
- too deep element recursion or too big document size
  programmers often test applications with small and simple XML instances;
- missing attributes
  are the attributes required or optional?;
- conflicting attributes
  are there attributes that can not be used together? DTD also misses flexibility on this situation;

If we are able to generate one or more documents covering all or most of the structure defined in the XML DTD or Schemas, the programmer can test his application robustness during development time.

This article will detail a Perl Module, named `Test::XML::Generator`, that provides a simple programmers interface for generating controlled random XML documents based on a DTD[1].

First we will present briefly other tools with similar objectives. Then, a section documenting `Test::XML::Generator` architecture and algorithm, focusing on the tuning parameters to control the XML documents structure, size, depth and other. Follows a section on the tool analysis. Finally we will reflect and conclude about the tool usefulness.

2 Similar Tools

While there are some XML generator tools like Stylus Studio® 2010 XML, Visual Age for Java or Sun XML Instance Generator, these tools generate mostly simple (or dummy) XML documents, with no possibility to force their size or behavior. They are integrated in IDE tools and are used for small XML documents to be enlarged by the user, and not with testing purposes.

Other tools, like IBM XML Generator or ToXgene [1,2] use annotated templates (annotated DTD or annotated XML template, respectively) to generate the documents. While this approach is especially interesting as they generate semantically correct and controlled documents, they lack on generality and simplicity, as the user needs to understand the templating language, and understand

[1] While we are aiming at XML Schemas as well their structure is much more rich and needs further study and evaluation.

the DTD before writing the template. Therefore, the problems of forgetting corner cases will arise as well.

3 `Test::XML::Generator` Approach

`Test::XML::Generator` has three main approaches:

- **Random document generation** – the DTD structure is traversed and an XML document is generated randomly. Optional elements can be, or not, generated, lists will include a medium number of elements, and random content will be generated for each element. This approach tries to mimic the common XML documents that the applications will need to process.
- **Minimum document generation** – when generating the minimum document, the algorithm will choose the minimum content possible for each element. Optional elements will not be present, lists will include just one or no elements, depending on the list arity, and attributes will be present only if required.
- **Maximum document generation** – this approach is very similar to the first random document generation, but lists, repetitions and recursion will be forced to the limits. As recursion can, on some kind of documents, be infinite, `Test::XML::Generator` uses some variables to control maximum recursion and list size: one limit for list sizes (`LIST_SIZE`), one limit for recursion per element name (`TAG_DEPTH`), and one other to control full recursion size (`FULL_DEPTH`).

To help on the description of the algorithm, the following structure will be used: for each element or attribute type a list of items will describe the algorithm behavior for that element type in this order: random document (RAND), minimum document (MIN) and maximum document (MAX) generation.

At the present moment `Test::XML::Generator` reacts to the following child types:

Enumerated sequence of elements – An enumerated sequence of child elements has the same reaction for the three approaches as all children are required: each element is processed and the result is concatenated together:

- **RAND:** concatenation of processing all children;
- **MIN:** concatenation of processing all children;
- **MAX:** concatenation of processing all children;

Lists of elements – There are two kind of lists, the ones accepting zero elements and the others where one element is required. As already stated, the algorithm uses the parameter `LIST_SIZE` to control the maximum size of lists, and generates:

- **RAND:** a pseudo-random number of elements, ranging from 0 or 1 (depending on the list type) to `LIST_SIZE` The algorithm forces this value to lower with the recursion depth (again, trying to mimic most common XML documents), and helping to control the document size;
- **MIN:** one or no element depending of the list size;
- **MAX:** a list with `LIST_SIZE` elements, unless that would make any of the recursion depth control variables to overflow.

Optional elements – Optional elements are a special kind of lists where the maximum number of child elements is just one:

- **RAND:** one or no element will be generated. Probability to generate empty elements will raise with the recursion level;
- **MIN:** no element will be generated;
- **MAX:** one element is always generated unless that would make any of the recursion depth control variables to overflow.

Choice or Mixed Content Elements – Yet again, mixed content elements can be considered lists but, instead of homogeneous content, they support different kind of elements. These elements can be divided in just choice elements, where children are elements, and mixed content elements, where children are elements or textual content. In any algorithm if the element supports mixed content, then textual content will have the same probability to be generated as the sum of the probabilities of the other elements. This will force the element to include the same number of text and element children. All other elements will have same probability to be generated.

- **RAND:** a random number of children will be generated, ranging from 0 to $n \times$ `LIST_SIZE` where n stands for the number of the optional elements. Recursion control variables will be used to ensure the algorithm ends.
- **MIN:** no element will be generated;
- **MAX:** $n \times$ `LIST_SIZE` elements will be generated unless recursion limits are reached.

Text nodes – Text nodes will be filled with random text. In this case, the well known "*Lorem Ipsum*" text generator [3] is used.

- **RAND:** a paragraph of *Lorem Ipsum* is generated.
- **MIN:** one or two words are generated. `Test::XML::Generator` supports a special flag to tell the algorithm that empty text nodes are possible;
- **MAX:** a paragraph of *Lorem Ipsum* is generated.

Any Elements – These elements are similar to text nodes but can contain tags (defined or not in the DTD). At the moment the tool considers these elements are standard text nodes. Future versions will include some random generated tags, so applications can test their handling of unknown elements.

Empty Elements – These elements can not include any content in any case, therefore no further discussion is necessary about them.

Required attributes – These attributes will be always added for the elements they are defined, as no other option is possible.

Optional attributes – Option attributes are both `IMPLIED` and `DEFAULT` DTD attributes and will be:

- **RAND:** randomly generated (50% of probability for being generated);
- **MIN:** not be generated;
- **MAX:** always be generated.

Regarding the attribute generation some caution should be taken when generating the attribute value. DTD does not define much differences on the attributes values, defining a few choices. The more used kinds are the enumerated, `CDATA`, `ID` and `IDREF` (or `IDREFS`).

While the generation of content for the enumerated and `CDATA` types is simple, the generation of identifiers and their reference is not that easy. This is specially true because DTD does not specify semantic information about what elements can reference what elements. Therefore, `Test::XML::Generator` is only able to ensure every `IDREF` has an existing `ID` (semantically valid or not).

4 Testing `Test::XML::Generator`

The overall algorithm is relatively simple. The main problem resides on the structure and complexness of the supplied DTD.

The generation of minimum document are fast and reliable. That might be the reason most applications include that feature (check the conclusion for that discussion). The problem is with the generation of random documents. If the recursion depth is big, or the XML document complex, the process can take several minutes.

Consider the following simple yet recursive DTD:

```
<!ELEMENT x (bar, zbr)>
<!ELEMENT bar (zbr+)>
<!ELEMENT zbr (foo|bar)>
<!ELEMENT foo (#PCDATA)>
```

The generation of a random document with list sizes of 30 elements and maximum depth of 10 per tag, and 20 in total, can take 15 seconds to generate a 7MB file. This file includes 217 456 elements `zbr`, 54 304 elements `foo` and 20 615 elements `bar`, with a total of 292 376 elements.

While we could use other well known DTD like RSS, Atom XML or TMX, they are not recursive. Other more recursive standards have their structure defined in XML Schemas or in complex DTD documents, that would make tool experiments harder (different points of rupture). Nevertheless, as soon as the tool gets stable these tests are indispensable.

5 Conclusions

This tool is not yet ready and still has problems. At the moment it is mostly a proof of concept, showing the possibility of generating different kind of dummy documents with different testing goals.

Main problems on the document generation task is the generation of non-empty documents. Questions like "where to stop recursion" or "how many tags should be used in a repetition" should be addressed. While we can use some limits (as we did in `Test::XML::Generator`), the generation time is not predictable given the randomness of the algorithm.

Finally, it is our claim that XML generator should be promoted as a stand alone tool instead of a toy option from IDE applications.

Future Work

The presented tool is under alpha stage, and further tests are needed, mainly to find out what the best values to be used on the controlling variables, namely recursion ones.

It is expected that the DTD parser module handle automatically parametric entities, expanding them into standard DTD constructions. Unfortunately this was not yet performed.

When the generation process gets stable, the tool should be expanded to parse XML Schemas as well, as they include more semantic and type information that can be used during documents generation. As already stated, this type of document structure will raise a lot of problems given its flexibility, data types and complex construction operators.

References

1. Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: A template-based data generator for XML. In *In Proc. Fifth Intl. Workshop on the Web and Databases (WebDB)*, pages 6–7, 2002.
2. Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly Lyons. Toxgene: An extensible template-based data generator for xml. In *In WebDB*, pages 49–54, 2002.
3. Wikipedia. Lorem ipsum — wikipedia, the free encyclopedia, 2010. [Online; accessed 12-March-2010].

Visual Programming of XSLT from examples

José Paulo Leal[1] and Ricardo Queirós[2]

[1] CRACS/INESC-Porto & DCC/FCUP, University of Porto, Portugal,
zp@dcc.fc.up.pt

[2] CRACS/INESC-Porto & DI/ESEIG/IPP, Porto, Portugal
ricardo.queiros@eu.ipp.pt

Abstract. Vishnu is a tool for XSLT visual programming in Eclipse - a popular and extensible integrated development environment. Rather than writing the XSLT transformations, the programmer loads or edits two document instances, a source document and its corresponding target document, and pairs texts between then by drawing lines over the documents. This form of XSLT programming is intended for simple transformations between related document types, such as HTML formatting or conversion among similar formats. Complex XSLT programs involving, for instance, recursive templates or second order transformations are out of the scope of Vishnu. We present the architecture of Vishnu composed by a graphical editor and a programming engine. The editor is an Eclipse plug-in where the programmer loads and edits document examples and pairs their content using graphical primitives. The programming engine receives the data collected by the editor and produces an XSLT program. The design of the engine and the process of creation of an XSLT program from examples are also detailed. It starts with the generation of an initial transformation that maps source document to the target document. This transformation is fed to a rewrite process where each step produces a refined version of the transformation. Finally, the transformation is simplified before being presented to the programmer for further editing.

Keywords: XML, XSL Transformations, Second order.

1 Introduction

Computer programs are texts written in a programming language. They are an interrelated set of instructions that a computer must execute autonomously. The relationships among instructions are expressed textually using identifiers. Visual programming languages provide an alternative to the use of textual identifiers by using graphical elements to represent structure and connect related parts. The goal of visual programming is to support non-textual interaction to help novice programmers and/or to increase productivity.

The majority of the "visual" programming environments focus only for the creation of graphical interfaces, using text editors for programming the logic of

programs. In fact, truly visual programming environments were not successfully implemented for the general-purpose languages such as Java or C++. They are mostly used for introductory programming languages used by children or non-programmers. They are sometimes used for domain specific languages and languages for graphical interface programming, as is arguably the case of XSLT.

The aim of this paper is the design of a visual programming tool for XSLT. This language was designed for XML stylesheet definition, but has won a leading role as a tool for conversion among document types in this formalism. The visual programming tool is based on examples. The programmer provides instances of the source document and the corresponding target document and pairs corresponding texts in both documents using graphical primitives. This data is supplied to a generator that creates an initial XSLT program. Second order XSLT transformations refine it to produce a more structured and generic XSLT program.

This visual approach of XSLT programming has obvious limitations. Only a subset of all possible XSLT transformations is programmable by pairing texts on a source and target documents. For instance, second order transformations or recursive templates are out of its scope. Use cases for Vishnu are formatting XML documents in XHTML and conversion among similar formats. For instance, creating an XHTML view of an RSS feed and converting metadata among several XML formats are among the possible uses of Vishnu. Moreover, we do not expect that the automated features of Vishnu to produce the final version of an XSLT program. We view its final result as a skeleton of a transformation that can be further refined using other tools already available in Eclipse.

The remainder of this paper is organized as follows: section 2 explores some of the related work in this area. In the following section we present the design of a visual programming environment for XSLT, more precisely, its architecture and its internal components used to produce and refine an XSLT program. Finally, we conclude with a summary of the major contributions of this paper and a prospect of future work.

2 Related work

There are several environments for programming in XSLT. Usually these tools are integrated in XML IDE's or in general purpose IDE's such as Eclipse. In the former we can highlight StyleVision and Stylus Studio. StyleVision [1] is a commercial visual stylesheet designer for transforming XML. It allows drag and dropping XML data elements within an WYSIWYG interface. An XSLT stylesheet is automatically generated and can be previewed using the FOP built-in browser. Stylus Studio's [2] is another commercial XML IDE that includes a WYSIWYG XSLT designer. The edition process is guided by simple drag-and-drop operations without requiring prior knowledge of XSLT.

There are also several plug-ins for Eclipse for editing XSLT and the Tiger XSLT Mapper [3] is the most prominent. It is a simple development environment that supports automatic mappings between XML structures and can be edited using the drag-and-drop visual interface. While the mappings and XML structures are modified,

the XSLT template is automatically generated and modified. Other examples of Eclipse plug-ins address the XSLT edition [4,5,6] and the XSLT execution [7,8].

There are other tools analogue to Vishnu that are not integrated into Eclipse, as the dexter-xsl [9] which is intended to be used from the command line, the VXT [10] a visual programming language for the specification of XML transformations in an interactive environment and FOA [11] an XSL-FO graphical authoring tool to create XSL-FO stylesheet. It includes a tree visualization scheme to represent the source XML document and the target FO tree structure. FOA generates an XSLT stylesheet that transforms XML content into an XSL-FO document.

Despite the existence of several environments for programming in XSLT, usually integrated into IDE's, they do not use visual editing for programming. Moreover, as far as we know, none of the graphical XSLT programming environment generates programs from examples.

3 The Vishnu application

In this section we present the design of a visual XSLT programming tool called Vishnu. The Vishnu application aims to generate XSLT transformations from pairs of related documents, corresponding to source and target documents of an XSLT transformation.

3.1 Architecture

The architecture of the Vishnu application is divided in two parts - the Editor and the Engine - as depicted in Figure 1. It includes the following components:

- **the Editor** is an Eclipse plug-in that loads a pair of XML documents, respectively, a source and a target document of an XSLT transformation. The Editor uses the Vishnu API to interact with the Vishnu Engine (e.g. set of the imported documents, set a new map, generate the XSLT program);
- **the Mapper** maintains an XML map file identifying the correspondences between the two documents. These identifications can be inferred automatically by the Mapper or manually set through the Editor;
- **the Generator** uses a second order transformation to generate a specific XSLT program, based on the correspondences set by the Mapper;
- **the Refiner** receives the previous XSLT file and uses it as input for a set of transformations. These transformations aim to refine the previously achieved XSLT program. The final result would be a generic XSLT file representing a more structured and generic transformation between any document instances of the respective types;
- **the Cleaner** receives the final transformation produced by the Refiner and replaces XSLT instructions by corresponding constructions in the target language. For instance, attributes constructed with the `xsl:attribute` element and the `xsl:value-of` are replaced by an attribute-value pair with the XPath expressions surrounded by brackets.

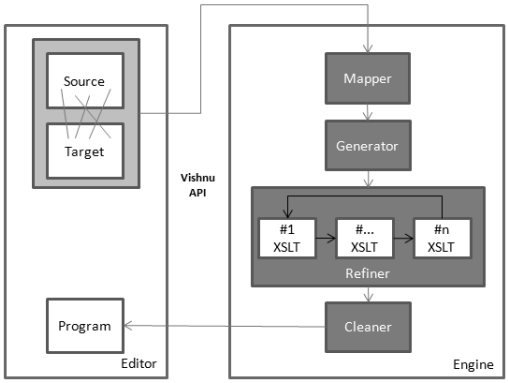


Fig. 1. Vishnu architecture.

In the following subsections we detail the internal components of Vishnu.

3.2 Editing

The front-end of Vishnu is an Eclipse plug-in. In this plug-in the "programmer" edits a pair of XML documents as examples of source and a target documents for the intended XSLT transformation. These XML instances may be created:

- from scratch, using the two XML Editors included in the GUI;
- guided by their type definitions, using the Eclipse completion mechanism.

In the last approach the built-in XML instance generator receives a schema file and automatically generates a valid instance. The user can also define several configuration options such as create optional elements/attributes, create a first choice of a required choice or even fill elements and attributes with data.

Regardless of the choice, the user can identify the correspondences between the two documents by clicking in the source and target text, respectively. The Editor component draws a line connecting both texts. A scenario of the graphical user interface (GUI) of the plug-in for Eclipse is shown in Figure 2.

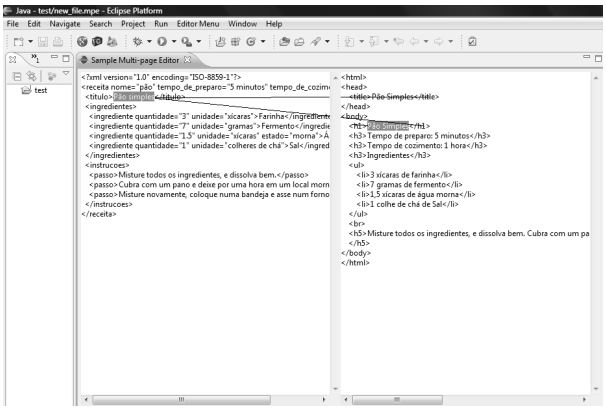


Fig. 2. The Vishnu Eclipse plug-in.

The GUI will include two side-by-side windows for editing respectively the source and target transformations. The widgets of these windows support XML editing for highlighting the XML tags and enable completion based on DTD or XML Schema, if declared for the document.

The programmer is able to pair contents on these windows by drawing links where the origin is on the source document and the destination is on the target window. Origin and destination must be character data, either text nodes or attribute values. As can be seen in the example, this correspondence in not a mapping since the same text on the source document may be used in several points of the target document.

3.3 Pairing

Correspondences can be set manually through the Editor GUI or inferred by the Engine. When in automatic pairing mode Vishnu tries to identify pairs based on:

- Text matches (text or attribute nodes);
- Text aggregation.

In the first mode strings occurring on text and attribute type nodes on the source document are searched on the text and attribute nodes of the target document, and only exact matches are considered. In this mode a single occurrence of a string in the source document may be paired with several occurrences in the target document, as depicted in Figure 3.

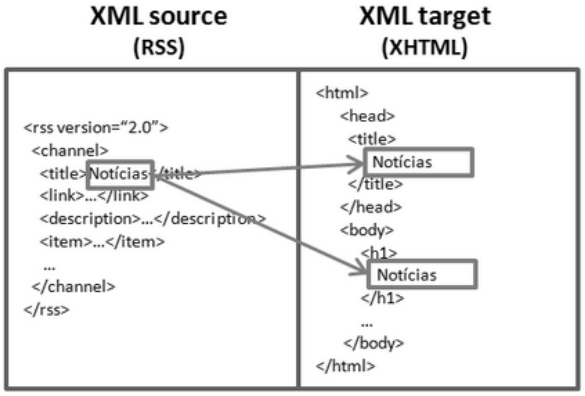


Fig. 3. Automatic mapping - exact match between single texts.

In the second mode Vishnu tries to aggregate strings in the source document to create a string in the target document. In Figure 4 we illustrate with a simple case where 3 strings occurring in attributes and text nodes can be concatenated into a part of the text node on the target document. In this mode several strings on the source document can be paired with strings on the target document.

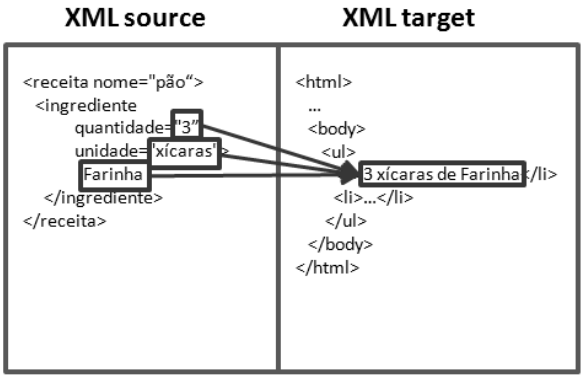


Fig. 4. Automatic mapping - subset of aggregation of texts.

After automatic pairing, the inferred correspondences are presented in the GUI with lines connecting the two XML documents. The user can then manually reconstruct the pairing of string between both documents, as explained in the previous sub-section.

The result of pairing the examples is a document including the actual documents and a list of pairs of XPath expressions relating them. This document is formally defined by an XML schema depicted diagrammatically in Figure 5.

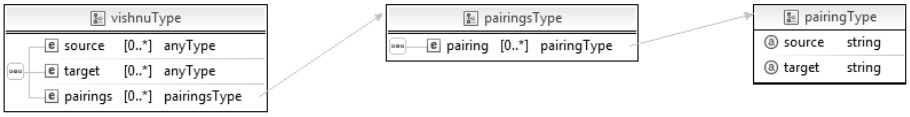


Fig. 5. The pairing XML language.

The pairing XML language has an element called `vishnu` as the root element with three top elements:

- `source` - a copy of the source document;
- `target` - a copy of the target document;
- `pairings` - list of pairing relating the two documents.

Each correspondence is defined by a `pairing` element with two attributes for selecting textual occurrences in both documents: source and target. The source attribute includes a valid XPath expression selecting the text to pair in the source document. The target attribute includes a valid XPath expression selecting the text of the target document. Based on the example of Figure 3, we present the correspondent XML pairing language:

```
<vishnu xmlns="http://www.dcc.fc.up.pt/vishnu">
  <source>
   <rss version="2.0" xmlns="http://backend.userland.com/rss2"/>
     <channel>
       <title>Notícias</title>
       <link>…</link>
       <description>…</description>
       <item>
```

```
      …
      </item>
    </channel>
  </rss>
</source>
<target>
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title>Notícias</title>
    </head>
    <body>
      <h1>Notícias</h1>
       …
    </body>
  </html>
</target>
<pairings>
    <pairing
       source = "/rss[1]/channel[1]/title[1]/text()"
       target = "/html[1]/head[1]/title[1]/text()"/>
    <pairing
       source = "/rss[1]/channel[1]/title[1]/text()"
       target = "/html[1]/body[1]/h1[1]/text()"/>
 </pairings>
</vishnu>
```

3.4 Generating

The Generator component is responsible for the generation of a specific XSLT program based on a given pairing. The component receives as input a document in the paring language introduced in the previous section and, using a second order transformation, produces a specific XSLT program. This program is already a transformation that applied to the source document produces the target document, but is too specific and almost illegible.

The initial program produced by the generator has a single template. The generator iterates over all elements in `//vishnu/target` elements while checking if its absolute path corresponds to any pairing defined in the `//pairing/@target` attributes. In case of correspondence, it includes a `value-of` XSLT element with the respective `//pairing/@source` attribute, otherwise it just copies the element.

As an illustration we present the output of this second order stylesheet based on the example included in the previous subsection.

```
<xsl:template match="/">
  <html>
    <head>
      <title>
        <xsl:value-of
select="/vishnu/source/rss[1]/channel[1]/title[1]/text()"/>
      </title>
    </head>
```

```
    <body>
      <h1>
        <xsl:value-of
select="/vishnu/source/rss[1]/channel[1]/title[1]/text()"/>
      </h1>
      …
    </body>
  </html>
</xsl:template>
```

3.5 Refining

The goal of the Refiner component is to produce a high quality XSLT program from the initial program received by the Generator. The refinement of the program is achieved through the application of a set of second order transformations - called refinements - that simplify and generalize the initial program. These second order transformations act as rewrite rules of a rewrite process that normalizes the initial program.

Each refinement introduces either a small change to the XSLT program or preserves it as it was. The refinement algorithm is rather simple: the set of refinements is repeatability applied to the program until it converges. Convergence occurs when none of the available refinements is able to introduce a modification.

The control of the refinement process is implemented in Java, rather than in XSLT. This separation encourages the modularity and reusability of the refinement transformations which would be harder to achieve if the whole refinement process was encoded in a single XSLT. With this approach is easy to introduce new refinements or to temporarily switch them off. It is easier to change a single and simple XSLT file than to change the code and recompile the application. There are two types of refinements - simplifications and abstractions – that are presented in the following sub-subsections.

3.5.1 Simplifications

Simplifications are refinements that preserve the semantics of the program while changing its syntax. Preserving the semantics means that, for all documents S and T, if a program P transforms document S in document T then the program P', resulting from a simplification refinement, will also transform S to T.

Simplifications can be used for different purposes. They can be used to improve the readability of XPath expressions or to extract global variables. The following paragraphs illustrate this concept with concrete simplifications and examples of the refinements they introduce.

- **Context:** extracts the common prefix of all the XPath expressions from `value-of` elements in the same template and append it as a suffix of the `match` attribute on the `template` element.

Table 1. The Context stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a"> ... *...<xsl:value-of select="b/c"/>* *...<xsl:value-of select="b/d"/>* *</xsl:template>*	*<xsl:template match="a/b"> ...* *...<xsl:value-of select="c"/>* *...<xsl:value-of select="d"/>* *</xsl:template>*

- **Melt:** two or more templates with the same containers are merged into one in which the `match` attribute is an expression that combines the terms of the original attributes match using the operator (|) that computes two or more node-sets.

Table 2. The Melt stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a"> ... *</xsl:template>* *<xsl:template match="b"> ...* *</xsl:template>*	*<xsl:template match="a | b"> ...* *</xsl:template>*

- **Extract:** strings inside the templates are assigned to global variables.

Table 3. The Extract stylesheet.

Source XSLT	Result XSLT
<xsl:template ...> xpto </xsl:template>	*<xsl:variable name="x" select="'xpto'"/>* *...* *<xsl:template ...>* *<xsl:value-of select="$x">* *</xsl:template>*

- **Join:** different variables within the same scope and the same content are merged into a single variable.

Table 4. The Join stylesheet.

Source XSLT	Result XSLT
<xsl:variable name="x1" select="'xpto'"/> *<xsl:variable name="x2" select="'xpto'"/>*	*<xsl:variable name="x1" select="'xpto'"/>*

3.5.2 Abstractions

Abstractions are refinements that change both the syntax and the semantics of the program, although the refined program must retain the intended semantics of the example documents. This means that, for the documents S and T given as example, if

a program P transforms document S in document T then the program P', resulting from a abstraction refinement, will also transform S to T.

Abstractions can be used for different purposes. For instance, they can be used to generalize templates and to restructure large templates in several smaller ones. The following paragraphs illustrate this concept with concrete abstractions and examples of the refinements they introduce:

- **Generalize:** two or more templates with the same container and a `match` attribute differing only in the "index" are merged into one and is removed the last predicate of the attribute `match`.

Table 5. The Generalize stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a[1]"> ... *</xsl:template>* *<xsl:template match="a[2]"> ...* *</xsl:template>* *<xsl:template match="a[3]"> ...* *</xsl:template>*	*<xsl:template match="a"> ... </xsl:template>*

- **Structure:** fragment templates that contain XPath expressions with a common prefix.

Table 6. The Structure stylesheet.

Source XSLT	Result XSLT
<xsl:template match="a"> *<X>* *<xsl:value-of select="b/x">* *<xsl:value-of select="b/y">* *</X>* *<xsl:value-of select="c">* *</xsl:template>*	*<xsl:template match="a">* *<xsl:apply-templates select="b"/>* *...* *<xsl:value-of select="c">* *</xsl:template>* *<xsl:template match="b">* *<X>* *<xsl:value-of select="x">* *<xsl:value-of select="y">* *</X>* *</xsl:template>*

3.6 The Vishnu API

Vishnu was conceived as an interactive tool integrated in Eclipse. Nevertheless, it was designed as two autonomous components: the editor and the engine. The editor is an Eclipse plug-in and concentrates all the tasks related with user interaction and integration with other Eclipse tools. The engine concentrates all the tasks related with the automatic creation of an XSLT program from examples using second order

transformations. The communication between these two components is regulated by the Vishnu API.

By separating concerns in these two components we enable the non-interactive use of Vishnu. The engine has a command line interface to create XSLT programs from example files. Using Vishnu in this mode is as simple as executing the following command line.

$ java vishnu.jar source.xml target.xml > program.xsl

The Vishnu engine can also be invoked from other Java programs through the Vishnu API. This API may be used to create new user interfaces for Vishnu. For instance, a web interface based on the Google Web Toolkit (GWT) or a Swing based desktop interface. In general Vishnu may used by any application needing to create XSL transformations from examples. Java programs using the API must instantiate the engine using the static method *Engine.getEngine()* and use the following methods exposed by the Vishnu API:

`void setSource(Document source)`
Set source document example for the intended transformation.
`Document getSource()`
Get given source document example for the intended transformation .
`void setTarget(Document target)`
Set target document example for the intended transformation.
`Document getTarget()`
Get given example of target document for the intended transformation.
`void resetPairings()`
Reset all previously defined pairings.
`void addPairing(String exprSource, String exprTarget)`
Add a pair of XPath location respectively on the source and target documents.
`List<Pair> getPairings()`
Returns the list of pairings.
`void inferPairings()`
Infer pairings from the given source and target documents.
`Document program()`
Produce a XSLT program from the examples and their pairings.
`Set<String> getFeatureNames()`
Return a list of names of features controlling the refinement process.
`public boolean getFeature(String name)`
Get the given feature status.
`public void setFeature(String name, boolean value)`
Set the given feature status.

4 Conclusions and Future Work

We presented the design of Vishnu - a visual XSLT programming tool for Eclipse based on examples. Visual XSLT programming in Vishnu is based on drawing correspondences on examples of source and target documents. This data is fed to a

generator that produces an illegible and over specialized XSLT program. The initial program is then rewritten into a more legible and general XSLT program using a set of elementary second order transformations called refinements. When no more refinements can be introduced the process stops and the last version of the program is cleaned up. This final version is then presented to the programmer on the user's interface of Vishnu.

The Vishnu project is in the design phase. At this stage we have a design and a prototype implementation of the engine. The generator is already implemented and XSLT programs are produced from examples. The main part of this project - creating the second order transformations to process the program - is just starting. We identified the main types of refinements - simplifications and abstractions - and examples of transformations of each type. We are currently writing a library of templates to support the development of refinements.

Developing a good set of refinements will be a challenge in itself. Proving that a particular set of refinements is confluent may be an even harder task. Confluence is an important property for the rewrite process to ensure its termination and to create a normal form for XSLT programs. A confluent set of refinements would open the use of the refiner to any XSLT program and not just to those produced by the generator. The refiner would be a tool in itself and could be used to refactor XSLT programs within an XSLT editor.

Vishnu will incorporate also a user interface as an Eclipse plug-in. Currently the plug-in prototype is less mature than the engine. The main challenge is editing graphical primitives, such as lines, across XML editing widgets. We plan also to experiment with web interfaces for developing XSLT transformations based on GWT.

References

1. Stylus Studio, http://www.stylusstudio.com/
2. Altova Working Group: Getting More from Your Content with Single Source Publishing: Scoping out the benefits, obstacles, and a unique starting point, Altova StyleVision - White Paper (2008)
3. AXIZON Working Group: Tiger XSLT Mapper Plugin Manual, AXIZON Technologies, Inc., Version 2.0.0 (2007)
4. XSL Tools, http://marketplace.eclipse.org/content/xsl-tools
5. SyncRO Soft Ltd.: <oXygen/> XML Editor 11.2 User Manual for Eclipse (2009)
6. XMLSpy Eclipse editor, http://www.altova.com/xmlspy/eclipse-xml-editor.html
7. OrangevoltXSLT, http://eclipsexslt.sourceforge.net/
8. X-Assist, http://sourceforge.net/projects/x-assist/
9. Dykman, M.: Descriptive XSL Transform Emitter, http://code.google.com/p/dexter-xsl/ (2008)
10. Pietriga, E., Vion-Dury, J. and Quint, V.: VXT: A Visual Approach to XML Transformations. ACM Symposium on Document engineering, USA (2001)
11. FOA - Formatting Objects Authoring tool, http://foa.sourceforge.net

Integration of repositories in Moodle

José Paulo Leal[1] and Ricardo Queirós[2]

[1] CRACS/INESC-Porto & DCC/FCUP, University of Porto, Portugal,
zp@dcc.fc.up.pt

[2] CRACS/INESC-Porto & DI/ESEIG/IPP, Porto, Portugal
ricardo.queiros@eu.ipp.pt

Abstract. Current Learning Management Systems focus on the management of students, keeping track of their progress across all types of training activities. This type of systems lacks integration with other e-Learning systems. For instance, learning objects stored in a centralized repository are unavailable throughout an organization for potential reuse. In this paper we present the interoperability features of crimsonHex - a service oriented repository of learning objects - highlighting the use of XML languages. Its interoperability features are compliant with the existing standards and we propose extensions to the IMS interoperability recommendation, adding new functions, formalizing an XML message interchange and providing also a REST interface. To validate the proposed extensions and its implementation in crimsonHex we designed two repository plugins for Moodle 2.0, the first of which is already implemented and is expected to be included in the next release of this popular learning management system.

Keywords: e-Learning, Repositories, Learning Objects, LMS, Portfolio, Interoperability.

1 Introduction

The main goal of a Learning Management Systems (LMS) is to manage processes regarding the delivery and administration of training and education [1]. Both usage scenarios are relevant: the learners can use the LMS to plan their learning experience and to collaborate with their colleagues; the teachers can deliver educational content and track, analyze and report the learner evolution within an organization. Most LMS's are structured around courses rather than courses' content thus, they only support reusability at the course level. This issue does not allow Moodle users, for instance, to easily bring content into Moodle from external repositories.

This paper builds upon previous work [2] on the design and implementation of crimsonHex - a service oriented repository of learning objects (LO). It provides services to a broad range of e-Learning systems, exposing its functions based on the IMS Digital Repositories Interoperability (DRI) [3] using two alternative web services flavours. Our experience in using these standards lead us to propose extensions to its set of functions and to the XML binding that currently lacks a formal

definition. To evaluate the proposed extensions to the IMS DRI specification and its implementation in the crimsonHex repository, we developed two crimsonHex plugins for the 2.0 release of the popular Moodle LMS. Moodle 2.0 users will be able to download/upload LO from/to crimsonHex repositories, since this LMS is expected to include the plugins described in this paper in its distribution.

The remainder of this paper is organized as follows: Section 2 traces the evolution of e-Learning systems with emphasis on the existing repositories. In the following section we introduce the architecture of crimsonHex and its application interfaces. Then, we provide implementation details of two crimsonHex plugins for Moodle 2.0 using the proposed IMS DRI extensions. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 e-Learning evolution

e-Learning or Electronic Learning can be defined as the delivery of educational content via any electronic media, including the Internet, satellite broadcast, audio/video tape, interactive TV, CD-Rom and others [4]. Despite some efforts [5] to potentiate remote education, the genesis of e-Learning can be traced with the development of network communication in the late 1960s, more precisely, with the invention of e-mail and computer conferencing (1971). These innovations contribute to the collaboration between teachers and students and initiate a new education paradigm shift [6]. During the 1980s and 1990s, there was a significant growth in the number of students studying part-time and also in non-traditional learners, such as, typical 18/24 years old students seeking the university demand and women's returning to the workforce after child rearing [6]. The growth in lifelong learning has made the educational institutions to seek for flexible education delivery to satisfy these non-traditional students. In the end of the century, this delivery has accentuated with the emergence of new forms of distance delivery based in ICTs advances, such as, the Internet.

In their first generation e-Learning systems were developed for a specific learning domain and had a monolithic architecture. Gradually, these systems evolved and became domain-independent, featuring reusable tools that can be effectively used virtually in any e-Learning course. The systems that reach this level of maturity usually follow a component oriented architecture in order to facilitate tool integration. Different kinds of component based e-Learning systems target specific aspects of e-Learning, such as student or course management. This architectural model structures software around pluggable and interchangeable components, thus enabling the development of larger systems, resulting from the collaboration of different teams. In some cases component oriented architectures led to oversized systems that are difficult to reconvert to changing roles and new demands. This is particularly true in e-Learning. A criticism to this approach [7] is that it reduced e-Learning to the use of one-size-fits-all systems, i.e., systems that 1) can be used on any learning subject but fails to address specific need of each of them, and 2) can be used by any student but is not able to adapt to unique characteristics of individuals.

In parallel with the development component-based systems, practitioners of e-Learning start valuing more the interchange of course content and learners' information, which led to the definition of standards for e-Learning content sharing and interoperability. Standards can be viewed as "documented agreements containing technical specifications or other precise criteria to be used consistently as guidelines to ensure that materials and services are fit for their purpose" [8]. In the e-Learning context, standards are generally developed for the purposes of ensuring interoperability and reusability in systems and of the content and meta-data they manage. In this context, several organizations (IMS, IEEE, ISO/IEC) have develop specifications and standards in the last years [9]. These specifications define, among many others, standards related to learning objects, such as packaging them, describing their content, organizing them in modules and courses and communicating with other e-Learning systems.

The Service Oriented Architecture (SOA) [10] is already a mature architectural pattern with established principles and technologies and can be defined as a systematic approach to system development and integration. The general trend towards SOA was also followed by the e-Learning community. In the last few years there have been initiatives to adapt SOA to e-Learning [11, 12, 13]. These new frameworks and APIs contributed with the identification of service usage models and are generally grouped into logical clusters according to their functionality [14]. For instance, the e-Framework for Education and Research is a joint initiative by JISC, Australia's Department of Education, Science and Training (DEST) and other international partners, to facilitate technical interoperability in education and research fields using SOA.

3 crimsonHex repository

In this section we introduce the crimsonHex repository, its architecture and main components, and we present its application interface (API) used both internally and externally. Internally the API links the main components of the repository. Externally the API exposes the functions of the repository to third party systems. To promote the integration with other e-Learning systems, the API of the repository adheres to the IMS DRI specification. The IMS DRI specifies a set of core functions and an XML binding for these functions. In the definition of API of crimsonHex we needed to create new functions and to extend the XML binding with a Response Specification language. The complete set of functions of the API and the extension to the XML binding are both detailed in this section.

3.1 Architecture

The architecture of the crimsonHex repository is divided in three main components:

- **the Core** exposes the main features of the repository, both to external services, such as the LMS and the Evaluation Engine (EE), and to internal components - the Web Manager and the Importer;
- **the Web Manager** allows the searching, previewing, uploading and downloading of LOs and related usage data;
- **the Importer** populates the repository with content from existing legacy repositories, while converting it to LOs.

Using the **API crimsonHex**, the repository exposes a set of functions implemented by a core component that was designed for efficiency and reliability. All other features are relegated to auxiliary components, connected to the central component using this API. Other e-Learning systems can be plugged into the repository using also this API. In the remainder we focus on the Core component, more precisely, its API and we introduce a new language for message interchange.

3.2 Applications Interface

The IMS DRI recommends exposing core functions as SOAP web services. Although not explicitly recommended, other web service interfaces may be used, such as the Representational State Transfer (REST) [15]. We chose to expose the repository functions in these two distinct flavours. SOAP web services are usually action oriented, especially when used in Remote Procedure Call (RPC) mode and implemented by an off-the-shelf SOAP engine such as Axis. REST web services are object (resource) oriented and implemented directly over the HTTP protocol, mostly to put and get resources.

The reason to provide two distinct web service flavours is to encourage the use of the repository by developers with different interoperability requirements. A system requiring a formal and explicit definition of the API in Web Services Description Language (WSDL), to use automated tools to create stubs, will select the SOAP flavour. A lightweight system seeking a small memory footprint at the expense of a less formal definition of the API will select the REST flavour.

The repository functions exposed by the Core component are summarized in Table 1.

Table 1. Core functions of the repository.

Function	SOAP	REST
Reserve	*XML getNextId(URL colectionl)*	*GET URL?nextId > URL*
Submit	XML submit(URL loid, LO lo)	PUT URL < LO
Request	LO retrieve(URL loid)	GET URL > *LO*
Search	XML search(XQuery query)	POST URL < XQUERY > XML GET URL?name1=value1&…> XML
Alert	RSS getUpdates()	GET URL?alert+seconds > RSS
Report	*XML Report(URL loid,Report rp)*	*PUT URL < LOREPORT*
Create	*XML Create(URL collection)*	*PUT URL*
Remove	*XML Remove(URL collection)*	*DELETE URL*
Status	*XML getStatus()*	*GET URL?status > XML*

Each function is associated with the corresponding operations in both SOAP and REST. The lines formatted in italics correspond to the new functions added to the DRI specification, to improve the repository communication with other systems.

To describe the responses generated by the repository we defined a Response Specification as a new XML document type formalized in XML Schema.

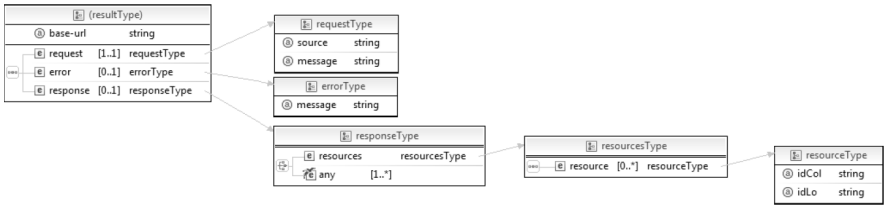


Fig. 1. Response specification schema.

The advantage of this approach is to enable client systems to achieve more information from the server and be able to standardize the parsing and validation of the HTTP responses. Fig. 1 shows the elements of the new language and their types.

The schema defines two top level elements: `result` and `rss`. The former will be used by all the functions except the Alert function that returns a feed compliant with the Really Simple Syndication (RSS) 2.0 specification. The `result` element contains the following child components:

- `base-url` attribute, defining a base URL for the relative URLs in the response;
- `request` element, containing the full request URL and an human readable request message;
- `error` element, containing an error message - client systems will search for this element to verify the existence of errors;
- `response` element, describing a successful execution of the function - it's composed by an human readable response message and, for some functions, by a `resources` element that groups a set of resources defined individually in `resource` elements.

A `resource` element contains an identification of the collection absolute path (attribute `idCol`) and an identification of the LO itself (attribute `idLo`).

In the remainder of this section we enumerate the Core functions of the repository, describing both the request and response data. For sake of simplicity we illustrate the requests using the REST interface since these can be used as command lines in a Linux system shell.

The **Register/Reserve** function requests a unique ID from the repository. We separated this function from Submit/Store in order to allow the inclusion of the ID in the meta-data of the LO itself. This ID is an URL that must be used for submitting or retrieving an LO. The producer may use this URL as an ID with the guarantee of its uniqueness and with the advantage of being a network location from where the LO can be downloaded. This action is performed, for instance, by sending a GET HTTP request to the server, as in the following example.

```
GET http://server/ch/lo?nextId > URL
```

The HTTP response includes an XML file complying with the Response Specification and containing all the details of the response generated by the Core. Nevertheless, in this particular function and for convenience of programmers using REST, the HTTP *Location* header contains the URL returned by the server.

```
Location: http://server/ch/lo/3
```

The **Submit/Store** function uploads an LO to a repository and makes it available for future access. This operation receives as argument an IMS CP compliant file and an URL generated by the Reserve function. This operation validates the LO conformity to the IMS Package Conformance and stores the LO in the internal database. To send the LO to the server we could use, in the REST flavour, the PUT or the POST HTTP methods. An example using the POST syntax is the following.

```
POST http://server/ch/lo/3 < LO
```

The repository responds with submission status data compliant with the Response Specification.

The **Search/Expose** function enables the e-Learning systems to query the repository using the XQuery language, as recommended by the IMS DRI. This approach gives more flexibility to the client systems to perform any queries supported by the repository's data. To write queries in XQuery the programmers of the client systems need to know the repository's database schema. These queries are based on both the LO manifest and its usage reports, and can combine the two document types. The client developer needs also to know that the database is structured in collections. A collection is a kind of a folder containing several resources and sub-folders. From the XQuery point of view the database is a collection of manifest files. For each manifest file there is a nested collection containing the usage reports. As an example of a simple search, suppose you want to find all the titles of LOs in the root collection whose author is *Manzoor*. The XQuery file would contain the following data.

```
declare namespace imsmd = “http://...”;
for     $p in //imsmd:lom
where   contains($p//imsmd:author,’Manzoor’)
return  $p//imsmd:title//text()
```

After creating the XQuery file you can use the following POST request.

```
POST http://server/ch/lo < XQUERY
```

Alternatively, you can use a GET request with the searched fields and respective values as part of the URL query string, as in the following example.

```
GET http://server/ch/lo?author=Manzoor
```

Queries using the GET method are convenient for simple cases but for complex queries the programmer must resort to the use of XQuery and the POST method. In both approaches the result is a valid XML document such as the following.

```
<result base-url="http://server/ch/lo/">
 <request source="http://server/ch/lo/" message="Querying LOR" />
 <response message="3 LOs found...">
  <resources>
     <resource idCol="" idLo="5">
       Hashmat the Brave Warrior
     </resource>
     <resource idCol="" idLo="123">
        Summation of Four Primes
      </resource>
     <resource idCol="graphs/" idLo="2">
        InCircle
      </resource>
  </resources>
 </response>
</result>
```

The **Report/Store** function associates a usage report to an existing LO. This function is invoked by the LMS to submit a final report, summarizing the use of an LO by a single student. This report includes both general data on the student's attempt to solve the programming exercise (e.g. data, number of evaluations, success) and particular data on the student's characteristics (e.g. gender, age, instructional level). With this data, the LMS will be able to dynamically generate presentation orders based on previous uses of LO, instead of fixed presentation orders. This function is an extension of the IMS DRI.

The **Alert/Expose** function notifies users of changes in the state of the repository using a RSS feed. With this option a user can have up-to-date information through a feed reader. Next, we present an example of a GET HTTP request.

```
GET http://server/ch/lo?alert+seconds > RSS
```

The repository responds with an RSS document.

The **Create** function adds new collections to the repository. To invoke this function in the REST interface the programmer must use the PUT request method of HTTP. The only parameter is the URL of the collection.

```
PUT http://server/ch/lo/newCol
```

The following is an example of the repository response to a create function.

```
<result base-url="http://server/ch/lo/" ...>
  <request source="http://server/ch/lo/newCol" message="New col" />
  <response message="Collection created">
    <resource idCol="newCol" idLo=""/>
  </response>
</result>
```

The **Remove** function removes an existent collection or learning object. This function uses the DELETE request method of HTTP. The only parameter is an URL identifying the collection or LO, as in the following example.

```
DELETE http://server/ch/lo/123
```

The following is an example of the repository response to a remove function.

```
<result base-url="http://server/ch/lo/" ...>
  <request
    source="http://server/ch/lo/123"
    message="Deleting a LO" />
  <response message="LO deleted">
   <resource idCol="" idLo="123" />
  </response>
</result>
```

The **Status** function returns a general status of the repository, including versions of the components, their capabilities and statistics. This function uses the GET request method of HTTP, as in the following example.

```
GET http://server/ch/lo?status
```

The repository responds with data compliant with the Response Specification.

4 Integration with Moodle

In this section we validate the interoperability features of the crimsonHex repository by integrating it with Moodle, arguably the most popular LMS nowadays. For this validation we present the new APIs for Moodle 2.0 plugins and we provide implementation details of a plugin for crimsonHex repositories.

The development of this plugin was straightforward. In terms of programming effort we spent half a day to produce approximately 100 new lines of code. This quick and simple integration benefited from the new interoperability features of the repository.

The beta version of Moodle 2.0 is due in February 2010 and will include support for different types of repositories. Several API are already available to enable the development of plugins by third parties to access repositories, including:

- **Repository API** for browsing and retrieving files from external repositories;
- **Portfolio API** for exporting Moodle content to external repositories.

4.1 Repository API

We chose the Repository API for test the integration features of the crimsonHex repository in Moodle. The goal of this particular API is to support the development of plugins to import content from external repositories. The Repository API is organized

in two parts: Administration, for administrators to configure their repositories, and; File picker, for teachers to interact with the available repositories.

To create a plugin for Moodle using the Repository API one must implement a set of related files. The steps are the following:

1. to create a folder for the plugin (*moodle/repository/crimsonHex);*
2. to add to the plugin folder the files
    a. *repository.class.php* – sub-classing a standard API class and overriding its default methods;
    b. *icon.png* – providing the icon displayed in the file picker;
3. to create the language file *repository_crimsonHex.php* and add it to the folder *moodle/repository/lang/en_utf8/.*

The *repository.class.php* is responsible for handling the communication between Moodle and all repository servers of that type. In this case the repository type is crimsonHex but other types are being developed for other types of repository, such as Merlot, YouTube, Flickr and DSpace.

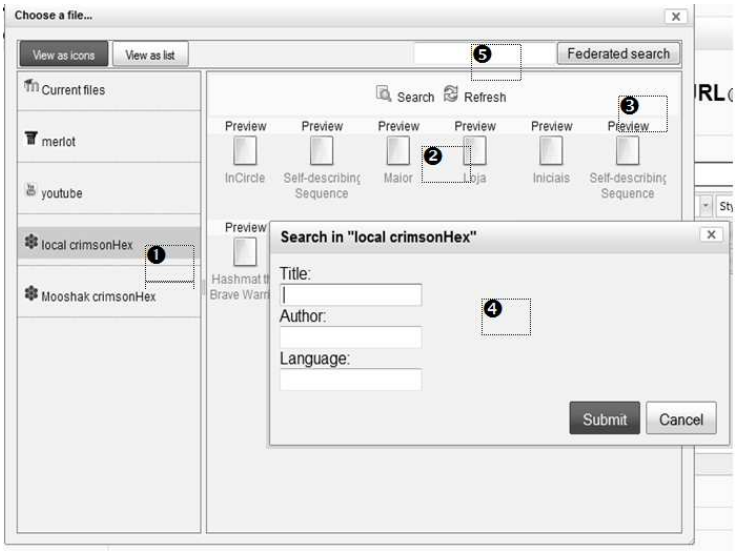


Fig. 2. crimsonHex repository plugin interface.

As explained before, the Repository API has two parts – Administration and File Picker – each with its own graphical user interface (GUI). In Figure 2 we present the file picker GUI of the crimsonHex plugin. On the left panel are listed the available repositories as defined by the administrator. Two crimsonHex repository instances are marked with label 1. Label 2 marks the default listing of the selected repository. Pressing the "Preview" link marked with 3 presents a preview of the respective LO. Pressing the "Search" link pops-up a simple search form, marked as 4 in Figure 2. For federated search in all available crimsonHex repositories is used the text box marked as 5.

For Moodle, each repository is just a hierarchy of nodes. This allows Moodle to construct a standard browse interface. The repository server must provide:

- a URI to download each node (e.g. a LO);
- a list of nodes (e.g. LO and collections) under a given node (e.g. collection).

In addition to these requirements, a repository can optionally support authentication, provide additional metadata for each node (mime type, size, related files, etc.), describe a search facility or even provide copyright and usage rules.

Each feature of the plugin is implemented by a method in the *repository.class.php* file referred in the previous sub-section. A typical method includes: a repository invocation (SOAP or REST), the parsing of its response (using the PHP *simplexml_load_string* function to parse the XML data), a selection of the pertinent data (using XPath) and an iteration over the new results (for instance, populating an array with the relevant data). The next example shows an excerpt of the overridden search function.

```
private function _search($queryString) {
   $list  = array();
   $c = new curl();
   $content=$c->get($this->options['url'] . $queryString);
   $xml = simplexml_load_string($content);
   $result = $xml->xpath("//resource");
   foreach ($result as $entry) {
     $attr = $entry->attributes();
      $list[] = array(
       'title'=>(string)$entry,
       'thumbnail'=>$OUTPUT->icon_url(path),
       'date'=>'',
       'size'=>'',
      'source'=>$attr['url'].$attr['idCol']
      .$attr['idLo']);
   }
   return $list;
}
```

4.2 Portfolio API

The Portfolio API is a core set of interfaces that should be implemented to easily publish files to all kinds of external repository systems. We chose the Portfolio API for test the integration features of crimsonHex repository in Moodle, more precisely, to export Moodle's content to crimsonHex. A typical user story would be:

1. When portfolios are enabled, every page or major piece of content in Moodle would have a "Save" button beside it.
2. User clicks on it and chooses from a list of configured portfolios (this step will be skipped if there's only one).
3. User may be asked to define the format of the captured content (e.g. IMS CP, IMS LD, PDF, HTML, XML).
4. User may be asked to define some metadata to go with the captured content (some will be generated automatically).
5. The content and metadata is COPIED to the external portfolio system.
6. User has an option to "Return to the page you left" or "Visit their portfolio".

To create a plugin for Moodle using the Portfolio API one must implement a set of related files.

The steps are the following:

1. to create a folder for the plugin (*moodle/portfolio/type/crimsonHex);*
2. to add to the plugin folder the file *lib.php* – sub-classing a standard API class and overriding its default methods;
3. to create the language file *repository_crimsonHex.php* and add it to the folder *moodle/portfolio/type/crimsonHex/lang/en_utf8/.*

The *lib.php* is responsible for handling the communication between Moodle and all external repository systems of that type. In this case the repository type is crimsonHex but other types are being developed for other types of repository, such as Mahara, GoogleDocs, and Box.net.

This plugin is still in early development and for that reason we only detail the methods that must be overridden in the class `portfolio_plugin_crimsonHex` whose must extend either `portfolio_plugin_push_base` or `portfolio_plugin_pull_base`. The real differences between them are that one pushes the package directly to the remote system (usually through a HTTP POST), and the other type (pull) requires the remote system to request it. The methods are the following:

- `prepare_package`: prepares the package for sending. This might be writing out a metadata manifest file and zip up all the files in a temporary directory. This is called after writing the files out to a temporary location. You can zip files using `$this->get('exporter')->zip_tempfiles`;
- `send_package`: send the package to the remote system. You can retrieve the files the caller has written out using `$this->get('exporter')->get_tempfiles()` which returns an array of stored_file objects;
- `get_interactive_continue_url`: return an URL to present to the user as a 'continue to their portfolio' link;
- `expected_time`: get the transfer time.

5 Conclusions

In this paper we present the interoperability features of crimsonHex - a repository of learning objects. In its current status crimsonHex is available for test and download at the site of the project [16]. The features of crimsonHex were designed based on the IMS Digital Repository Interoperability and we propose several extensions to this specification. These extensions include new functions and a formal definition of a response specification for the complete function set. To evaluate these extensions we designed two plugins for the 2.0 Moodle's release that uses the interoperability features of crimsonHex and will facilitate the use of crimsonHex by Moodle users.

The main contributions of this work are the proposed extensions to the IMS DRI specification and the design of the two plugins to be included in the Moodle 2.0 distribution.

Completing the implementation of the Portfolio API is the next step in this research. This Moodle API is in early development cycle and we are looking forward to finish the plugin. Adding authoring features to the crimsonHex is another research direction. Creating LOs with metadata of good quality is a challenge since the typical author of e-Learning content usually lacks the knowledge of metadata standards. This

is also an interoperability issue since the LMS is where e-Learning content is tested and used in first place but repositories are the appropriate place to promote content reuse as LOs. Using the plugin based on the Portfolio API will enable the content author to upload learning content to crimsonHex and create a new LO with the essential metadata. Them, using the authoring features of crimsonHex, the content author will be assisted in refining the LO metadata.

References

1. Harman, K., Koohang, A., Learning Objects: Standards, Metadata, Repositories, and LCMS, Informing Science Institute, Edição de Informing Science, 2007. ISBN 8392233751, 9788392233756.
2. Leal, J.P., Queirós, R., 2009. CrimsonHex: a Service Oriented Repository of Specialised Learning Objects. In: *ICEIS 2009: 11th International Conference on Enterprise Information Systems, Milan*.
3. IMS DRI - IMS Digital Repositories Interoperability, 2003. Core Functions Information Model, URL: http://www.imsglobal.org/digitalrepositories.
4. Tastle, J., White, A. And Shackleton, P., E-Learnintens g in Higher Education: the challenge, effort, and return of investment, International Journal on ELearning, 2005.
5. Harasim, L.: History of E-learning: Shift Happened, The International Handbook of Virtual Learning Environments, Springer, 2006.
6. Williams, J., Goldberg, M.: The evolution of e-learning. Universitas 21 Global, 2005.
7. Dagger,D., O'Connor, A., Lawless,S., Walsh,E., Wade,V.: "Service-Oriented E-Learning Platforms: From Monolithic Systems to Flexible Services," IEEE Internet Computing, vol. 11, no. 3, pp. 28-35, May/June 2007.
8. Bryden, A. Open and Global Standards for Achieving an Inclusive Information Society.
9. Friesen, N. Interoperability and Learning Objects: An Overview of E-Learning Standardization". Interdisciplinary Journal of Knowledge and Learning Objects. 2005
10. Earl, T.: Service-oriented architecture - Concepts, Technology and Design, Prentice Hall, 2005, ISBN 0-13-185858-0
11. C. Smythe, “IMS Abstract Framework - A review", IMS Global Learning Consortium, Inc. 2003.
12. Open Knowledge Initiative Website, http://www.okiproject.org.
13. S. Wilson, K. Blinco, D. Rehak, “An e-Learning Framework” - Paper prepared on behalf of DEST (Australia), JISC-CETIS (UK), and Industry Canada, July 2004.
14. S. Aguirre, J. Salvachúa, A. Fumero, A. Tapiador. "Joint Degrees in E-Learning Systems: A Web Services Approach". Collaborative Computing: Networking, Applications and Worksharing, 2006.
15. Fielding, R., 2000. Architectural Styles and the Design of Network-based Software Architectures, Phd. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation.
16. crimsonHex – Project Web site, 2009. URL: http://www.dcc.fc.up.pt/crimsonHex.

XML to paper publishing with manual intervention

Oleg Parashchenko

bitplant.de GmbH
Fabrikstr. 15, 89520 Heidenheim, Germany
olpa@uucode.com

Abstract. Existing tools consider XML to paper conversion as one black box step and provide control only through predefined options. With this approach, tuning the layout of output documents is a burdensome task.
The paper advocates a new workflow for XML to paper conversion, in which a separate step allows the user to fine control the layout. The changes made by the user are remembered and later can be automatically re-applied during publishing an updated version of the document.
A possible technical implementation for the workflow is suggested. TEX is used as a typesetting engine. XML to TEX conversion is made using XSLT and TEXML. The management of changes is performed by diff and patch tools.

1 Introduction

Publishing XML on paper, especially creating books, requires that the resulting layout is perfect. Unfortunately, sometimes the layout is unsatisfactory, regardless how good the conversion scripts are. In such cases human assistance is required.

Manual intervention is possible on three levels: as hints for conversion scripts in the source XML, as corrections in the final format, or somewhere inbetween. The first, putting hints in the source, is the simplest, but it pollutes the logical markup with presentation details. Also, a set of possible hints can be not enough for fine tuning. The second option, correcting layout in the final format (usually PostScript or PDF) is good for local changes, but does not suitable for such tasks as inserting a page break. I think that the last option, tuning somewhere inbetween, could be best.

XSL [1] is the main candidate. To publish XML on paper, W3C recommends the XSL standard: an XSLT program converts a source XML to XSL formatting objects (XSL-FO), and some tool performs actual typesetting as specified. Unfortunately, XSL-FO files are not intended and not suitable for editing by hand, and there are no specialized editors.

An important possibility is importing XML to a desktop publishing program and tuning the layout there. The problem, which also exists in the case of an

May, 2010 - Page. 69–76 ≍ATA 2010 — ISBN: 978-972-99166-9-4

imaginary XSL-FO editor, is that if the source XML is changed, then the user should re-import XML again and repeat all the changes.

As a solution, we propose to use the typesetting system TEX [2] as an intermediate between XML and paper. A TEX document is an usual text file, and no special editor is required for it. With some effort, XML can be converted to a good TEX file, such that it does not look like an autogenerated mess. The user can edit this file to tune the layout.

The diff/patch [3] mechanism solves the second problem, the need to repeat the changes in the updated document. In software development, diff and patch tools are used to join changes from different developers. But the tools are not restricted to programming code, they work with any text files. For our needs, the changes in the layout and the changes in the document also can be merged. Effectively, it looks like the user-made changes are automatically applied to the new version of the document.

The idea of using TEX for XML publishing isn't new (see [4]), but all the existing proposals consider XML to paper conversion as one logical step. The user has to use only pre-defined options and is not supposed to interfere with the generation process. The same is for non-TEX software: I'm not aware of tools which encourage the user to create and manage changes.

The rest of the paper describes the complete workflow. Different technical implementations are possible. I propose one based on the standard tools (XSLT, diff, patch), on TEXML [5], a tool for supporting XML to TEX conversion, and finally on the own experience.

2 Workflow

A graphical illustration of the workflow is shown on the Fig. 1. Version numbers are for illustrative purpose, not the real ones. "PDF" means any suitable output format for printing.

1. Convert an XML document version 1.0 to a TEX document version 1.0.0 and then to a PDF document version 1.0.0.
2. Change the layout of the document version 1.0.0. This gives the document version 1.0.1.
3. Repeat the step 2 as long as necessary. The final document version is 1.0.N.
4. Remember the differences between the TEX documents 1.0.0 and 1.0.N in a patch file 1.0.0-1.0.N.
5. If the source XML document is updated to the version 1.1:
   - Generate a TEX document version 1.1.0
   - Apply patch 1.0.0-1.0.N to the generated file, this gives the document version 1.1.N. Restart the workflow at the step 3, using the new version numbers.

The workflow description is high-level. It explains what to do, but doesn't specify how to do it. Different technical implementations are possible, the one used by the author is described in the next section.

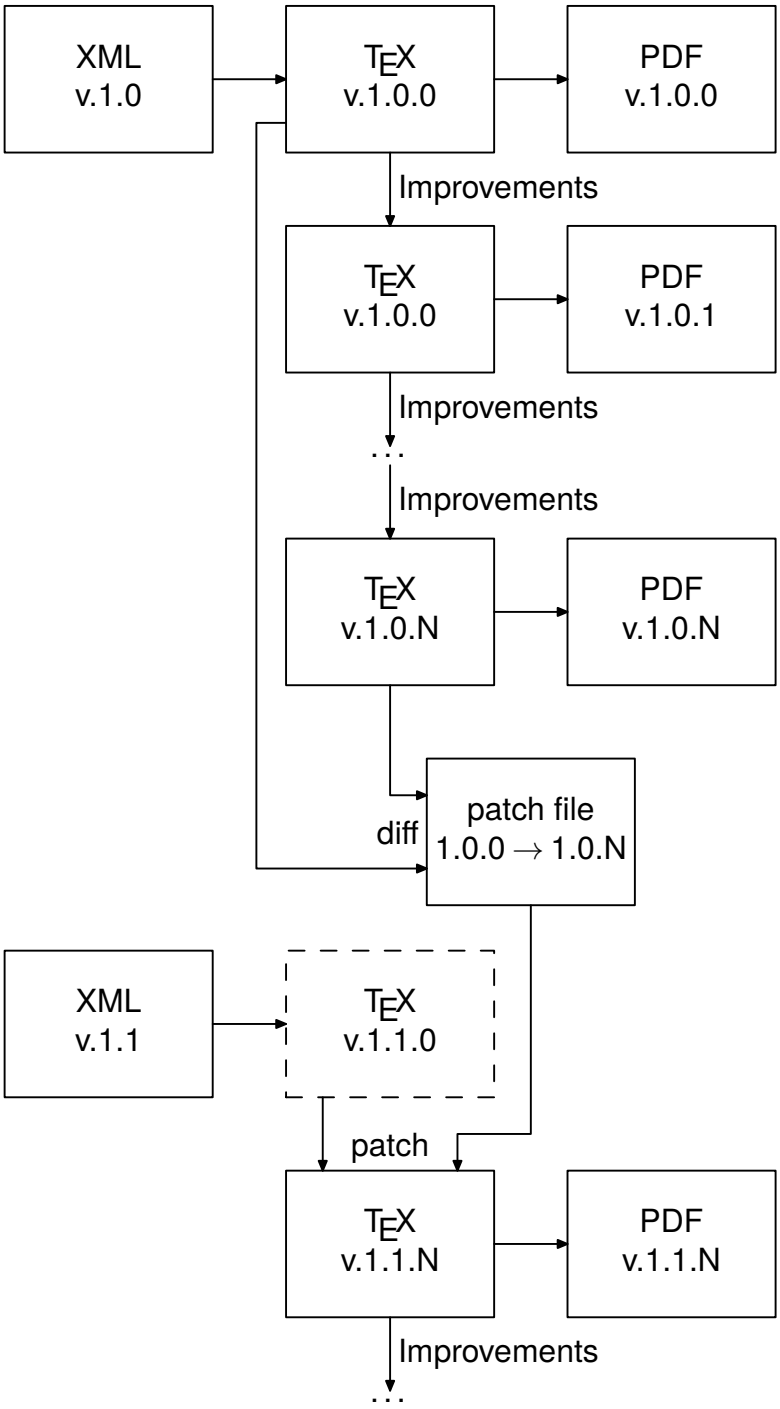


Fig. 1. XML to paper workflow

3 Possible Technical Implementation

3.1 XML to TEX

Different approaches for converting XML to TEX are possible. My personal choice is TEXML: an XSLT program converts XML to an intermediate XML-based format TEXML, and a serializer makes a TEX file from TEXML. This approach is advocated in the article [5]. In short:

- XSLT is the standard for transforming XML,
- Producing the correct TEX syntax is a hard task, better to be delegated to a specialized tool.
- The TEXML serializer automatically makes a human-friendly TEX file with good formatting.

As an example, consider the following source XML file `article.xml`.

```
<article>
<title>De finibus bonorum et malorum</title>
<para>Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
sed diam nonumy eirmod tempor invidunt ut labore et dolore
magna aliquyam erat, sed diam voluptua. ...</para>
<para>Ut wisi enim ad minim veniam, quis nostrud exerci tation
ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo
consequat. ...</para>
</article>
```

An XSLT program `democonv.xsl` to convert the XML to TEXML:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="article">
  <TeXML>
    <xsl:call-template name="generate-header"/>
    <env name="document">
      <xsl:apply-templates/>
    </env>
  </TeXML>
</xsl:template>

<xsl:template match="title">
  <cmd name="title">
    <parm><xsl:apply-templates/></parm>
  </cmd>
</xsl:template>
```

```
<xsl:template match="para">
  <env name="para">
    <xsl:apply-templates/>
  </env>
</xsl:template>

<xsl:template name="generate-header">
  <TeXML escape="0">
\documentclass{article}
\usepackage{democonv}
% ... more document-specific settings ...
  </TeXML>
</xsl:template>

</xsl:stylesheet>
```

The generated TEX file `article.tex` is:

```
\documentclass{article}
\usepackage{democonv}
% ... more document-specific settings ...
\begin{document}
\title{De finibus bonorum et malorum}
\begin{para}
Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
sed diam nonumy eirmod tempor invidunt ut labore et dolore
magna aliquyam erat, sed diam voluptua. ...
\end{para}
\begin{para}
Ut wisi enim ad minim veniam, quis nostrud exerci tation
ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo
consequat. ...
\end{para}
\end{document}
```

This conversion example shows that one can get one-to-one mapping between XML and TEX structural elements. (More precisely, TEX is used here in the form of its macro package LATEX.) Correspondingly, the conversion XSLT-program is trivial.

Formatting is described in the file `democonv.sty`:

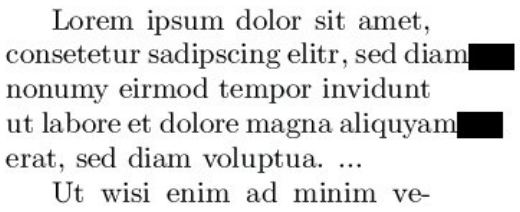
```
\ProvidesPackage{democonv}[Demo article]

\usepackage[paper=a5paper,textwidth=5cm]{geometry}
\setlength\overfullrule{15pt}

\newenvironment{para}{\ignorespaces}{\par}
```

First comes the service header, then, for illustrative purposes, definition of page size and narrow text width. The text will not fit, and TEX will complain and show a dick marker near such places.

Paragraphs in TEX are usually separated by empty lines. But my experience shows that in case of automatic conversion, it is convenient to wrap the paragraphs in environments.

3.2 Tuning the Layout, Remembering the Changes

Compiling the TEX file gives two warning messages about overfull lines, the corresponding places are marked with black squares:



The forward search and reverse search feature [6] of TEX tools helps for big documents. If the feature is enabled, one can jump from a location in the source TEX file to the corresponding place in the output (forward search). And reverse, one can jump from a location in the output to the corresponding place in the source.

For this simple example, no search is required to find the code that should be tuned. A simple correction is enough: just add `\break` after the words "elitr, sed". Now the file is compiled without warnings.

After the layout is corrected, generate a patch file `article.patch` from the original TEX code `article.tex.orig` and the final version `article.tex`:

```
diff -u article.tex.orig article.tex > article.patch
```

The patch file looks as follows:

```
--- article.tex.orig    2010-03-10 04:53:46 +0100
+++ article.tex 2010-03-10 04:56:24 +0100
@@ -5,7 +5,7 @@
 \title{De finibus bonorum et malorum}
 \begin{para}
 Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
-sed diam nonumy eirmod tempor invidunt ut labore et dolore
+sed\break diam nonumy eirmod tempor invidunt ut labore et dolore
 magna aliquyam erat, sed diam voluptua. ...
 \end{para}
 \begin{para}
```

3.3 Publishing an Updated Version

Update the sample XML by adding a paragraph:

```
...
magna aliquyam erat, sed diam voluptua. ...</para>
<para>Duis autem vel eum iriure dolor in hendrerit in vulputate
velit esse molestie consequat, vel illum dolore eu feugiat nulla
facilisis at vero eros et accumsan et ...</para>
<para>Ut wisi enim ad minim veniam, quis nostrud exerci tation
...
```

The output contains the same layout problem as in the original version. The user should repeat the changes. This can be done automatically by applying the patch:

```
patch article2.tex <article.patch;
```

In good cases, like this example, all the changes are automatically applied. But it is possible that some changes are rejected. In this case, the user has to revise the rejections manually. On the bright side, even if something can not be applied automatically, it works as a reminder where to check the layout.

4 Conclusions and Future Work

This paper proposes a new XML-to-paper workflow. The main contribution is emphasis on manual intervention in the generation process. Unlike existing approaches, the workflow helps the user to create and manage layout changes. As a technical implementation, we convert XML to human-friendly TEX code using XSLT and TEXML, and control the changes using diff and patch tools.

The author successfully uses the workflow in internal publishing services in industry. To support the workflow, a helper software named Consodoc (CONStructor Of DOCumentation, `http://getfo.org/consodoc/`) was developed and released to public. But now the software requires a major revision to allow simple creation of processing steps in additional to the default ones.

A part of the workflow, the TEXML-to-TEX converter is an open-source tool (MIT/X Consortium license) written in Python. The software and documentation are available from the home page of TEXML project `http://getfo.org/texml/`, a number of code repositories includes the tool. The private reports from the independent developers indicate that the tool works well in theirs projects.

The future work on the project consist of:

- development of sample TEXML stylesheets for popular XML formats, for example, for DocBook [7];
- revision of Consodoc, development of sample publishing projects, for example, for the book "DocBook 5.0: The Definitive Guide" [8];
- further improvement of the TEXML tool.

References

1. W3C: Extensible Stylesheet Language (XSL), Version 1.0. W3C Recommendation 15 October 2001. http://www.w3.org/TR/2001/REC-xsl-20011015/
2. Knuth, D.: TEX and METAFONT: New Directions in Typesetting, American Mathematical Society and Digital Press, Bedford, MA, 1979.
3. Johnson, M.: Diff, Patch, and Friends. Linux Journal, Volume 1996, Issue 28es (August 1996)
4. Pepping, S.: From XML to TEX, a short overview of available methods, EuroTEX 2001. http://www.ntg.nl/eurotex/PeppingXML.pdf
5. Parashchenko, O: TeXML: Resurrecting TeX in the XML world, TUGboat 28:1, 2007.
6. Laurens, J.: Direct and reverse synchronization with SyncTEX, TUGboat 29:3, 2008.
7. Walsh, N., Muellner, L.: DocBook: The Definitive Guide, O'Reilly & Associates, 1999.
8. Walsh, N.: DocBook 5.0: The Definitive Guide, work in progress, http://www.docbook.org/tdg5/.

XML Description for Automata Manipulations*

José Alves Nelma Moreira Rogério Reis
{sobuy,nam,rvr}@ncc.up.pt

DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal

Abstract. GUItar is a visualization software tool for various types of automata (standard, weighted, pushdown, transducers, Turing machines, etc.). It provides interactive manipulation of diagrams, comprehensive graphic style creation, multiple export/import filters, and a generic *foreign function calls* (FFC) interface with external systems. In this paper we describe GUItar's XML framework and show how it allows for extensibility, modularity and interoperability.

1 Introduction

FAdo [1,2,3] is an ongoing project which aims to provide a set of tools for symbolic manipulation of automata and other models of computation. For diagram graphical visualization and interactive manipulation the GUItar application [2] is being developed in Python [4] and using the wxPython [5] graphical toolkit. GUItar provides assisted drawing, interactive manipulation of diagrams, comprehensive graphic style creation and manipulation facilities, multiple export/import filters and extensibility, mainly through a generic *foreign function call* (FFC) mechanism. Figure 1 shows the GUItar architecture. The basic frame of its interactive diagram editor has a notebook that manipulates multiple pages, which one containing a canvas for diagram drawing and manipulation.

In this paper we describe GUItar's XML framework and show how it allows for extensibility, modularity and interoperability. In Section 2 we present GUItarXML, an XML format for the description of diagrams that allow several information layers. This format is the base for the export/import methods described in Section 3, where a generic mechanism for add new methods is also presented. Section 4 presents the FFC's configuration and manipulation mechanisms. These allow the interoperability with external software tools. As an example we consider the interface with the FAdo toolkit [2]. Finally in Section 5 we briefly present the *object library* which will allow the dynamic construction of an automata database.

* This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, and by project ASA (PTDC/MAT/65481/2006).

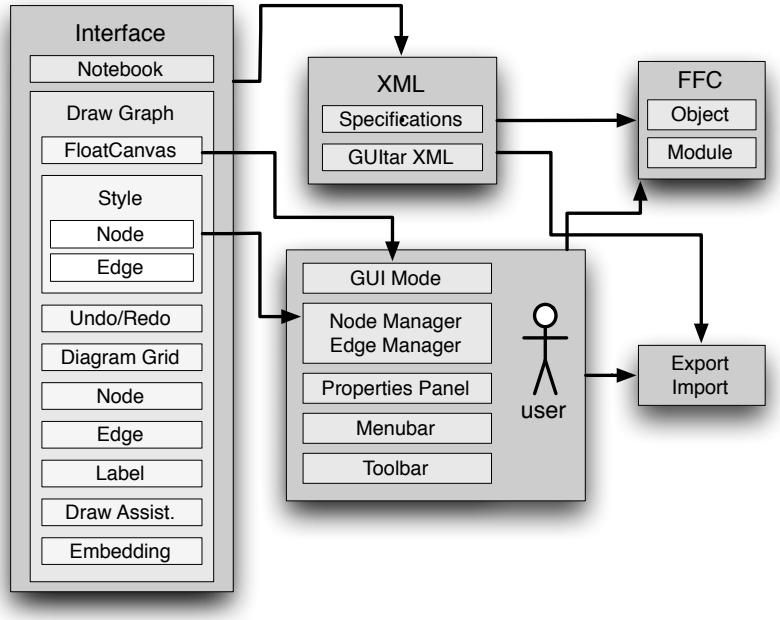


Fig. 1. GUItar architecture

2 GUItarXML

GUItarXML is an XML [6] format for the description of diagrams originally designed for the GUItar application, but that aims to be as generic as possible. It is based on GraphML [7], a simple language to describe the structural properties of graphs. Although GraphML has an extension mechanism based on key/value pairs to encode application specific data, because of efficiency and clarity reasons, we chose not to use it. Instead, we encode that data directly as new elements. GUItarXML describes graphs and digraphs or graph-like diagrams, such as automata, and contains graphical information, style information, and specific automata information. A fragment of the GUItarXML Relax NG schema [8] is presented in Figure 2. Diagrams are composed of a set of nodes connected by a set of edges. They are encoded in `graph` elements and there can be an arbitrary number of diagrams per GUItarXML document. The `graph` element can contain an arbitrary number of `node` and `edge` elements, each of them identified by an `id` attribute that must be unique. Edges have the `source` and `target` attributes that are the `id`s of the endpoints of the edge. Although this information is enough for some applications, some additional data may be required. Automata, for instance, require states and edges to have labels. The nodes and edges have a `label` element. The labels can be either simple or compound. Simple labels are just text strings. Compound labels have fields that can assume user-specified values. This is used,

```
include "styles.rnc"
include "defaults.rnc"

start = element guitarxml {
  attribute version {text},
  graph*,
  style*,
  state_object_group*
}
graph = element graph {
  attribute id {text},
  node*,
  edge*,
  graph_automata?,
  defaults?
}
node = element node{
  attribute id {text},
  node_diag?,
  node_draw?,
  label?,
  node_aut?
}
node_diag = element
    diagram_data {
  attribute x {text},
  attribute y {text}
}
node_draw = element draw_data {
  attribute obgroup {text},
  attribute x {xsd:long},
  attribute y {xsd:long},
  attribute scalex {xsd:long}?,
  attribute scaley {xsd:long}?
}
node_aut = element
  automata_data {
  attribute initial {"1"|"0"}?,
  attribute final {"1"|"0"}?
}
```

```
edge = element edge{
  attribute id {text},
  attribute source {text},
  attribute target {text},
  element diagram_data{...},
  edge_draw?,
  label?
}
edge_draw = element draw_data {
  attribute arrowlinestyle {text}?
  attribute head1style {text}?,
  attribute head2style {text}?,
  attribute numheads {"0"|"1"|"2"}?,
  attribute swapheads {"1"|"0"}?,
  (point*)?,
  ...
}
label = element label {
   attribute type {"simp"|"comp"},
   attribute text {text}?,
   attribute layout {text},
   attribute style {text}?,
   dict*
}
dict = element dict {
    attribute key {text},
    attribute value {text}
}
graph_automata = element
   automata_data{
  element sigma{
   element symbol{
     attribute value {text}
    }*
  }?,
  element classification{
    element class{
      attribute value {text}
    }*
  }?
}
```

Fig. 2. A fragment of the Relax NG schema for diagrams.

for example, with weighted automata, transducers or Turing machines. Figure 3 shows a compound label with two fields: label and weight. These fields have the values a and 0.3, respectively, so the final label value is $a : 0.3$.

```
<label type="Compound" layout="$label : $weight"
    style="style1">
  <dict key="label" value="a"/>
  <dict key="weight" value="0.3"/>
</label>
```

Fig. 3. Example of a compound label.

Embedding and drawing information are described in the `diagram_data` and the `draw_data` elements, respectively. For the nodes, `diagram_data` contains the embedding coordinates. The `draw_data` elements contain graphical data such as the graphical object styles and drawing properties such as the node's draw coordinates and scale. The `automata_data` elements describe specific automata properties, like boolean attributes `final` and `initial` used to indicate, for nodes, if a state is final or initial, respectively. Within `graph` element, general automata information (if applicable) such as the alphabet is specified. The `graph` elements also have the `defaults` element that contains style the default values.

```
<graph>
  ...
<edge ...>
      ...
    <draw_data linestyle="red" head1style="red"
        numberofheads="1" ...>
      ...
</edge>
  ...
</graph>
  ...
<style name="red" basestyle="default" linewidth="2"
    fillstyle="Dot">
    <fillcolor r="255" g="0" b="0"/>
    <linecolor r="255" g="0" b="0"/>
</style>
```

Fig. 4. Style usage example.

2.1 Styles

GUItar has a rich and powerful style set of facilities that allow the creation and manipulation of the graphical representation of nodes and edges. GUItarXML styles are similar in concept to *cascading style sheets* (CSS) [9]. CSS provide a way for markup languages to describe their visual properties and formatting. GUItar styles only focus on visual properties and allow the definition of style classes. The `style` elements have the `name` attribute that can be used when the style is to be applied to an object. An example is shown in Figure 4. The `basestyle` attribute is the name of the base style for the style. Styles inherit their properties from their base style. Besides those attributes, the `style` can have the actual styling attributes such as line color, fill color, line width, etc.

```
<graph>
  ...
<node ...>
      ...
    <draw_data obgroup="final" ...>
      ...
</node>
  ...
</graph>
  ...
<state_object_group name="final">
    <ellipse style="default" primary="True">
        <size x="35" y="15"/>
    </ellipse>
    <ellipse style="default">
        <size x="40" y="20"/>
    </ellipse>
</state_object_group>
```

Fig. 5. A node object group for final states.

Node styles are more complex than the edges styles. A node can be composed of several sub-objects. Consider the classic representation of a final state pre-

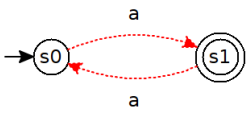


Fig. 6. A GUItar diagram with `red` style edges and nodes with an `initial` and a `final` state object group.

sented in Figure 5. That state representation is composed of two sub-objects that are two concentric circles (or ellipses). These complex graphical objects are defined in `state_object_group` elements. These elements contain some state style specific options and one or more shape elements. Available shapes are: `ellipse`, `rectangle`, `floatingtext` (a static label), and `arrowspline` (a multiple control point spline arrow). Figure 6 shows an example of an automaton that makes use of these styles.

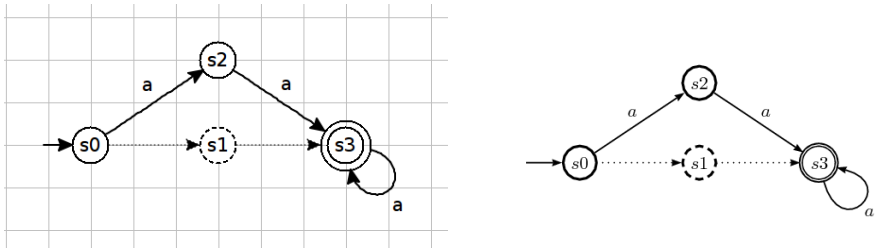


Fig. 7. A GUItar diagram and the Vaucanson-G LaTeX output.

3 Format Conversions

GUItarXML generic format is used for the conversion to, and from, other formats. Currently GUItar implements conversion methods to export to the following formats: GraphML, dot [10], Vaucanson-G [11] and FAdo. It has, also, methods to import from all those formats except from Vaucanson-G. Conversion to GraphML is simple, since GUItarXML is based on GraphML. An XSL transformation is used to remove the styles and extra elements in nodes, edges, and graphs. The result is a basic GraphML file with the basic topological data. The dot is a language for the specification of graphs that is part of Graphviz, a package of graph visualization tools [10]. GUItar exports to dot and currently dot graphs will retain all data except for style data. The Vaucanson-G is a LaTeXpackage that allows the inclusion of automata-like diagrams in LaTeX documents. When GUItar exports to Vaucanson-G it creates a document with one `VCDraw` environment containing the automata data. GUItar styles are converted to LaTeX macros (see Figure 7). Whenever an exact conversion is not possible, a reasonable approximation is used. For example, Vaucanson-G doesn't support drawing of complex states, so the primary object is used as a base for the Vaucanson-G state.

FAdo's format is used for the representation of deterministic finite automata (DFA) and nondeterministic finite automata (NFA). GUItar uses its Xport mechanism to handle the export and import from that format.

Conversions to SVG (Scalable Vector Graphics) [12] and FSMXML [13] are currently being implemented. The first format is a generic XML language for describing two-dimensional graphics. The second one is an XML format proposal for

the description of weighted automata, where the graphical information is based on the Vaucanson-G styling and is mainly oriented for algebraic applications of automata.

3.1 Xport Mechanism

The Xport mechanism allows an easy way to add new export and import methods to GUItar, either coded in Python or as XSL transformations. This mechanism is configured using an XML specification (see Figure 8).

The specification allows for multiple Xport to be defined. Each one has a `name`, that is the string that will appear in the GUItar interface export/import menus. The `wildcard` attribute can be a file wildcard used in the wxPython's file dialog wildcard argument.

Depending on what type of Xport it is, additional attributes are required. XSL Xport require the attributes `expfile` (export) and `impfile` (import) that are file `paths` of the XSL transformations. For regular Xport, the attribute `import` must exist and contains the Python import statement for the module that contains the conversion methods. The element `export` and the element `import` indicate the methods used to perform the conversions.

4 Foreign Function Calls

The *foreign function call* (FFC) mechanism provides GUItar with a generic interface to external Python libraries, or programs and mechanisms to interact with foreign objects. In the first case, the FFC mechanism calls a function directly from an external Python module (Module FFC). In the second case, it creates a foreign object and then calls methods of that object (Object FFC). Both cases require an XML configuration file that specifies, among other things, the names of the available methods, their arguments, and their return values. Figure 9 presents the general FFC mechanism.

```
<xport_data>
  <xport name="FAdo" import="GF" wildcard="FAdo files
      (*.fa)|*.fa">
    <import method="read_o"/>
    <export method="save_o"/>
  </xport>
  <xslxport name="GraphML" expfile="guitar-graphml.xsl"
      impfile="graphml-guitar.xsl" wildcard="xml files
      (*.xml)|*.xml"/>
 </xslxport>
<xport_data>
```

Fig. 8. Xport configuration for FAdo and for GraphML files.

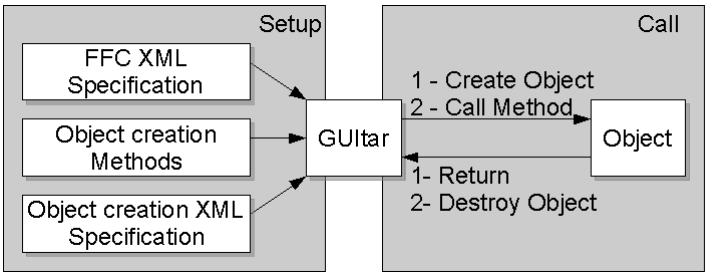


Fig. 9. Module and Object FFC.

```
<foreign_function_call>
 <depends import=...>
  ...
 <module import=...> or <object creatorname=...>
  <method>
   ...
  <menu_data>
    ...
  </menu_data>
   ...
 </module> or </object>
  ...
</foreign_function_call>
```

Fig. 10. An FFC configuration file.

4.1 FFC Configuration Specification

FFC configuration files can contain several FFC definitions (module or object). Module FFC's must indicate the module they import (the `import` attribute that uses Python's import syntax), while object FFC's must indicate the name of the object creator they will use to create the object (`creatorname` attribute). A more user friendly name and a description of the FFC method may be given in the optional attributes `name` and `description`, respectively.

Each module or object can contain multiple method definitions. For each method, a `name`, an `id`, it's `arguments`, and it's `return` values are needed (see Figure 11). Arguments require the `type` attribute, that states its type. An optional `default_value` attribute can be given with the default value for this argument. Return values only require the `type` attribute. A FFC method can have multiple arguments and multiple return values with an order that must agree with their appearing order in the definition. The types for arguments and return values can be `Int`, `Float`, `Boolean`, `String`, and also, one of the following:

`File`: a file. It has two additional attributes: `dialogmode` and `filemode`. The `dialogmode` can be either `save` or `load`, and indicates the type of dialog to

```
<method name=... id=...>
  <argument type=... default_value=.../>
    ...
  <return_value type=.../>
    ...
</method>
```

Fig. 11. Structure of `method` elements.

```
<menu_data>
 <menu title="FAdo">
  <menu title="DFA">
    <menu_entry descr1="Minimal" descr2="Minimize automata"
        action="minimal"/>
    <menu_entry descr1="Union" descr2="Returns union of two
        DFAs" action="union"/>
  </menu>
 </menu>
</menu_data>
```

Fig. 12. A menu data example. Nested menu elements will create sub-menus.

show. The `filemode` can be either `path` or `file`, and indicates if the value is expected to be the path to the file (string) or a Python file object.

`Canvas:` a GUItarXML representation of a diagram in a canvas.

`Object:` a foreign object. The attribute `creatorname` is required to know which *object creator* will be used.

FFC's can, optionally, define their own menus. Those menus will be dynamically created by GUItar on startup, just like GUItar's own native menus. The structure of the `menu_data` element is presented in Figure 12.

Optionally, FFC's can also include multiple `depends` elements that have the `import` attribute and are used to indicate any module dependencies that the FFC has. These dependencies are checked when the FFC's are being loaded and a warning is raised for the user, in case of failure.

4.2 Foreign Objects

A *foreign object* is a Python object which type is not recognized by GUItar. Its GUItar type (see Section 4.1) will be `Object`, and to convert to and from a GUItar object *object creators* are used. Object creators require two components: an XML configuration file and a Python file containing the conversion methods. Object creator configuration files may define multiple object creators, under the condition that they are methods defined in the same module (see Figure 13). The module containing the methods' definition is the value of the `import` attribute.

```
<object_creator_group  import="GF">
  <depends import="FAdo" />
  <depends import="yappy" />
  <object_creator name="FAdoDFA" class="DFA">
    <to_method method="GuitarToFA">
      <argument type="Canvas" default_value="Current" />
    </to_method>
    <from_method method="FAToGuitar">
      <returns type="Canvas" />
    </from_method>
  </object_creator>
</object_creator_group>
```

Fig. 13. Example object creator configuration file for FAdo DFA objects.

Each object creator has a `name` attribute, which value is used in arguments and return values of FFC methods. The `class` attribute contains the foreign object Python class name. The attributes `to_method` and `from_method` name the methods to be used in the conversions.

4.3 A FFC Example

To illustrate the use of a FFC object, we will use the FAdo DFA minimal method. This method computes the minimal DFA equivalent to a given automaton.

```
<foreign_function_call>
  <depends import="FAdo" />
  <object creatorname="FAdoDFA">
  <method name="minimal" id="minimal" friendly_name="Minimal"
      description="Returns equivalent minimal DFA">
    <return_value type="Object" creatorname="FAdoDFA" />
  </method>
    ...
</foreign_function_call>
```

Fig. 14. Fragment of the FAdo DFA interface.

Figure 14 shows part of the definition of the FAdo DFA interface, highlighting the *minimal* method.

Figure 15 shows the GUItar graphical interface and the menu selection of the *Minimal* method. The user will be asked to choose which GUItar object wants to be minimized and the object creator will create the correspondent DFA object. The FAdo DFA minimal method takes no extra arguments, but if there were any arguments, they would also be created using their appropriate creator. After

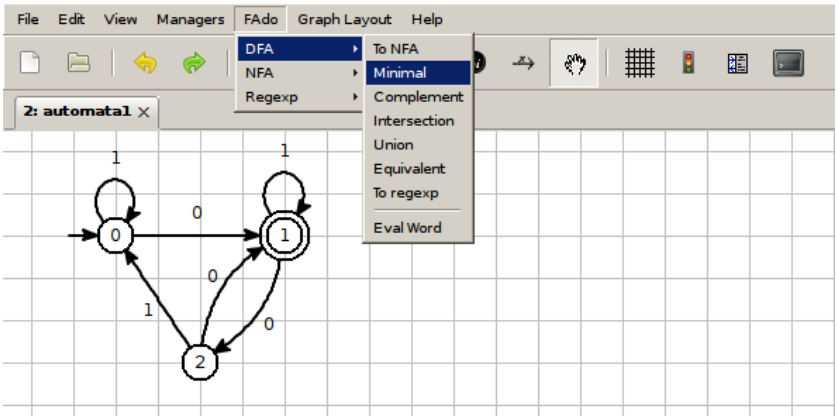


Fig. 15. GUItar graphical interface and the FAdo FFC interaction.

the termination of the FFC execution, the returned values are processed. In this example, a DFA object is returned. That object is converted into a GUItar object (GUItarXML string) which is automatically drawn in a new canvas.

5 Object Library

As seen in the example of Section 4.3, the GUItar framework allows the operation over several automata, using the FFC mechanisms. It is possible to trace the operations (methods) used to generate objects and which objects are arguments for that operations. In this way it is possible to relate the various objects manipulated, and also determine some of their properties. This information can be used for constructing automata databases, and store it in the `automata_data` elements (of the GUItarXML format). Currently we are developing a XML specification for this information. However, we can already automatically display in the GUItar graphical interface the relationships between the several objects, and save them as GUItarXML diagrams. Each diagram has as nodes the created objects. Each arc is labeled with the operation that takes the object that constitutes arc source, to produce the one that constitutes its target. For each operation, there are the same number of arcs (edges) as arguments. In Figure 16 we present two object dependence diagrams. One for an application of the FAdo minimal method and the other for the union of two automata.

6 Conclusions

In this paper we presented an ongoing work for the development of a set of tools for the visualization, conversion and manipulation of automata. The XML framework provides a means of obtaining extensibility and interoperability with external automata manipulation tools. Although GUItar is already a functional

prototype, several improvements are needed. More format conversions must be implemented and some of the existent ones must be extended. The FFC configuration must be automated in order to easy the interaction with external systems. The object library must be fully implemented.

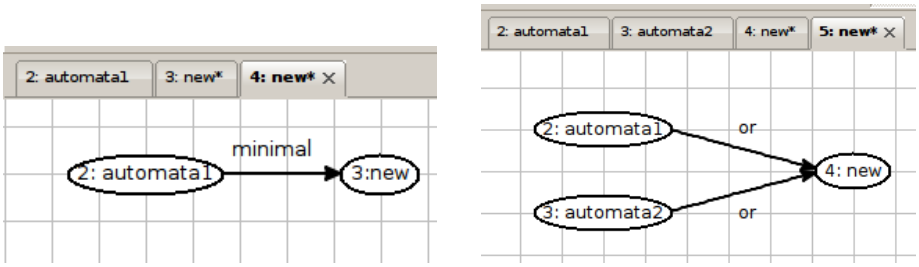


Fig. 16. Examples of object dependence diagrams.

References

1. Moreira, N., Reis, R.: Interactive manipulation of regular objects with FAdo. In: Proceedings of 2005 Innovation and Technology in Computer Science Education (ITiCSE 2005), ACM (2005) 335–339
2. Almeida, A., Almeida, M., Alves, J., Moreira, N., Reis, R.: FAdo and GUItar: tools for automata manipulation and visualization. In Maneth, S., ed.: 14th CIAA'09. Volume 5642 of LNCS., Springer (2009) 65–74
3. FAdo: tools for formal languages manipulation. `http://www.ncc.up.pt/FAdo` (Access date:1.1.2010)
4. Foundation, P.S.: Python language website. `http://python.org` (Access date:1.12.2009)
5. Smart, J., Roebling, R., Zeitlin, V., Dunn, R.: wxWidgets 2.6.3: A portable C++ and Python GUI toolkit. (2006)
6. WWW Consortium: XML specification WWW page. `http://www.w3.org/TR/xml` (Access date:1.12.2008)
7. GraphML Working Group: The GraphML file format. `http://graphml.graphdrawing.org` (Access date: 01.12.2009)
8. van der Vlist, E.: RELAX NG. O'Reilly (2003)
9. WWW Consortium: CSS WWW page. `http://www.w3.org/Style/CSS` (Access date:13.03.2010)
10. Graph Visualization Software: The dot language. `http://www.graphviz.org` (Access date:1.12.2009)
11. Lombardy, S., Sakarovitch, J.: Vaucanson-G. `http://igm.univ-mlv.fr/~lombardy` (Access date:1.12.2009)
12. WWW Consortium: Scalable vector graphics. `http://www.w3.org/Graphics/SVG/` (Access date: 01.12.2009)
13. Vaucanson Group: FSMXML. `http://www.lrde.epita.fr/cgi-bin/twiki/view/Vaucanson/XML` (Access date: 01.12.2009)

A Performance-based Approach for Processing Large XML Files in Multicore Machines

Filipe Felisberto, Ricardo Silva, Patricio Domingues, Ricardo Vardasca and Antonio Pereira

Research Center for Informatics and Communications
School of Technology and Management
Polytechnic Institute of Leiria
Leiria, Portugal
patricio@estg.ipleiria.pt

Abstract. Due to its ubiquity, XML is used in many areas of computing, contributing to partially solve the problem of universal data representation across platforms. Although the parsing of XML files is a relatively well studied subject, processing large XML files with more than hundreds of megabytes pose many challenges. In this paper, we tackle several approaches focusing on how the performance can be improved when parsing very large XML files (hundreds of megabytes or even some gigabytes). We present a multithreaded block strategy that yields a roughly 2.19 relative speedup in a quad core machine when processing a 2.6 GB XML file.

1 Introduction

The eXtensible Markup Language (XML) has contributed to partially solve the problem of data representation, allowing for a self-contained and portable representation of information, across multiple hardware and software platforms. Therefore, and despite its cumbersome size overhead, XML has been widely deployed in the last decade. Thus, XML has become pervasive in modern computing, being used for the representation of data structures across several languages and programming interfaces, fostering the interoperability among applications. For instance, XML is used as the base format for several office application suites such as Microsoft Office and OpenOffice [1].

As the adoption of XML rises, so does the amount of data generated and collected, with files of several hundred megabytes becoming increasingly common, especially because many XML files are now generated by automatic tools that can produce vast amounts of data in short amount of time. This is for instance the case of logging mechanisms. A similar trend also exists for scientific applications of XML, where many datasets are being made available in XML to overcome portability issues. For instance, the XML files in the protein sequence database are close to 1 GB in size [2].

As the role of XML in applications has increased, parsing XML files has also become an important issue. Although many libraries and applications exist for

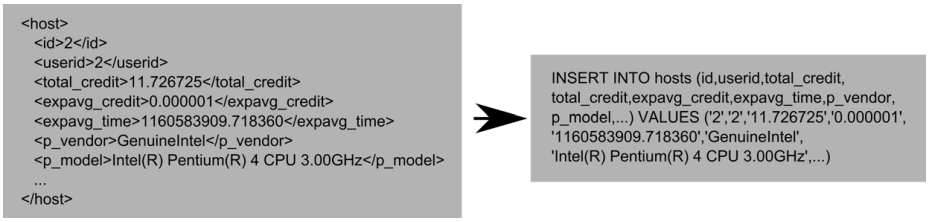


Fig. 1. A flat XML <host> entry (left) and its equivalent SQL representation (right)

dealing with XML, namely creating, editing and parsing, most of them target small (few kilobytes) to medium XML files (some megabytes), failing to deliver a proper performance or even to process large XML files (hundreds of megabytes and possibly gigabytes). Indeed, dealing with large XML files poses its own challenges. For instance, the Document Object Model (DOM) interface [3], which is widely used for processing small XML files is unsuitable for very large files, failing due to its large memory requirements. Almeida et al. report that the memory usage for a DOM representation of an XML file is roughly 1.15 times the file size [4].

The computing landscape has radically evolved in the last years, with affordable multicore machines becoming mainstream and single-core being confined to the ultra low power consumption devices. Although multicore CPUs provide more computing power, exploring their computing power requires multithreading applications, meaning that important changes are needed to adapt single-core applications to multicore computing environments [5].

In this paper, we tackle the performance of processing very large (few gigabytes) linear XML files in multicore environments. By linear XML files, we mean XML files that have a rather simple structure, with most of the nodes being at the same level. These files are commonly used in log formats. The XML processing of such files can occur for instance, in the conversion of the XML data to another format. Specifically, the motivation for this work was the conversion of a 2.6 GB file from the SETI@home volunteer computing project [6] to SQL insert statements, in order to insert the data into a relational database. An example of conversion is given in Figure 1, with the linear (flat) XML <host> entry on the left side and the corresponding SQL representation on the right side.

For this purpose, in this paper, we first explore several programming languages (C++ and Java) and analyze a naïve I/O parallelization scheme. We then move on to tune parsing algorithms for multithreaded environments, exploiting the multicore parsing of XML files. The main contributions of this work are *i*) an analysis of the requirements that drive efficient parsing of large XML files, *ii*) the proposition and evaluation of a multithreaded block-based algorithm devised to speed up the processing of large XML files in multicore machines. A further contribution derives from the knowledge gained in dealing with large files (over 2 GB) in mainstream multicore machines.

The remainder of this paper is organized as follows. We review related work in Section 2. In Section 3, we describe the testbed used in this work, and then test the C++ and the Java programming languages to evaluate which one delivers the best performance. We also analyze a naïve approach for speeding up I/O. Next, in Section 4, we detail the multithreaded block-based algorithm and present the main performance results. Finally, Section 5 concludes the paper and discusses venues for future work.

2 Related Work

2.1 Processing XML

There are two main APIs for parsing XML: *i*) the Document Object Model (DOM) [3] and the *ii*) Simple API for XML (SAX) [7]. DOM is a tree-traversal API which requires loading the whole XML document's structure into memory before it can be traversed and parsed. Although DOM provides for a convenient programming interface, it does so at the cost of a large memory footprint. Indeed, the memory requirement of DOM makes its usage inefficient for large XML files thus precluding the processing of very large XML files (this is the case for instance if the file is larger than the available memory). Due to this fact, DOM was not considered in this work, solely the SAX API.

SAX works by associating a number of callback methods to specific types of XML elements. When the start or the end of a given element is encountered, the correspondent event is fired. By default, the parser only has access to the element that triggered the event and, due to its streaming, unidirectional nature, previously read data cannot be read again without restarting the parsing process [8].

2.2 Processing Very Large XML Files

Although XML parsing is the subject of a plethora of research, few of this research is devoted to the field of multicore processing of very large XML files under a SAX-like paradigm. An important contribution is given by Head et al. [2] who have developed the *PiXiMaL* framework (http://www.piximal.org) that targets the parsing of large scientific XML files in multicore systems. For instance, in [2], the authors resorts to *PiXiMal* for processing a 109 MB XML file holding a protein sequence database. Unfortunately, *PiXiMaL* does not seem yet available and thus we could not compare our work with the framework. It should also be noted that [2] focuses on somewhat complex hierarchical structures of XML, with several levels of nodes existing within other nodes and thus with the existence of dependencies within the document. As stated before, our work targets simpler (almost flat) very large XML files, and thus our main priority is to process the document from start to end in the fastest possible way. We also resort to a 2.6 GB test XML file which is 25 times larger than the 109 MB file used in [2].

Fadika et al. build up on the work of the PiXiMaL framework and propose a distributed approach for parsing large XML files [9]. Specifically, they resort to the MapReduce model [10], using the open source Apache Hadoop [11] framework to distribute the parsing of large XML files in what they call *macro-parallelization techniques*. They point out that Apache Hadoop can bring benefits but care needs to be taken with the startup costs and that a proper balance between computation and I/O needs to be achieved in each machine, otherwise no proper performance gains can be achieved.

An alternative approach to the SAX API for processing large XML documents is XmlTextReader[1] available under the Gnome Project's libxml2. XmlTextReader is an adaptation of Microsoft's XmlTextReader that was first made available under the C# programming language. The authors point out that XmlTextReader provides direct support for namespaces, xml:base, entity handling and DTD validation, thus providing an interface that is easier to use than SAX when dealing with large XML documents. However, as we show later on in section 4.3, XmlTextReader performs quite slowly when compared to SAX.

Almeida et al. [4] pursue an interesting path for processing large TMX files (XML files that hold translation memories used in text translation), merging the SAX and DOM methodologies. Specifically, they process large TMX files in an incremental way, splitting up the file in individual blocks, with individual blocks being processed through DOM. This is made possible due to the fact that a TMX file is comprised of several blocks, each block being a valid XML file. Although their hybrid SAX-DOM approach is slower than a pure DOM methodology, it allows for the processing of large TMX files since its memory requirements are fixed, while processing a large TMX file via DOM becomes impossible as soon as the file size is close to the amount of system memory. The authors do not target multithreaded environments.

3 Sequential Approach

In this section, we first present the testbed computing environment, and then move on to compare the C++ and Java programming languages to determine the best suited one for processing large flat XML files. We then analyze a read/write-based I/O separation strategy.

3.1 Testbed Environment

All the experiments described in this paper were conducted on an Intel Core 2 Quad processor model Q6600@2.4 GHz with four cores and eight MB of L2 cache, fitted with four gigabytes of DDR2 RAM and a 160 gigabyte SATA harddrive (the harddrive has an internal cache of 8 MB). The operating system was the 64 bits version of the Linux Ubuntu 9.10, kernel 2.6.31 and an ext4 filesystem. The system was fitted with GCC 4.4.1, libxml2 2.7.5 and OpenJDK 1.6.1.

[1] http://xmlsoft.org/xmlreader.html

The performance tests were based on parsing a large flat XML file building an equivalent SQL representation. The process of building the SQL file works as follows: *i*) an XML <host> node is read from the input file, *ii*) converted to its equivalent SQL expression version, and then *iii*) this SQL expression is appended to an output file. All these steps are repeated until the whole XML document has been processed. Figure 1 gives an example of an XML input (<host> node) and the corresponding SQL output.

As XML input for the execution tests, we used two so-called <host> log files from two BOINC-based volunteer computing projects [12]. As the name implies, in a BOINC-based project, a host log file accumulates data related to the computers that participate (or have participated) in a volunteer computing project, with an <host> XML node existing for each of this computer. To contain execution times (each test was run at least 30 times), we used a 100 MB file from the QMC@home project (Quantum Monte Carlo [13]) and a larger one, with 2.6 GB from the SETI@home project [14]. The larger file was solely used for assessing the performance of the final version. The 100 MB files had 98,615 <host> nodes, while the 2.6 GB files held 2,588,559 <host> entries with a depth level of three. Both files are freely available respectively, at http://qah.uni-muenster.de/stats/host.gz and at http://setiathome.berkeley.edu/stats/host.gz, although it should be pointed out that both files are updated daily and thus the current version are most certainly larger than the ones we used.

3.2 The Programming Language

The first stage of our study was to select the most suited programming language for processing large XML files, with the selection parameter being the yielded performance. For this stage, two languages were considered: C++ and Java. While it might seem strange to even consider an interpreted language like Java for high performance data processing in opposition of the C++ programming language, the decision to analyze both languages was based on the fact that the SAX paradigm, being event driven, is better suited to oriented object languages. Additionally, the higher abstraction level of Java allows for a smoother development cycle.

To compare both languages, two test programs were developed, one for each language, both of them with the sole purpose of processing the XML document in a linear fashion. It should be pointed out that we used the C++ language, although no real object oriented model was followed. The rationale for resorting to C++ was to use commodity classes like string and the queue class. The former automatically takes care of the memory management involved when dealing with the concatenation of the strings needed for the creation of the SQL INSERT statement, while the queue class allows to store the SQL code in an automatic way. Another motivation for using C++ was the availability of the BOOST [15] library that substantially eases the task of developing multithreaded applications.

The results shown in Table 1, corresponding to the processing of the 100 MB XML test file, unequivocally proves that the C++ version is more than twice

-	Average (s)	Median (s)	Std. deviation (s)
Java	5.0027	5.0035	0.3017
C++	2.2340	2.2933	0.2750

Table 1. Execution times for the C++ and Java version (100 MB XML)

-	Average (s)	Median (s)	Std. deviation (s)
Dual I/O thread	2.2842	2.2788	0.2177

Table 2. Execution times for the dual I/O threading approach (100 MB XML)

faster than the Java implementation. Therefore, in the remainder of this work, we solely focus on the C++ version.

3.3 I/O Separation

To understand the execution profile, the C++ application was run under the GNU profiler (gprof). The profiling indicated that a large part of the execution time were I/O operations (read and write). This is due to the fact of reading/writing a large file from/to disk, and also because of the reduced volume of computing operations – no computations are performed over the XML data, solely some string manipulations to generate the SQL. Moreover, since SAX processes the input file in a sequential, single-threaded way, the potential of a multicore platform can not be directly exploited.

To reduce the execution time, the natural step was trying to overlap I/O operations with computing operations. For this purpose, a two-thread scheme was devised. On one side, a thread would read the data from the disk, process it, and put the output into a stack of buffers, moving to the next chunk of the input file. On the other side, the writing thread would write the content of the buffers to disk. Results of the read/write threads approach are given in Table 2. It can be seen, that the wall clock time is similar (even slightly worse) than the dummy approach. The explanation for this behavior is that resorting to a reading thread and a writing thread does not eliminate the competition for disk access, since both threads will try to concurrently access the disk, therefore invalidating the gain yield by separately reading and writing operations. It should also be pointed out that the read/write threads scheme does not scale, since it could at most take advantage of a two-core machine, with cores being wasted when more than two exist.

4 Parallel Approach

4.1 The Block-based Algorithm

In order to parallelize the XML parsing operations, a block-based approach was devised. Although the SAX API does not allow for the parallel processing of the input document, the API supports processing data fed via buffers. Thus, the block-based approach consists in splitting up the input file in blocks, each block being assigned to a thread. A dummy approach would be to split the file in as many blocks as the number of wanted threads, each block being individually processed by its associated thread. However, this solution is not acceptable, as it does not eliminate the multiple concurrent I/O accesses and thus the performance impact of I/O. Moreover, this would require a pre-processing stage where the XML file would be read, split in blocks, with the generated output written to the disk.

A viable approach is to split the processing of the input file in small blocks, with each thread successively reading, processing and writing a block and then repeating these actions again and again. Thus, each data block is independently processed by a thread. Figure 2 illustrates the block-based algorithm.

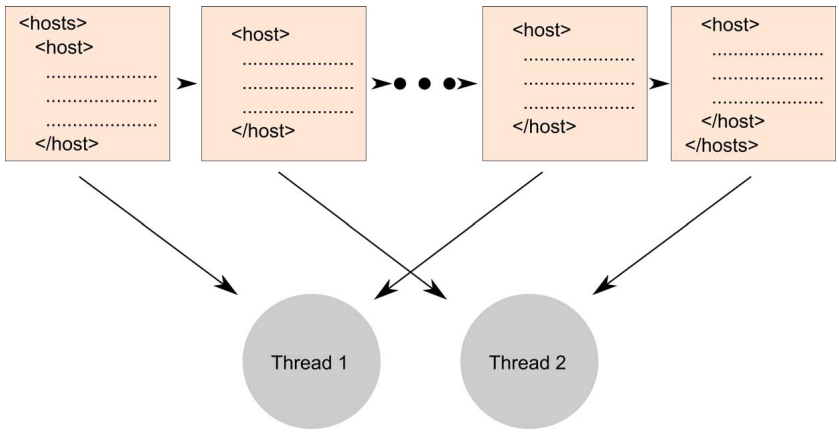


Fig. 2. The block-based algorithm

An important issue regarding the division of an XML file is that the blocks need to take into account the boundaries of each node, since an XML node can not be divided in half, otherwise the parsing is no longer coherent. Moreover, the root XML tag <hosts> needs to be added at the beginning and its counterpart (</hosts>) at the end of each block. Under the C/C++ programming language, filling a memory block previously initiated with the root tag is a simple matter of passing the appropriate offset (in this case, the offset corresponds to the string length of the initiating tag) to the fread() functions.

# of threads	Average (s)	Median (s)	Std. deviation (s)
1	8.0851	8.3045	0.9928
4	5.9015	6.0684	0.6162

Table 3. Multithreaded XmlTextReader (100 MB XML file)

To ensure the thread safeness of the block-based approach, we resorted to the xmlSAXUserParseMemory() function that allows for the specification of buffers holding the control variables and user data needed for each instance of the SAX parser.

4.2 Main Results

We now present the main results for the multithreaded versions. We first start by assessing the performance of a multithreaded XmlTextReader-based version and then analyze our own multithreaded approach.

4.3 The XmlTextReader Approach

As stated earlier in the related work section, the XmlTextReader approach targets the processing of large XML documents, offering an higher level of abstraction than the SAX paradigm[2]. Therefore, we assessed the performance of the XmlTextReader in order to decide whether this approach was worth considering from the performance point of view. For this purpose, we developed a XmlTextReader-based version of our application and ran it over the 100 MB XML document with one thread and then with four threads. We used XmlTextReader that is shipped with libxml2.

The results (Table 3) show that the performance delivered by XmlTextReader is inferior to the SAX-based sequential version, even when considering the sequential Java-based version. Therefore, XmlTextReader does not suit our performance-oriented purposes. It is also worthy to note that some stability problems were found with the four-thread executions (crashes related to memory corruption). Nonetheless, it should be pointed out that the XmlTextReader approach allows for an easier implementation of the application, bearing some resemblances with recursive programming. Therefore, in environments where performance is not a critical issue, XmlTextReader might be an adequate choice.

4.4 Multithreaded Block-based Version

To assess the performance of the block-based algorithm, the parallel version was run over the 100 MB and the 2.6 GB input data file. The size for each individual block was set to 5 MB, with tests conducted with 10, 15 and 20 MB yielding similar results. Execution times ranging from one to four threads are shown in Table 4 (100 MB input XML file) and in Table 6 (2.6 GB input XML file).

# of threads	Average (s)	Median (s)	Std. deviation (s)
1	1.8364	1.8378	0.0084
2	1.1694	1.0418	0.2840
3	0.9958	0.9893	0.1888
4	1.0516	0.9445	0.2806

Table 4. Multithreaded block-based version (100 MB XML file)

Average (s)	Median (s)	Std. deviation (s)
0.7679	0.7041	0.1140

Table 5. Execution time due to I/O Overhead (100 MB XML file)

As shown in Table 4 (100 MB input file), the execution times progress from one to two threads (1.8364 to 1.1694 seconds), and then again, from two to three threads (1.1694 to 0.9958 seconds). However, adding a fourth thread worsen the execution time, meaning that the algorithm does not scale beyond three threads. To determine if I/O was limiting the scalability of the algorithm, we devised a simple read/write applications, that performed the I/O operations, reading and writing individual blocks of 5 MB. Execution times of the I/O application are shown for the 100 MB file in Table 5 and confirms that a significant time (slightly less than 0.77 seconds) is spent in I/O operations, impairing the scalability of the application beyond three threads.

# of threads	Average (s)	Median (s)	Std. deviation (s)	Relative speedup
1	125.8304	124.3780	3.2778	–
2	78.9234	78.0115	4.1704	1.5943
3	66.5678	67.5766	2.3982	1.8903
4	57.35964	56.9639	2.2142	2.1937

Table 6. Multithreaded block-based version (2.6 GB XML file)

The execution times of the parsing of the 2.6 GB input XML file (Table 6) follow the trend observed with the 100 MB files, although with a difference: adding a fourth thread still diminishes the execution times. This means that larger files yield larger scalability of the algorithm, although the gain beyond the $2^n d$ thread are somehow reduced. Nonetheless, the applied strategy yields a two-fold improvement, with the execution time being reduced from slightly less than 126 seconds (one thread) to roughly 57 seconds (four threads). Table 6 also

[2] http://xmlsoft.org/xmlreader.html

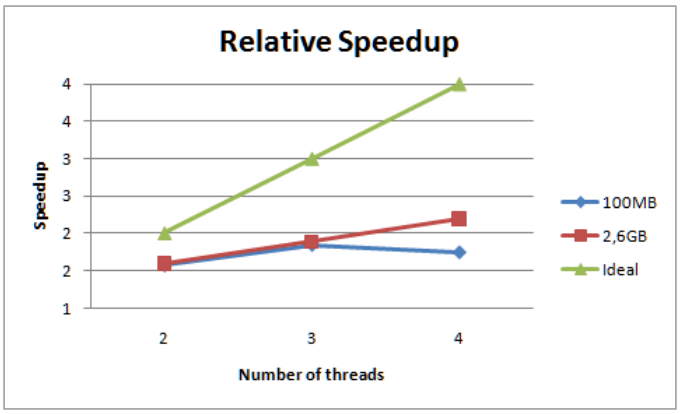


Fig. 3. Relative speedup for the block-based version

shows the relative speedups. Relative speedups for both the 100 MB and 2.6 GB input files are shown in Figure 3.

5 Conclusion and Future Work

In this paper, we study the processing of very large XML files, an issue that has gained relevance in the last years with the continuous increasing growth of datasets and the generalized use of the XML markup language for representing data. We present and study a multithreaded block-based strategy targeted at multicore machines when processing very large XML files. Our approach delivers some performance improvement achieving a 1.59 relative speedup with two threads and nearly 2.19 with four threads when processing a 2.6 GB XML files.

We have also confirmed that the performance of multicore machines for processing very large stream of XML data are bounded to their I/O subsystems, with the block-based approach failing to scale beyond the contention that exists due to the read and write I/O operations.

In future work, besides studying the effects of using a number of threads greater than the number of cores, we plan to address the issue of creating an extension to the libxml2 libraries for processing very large XML files in multicore machines. We also plan to study more advanced I/O subsystems, like the use of separate disks for the input and output streams.

References

1. Ditch, W.: XML-based Office Document Standards. JISC Technology & Standards Watch (2007)
2. Head, M.R. Govindaraju, M.: Parallel processing of large-scale xml-based application documents on multi-core architectures with piximal. In: eScience, 2008. eScience '08. IEEE Fourth International Conference on, Binghamton, NY, USA, Grid Comput. Res. Lab., SUNY Binghamton (2008) 261–268
3. W3C: Document Object Model (DOM) (2010)
4. Almeida, J.J., Simes, A.: XML::TMX — processamento de memrias de traduo de grandes dimenses. In Ramalho, J.C., Lopes, J.C., Carro, L., eds.: XATA 2007 — 5 Conferncia Nacional em XML, Aplicaes e Tecnologias Aplicadas. (2007) 83–93
5. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobbs Journal **30**(3) (2005) 16–20
6. Anderson, D., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: SETI@home: an experiment in public-resource computing. Communications of the ACM **45**(11) (2002) 56–61
7. XMLSoft: Module SAX2 from libxml2 (2010)
8. The Simple API for XML: About SAX (2010)
9. Fadika, Z., Head, M., Govindaraju, M.: Parallel and Distributed Approach for Processing Large-Scale XML Datasets. (2009)
10. Dean, J., Gemawhat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th OSDI. (2004) 137–150
11. Apache: Hadoop - Open Source Implementation of MapReduce. http://lucene.apache.org/hadoop/ (2009)
12. Anderson, D.: BOINC: A System for Public-Resource Computing and Storage. In: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004, Pittsburgh, USA. (2004) 4–10
13. WWU Munster: Quantum Monte Carlo at Home (http://qah.uni-muenster.de/) (2010)
14. Berkeley, U.o.C.: The SETI@home Project (http://setiathome.berkeley.edu/) (2010)
15. Karlsson, B.: Beyond the C++ standard library. Addison-Wesley Professional (2005)

Short Papers

Parsing XML Documents in Java using Annotations

Renzo Nuccitelli, Eduardo Guerra, Clovis Fernandes

Aeronautical Institute of Technology, Praça Marechal Eduardo Gomes, 50
Vila das Acacias - CEP 12.228-900 – São José dos Campos – SP, Brazil
{renzon,guerraem}@gmail.com, clovistf@uol.com.br

Abstract. In Java language, to parse XML documents DOM, or some framework based on it, is a widely used solution. However, for large documents it is not possible use one of this approaches once they load the full XML tree, causing memory problems. In this context, it is advised the use of SAX, which is a pull parser and has an architecture based on Observer pattern. However, it has some drawbacks in its Java implementation, which make the parsing classes poorly extensible and granular. Intending to make pull parser development in Java easier, this paper proposes JColtrane, which is a framework that implements the parsing of XML documents based on code annotations. This paper also presents a case study which compares this approach with the regular SAX usage.

Keywords: XML, Java, SAX, JColtrane, DOM, Parser

1 Introduction

Nowadays XML is adopted in a lot of applications. Hundreds of XML-based languages have been developed, including RSS, Atom, SOAP and XHTML [1]. With the increasing use of XML-based languages tools to process them are gaining an increasing importance.

SAX, which stands for Simple API for XML, is a serial access parser API for XML. It provides a mechanism for reading data from an XML document [2] based on events. It reads XML nodes in the order they appear in document and trigger API methods in the handler classes, which are responsible to handle the information received for each element. This structure is based on the Observer pattern [3], where the handler class must extend DefaultHandler and receives an invocation in the callback methods each time that the SAX parser reach a node. All the information about each XML element are passed in the arguments for the developer to be able to build the parsing logic. This kind of Observer implementation for this context brings some issues, like a high cyclomatic complexity in each method and a high coupling with the API [4].

The present work tried to solve these issues by the creation of JColtrane [5], which is a metadata-based framework [6] which implements a pull parser that uses SAX in background. It provides annotations in the handler methods that can define which kind of element that method should handle. Parameter annotations can also be used the determine the information which need to be received as arguments. Using this

approach, the parser code becomes cleaner, less complex and more cohesive.

This paper is structured as follows: section 2 presents the existent approaches for XML parsing in the Java language; section 3 introduces the idea of using annotations for event-based parsing; section 4 presents the features of JColtrane and its internal structure as well; section 5 presents a case study which compares JColtrane with regular SAX usage; and section 6 concludes the paper.

2 XML Parsing in Java Language

XML parsing in Java platform is usually solved by three main approaches: DOM[7], SAX [2] and JAXB [12].

Using DOM [7], the XML document is loaded in memory, so each node is accessed through an object representation similar to a tree [2]. From one node on that tree, it is possible to reach any node in this structure. The main drawbacks of this process are: (a) every time that a document needs to be parsed, it is necessary to implement the logic to visit the nodes, such as a deep or a breadth search; (b) all nodes are loaded in memory and if the XML document is too large or there are a lot of documents being processed at the same time, it can cause the program to use an excessive amount of memory [2].

The first issue can be solved using Visitor pattern [8], but this solution can still generate a complex solution. The second issue can be solved increasing the available memory, which are not always possible due to hardware limitations.

The JAXB [12] approach, use code annotations to map the application classes to XML elements defined in an XML Schema. It shares with DOM one of its drawbacks, which is to load all the information of the XML document in the application memory. It is also are only suitable for documents which have a defined and fixed format, since the application classes should also follow it.

For situations where DOM or JAXB are not suitable, such as large documents, documents without a fixed format and parsing where only a piece of the information is relevant, the use of SAX [2] is advised. It reads the XML document throwing events for each piece of information found, such as the begin of elements, characters inside elements and the end of elements. These events generated by the parser are handled by an application class responsible to capture the necessary information

Since only the information about the current element are stored by the parser, it does not suffer from memory issues. Further, it does not need the document to have a fixed format because every element is handled by the methods. This approach is indicated also when the parsing does not need every information in the document, since it is possible to ignore the method invocations related to undesired elements.

The following presents the main handler methods that could be inherited from the *DefaultHandler* class, which should be extended by the SAX handlers:

- *startDocument()* receives the notification of the beginning of the document.
- *endDocument()* receives the notification of the end of the document.
- *startElement(String uri, String localName, String qName, Attributes attributes)* receives the notification of the start of an element.

- *endElement(String uri, String localName, String qName)* receives the notification of the end of an element.
- *characters(char[] ch, int start, int length)* receives the notification of character data inside an element.

To illustrate how a SAX handler works, Fig. 1 presents an example of an XML document and Fig. 2 presents the source code used to parse this document. It prints in the console an announce of the begin and end of the document. Additionally, it also prints the path of each element in its beginning and the body content followed by the path in the end. To enable that, the attributes *currentBranch* and *body* are used to store information along the method invocations.

```
<?xml version="1.0" encoding="UTF-8"?>
<beanDescriptor>
   <line>
      <property  mandatory="true" >
         body of the property
      </property >
   </line>
</beanDescriptor>
```

Fig. 1. Example of XML document

Despite SAX is the most appropriate solution for some situations, the class created as the SAX handler faces some issues that makes difficult its development and maintenance. The following describes these issues:

- Since the same method is used to receive the events generated by different elements, in some cases it is necessary to use nested *if*/*else* statements to differentiate their processing, increasing the class complexity.
- When the same event is used for more than one concern, their processing should be triggered from the same method reducing the method cohesion.
- When a new node is read, the parser do not keep the states of past nodes. Consequently, if it is needed to some information of the past nodes, such as the parent, the handler class should store them in attributes, which also increases the class complexity.
- It is mandatory to receive all attributes and informations from read node, even if they are not necessary in the processing.
- The node attributes are received as a String, forcing their cast values to desired types, such as *Integer* or *Date*.
- It is necessary to build and store manually the node's body content, since the content of a single element can generate more than one call to the *characters()* method.

Some of the presented issues can be cleared perceived on the simple example presented on Fig. 2. The methods *startElement()* and *endElement()*, for instance, receives parameters that are not necessary in the implemented logic. Another example is the manually construction of the body content and the necessity of attributes to keep past informations, such as *currentBranch*.

```
public class SAXHandler extends DefaultHandler {
   private StringBuilder currentBranch=new StringBuilder("/");
   private StringBuilder body= new StringBuilder();

   public void startDocument(){
      System.out.println("Starting Document: Hello World!!!\n");
   }
   public void endDocument(){
      System.out.println("End Document: Bye bye World!!!\n");
   }
   public void startElement(String uri, String localName,
                  String tag,Attributes atributos)throws SAXException{
      currentBranch.append(tag+"/");
      System.out.println(currentBranch.toString());
      System.out.println("Executing some action in start element\n");
      body.delete(0,body.length());
   }
   public void endElement(String uri, String localName,
                  String tag)throws SAXException{
      if(!body.toString().equals("\n")&&body.length()!=2)
         System.out.println(body.toString().trim());
      System.out.println(currentBranch);
      body.delete(0, body.length());
      System.out.println("Executing some action in end element\n");
      currentBranch.delete(currentBranch.length()-tag.length()-1,
                           currentBrunch.length());
   }
   public void characters(char[] ch, int start, int lenght)
                  throws SAXException{
      body.append(ch,start,lenght);
   }
}
```

Fig. 2. Example of SAX parsing class

3 Metadata-based SAX Parsing

The existence of these difficulties in the SAX programming model motivated the creation of a framework with a metadata-based approach [6] that addresses them. It uses code annotations [9] to define in the handler class methods conditions to that method execution. Annotations in the method arguments could also be used to identify which information should be passed there. The following describe the main functionalities implemented with their rationale:

- Annotations in the handler methods are used to express the conditions to their execution. The conditions can use contextual data such as the element name, its parents and its attribute values. Consequently, the processing of elements can be separated in different methods by similarity groups. Additionally, the same element can be handled by more than one method with different concerns.
- A parsing context variable are provided to allow access to parent nodes

informations. While SAX do not store any information about the past nodes, the proposed approach stores only the information about the parent nodes. It does not consume a large amount of memory and can be useful for the method conditions and the information processing.

- A general use map kept by the framework can be passed as an argument to the methods to avoid the creation of class attributes. It also can be used to share information among different handler classes.
- Parameter annotations can be used to enable the method to receive only the arguments needed for its logic. Each parameter must be annotated to identify which information it should receive. Examples of supported information are the element name, an attribute value, the attribute map or the general use map. This feature avoid the method to receive unnecessary arguments.
- An automatic casting of attributes and information from XML to primitive Java classes are performed by the framework. It uses the declared parameter type which should receives that information and perform the convertion.

To exemplify the use of the proposed approach, Fig. 3 presents a handler with the same functionality of the presented in Fig. 2. This class needs neither to extend a framework class nor to override its methods. The methods can be freely created and configured by one of the conditional annotations. The parameters also should be annotated to configure what should be received. The element body, for instance, can be received as a parameter in methods annotated with the *@EndElement* annotation.

```
public class HelloWorld {

   @StartDocument
   public void executeInStartDocument(){
      System.out.println("Hello World!!!\n");
   }
   @EndDocument
   public void executeInEndDocument(){
      System.out.println("Bye bye World!!!\n");
   }
   @StartElement
   public void executeInStartElement(
                  @CurrentBranch String currentBranch){
      System.out.println(currentBranch);
      System.out.println("Executing something in start element\n");
   }
   @EndElement
   public void executeInEndElement(
                  @CurrentBranch String currentBranch,
                  @Body(tab=false, newLine=false) String body){
      System.out.println(currentBranch);
      if(body.length()!=0)
         System.out.println(body);
      System.out.println("Executing something in end element\n");
   }
}
```

Fig. 3. Example of JColtrane parsing class

4 JColtrane

JColtrane is an open source framework which implements the concept of the metadata-based SAX parser presented in previous section. Its name is a tribute to the famous sax player John Coltrane. It had already been used in production software, like frameworks which reads a XML descriptor and applications which interprets Brazilian electronic invoices.

The next subsections presents the JColtrane main features and details about its internal structure as well.

4.1 Main Annotations

The main annotations defined in JColtrane correspond to the methods which can be inherited from the *DefaultHandler*. The names chosen for the annotations intend to decrease the learning curve for those who already had been used SAX. Table 1 presents those annotations with its description.

Annotation	Description
@StartDocument	annotated method must execute in the beginning of the XML document
@EndDocument	annotated method must execute in the end of the XML document
@StartElement	annotated method must execute in the beginning of an XML element
@EndElement	annotated method must execute in the end of an XML element

Table 1. Main JColtrane's annotations

The annotations *@StartElement* and *@EndElement* provides attributes that can be configured to filter which elements should be processed by that method. Table 2 detail these options. Fig 4 presents an example of these attributes usage, where the method should be executed in the end of elements with attributes valued as *"pt"*.

Attribute	Arguments	Meaning
tag	regex	element must have a tag name that matches the given regular expression
uri	regex	element must have an uri that matches the given regular expression
localName	regex	element must have a local name that matches the given regular expression
priority	int	methods with higher priorities are executed before those with lower priorities
containAtribute	*@ContainAttribute*	element must have an attribute that matches the given regular expression in the name and value arguments

Table 2. Attributes of *@StartElement* and *@EndElement* annotations

```
@EndElement(
   priority=3,
   attributes={@ContainAttribute(value="pt")}
)
```

Fig. 4. Example of @EndElement conditions

4.2 Filter Annotations

JColtrane also provide other annotations which can be combined with *@StartElement* and *@EndElement* annotations to enable a more precise filtering. They are based on the element parents and are presented in Table 3. The Fig. 5 presents an example of the usage of the *@BeforeElement* usage. The presented method should only execute for elements which the parent node are named *"paragraphy"*.

Annotation	Description
@BeforeElement	annotated method must execute when current element has the element specified in the annotation as an ancestor in specified level
@InsideElement	annotated method must execute when current element has the element specified in the annotation as an ancestor in any level

Table 3. Filter Annotations

```
@BeforeElement(elementDeep=1,tag="paragraphy")
@StartElement
public void executeInStartElement(
               @CurrentBranch String currentBranch){
   System.out.println(currentBranch);
   System.out.println("Executing something in start element\n");
}
```

Fig. 5. Example @BeforeElement Annotation

The framework also allow the creation of custom annotations which can define personalized conditions in the parsing. The custom annotation should be annotated with *@ConditionFactoryAnnotation* passing as an attribute a class that interprets the annotation. This class should implement the interface *ConditionsFactory* and its method *getContidions()*, which returns a list of instances of the type *Condition* that should implement the logic to indicate if the method should be executed or not.

4.3 Handler Method's Parameters

Another important issue in JColtrane is how the parsing information are passed to the methods. It has an special class, named *ContextVariables*, which provides some import informations about the XML document and the context of the current element. The instance of this class can be received in any method annotated with

@StartElement or *@EndElement*. In order to receive it in a method, it is only necessary to declare an argument of the type *ContextVariables*.

Besides this class instance, it is also possible to receive separately any piece of information about the current element. In order to do that, the framework provides some annotations which can be used to configure in the parameters which information is needed. The Table 4 presents the JColtrane's parameter annotations and the value which is returned to be passed in the argument.

Annotation	Description
@Attribute(qName)	Returns the value of the attribute passed as the annotation attribute. Make automatic convertion for all primitive types or their corresponding wrapper classes. It returns *null* for objects or the correspondent MIN_VALUE for primitive types if element does not contain the attribute.
@AttributeMap	Returns a map containing all the attributes from the current element.
@CurrentBranch	Returns the current branch from the current element
@Uri	Returns the uri from the current element
@Tag	Returns the tag from the current element
@LocalName	Returns the local name from the current element
@GeneralUseMap	Returns the *GeneralUseMap* from *ContextVariables*. This map can be used to share information among the methods, and even among different handler classes.
@Body	Returns the body of current element and can only be used at methods annotated with *@EndElement*. It provides the attributes *newLine* and *tab* to configure if each kind of character should be or not included in the returned String.

Table 4. Parameter annotations provided by JColtrane

Fig. 6 presents an example of the use of a parameter annotation to configure what should be received by the method. In the example, the integer value of the *"page"* attribute should be passed in the first argument. Other examples can also be found in Fig. 3.

```
@StartElement
public void executeUserStartTest(
                @Attribute("page") int pageNumber){
   //method implementation
}
```

Fig. 6. Example of an argument's annotation usage

4.4 Framework Structure

JColtrane's main class is a regular SAX parser which receives the SAX events and invoke the correspondent methods in its handler classes. Consequently, it can be used with any existent SAX parser. Fig. 7 presents a code example of how to use a

JColtrane's handler to parse an XML document. The class JColtraneXMLHandler encapsulates the framework functionalities as a regular SAX handler. The class MyJColtraneHandler represents the application class developed using the JColtrane annotations, which is responsible to extract the information from the XML file.

```
public static void parseXMLStream(InputStream in){
  try {
    SAXParser parser = SAXParserFactory.newInstance().newSAXParser();
    InputSource input = new InputSource(in);
    parser.parse(input, new JColtraneXMLHandler(
                                  new MyJColtraneHandler()));
  } catch (Exception e) {
    throw new RuntimeException("Problems loading XML file", e);
  }
}
```

Fig. 7. Example of JColtrane's Usage

Fig. 8 presents the main steps in the framework execution. When JColtrane receives the the annotated classes in its constructor, it reads and interpret the annotations storing the metadata in internal descriptors. After the beginning of the XML parsing, for each SAX event received by the JColtrane main handler, it perform the following steps: (a) it iterates through the handler methods list and selects the ones which the conditions are satisfied by the current element; (b) the framework verifies which information should be passed in each parameter and retrieves them; finally (c) the methods are executed using the Reflection API in the priority order.

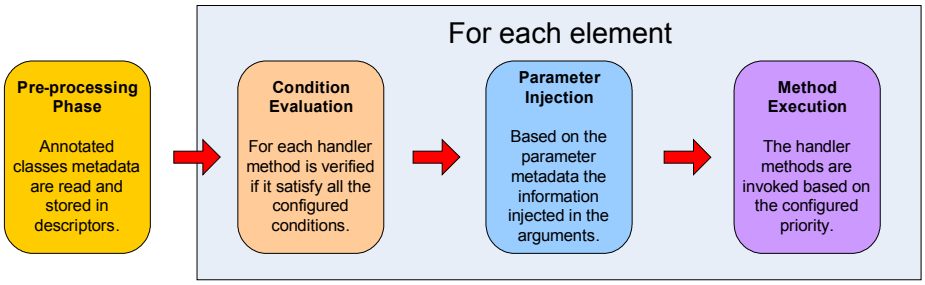


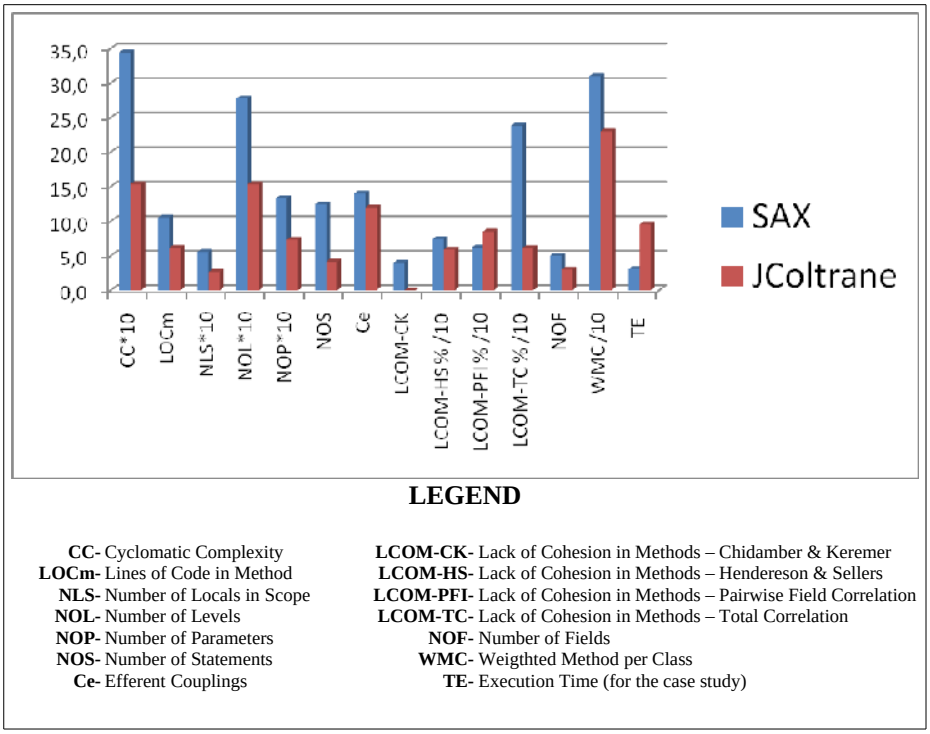
Fig. 8. JColtrane´s Internal Structure Overview

5 Case Study

To compare the use of the JColtrane framework with the regular SAX parsing, the authors used an existent parsing code developed for an open source framework and generated a similar solution using the proposed approach. The Esfinge Framework [11] had been chosen for the following reasons: (a) it uses SAX for parsing; (b) the XML structure are neither too complex or too trivial; and (c) it has unit testing for the parsing class. These unit tests were used to ensure that the parser developed with JColtrane are implementing the same functionality.

The chart presented in Fig. 9 presents the metrics obtained from both solutions.

The Eclipse plugin Metrics were used to extract these values from the source code. The execution time were measured executing each parser two hundred times and calculating the average value.



LEGEND

CC- Cyclomatic Complexity
LOCm- Lines of Code in Method
NLS- Number of Locals in Scope
NOL- Number of Levels
NOP- Number of Parameters
NOS- Number of Statements
Ce- Efferent Couplings

LCOM-CK- Lack of Cohesion in Methods – Chidamber & Keremer
LCOM-HS- Lack of Cohesion in Methods – Hendereson & Sellers
LCOM-PFI- Lack of Cohesion in Methods – Pairwise Field Correlation
LCOM-TC- Lack of Cohesion in Methods – Total Correlation
NOF- Number of Fields
WMC- Weigthted Method per Class
TE- Execution Time (for the case study)

Fig. 9. SAX and JColtrane metrics for Esfinge's XML document parsing

Based on the chart, it is possible to conclude that JColtrane decreases the general code complexity. Not only the size metrics, such as lines of code and number of parameters, reduced, but also the cyclomatic complexity and the coupling. From the cohesion metrics, only in the Pairwise Field Correlation the JColtrane presented a higher value than SAX. That can be explained by the fact that in the proposed approach the use of attributes is reduced, and this metric measure how different are the attribute sets accessed by the class methods.

These results assessed by the metrics were expected, since JColtrane addresses issues related to the difficulty of building the SAX parsing code. The use of method annotations for filtering the elements reduce the number of conditional statements inside the method. Further, the possibility to trigger the execution of more than one method from one event allows a better responsibility division among them. Additionally, the use of parameter annotations enables the method to receive only the necessary information for its execution. Based on those facts, it is possible to state that using the proposed approach the source code becomes cleaner and less complex.

The main drawback of JColtrane was the execution time, which is almost three

times slower than the regular SAX. This fact can be attributed to the internal overhead for executing the framework functionalities and to the reflection usage in the methods invocation.

6 Conclusion

This paper presented the JColtrane framework which uses code annotations to simplify the development of event-based XML parsers. The annotations are used to define conditions on the handler method execution based on the current element information. They can also be used on the parameters to configure which information should be passed as an argument.

A simple comparative study was performed to compare an existent SAX parser implementation with a similar one using JColtrane. On one hand, the source code developed with the proposed approach had a lower complexity and was more simple than the existent one. On the other hand, it also were almost three times slower than the regular SAX usage. Future works can measure other issues, such as development time, in different case studies and environments. Another further work is the comparison of JColtrane with other frameworks.

JColtrane is an open source project and has more than 350 downloads by the time this article was writen. It also was the first place over 70,000 projects in XML challenge in 2008, promoted by IDUG and IBM. It was already used by production frameworks and applications in the parsing of XML files. These facts highlights the contribution of JColtrane to the XML development community.

References

1. Wikipedia: XML (2010).http://en.wikipedia.org/wiki/XML
2. SAX. SAX Project, 2004. Available on http://www.saxproject.org/ accessed in 2010-01-03.
3. Gamma, E.; Helm, R.; Johnson, R. ;Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented. Ed. Addison-Wesley (1994).
4. Nuccitelli, R.: Tratamento de Evento com Uso de Metadados. Relatório Técnico - Instituto Tecnológico de Aeronáutica (2008).
5. Nuccitelli, R.; Guerra, E. M.: JColtrane – better than SAX alone (2008). http://jcoltrane.sourceforge.net/
6. Guerra, E. M. ; Souza, J. T. ; Fernandes, C. T. . A Pattern Language for Metadata-based Frameworks. In: 16th Conference on Pattern Languages of Programs, 2009, Chicago.
7. Sun Microsystem. DOM. http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/package-summary.html
8. O´Gara, M.; Geelan, J.; Gardner, D.; Katz, M.: Visiting the DOM (2004). http://xml.sys-con.com/node/47287
9. JSR175. JSR 175: A Metadata Facility for the Java Programming Language. 2003. Available at http://www.jcp.org/en/jsr/detail?id=175 acessed in 2009-

12-17.
10. Sun Microsystem. DefaultHandler class documentation. http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/helpers/DefaultHandler.html
11. Guerra, E. M.: Esfinge Framework (2007). http://esfinge.sourceforge.net/
12. JSR 222: Java Architecture for XML Binding (JAXB) 2.0. 2006. Available at http://jcp.org/en/jsr/detail?id=222 acessed in 2009-12-17.

XML, Annotations and Database: a Comparative Study of Metadata Definition Strategies for Frameworks

Clovis Fernandes, Douglas Ribeiro, Eduardo Guerra and Emil Nakao

Aeronautical Institute of Technology, Praça Marechal Eduardo Gomes, 50
Vila das Acacias - CEP 12.228-900 – São José dos Campos – SP, Brazil
{ douglasribeiro.sh, emilnakao, guerraem}@gmail.com
clovistf@uol.com.br

Abstract. The use of metadata in frameworks contributes for a cleaner code with properties unbounded from the implemented logic, helping for a better program development. An incorrect choice of metadata definition form can compromise the framework's extensibility and usage, but there is no study yet that compares different metadata definition forms. To address this issue, the present work compares three metadata definition strategies in face of different requirements, by developing frameworks that uses them and comparing the difficulty for its implementation and its usage. The conclusion of this work clarifies how and with which strategy metadata should be used according to the requirements to avoid future problems. As a result, developers can prevent implementation troubles and unnecessary work because of a bad choice for the metadata definition strategy.

Keywords: Metadata, Java, Framework, XML, Annotations, Database

1 Introduction

Frameworks are implementations of systems or subsystems for solving a specific problem with great reusability [1]. A reflective framework uses techniques to introspect or modify classes during runtime to adapt itself for the application classes, consequently reducing the coupling [2]. Some of them uses code conventions [3], such as naming patterns, to customize this adaptation for each class, however in some contexts this information is not enough. In those situations, the framework can demand the creation of additional information for each class, which is known as metadata [4].

The use of metadata is a common practice in recent software development; it can be evidenced by its usage on highly implemented frameworks and APIs such as Hibernate [5], Struts [6], Spring [7] and EJB 3 [8]. In the previous versions of those frameworks, XML was largely used as the metadata definition type, however, in the recent versions, its use has been reduced because the use of annotations became popular among framework developers.

According to the framework requirements, a different strategy for metadata definition could be most appropriate. However, there was no comparative study that

investigates how each one can be implemented facing different requirements about class metadata. This work's in an approach to do such a comparison, a base metadata-based framework that supports annotations, XML and databases for metadata definition was implemented. This base framework was refactored to support the following three requirements in distinct scenarios: (a) runtime metadata modification; (b) distinct metadata sets for the same class; and (c) custom metadata definition. As a result, benefits and drawbacks in each situation were identified for each different metadata definition strategy, providing an useful information to support decisions about the best choice to define metadata.

Section 2 presents an overview of metadata definition strategies; section 3 presents the case study, with a description of the developed framework and its structure. Some important concepts that were used for the case study are also described. Section 4 describes the three refactoring made to the base framework, presenting the benefits and drawbacks for each definition strategy, and Section 5 provides the work conclusions.

2 Metadata Definition Strategies

Metadata is an overloaded term in computer science and can be interpreted differently according to the context. In the context of object-oriented programming, metadata is information about the program structure itself such as classes, methods and attributes. A class, for example, has intrinsic metadata like its name, its superclass, its interfaces, its methods and its attributes. In metadata-based frameworks, the developer also must define some additional application-specific or domain-specific metadata. Even in this context, metadata can be used for many purposes. There are several examples of this, such as source code generation [15]; compile-time verifications [16][17]; and class transformation [18].

There are many ways to define a metadata. For instance, it can be embedded in the source code, using name standards or annotations, or stored in external sources, like a XML file or a database. Three main metadata definition types that are considered in this paper are code annotations, XML documents and databases.

The most simple metadata definition type is the use of code conventions [3]. For instance, the Java Beans [9] standard's define that methods for acquiring values must begin with the word "get", or methods for setting values must begin with "set". Another example is JUnit 3 [10], which interprets methods beginning with 'test' as test cases implementation. Ruby on Rails [11] is an example of a framework known by the naming conventions usage. This metadata definition type is not expressive enough to represent complex metadata structure and for that reason it is not included in the comparison.

Another option for metadata definition is the use of an external file. The XML file type is the most popular, because of its simplicity and flexibility. Despite of being essentially a plain text document, the data in an XML file is self-describing, because its markups are meaningful. This characteristic makes XML documents easier to be read by humans than other existent data storage file formats.

It's also possible to store the metadata externally using a database system. That option is not simple as using an XML file, but the data can be stored more secure and with a transaction access. For a large amount of data, searching and editing them become's simpler by its access using the SQL language.

The possibility to modify the metadata during deploy-time or even during runtime without recompiling the code is an advantage of external approaches. However, the definition is more verbose because it has to reference and identify the program elements. Furthermore, the distance which the configuration keeps from the source code is not intuitive for some developers.

Another alternative that is becoming popular in the software community is the use of code annotations, which is supported by some programming languages like Java [12] and C# [13]. Using this technique the developer can add custom metadata elements directly into the class source code, keeping this definition less verbose and closer to the source code. The use of code annotations is also called attribute-oriented programming [14].

3 Case study Presentation

The case study starting point was the creation of a base framework, which was further modified in order to study the implementation of three different requirements for metadata definition. The base framework, called **ProfilePrinter**, uses metadata to structure the data of a class instance into a String. It reads metadata that defines a formatting specification and uses it to write a text in a custom format with the information of an object passed to it as a parameter.

Section 3.1 shows a usage example for the framework, explaining its main functionality. Section 3.2 presents some design patterns used, for a better understanding of the internal structure described in Section 3.3.

3.1 ProfilePrinter Usage Example

The **ProfilePrinter** framework main class is called *BeanPrinter*. The method *printBean()* returns formatted String with all instance properties. The metadata that should be defined for this framework must be related to how each property should be formatted. For instance, it supports the definition of labels, prefixes, suffixes and different kinds of formatting.

An example of the framework usage is presented in Fig. 1. An instance of the class *Person* is created, and further an instance of *BeanPrinter* with the generic type *Person*. The method *setBeanType()* is used to determine the source that should be used to read the metadata. In this example, the framework is configured to read from an XML document. At last, the output was generated with the method *printBean(person)*. The expected output for the metadata defined further in this section is presented in Fig. 2.

```
public class PrintingDemonstration {

   public static void main(String[] args) {
      Person person = new Person("John", 56.0987f, 1.90f,
                           new Date(System.currentTimeMillis()));

      BeanPrinter<Person> personPrinter = new BeanPrinter<Person>();
      personPrinter.setMetadataType(BeanPrinter.MetadataType.XML);
      System.out.println(personPrinter.printBean(person));

   }
}
```

Fig. 1. Example of a BeanPrinter class usage.

```
Weight: 56.10 kg
Birthday: 19/11/2009
Name: Mr. John
Height: 1.90 m
```

Fig. 2. Expected output for the example.

Metadata can be defined using code annotations, an XML document or a database table. In the XML definition, the formatting options are defined in a file named formatter.xml and is represented in Fig. 3. In addition, the example of the code annotations used with the Person class itself is illustrated in Fig. 4. At last, the table information that defines the metadata is presented in Fig. 5. It is important to notice that despite the format being different, the information represented is the same.

```
<?xml version="1.0" encoding="UTF-8"?>
<formatter>
  <property class-fullname="beans.Person"  attribute="name"
            label="Name" prefix="Mr."/>
  <property class-fullname="beans.Person"  attribute="birthday"
            label="Birthday" date-mask="dd/MM/yyyy"/>
  <property class-fullname="beans.Person"  attribute="height"
            label="Height" number-format="##.00" suffix="m"/>
  <property class-fullname="beans.Person"  attribute="weight"
            label="Weight" number-format="##.00" suffix="kg"/>
</formatter>
```

Fig. 3. Metadata definition for the class Person using XML.

```
public class Person {

   private String name;
   private float weight;
   private float height;
   private Date birthday;

   @Prefix("Mr.")
   @Label("Name")
   public String getName() {
        return name;
   }

   @NumberMask("##.00")
   @Label("Weight")
   @Suffix("kg")
   public float getWeight() {
        return weight;
   }

   @NumberMask("##.00")
   @Label("Height")
   @Suffix("m")
   public float getHeight() {
        return height;
   }

   @DateMask("dd/MM/yyyy")
   @Label("Birthday")
   public Date getBirthday() {
        return birthday;
   }
}
```

Fig. 4. Metadata definition for the class Person using Annotations.

	id [PK]	attribute character	class-fullname character varyi	date-mask character v	label character	number-format character varyii	prefix charac	suffix chara
1	1	name	beans.Person		Name		Mr.	
2	2	birthday	beans.Person	dd/MM/yyyy	Birthday			
3	3	height	beans.Person		Height	##.00		m
4	4	weight	beans.Person		Weight	##.00		kg
*								

Fig. 5. Metadata definition for the class Person using a Database.

3.2 Design Patterns Used

The framework architecture was conceived to minimize the amount of work required for comparative study refactoring. In particular, some design patterns from **Pattern Language for Metadata-based Frameworks** [4] were used, such as Metadata Container, Reader Strategy, Processing Layers and Delegate Metadata Reader. The following is a brief explanation of each pattern:

- **Metadata Container** separates framework logic from metadata reading logic, through specialized classes, called Metadata Containers, that stores the information read from a metadata source, which can be annotations, XML or a database, for example.
- **Metadata Reader Strategy** provides a structure to allow metadata reading from different sources and definition types, such as XML and annotations. It can be considered a specialization of the Strategy pattern.
- **Metadata Processing Layers** focus on the functionality expansion problem for this kind of framework, dividing the logic in many processing layers. The main execution logic is organized in execution layers, enabling their individually development and insertion.
- **Delegate Metadata Reader.** Approaches the metadata schema extension, delegating to configurable classes, named Concrete Metadata Readers, the responsibility for interpreting metadata elements. A metadata element can be an annotation or an element in a XML file, for example.

3.3 Framework structure

The **ProfilePrinter** has four modules: *StringGenerator*, *AnnotationsCore*, *XMLCore* and *DatabaseCore*. The "core" modules are responsible for reading metadata in a specific format and making them available in a standard way defined by the *StringGenerator*. Fig. 6 illustrates the main classes of the framework.

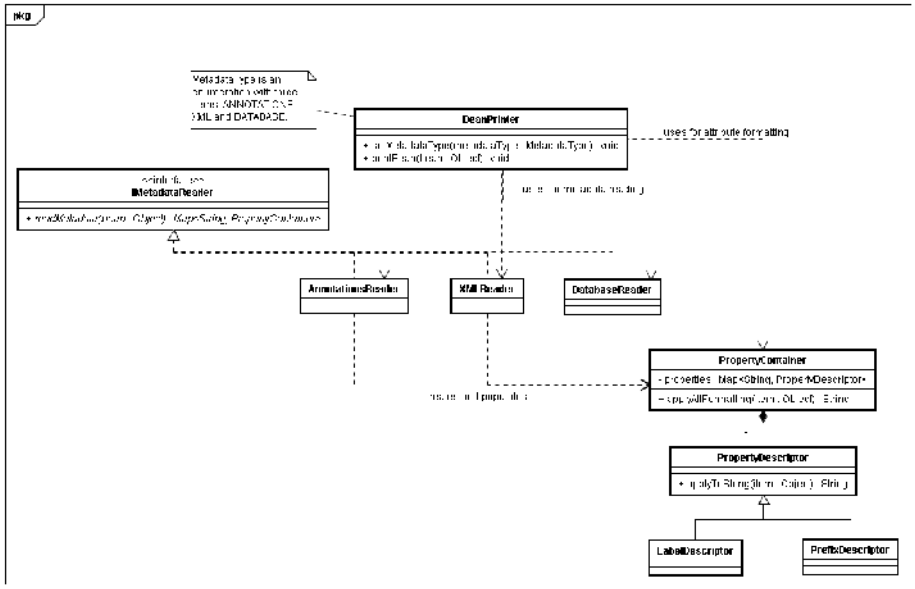


Fig. 6. Framework Structure.

The Metadata Reader Strategy pattern is used to read metadata from the source type specified by the method *setMetadataType()*, from the *BeanPrinter* class. Each core module provides one implementation for the interface *IMetadataReader*, such as *AnnotationsReader* and *XMLReader*.

The information read from each source is stored in *PropertyContainer* class instances, based on the **Metadata Container** pattern. Each *PropertyContainer* contains a set of subclasses of *PropertyDescriptor*, for instance *LabelDescriptor* and *PrefixDescriptor*. Those classes are responsible for storing specific information about a metadata type, such as its label or its prefix, and for applying the formatting to the object's properties to generate the returned *String*.

Each *PropertyContainer*, besides storing metadata, also provides a method to apply the formatting, which delegates the execution to the associated *PropertyDescriptor* instance. That structure is an implementation of the **Metadata Processor** and **Metadata Processing Layers** pattern.

4 Comparative Study

The following three different requirements were considered to extend the framework: runtime metadata modification, distinct metadata sets for the same class and custom metadata definition. They represent requirements that can influence the decision of metadata definition type. The comparative study focuses, when possible, on the amount of work necessary to perform the extension for each metadata definition type in the framework and be implemented by the framework users.

4.1 Runtime Metadata Modification

Runtime metadata modification is necessary for applications that cannot be stopped frequently, but needs to have parameter adjustments during runtime in order to change their execution behavior. Server applications are an example of several applications' that needs to be in execution without interruption, but can use metadata in order to configure themselves at runtime. Security is an instance of concern that is usually configured by metadata, which in some applications can be changed at runtime.

For each metadata definition type the possibility of runtime modification and advantages were analyzed, and the conclusions are displayed in Table 1. Since annotations cannot be changed in runtime, it is not included in that table.

Table 1. Runtime Metadata Modification Comparison

Characteristic	XML	Database
Pros	Simpler and faster direct editing	More security in information storage. Data is well organized, which makes changing of data easier. Support to transactional access.
Cons	It can be disordered in excessively large XML files, or among a great amount of files, which can make changes more difficult.	Metadata information change cannot be performed directly so easily as in XML files, and consequently is not well suited when this change is not performed through the application itself.

It was not necessary to change the framework to support this modification using XML and databases. If the framework uses cache for metadata, it should be synchronized with those modifications to change the behavior at runtime.

The differences between these two approaches are more related to the characteristics of each one and not to the framework implementation. In one hand, XML files are more appropriate when those changes are performed directly by developers in the production environment. On the other hand, the database storage is more suitable when metadata is changed by the application itself, since it provides services like security and transaction management.

4.2 Distinct Metadata Sets for the Same Class

For some domains it is important to enable the definition of different metadata sets for the same class to be used in different contexts. An example of this need is faced by SwingBean [19] framework which uses metadata to generate forms and tables for a class. In this context, it is usual to have in an application different forms that fills information into the same class. This situation is also common for instance validation, where the rules can vary with the context.

In the *ProfilePrinter*, it is possible to consider that the application's numeric data and dates can vary according to the country. In order to use the framework in this context, each country requires a distinct metadata set. This requires a change in the framework to support the distinction and the reading of the different ones.

Metadata set implementation have different difficulty levels according to the metadata definition type being used. The modification in the base framework consisted in adding a new parameter to the print bean methods, which started to receive the metadata set name to be used. In addition, for each metadata definition type, the *IMetadataReader* class had to be changed, and the conclusion obtained by the comparison for each metadata definition type module is resumed in Table 2.

Table 2. Distinct Metadata Sets for the Same Class Modification Comparison

Characteristic	Annotations	XML	Database
Framework Implementation	It requires changes in all annotations and in their reading. Each annotation must have the 'set' information.	The framework does not need to be changed, since the sets can be in different files. It can also support the definition of metadata sets in the same file.	A column to represent the set should be added to the table, which is used to retrieve the desired information.
Framework Usage	It is necessary to define all metadata with the source code, which compromise its readability. The information of all sets became mixed in the source code.	The metadata sets can be organized conveniently, since they can be on the same or different files.	The set information must be configured for each metadata row.

In this scenario the definition in XML presented more advantages than the other alternatives. The fact that the sets can be defined in different files simplifies both the implementation and the use of this approach. The use of databases is also simple, but since metadata sets must be defined in the same table, a column should be created in order to differentiate them.

The use of annotations for different sets of metadata generates a large number of annotations that compromise the class readability. In Java this problem is even worse, since the same annotation cannot configure the same element. As a result, in order to insert metadata for different sets, it is necessary to create a new annotation to group these different definitions.

4.3 Custom Metadata Definition

Extensibility is a desirable characteristic of a framework [20]. Extending a framework which uses metadata implies, supporting the extension of the metadata schema and its respective processing. It is especially useful when the framework works in a domain where the metadata's are close to the application domain and cannot be anticipated by the framework developers. An example of a framework that supports such extension is Hibernate Validator [21], which allows the user to define its own validation rules with new annotations.

In order to perform the metadata definition, the framework user has to extend the class *PropertyDescriptor* to store this new information and to implement the new formatting specification. To create this extension it is also necessary to implement a *MetadataReader* for the new metadata element that are used according to the **Delegate Metadata Reader** pattern. The framework was refactored to prepare each module to accept such customized readers, and associated metadata elements. The Table 3 resumes the conclusion obtained from the **ProfilePrinter** refactoring.

Table 3. Custom Metadata Definition Modification Comparison

Characteristic	Annotations	XML	Database
Framework Implementation	It should support the creation of new annotations and their mapping to a *MetadataReader* class.	The solution can be implemented using the metadata reading as base and enabling extensions on the XML parsing.	Queries should be flexible to be changed in the framework.
Framework Usage	It just needs a new annotation and a new *MetadataReader* class.	New elements and attributes should be added in the document and mapped to reader classes.	It's hard to create the metadata. It needs a new database tables or columns and new database persistence classes.

In this context, the implementation with annotations was considered the most appropriate. This is related to the fact that the metadata extension includes parsing of the new metadata element, and the annotation interpretation by the framework user is easier than the other ones.

5 Conclusion

Metadata is used in many applications and frameworks, and contributes to more productive architectures. However, some metadata definition types, if not correctly used, can limit a framework's functionality and application. Until now there has been no study to compare the metadata definition types and present the advantages and disadvantages for them, under different scenarios, in order to help framework developers to decide which definition should be supported.

With this work, it was observed that there is not a metadata definition type that can be considered the most appropriate for all the situations. The study of modifications which can occur in frameworks that uses metadata has shown that in each situation a definition type could offer more advantages than the others types. Table 4 gives an overview of the conclusions extracted from the case study.

Table 4. Metadata Definition Types vs. Metadata Requirements

	Runtime metadata modification	**Distinct metadata sets for the same class**	**Custom metadata definition**
Annotations	**Does not support**	**Bad,** since the configuration of more than one set makes the source code unreadable.	**Good,** since needs only a new annotation, which contains which reader should be used to its interpretation.
XML	**Good**, especially in situations where metadata is changed directly by developers.	**Good,** since more than one XML file can be used, which facilitates organization.	**Average,** since it needs a new XML property or element, which must be mapped to the reader using also the XML file.
Database	**Good,** especially in situations where metadata is change by the application itself.	**Good,** since it is possible to store metadata sets, that can be easily filtered using database queries.	**Bad.** It needs a new column in the database table with the metadata. The creation of a new bean class that extends a default class defined by the framework is required.

According to Table 4, XML was the metadata definition type with the most reasonable balance between implementation effort and applicability for the three modifications performed in this work. Consequently, it can be recommended for frameworks that would like to implement all three modifications proposed.

The obtained conclusions should allow developers to choose more consciously the metadata definition types that are going to be supported by their frameworks. The difficulty level in maintenance and evolution of metadata support can be verified with the knowledge of what kind of operation with metadata and further framework's expansions are going to be needed.

Future related works approach the study of best practices for metadata modeling for each definition type and the study of combination of metadata definition strategies, where more than one metadata definition type is used as source at once to infer some information at runtime.

References

1. Johnson, R., Foote, B.: Designing Reusable Classes. In: Journal of Object-Oriented Programming, vol. 1, Number 2, pp. 22 – 35. (1988).
2. Foote, B., Yoder, J.: Evolution, Architecture, and Metamorphosis. In: Pattern Languages of Program Design 2, Chap. 13, pp. 295-314. Addison-Wesley Longman Publishing, Boston, USA (1996).

3. Chen, N.: Convention over Configuration, http://softwareengineering.vazexqi.com/files/pattern.html in 2009-12-17 . Accessed in April 4th 2010.
4. Guerra, E. M. ; Souza, J. T. ; Fernandes, C. T. . A Pattern Language for Metadata-based Frameworks. In: 16th Conference on Pattern Languages of Programs, 2009, Chicago.
5. Bauer, C., King, G.: Java Persistence with Hibernate Manning Publications. (2006).
6. Brown, D., Davis, C.M., Stanlick, S.: Struts 2 in Action. Manning Publications. (2008).
7. Walls, C., Breidenbach, R.: Spring in Action. Manning Publications, 2nd edition. (2007).
8. JSR220: Enterprise JavaBeans 3.0, http://www.jcp.org/en/jsr/detail?id=220 . Accessed in April 4th 2010.
9. JavaBeans(TM) Specification 1.01 Final Release, http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html . Accessed in April 4th 2010.
10. Massol, V., Husted, T.: JUnit in Action. Manning Publications. (2003).
11. Ruby, S., Thomas, D., Hansson, D., Hansson, D.H.: Agile Web Development with Rails. Pragmatic Bookshelf, Third Edition. (2009).
12. JSR175: A Metadata Facility for the Java Programming Language, http://www.jcp.org/en/jsr/detail?id=175 . Accessed in April 4th 2010.
13. Miller, J.: Common Language Infrastructure Annotated Standard. Addison-Wesley. (2003).
14. Schwarz, D.: Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5, http://missingmanuals.com/pub/a/onjava/2004/06/30/insidebox1.html . Accessed in April 4th 2010.
15. Dam Yanov, I., Holmes, N.: Metadata Driven Code Generation Using .NET Framework. In: 5th International Conference on Computer Systems and Technologies, pp. 1-6. Rousse, Bulgaria. (2004).
16. Quinonez, J., Tschantz, M., Ernst, M.: Inference of Reference Immutability. In: 22nd European Conference Object-Oriented Programming – ECOOP 2008, pp. 616 – 641. Paphos, Cyprus. (2008).
17. Ernst, M.: Type Annotations Specification (JSR308), http://types.cs.washington.edu/jsr308/specification/java-annotation-design.pdf . Accessed in April 4th 2010.
18. Project Lombok, http://projectlombok.org/. Accessed in April 4th 2010.
19. SwingBean, http://swingbean.sourceforge.net/ . Accessed in April 4th 2010.
20. Froehlich, G., Hoover, H.J., Liu, L., Sorenson, P. Designing Object-Oriented Frameworks. University of Alberta (1998).
21. Hibernate Validator - http://www.hibernate.org/412.html Accessed in April 4th 2010.

XML Archive for Testing: a benchmark for GuessXQ

Daniela Fonte[1], Pedro Carvalho[1], Daniela da Cruz[1], Alda Lopes Gançarski[2], and Pedro Rangel Henriques[1]

[1] University of Minho, Department of Computer Science, CCTC, Campus de Gualtar, Braga, Portugal
[2] Institut TELECOM, TELECOM & Manangement SudParis, CNRS SAMOVAR 9 rue Charles Fourier, 91011 Évry, France

Abstract Aiming at making easier the process of information retrieval from structured documents, we developed an environment, GuessXQ, to provide query-by-example assistance. GuessXQ offers the user the chance to choose a family of documents, and to annotate graphically a sample document, picked from the collection. From the visual annotation, that exemplifies the user's needs, GuessXQ infers a XQuery query that automatically applies to the collection to retrieve all the documents satisfying the information required by the user.
However, after developing a GuessXQ prototype, it is mandatory to validate pragmatically the effectiveness of the environment. This can be done, drawing a set of experiments. The success of such assessment strongly relies on the quality of collections used. This claim led to the construction of XAT, an archive of XML collections specially gathered in order to be useful for testing XML tools. Looking for a generic purpose, XAT was organized according to a classification criteria developed for this specific purpose.
This article is about the design and implementation of XAT.

1 Introduction

XML information access is a very important research interest due to the increasing amount of XML collections used in organizations. Different XML retrieval languages appeared, being the basis for XQuery, the standard XML query language. However, specifying XQuery queries may be too difficult for users not being familiar with the language. In this context, we are developing a system which allows the user to specify his needs in a query-by-example (QBE) way. Using this paradigm, a sample document extracted from the document collection is shown to the user to select the desired elements or attributes, as well as restrictions over them. GuessXQ leverages the user task: in one way, GuessXQ offers a visual/graphical user friendly interface; in another way it restricts XQuery to the operations which are the simplest ones, assuming that more complex queries are needed by other kind of users who need to know deeply XQuery itself. The

May, 2010 - Page. 127–138 ≻ATA 2010 — ISBN: 978-972-99166-9-4

effectiveness of this system can be validated by drawing a set of experiments over an XML test collection.

In this paper we describe how XAT(XML Archive for Testing) came to live from the need to test the GuessXQ system through XML benchmarking. XML benchmarking consists of an XML test collection and a set of execution tests to measure used memory and processing time. For example, XML Test[1] is an XML processing test developed at Sun Microsystems [SMI09] aiming at comparing XML Processing Performance in Java and .NET. It is designed to mimic the processing that takes place in the lifecycle of an XML document. Typically that involves parsing (and possibly building an in-memory tree representation), accessing, modifying and serializing (converting the tree representation to a textual form that is written to a disk file) an XML document. This benchmark measures the throughput of a system, i.e. average number of XML transactions (complete lifecycle of an XML document) executed per second.

Concerning our system, the set of data we are interested on is composed by XML documents, with their XML schema(s); the set of operations concerns XQuery specification and processing using the QBE paradigm for XML. There are many Web available XML collections with different characteristics (e.g. MESH). Among them, we chose the appropriate ones for our case: it must include documents with various sizes or with rich XML Schema properties (e.g. variety of elements).

This paper is organized as follows. Section 2 presents an overview of XML Benchmarking approach and considerations. Section 3 introduce the query-by-example paradigm. Section 4 is a quick overview of GuessXQ, the XML query-by-example environment. GuessXQ architecture is also presented in this section. The requirements for a complete set of tests adequate to assess the effectiveness of GuessXQ, are identified and specified in Section 5. This section contains the motivation for the central piece of work here discussed, the XML documents repository XAT. Section 6 introduce the XAT design and the classification adopted. Section 7 concludes the paper with a summary, actual achievements and future work.

2 XML Benchmarking

In the last years, XML has undoubtedly reach a leading role among languages for data representation. Nowadays, we verify a significant increase on the number of techniques to work with XML; those technics mutually compete in speed, efficiency and memory requirements. So it is desirable to test them on different data sets to compare performance and allow an easier choice of the most suitable for each particular needs. Benchmarks are used for such a purpose. A benchmark, or a test suite, consists in a set of testing scenarios or test cases, i.e. data and related operations which enable one to compare versatility, efficiency or behavior of the *system under test* [VM09]. They should be simple, portable, scalable

[1] XML_Test_1.1 downloadable at `http://java.sun.com/performance/reference/codesamples/`

to different workload sizes, and allow the objective comparisons of competing systems and tools.[Gra93]

Nowadays, there exists a large set of XML query benchmarks. According to the work developed by Mlynkova and Vranec [VM09] the seven best known representatives are XMark [BCF+03], XOO7 [ea02], XMach-1 [BR02], MBench[ea06], XBench [YO03], XPathMark [Fra05] and TPoX [NKR+].

From the point of view of the type of data we can distinguish benchmarks which involve real-world data and benchmarks involving synthetic data [Mlý09]. According to a statical analysis of real XML data collections [MTP06]: they are usually very simple and do not cover all constructs allowed by XML specifications; contain a huge number of errors or are not well-formed; are often available without any associated schema. These characteristics imply that important features of the tools under assessment could not be covered by the test sets. This statement led some people to defend the use of synthetic data, like Mlynkova. Synthetic data is created by the benchmark generator; based on user-specific characteristics (amount of documents and size of the data), the generator produces documents covering as much as possible the various features of XML. These generators deal with marginal problems such as where to get the textual data or the elements/attributes name, to obtain data as natural as possible.

In opposition to Mlynkova approach, oriented towards synthetic data sets to build benchmarks, the work reported in this paper (on XAT design and implementation) follows a more pragmatic approach and is oriented towards real-world test collections. This is because: (1) we believe those collections better represent the data that most of the XML applications deal with; (2) the synthetic data is limited to the user-specific characteristics. However, real-world data does not have always the same and desired properties, as some times they have errors. This leads us to rank real-world XML documents concerning their validity (well-formed and compliant with the schema), complexity, diversity in the number and size of elements and the size of the document, in order to provide an easy way to find the real-data document that fits the user needs for his/her specific benchmark. Before describing the XAT, we introduce our GuessXQ system which motivated this benchmarking work.

3 Retrieving Information from XML document collections

Queries for XML retrieval allow the access to certain parts of documents based on content and structural restrictions. Examples of such queries are those defined by XPath [BBC+05] and XQuery [BCF+05], the standard proposed by W3C. These languages are expressive enough to allow to specify sophisticated structural and textual restrictions.

XQuery is formed by several kinds of expressions, including XPath location paths and `for..let..where..order by..return` (FLWOR) expressions based on typical database query languages, such as SQL. To pass information from one opera-

tor to another, variables are used. As an example, assume a document that stores information about articles, including title, author and publisher. The query bellow returns articles of author "Kevin" ordered by the respective title.

```
for $a in doc('articles.xml')/article
 where $a/author = 'Kevin'
 order by $a/title
return $a
```

XQuery operates in the abstract and logical structure of an XML document, rather than its surface syntax. The corresponding data model represents documents as trees where nodes can correspond to a document, an element, an attribute, a textual block, a namespace, a processing instruction or a comment. Each node has a unique identity.

However, structured queries construction is not always an easy process because, among other reasons, the user may not have a deep knowledge of the query language or of the documents collection structure. Moreover, after specifying a query, the user may get a final result that it is not what he expected. To solve these problems, many works are devoted to graphical user-friendly interfaces for query specification based on the "Query-by-example" paradigm, as will be discussed below.

Query-by-example for XML

Query-by-Example (QBE) is a language for querying, creating and modifying relational data [RG07]. It differs from SQL in terms of having a graphical user interface that allows users to write queries by creating example tables on the screen. A user needs minimal information to get started and the whole language contains few concepts. QBE is especially suited for queries that are not too complex and can be expressed in terms of a few relations. Through the years, the use of structured documents, like XML documents, in databases or as databases, led to an evolution of the QBE concept associated to XML retrieval.

Often we are interested in searching for particular information in a document, but the learning of a new language (the query language) can be a challenge. So, the idea of generating queries through an example seems the solution for this problem. Most of the works [BC05,NO04,LGB07] adapt the relational QBE model by showing the XML Schema Definition tree instead of the table skeleton. We developed a system, GuessXQ, that not only displays the XML Schema tree representation to the user, but also a sample document from the collection. Elements selection and restriction is done directly in the sample document, giving the user a concrete and realistic vision of the information he/she is searching for. Using the sample document interface, the user is able to specify two possible functionalities: selection and filtering. The selection corresponds to the return clause in XQuery, where the components to be retrieved are specified. The filtering corresponds to the where clause and allows to filter out components from

the result. In the user interface, the user may apply a filter to each component selection, thus generating a set of pairs <selection, filtering>, each one yielding a `for.. where.. return` query.

In the next section, we present GuessXQ interface and describe its architecture.

4 GuessXQ at a glance

GuessXQ system shows a sample document to the user to specify his query. Figure 1 depicts the interface showing the sample document where the user specifies the components (elements or attributes) he needs.

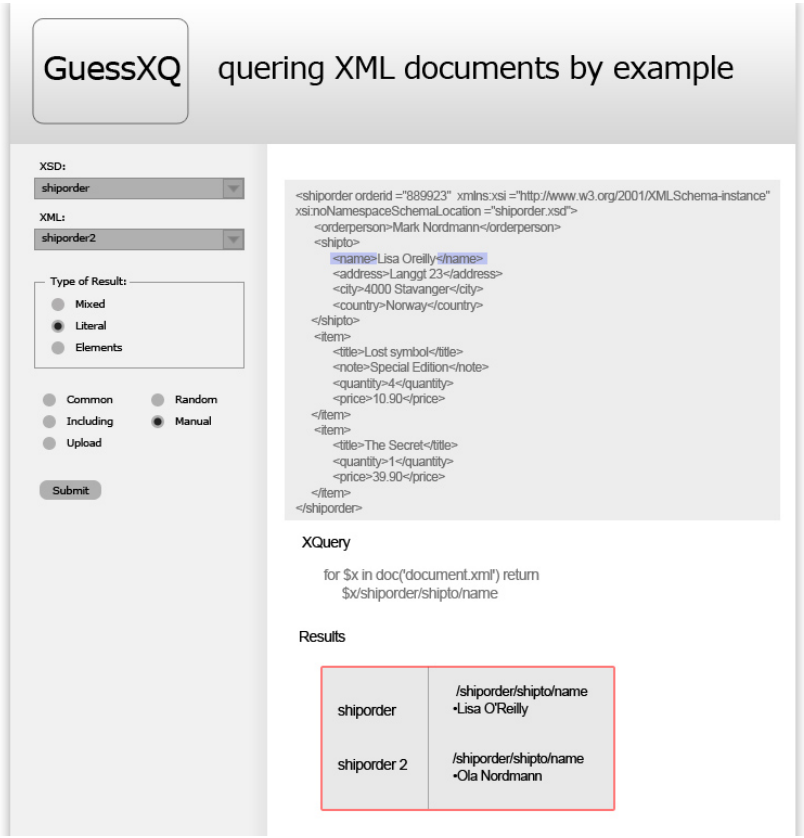


Figure1. GuessXQ sample document interface.

In this interface, we can see, on the left, options related to the XML Schema, the type of expected results (mixed, literal or elements) and the type of sample document choice (common, including, random or manual). The sample document

is the main frame of the interface. If the user selects, for example, element *name*, the corresponding generated XQuery query is

```
for $x in doc('document.xml')
return $x/shiporder/shipto/name
```

We describe next the metrics behind the sample document selection.

4.1 Sample document choice

The choice of the sampe document plays a major role in the user interaction with the search system. By now, the sample document is randomly picked from the collection. The user may also choose it manually. However, we are implementing a document choice component in the system based in the following metrics.

Document size: Big file sizes can slow down the system; also, smaller size files can contain too little information or elements to aid the user selection. This metric can be used as a delimiter to complement the others by not allowing a file bigger than a predefined size.

Number of elements/attributes: Taking into account the number of elements and attributes in the sample document is important. In one hand, if the file has too many components (elements or attributes), it can be too cluttered for the user to select his desired example. On the other hand, if the document has few components, it may not contain all those ones the user needs.

Number of different elements/attributes: To counteract some of the shortcoming of the previous metric, it may be interesting to look at the number of different elements and attributes in a file. This way, if a file contains almost all the elements and attributes present in the schema, the user gets a more complete variety of elements to specify his needs.

Diversity of Values: As stated before, the capacity of the user to see example data and not just the structure (schema) of the queried documents is the main innovation of our QBE approach. Therefore, a metric guaranteeing the diversity of data in the sample document is important. Having different values for the same element (or attribute) allows the user to better understand the fields in the document he is querying. However, similar to the other metrics, if there is too much diversity, the sample document may become too big.

As seen, each metric has its own merits and shortcomings, so they must be used together in a meaningfully way. The sample document should be diverse, which means that it must have a rich subset of the elements, attributes and possible values from the schema. However, it also must be a file contained in a predefined size. Therefore, we propose to use a combination of the 2nd, 3rd and 4th metrics restricted by a file size limitation (1st metric). We also intend to make a ranked list of possible sample documents, thus making easy for the user to retrieve the 'second best choice' when the previous document suggested by the QBE system is not suitable.

4.2 GuessXQ Architecture

Basically, the system has the architecture depicted in Figure 2.

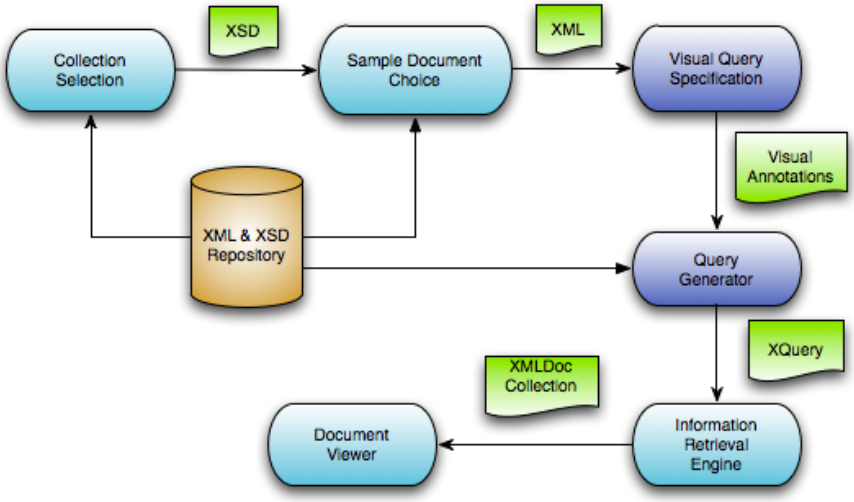


Figure2. GuessXQ system architecture.

After the *Sample Document Choice*, the selected sample document is shown to the user in an interface for *Visual Query Specification*. This interface corresponds to the one presented in Figure 2. The *Visual Annotations* made in the previous interface are, then, translated into XQuery by a *Query Generator*. The generated query is processed by an *Information Retrieval Engine* which searches in the documents collection for the components specified in the query. The returned components are given back to the user in a *Document Viewer* interface.

5 GuessXQ Assessment

In this section, we focus on the motivation for the central part of the work here discussed, the XML repository XAT. We also introduce the requirements for the complete set of tests that assess the effectiveness of GuessXQ.

5.1 The need for testing

As in every prototype, it is essential to assess the effectiveness of the system, to search and fix eventual system failures.
This need led us to do a pragmatical validation of the application performance,

and to the creation of a complete set of tests to assess the robustness of GuessXQ architecture. The analysis of the results meets the conditions to progress to the next level of the prototype and implement new algorithms. Thus, it is important to apply the concept to XML documents that represent real situations, to prove the GuessXQ usefulness and efficiency.

5.2 Requirements for the set of tests

As described in section 4, the logic under the choice of the sample document (conforming to the selected schema) is a focal point in our approach to QBE. Thus, the need to test the combination of metrics chosen (see subsection 4.1), led us to discuss the main question that motivates the XAT: which requirements are important to test, to ensure the effectiveness of GuessXQ system?
Whereas each metric has its own merits and shortcomings, the sample document should be diverse and have a rich subset of elements, attributes and values from the schema. It also must consider the size limitation, the deepness[2] and its syntactic and semantic correction. Thus, we decided to test the behavior of the prototype in three main directions:

- its performance working with files with different sizes;
- its ability for dealing with malformed XML files;
- its ability to process XML files that have a great diversity of elements, attributes, text data or complex deepness.

Concerning the first direction, the aim is to compare the execution time for large or small files. For this comparison, it is very important to collect as many information as possible to ensure that the system is able to work with large files in a reasonable time. Thus, it is important, not only to focus the analysis in the runtime, but also in the capacity of displaying the entire XML document to the user and generate a correct result.

Concerning the second direction, our tests help to model the behavior of the application when facing an unexpected input. The aim is to guarantee that GuessXQ prototype is able to recognize malformed XML files, and prevent a system failure when processing it.

Finally, concerning the third direction, the objective is to test the system response facing the diversity of information that can be present in a XML file. It ensures the reliability of the output produced by the GuessXQ prototype. If ok, it certifies GuessXQ main purpose: the information retrieval from structured documents with the query-by-example assistance.

Consequently, we had to search and store a large number of files that fits those criteria, in order to perform the required tests. This need led us to order and rank the documents to easily find which one fits each set of tests. So, we create a repository to neatly archive the files needed not only to test GuessXQ system, but also general XML applications, as described in the next section.

[2] We consider a complex deepness when a file has a depth superior to seven levels.

6 XAT, a XML Archive for Testing

The requirements described along the previous section enabled us to draw what should be a set of adequate XML documents for GuessXQ testing. From practical considerations rose the idea of creating a repository to archive those files. Generality criteria guided the design of a repository able to test in-house XML applications for performance and correction. There were however some open issues for us to answer: *what do we require, syntactically and semantically, from a document in order to test performance or correction*; *can we gather a set of documents that meet all the requirements and provide a consistent set of results and, along with a well defined XML benchmarking process, would effectively help in systemizing the testing of XML based applications.*

6.1 The XML collection

To build our XML collection, we searched for XML query benchmarks which provide a set of testing XML data collections and respective XML operations that are publicly available and well described. What follows is the list of available XML document collections we use:

- MESH[3], Medical Subject Headings vocabulary files.
- Shakespeare[4], the complete plays of Shakespeare.
- Eurovoc[5], a multilingual thesaurus covering the fields in which the European communities are active; it provides a means of indexing the documents in the documentation systems of the European institutions and of their users.
- Miscellaneous (Religion[6], etc.), a set of diverse documents collected from different sources.

At this point we should note that the archive is not restricted to these collections, as a work in progress it is in constant growth.

6.2 Classification

In our effort to classify XML documents for testing purposes we distinguished two subclasses: Correct Documents (i.e. documents well-formed and valid according to their DTD, or Schema); and Incorrect Documents (not valid XML documents). Each subclass has it is own criterion and a precise purpose in terms of testing, as it is discussed below.

[3] `http://www.nlm.nih.gov/mesh/xmlmesh.html`.
[4] `http://xml.coverpages.org/bosakShakespeare200.html`.
[5] `http://europa.eu/eurovoc`.
[6] `http://www.ibiblio.org/pub/sun-info/standards/xml/eg/`

Correct All documents under this class have an attached DTD or Schema for validation; all of them are syntactically and semantically correct. This class is crucial to test the performance of tools as well as particular features dependent on different characteristics of annotated documents; thus it is split into two other subclasses. Many documents falling in this category come from the MESH collection

Performance In order to access the requirements of XAT for performance testing we have to look into the technologies commonly used in the development of XML applications, and how they stack up against each other. With this we should have a clear understanding of what is needed to test them.

XML parsing techniques fall into two known methodologies, tree-based and event-based parsers. These two methodologies diverge primarily in the computer representation of the XML document. The two most proliferated parser types are DOM and SAX. DOM based parsers construct an in-memory tree representation of the document, where as SAX are event-driven, and do not create in-memory representations. More recently a new type of parser is rising up. A StAX (Streaming API for XML), like SAX, is a parser independent pure Java API based on interfaces that can be implemented by multiple parsers. It takes a mix of an event-based and tree-based approaches for working with XML data in Java. However, unlike its predecessors, StAX uses XML Streams to access the data. This mixed approach enables the parser to pulling data as it needs, even in large documents. This feature gives to application the full control of the parser, giving to StAX a very clear performance advantage.

We can assume that all applications that deal with XML documents are built on top of these three techniques. In general, the bigger the file the more resources it will take to be processed. This will affect in different ways applications built according to these three parsing techniques. Event-driven parsers (SAX) are progressive; they read a file sequentially, and thus performance will be proportional to the document size, even though some event-based parser applications are required to have some sort of intermediate internal representation. In DOM based parsers, we have to take into account the cost of building and storing the object tree in memory. StAX parsers, not suffering from the same limitations, can vary widely from implementation to implementation.

With the emphasis in document size, we have defined five categories:

- C1: from 0k to 10k
- C2: from 10k to 100k
- C3: from 100k to 1MB
- C4: from 1MB to 1GB
- C5: higher than 1GB

This division was based on common sense, our experience, and tests with different parsers. For each parser tested, we determined the document size above which its behavior is affected.

Many documents falling in this category come from the Shakespeare collection.

Diversity A XML document may be rich in diversity, i.e. in the number and size of elements, attributes, element depth and PCDATA. To prove that an application is working correctly, we can test it against documents that have been inspected for specific diversity values. If we have inspected the previous files with an application that it is known to be correct we would have a base of comparison, simplifying a shore that becomes tedious, exhaustive and not bullet proof, when performed by humans. Our investigation led us to the creation of such a tool, a simple script that counts all the diversity elements. We further extended this idea by creating information files: for each correct document in the archive, the script outputs a info file. This info file is currently composed of a simple format, and contains the number of elements, attributes, PCDATA as well as maximum element depth. Many documents falling in this category come from the MESH collection.

Incorrect To assess, or benchmark, completely document processing tools, it is also mandatory to prove that they cope adequately with erroneous inputs (it is necessary to verify that they detect and support invalid documents). So it was strictly necessary that the archive also includes incorrect documents. As previously said, we consider incorrect documents those that:

- Do not obey to the XML well-formed specification.
- Do not conform to a given Schema.

It should be noticed that the XML specification, though simple, is large enough to allow a wide variety of errors on a given document. For effective testing, each document in this class should have one and only one error. For identification purposes, the document filename directly reflects the type of error. This feature should help on indexing the archive, making it easy to narrow down searches. A more complete description of the malformed can be found inside the document, as a comment.

7 Conclusion

The work we present in this paper describes how we managed to build an XML benchmark adapted to a system we are developing called GuessXQ. For such purpose, we created a generic XML archive, XAT composed by documents with different characteristics split into two groups, correct (performance and diversity) and incorrect. GuessXQ allows for XML access in a QBE fashion: the user specifies his queries by selecting the desired components in a sample document view.

The benchmark we use is formed by documents coming from different Web available collections. The tests done measured the system capability for document processing.

As future work, we intend to make XAT available to public via a web browser. For this purpose we are developing an adequate interface.

References

[BBC+05] A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. Xml path language (xpath) 2.0 w3c working draft. `http//www.w3c.org/xpath20/`, 2005.

[BC05] Daniele Braga and Alessandro Campi. Xqbe: A graphical environment to query xml data. *World Wide Web*, 8(3):287–316, 2005.

[BCF+03] R. Busse, M. Carey, D. Florescu, M. Kersten, I. Manolescu, A. Schmidt, and F. Waas. Xmark - an xml benchmark project. `http://www.xml-benchmark.org`, 2003.

[BCF+05] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. w3c working draft. `http//www.w3c.org/TR/xquery/`, 2005.

[BR02] T. Bohme and E. Rahm. Xmach-1: A benchmark for xml data management. `http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html`, 2002.

[ea02] S. Bressan et al. `http://www.comp.nus.edu.sg/~ebh/XOO7.html`, 2002.

[ea06] K. Runapongsa et al. The michigan benchmark. `http://www.eecs.umich.edu/db/mbench/`, 2006.

[Fra05] M. Franceschet. Xpathmark. `http://users.dimi.uniud.it/~massimo.franceschet/xpathmark/`, 2005.

[Gra93] Jim Gray. *The Benchmark Handbook: for Database and Transaction Systems*. Morgan Kaufmann Pub, San Francisco CA USA, 2nd edition, 1993.

[LGB07] X. Li1, J. H. Gennari1, and J. F. Brinkley. Xgi: A graphical interface for xquery creation. In *Proceedings of the American Medical Informatics Association Anual Symposium*, pages 453–457. American Medical Informatics Association, 2007.

[Mlý09] Irena Mlýnková. *XML Benchmarking - the State of the Art and Possible Enhancements*. Idea Group Publishing, April 2009.

[MTP06] I. Mlynkova, K. Thoman, and J. Pokorny. Statistical analysis of real xml data collections. *COMAD'06: Proceedings of the 13th International Conference on Management of Data*, pages 20 – 31, 2006.

[NKR+] M. Nicola, I. Kogan, R. Raghu, A. Gonzalez, M. Liu, B. Schiefer, and G. Xie. Transaction processing over xml (tpox). `http://tpox.sourceforge.net`.

[NO04] Scott Newman and Z. Meral Ozsoyoglu. A tree-structured query interface for querying semi-structured data. *Scientific and Statistical Database Management, International Conference on*, 0:127, 2004.

[RG07] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*, chapter 6 - Query-by-Example (QBE). 2007.

[SMI09] The XML Performance Team Sun Microsystems Inc. Xml processing performance in java and .net. `http://java.sun.com/performance/reference/whitepapers/XML_Test-1_0.pdf`, 2009.

[VM09] Maros Vranec and Irena Mlýnková. Flexbench: A flexible xml query benchmark. In *DASFAA*, pages 421–435, 2009.

[YO03] B.B Yao and M. T. Ozsu. Xbench - a family of benchmarks for xml dbmss. `http://se.uwaterloo.ca/~ddbms/projects/xbench/`, 2003.

Integrating SVG and SMIL in DAISY DTB production to enhance the contents accessibility in the Open Library for Higher Education - Discussions and Conclusions

Bruno Giesteira[1], Inês Gomes[2], Alice Ribeiro[3], Diamantino Freitas[4]

[1] University of Porto - School of Fine Arts,
Av. Rodrigues de Freitas, 265
4049-021 Porto, Portugal, bgiesteira@fba.up.pt
[2] University of Porto – Faculty of Arts,
Via Panorâmica, s/n,
4150-564 Porto, Portugal, inesdgomes@gmail.com
[3]University of Porto – Faculty of Arts,
Via Panorâmica, s/n,
4150-564 Porto, Portugal, malice@letras.up.pt
[4]University of Porto – Faculty of Engineering,
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal, dfreitas@fe.up.pt

Abstract: Following BAES[1] investment in the use of DAISY[2] DTB[3], it has become notorious the lack of authoring tools and user agents based in the most recent updates of the DAISY standard. The software in use nowadays does not allow the integration of SVG[4] and SMIL[5] files, two formats that can improve DTB, making contents richer and more accessible. It would be an important step for Portuguese Higher Education Institutions to be able to produce DTB with SVG images and SMIL videos.

Keywords: Accessibility, DTB, DAISY, Authoring Tools, User Agents, Library

1. Introduction

According to the United Nations, one out of ten people in the world, 650 millions, have some kind of disability. 80% of those live in developing countries.[6] People with disabilities have less than 50% chances of reaching third level education when

[1] BAES is the acronym for *Biblioteca Aberta do Ensino Superior*, which means Open Library for Higher Education.
[2] DAISY is the acronym for Digital Accessible Information System.
[3] DTB is the acronym for Digital Talking Book.
[4] SVG is the acronym for Scalable Vector Graphic.
[5] SMIL is the acronym for Synchronized Multimedia Integration Language.
[6] *Some Facts about Persons with Disabilities*:
http://www.un.org/disabilities/convention/facts.shtml

compared with people with no disabilities.[7] These figures are too important to be ignored and require a policy that assures conditions for these people to feel fully integrated in society

The populations' aging tendency, which is more significant in first world countries, is also an alert to the necessity of adapting these technologies to the physical challenges that aging requires. A recent Eurostat study predicted that the percentage of people with more than 65 years old will raise from 17,1% to 30%, which corresponds to an increase of 84.6 millions in 2008 to 151.1 millions in 2060[8].

Nowadays, the Web accessibility is of the outmost importance to access information and is a major factor in social inclusion. Nevertheless, the Web is often an interdicted place to people with print disabilities. As the importance of the Web grows in fields such as education, governmental information, civic participation, and entertainment, the concerns about accessibility should increase as well.

"*Making the Web Accessible can dramatically improve people's lives and benefit society as a whole*" (THATCHER, BURKS, & al, 2006). That is why accessibility is firstly a social and cultural issue and secondly a technological one. When developing accessible websites, we are not only giving equal access but also equal opportunities to everyone (THATCHER, BURKS, & al, 2006).

1.1. Open Library for Higher Education: BAES

BAES has significant role in the context of learning and education. It was launched on the 3rd December of 2007 as a partnership between nine institutions of Higher Education. In spite of still being a developing structure, BAES has already around 3000 books in Braille, audio and full text. It aims to democratize the access to college education, giving online information in accessible formats to students with print disabilities

University of Porto (UP) concerns with students disabilities started in 2001, when a set of guidelines on evaluation and class attendance for students with special education needs was approved, adopted later by each faculty of UP.

In 2004, GTAEDES[9] started it's activity. This work group is an assembly of several Portuguese Higher Education institutions. This group has also the collaboration of the General Direction of Higher Education, UMIC – Agency for the Society of Knowledge and the National Institute for Rehabilitation.

In July 2006, the former group carried out a survey of students with special education needs in the Portuguese Higher Education Institutions. The results of this survey reinforced the goals of BAES and the need for its further development.

---

[7] *Facts and figures about disability*: http://www.edf-feph.org/Page_Generale.asp?DocID=12534

[8] *Ageing characterizes the demographic perspectives of the European societies:* http://epp.eurostat.ec.europa.eu/portal/page/portal/product_details/publication?p_product_code=KS-SF-08-072

[9] GTAEDES is the acronym for *Grupo de Trabalho para o Apoio a Estudantes com Deficiência no Ensino Superior*, which means Work Group for Support of Students with Disabilities in Higher Education

In 2008 a policy regarding the rights and duties of students with special education needs was approved. [10] by the University of Porto.

1.2 Formats used nowadays in BAES

The content digitalization is an advantage to accessibility, since it allows the information to be accessed through screen readers and Braille displays. In order to do that it is necessary for documents to be well structured. BAES defines some rules for the production of documents in digital support, so that the transformation of information is done in an exact, correct and coherent way, respecting the rules of accessibility.

Most contents created by BAES are PDF[11] and RTF[12]. Although these formats have some potential when it comes to accessibility, many of this kind of documents that we find on the Web, are not accessible since they do not have a navigable structure. BAES on the other hand tries to fight this tendency by providing structured documents that allow users to access them through alternative technology.

The PDF has a lot of advantages over other formats for various reasons: the document's creation is simple; sharing is easy for in order to read the document all you need is the free software of Adobe Reader; the format preserves the document's original layout; and it provides safety for the content since it cannot be altered.

In terms of accessibility, PDF offers, as it has been said before, some possibilities. Like HTML, PDF allows the definition of tags on the document's structure, vital for the use of screen readers and other assistive technologies. Moreover, PDF allows the user to change the colours, resize the document and search words within the file.

To make PDF accessible, the easiest way is to use Microsoft Word 2000, XP, 2003 or 2007 that allows tagging. On the other hand, if a document is digitalized through a scan, a screen reader will hardly read it, since text no longer exists in its structure. The file will simply be a PDF Image Only File.

However this type of documents is not enough to meet the needs of students that can't read printed information. The goal must be to create reading experiences close to those you get from reading a printed book. It is essential to follow the latest advances in the accessibility area. Therefore the producers of accessible contents cannot ignore the potential of DTB.

1.2. The beginning of DTB production by BAES

DTB represents a giant step to information access for people with print disabilities. Comparing to analogical talking books, the advantages are notorious. When referring

[10] http://sigarra.up.pt/up/conteudos_geral.conteudos_ver?pct_pag_id=122231&pct_parametros=p_pagina=122231&pct_disciplina=&pct_grupo=947#947

[11] PDF is the acronym for Portable Document Format.

[12] RTF is the acronym for Rich Text Format.

to DTB, it is also important to refer to DAISY, a consortium responsible for the creation of the format that defines the DTB structure.

In light of DAISY's advantages over PDF, RTF or even HTML[13] files, UP, through SAED[14], is now betting on the development of DTB based on this standard. This initiative consisted first of all in exploring the format, attempting to optimize it using SVG and SMIL standards and integrating them in DAISY, incorporating also video. This goal met with some obstacles, that shall be mentioned later in this article

Related Work: DAISY

The most recent DAISY update is called *DAISY/NISO[15] 2005 standard, Specifications for the Digital Talking Book*[16], also known as DAISY 3 or ANSI[17]/NISO Z39.86. DAISY DTB is a set of electronic files organized in a way that presents information through alternative media. Today, as far as the DAISY standard is concerned, DTB is clearly a conquest over the traditional audio books. With DTB, the navigation through the contents is simpler and more effective. The user can move from sentence to sentence, from chapter to chapter, from page to page. The navigation is as flexible as the mark-up structure of the document is more refined.

Therefore, unlike analogical audio books, the reading does not have to be linear. Besides, the interactivity of the user with the content is improved: the XML[18] allows the word searching and highlighting or making annotations on DTB, without affecting the original structure. The user may even have access to spelled words.

Also DTB can include text, images and animations simultaneously, something that would never be possible in an analogical audio book. Although images and animations cannot be accessed for instance by blind people, they are often very important for the comprehension of the whole. When the creator provides a textual description of these elements (through the attribute *alt* and *longdesc*), a user with print disabilities may also access them.

DAISY Consortium activities have always been developed in strict collaboration with W3C[19] and ISO[20], in order to let the standard have world wide recognition, therefore improving the information access by people with print disabilities. At some point, the adoption of this format by the contents' creators will eventually benefit the whole population. This is W3C-WAI[21] mission.

Proof of that partnership with W3C is the adoption of XML, XHTML[22] and SMIL (which supports the multimedia side of DAISY) as DAISY's basics. In fact, the

[13] HTML is the acronym for HyperText Markup Language.
[14] SAED is the acronym for Serviço de Apoio ao Estudante com Deficiência, meaning Disability Support Service.
[15] ANSI is the acronym for American National Standards Institute.
[16] *Specifications for the Digital Talking Book*: http://www.daisy.org/z3986/2005/Z3986-2005.html
[17] NISO is the acronym for National Information Standards Organization.
[18] XML is the acronym for extensible Markup Language.
[19] W3C is the acronym for World Wide Web Consortium.
[20] ISO is the acronym for International Organization for Standardization.
[21] WAI is the acronym for Web Accessibility Initiative.
[22] XHTML is the acronym for eXtensible HyperText Markup Language.

potential in accessibility that DAISY represents is mostly due to XML. It is this markup language that enables a reading experience closer to the reading of a print book. DAISY has specific tags that represent certain DTB structures. XML allows navigation between those tags, therefore allowing flexibility not possible in an analogical audio book or even in a digital audio book, which does not support XML.

2. Related Work: SVG and SMIL

SVG allows text to represent graphic information, combining XML potentialities with vector design. Describing an image as a series of geometric forms or mathematic vectors, the vector design allows it to be described through text, making the result an interpretation of an instructions' set.

One of the biggest advantages of the vector images over the raster images is the possibility of interactivity as it allows resizing it without any loss in quality. When a raster image is enlarged, one sees only dots of colour (pixels) that, if successively increased, will eventually lose its shape. If, on the other hand, we amplify a vector image, we will be able to see geometric drawing in detail, but its shape will never be lost. This possibility of resizing a SVG image as much as we want is especially important for people with low vision or also in contexts where the displays are very small. In the following hyperlink there is an example of an SVG image created by the authors. To understand the SVG potential, you must zoom the image: http://www.giesteira.net/XATA2010/SVG-Example.svg

If the vector image obeys a set of instructions that can be represented in text, that means that if those instructions are altered, the same will happen to the final result, which is the same as saying that the practical interpretation of the instructions will be modified. A vector image can be changed without making any direct intervention in the design, but only in the instructions that guide it. As far as accessibility is concerned, this possibility means a huge potential. For instance, a person with a visual disability may manipulate an image interfering only in the text that represents it. So, this user, who was once limited, has now the chance to create and edit images with a simple text editor.

Regarding SMIL, this is a format that enables the integration of a series of elements as text, sound, video and other graphics, in a multimedia synchronized presentation. SMIL, just as SVG, is a language created from XML. SMIL allows defining presentations layout, setting the spatial (X-axis and Y-axis) and temporal position of the different elements.

SMIL's function is to gather a set of independent files in a single presentation, which means the isolated elements in the server obey the SMIL's file instructions. They are brought together in a final product activated by SMIL player (NIEDERST, 2001). The original files remain unaltered, this way they can be reused in other multimedia presentations. The elements may also be easily replaced by modifying the text in the SMIL file.

The problem with the usual multimedia contents is the unified integration of several elements. If the different elements are initially different layers (sound, video, text and isolated graphics), the final result is a single and unified content.

Text/sound/graphics/video will no longer be separately accessed (ARCINIEGAS, 2002) – that's what happens with Adobe Premiere or Adobe Flash.

For instance, two videos can be integrated in a multimedia presentation, as it is shown in this hyperlink: http://www.giesteira.net/XATA2010/lgp_sinc.smil (this file was created by Jorge Fernandes, from UMIC's team). However, depending on the user's will, they can be accessed separetly. A deaf person can see the video with only sign language. Also, the SMIL file can integrate subtitles, like in the example in the following hyperlink: http://www.giesteira.net/XATA2010/caption_mac.smil (this file was also created by Jorge Fernandes, from UMIC's team). For a blind person, this is of an extreme importance, since the subtitles can be accessed using an text editor that allows the use of a screen reader.

In a time where multimedia contents become more and more important, SMIL comes as an answer to the accessibility problem that many of these contents represent. It is precisely this possibility to access separately different elements that enhances SMIL accessibility.

Multimedia files developers should guarantee its accessibility; therefore they should provide information in an alternative format, in such a way that is also synchronized with the rest of the elements. Only then an intelligible result for any kind of user can be assured.

4. SVG and SMIL integration in DAISY

4.1 DAISY: DTB Structure

These are the files that constitute a DTB, according to DAISY 3.0[23]:

1. *Package File*: a XML file, with OPF[24] extension, which describes DTB information and of all of its files. This is usually the main file during the playback. These are the elements of the XML file:
    - *metadata:* information about the DTB;
    - *manifest*: files list of the DTB,
    - *spine*: indicates the DTB SMIL files in a reading linear order;
    - *tours* and *guide:* optional elements.
2. *Textual Content File*: a XML file with the textual content, structured according to dtbook.dtd[25]. When the book is constituted only by audio, this file does not exist. An important element of this file is the element <level>. Levels describe the relative position of the text in the global book structure. There are only six level tags: <level1>…<level6>. The <level1> corresponds to the book's higher level; therefore <level 6> is the lowest level.

[23] *Specifications for the Digital Talking Book*: http://www.daisy.org/z3986/2005/Z3986-2005.html

[24] OPF is the acronym for Open Packaging Format.

[25] *DAISY/NISO Standard DTD and CSS files:* http://www.daisy.org/daisyniso-standard-dtd-and-css-files

3. *Audio files*: DAISY supports common formats such as WAV[26], MP3[27] and MP4[28].
4. *Image files*: JPEG[29], PNG[30] and SVG are supported.
5. *Multimedia Synchronization Files:* SMIL files that establishes the relation between text, sound and images, defining the presentation timing and order. SMIL allows the parallel and synchronized presentation of the different objects. The latest SMIL update refers to DAISY Profile, a SMIL adaptation to the needs of DAISY. SMIL 3.0 DAISY Profile[31] is a Strict Host-Language Conformant SMIL 3.0 Profile. In fact, SMIL as a whole is too complex for using in DTB; that is why it was created a specific profile to be used in DAISY.
6. *Control Navigation Files:* the NCX[32] file defines the DTB's navigation through specific items. These NCX items are connected by the SMIL files to the specific locations in the audio and textual content files. NCX establishes the main file structures. NCX enables the user to navigate through page to page, heading to heading and so on. When having a textual content file in the DTB, NCX must respect that XML files structure.

There may be also other kind of files such as bookmark or highlight; resource files; distribution information file; and presentation styles.

4.2 Experiences to integrate SVG and SMIL video in DAISY

In order to produce more complete contents for the University of Porto, we have tried to integrate SVG and SMIL in DAISY. For that purpose we used tools for DAISY DTB creation (authoring tools).

Dolphin Publisher is one of the main production tools of digital talking books based on DAISY standard. This software makes the DTB creation easier and more intuitive. By importing a text document and by defining the levels of reading through the creation of headings and other tags, a structure of the document can be established. Dolphin Publisher enables the association of a human voice to text. This is an important advantage over text synthesizers like for example JAWS.

[26] WAV stands for Waveform Audio Format.
[27] MP3 stands for MPEG-1 Audio Layer 3. MPEG is the acronym for Moving Pictures Experts Group.
[28] MP4 is the acronym for MPEG-4 Part 14.
[29] JPEG is the acronym for Joint Photographic Experts Group.
[30] PNG is the acronym for Portable Network Graphics.
[31] W3C, *SMIL 3.0 DAISY Profile:* http://www.w3.org/TR/SMIL/smil-daisy-profile.html
[32] NCX is the acronym Navigation Control File for XML.

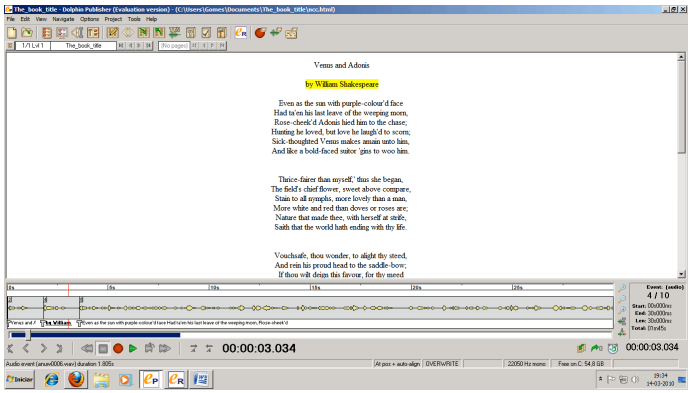


Image 1: Example of a DTB's creation in Dolphin Publisher

Each yellow selection in Dolphin Publisher is a text event, which is the smallest part of the DTB. Creating text events allows the user to navigate from event to event, as we will see further.

DAISY Pipeline is also a DAISY producer, but unlike Dolphin Publisher, this one is open source. Once we have the DTB XML, Pipeline can be used to create DAISY books in an automatic way.

However, none of these authoring tools allows the integration of SVG or SMIL files. Given this lack of appropriate software to reach our goals, we decided to use a simple text editor to integrate these two standards in DAISY.

In order to do that, we tried to use the elements <embed> and <object> in the textual content file, which is, as we have seen, a XML file with the textual content, structured according to dtbook.dtd. To insert the SVG file, we also tried the element <img>.

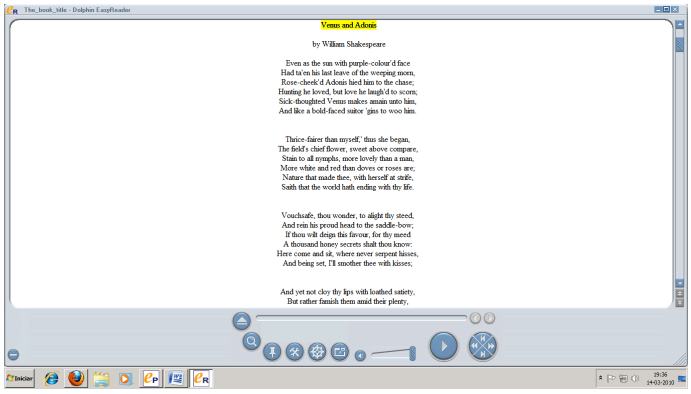


Image 2: Example of the process of reading a DTB in Easy Reader

But, even if the DAISY reader detects some object, it cannot read it, which means that the user agents (Dolphin Easy Reader and AMIS[33]) are not ready to read those formats.

Therefore, creating DAISY contents through a text editor did not solve the lack of software problem.

However, our will to produce better contents for BAES remained. To overcome this problem, we have inserted hyperlinks in the DTB so they would lead the reader to SVG and SMIL files. The following URL shows an example of a DTB with hyperlinks to those SVG and SMIL contents. You should download the file "DTB_Prototype.zip" and open the document "DTB_Prototype.html" with the user agents Easy Reader or Amis: www.giesteira.net/XATA2010/DTB

Although this solution is not yet totally satisfactory, because of the lack of efficiency (usability) for the reader who has to use several windows in order to access all types of documents, that is, in the current development state of the Daisy format, the best way to promote the access to multimedia contents like SVG images and SMIL videos, very useful for blind and deaf people, for people with low vision and/or in bad environment contexts.

Hyperlinks are identified by vision, since they are usually underlined and blue. For that reason blind people cannot identify hyperlink in the text. In order to circumvent that gap, we used synchronized voice giving that information. The drawback of this strategy is its interference with the rhythm and prosody of the text.

Moreover, reading SVG and SMIL files is not a simple task, when one has not the appropriate software. To read SVG, it is required to have installed the Adobe SVG Viewer, a plug-in that allows the reading of SVG by Internet Explorer, Mozilla Firefox, Opera and Safari.

In what SMIL is concerned, although it is recommended by W3C the use of the RealPlayer and QuickTime, only the last one (version 7) has worked in our tests.

4. Conclusion: Current lack of adequate Authoring Tools and User Agents to develop and access DAISY multimodal contents

The creation of authoring tools and user agents that follow the most recent updates of the standards is of the outmost importance. DAISY 3 has been already published in 2003 and it has been updated in 2005. In 2008, the SMIL standard update has also mentioned the SMIL DAISY Profile which, among other things, enabled the video integration in DAISY. This year, according to DAISY Consortium, it will be released DAISY 4.0, also known as ZedNext[34].

In light of all this, we have reached the conclusion that the basics for the development of richer and more complete multisensory content for people with disabilities have been established. It is now time to invest in the development of the necessary tools that can increase the capacity to create contents for a larger group of people.

---

[33] AMIS is the acronym for Adaptive Multimedia Information System.

[34] ZedNext Home Page: http://www.daisy.org/zw/ZedNext_Home_Page

DAISY, combined with images and videos, respectively, in SVG and SMIL formats, comes as an answer to the special needs of some students that have not yet been completely satisfied in UP and in other Portuguese Higher Education institutions. It is, therefore, important to make people aware of the present state of the art and to mobilize efforts and researchers towards this issue, so that education may become more accessible and personalized for everyone.

References

1. ARCINIEGAS, Fabio, *A Realist's SMIL Manifesto*, 2002. Retrieved from: http://www.xml.com/pub/a/2002/05/29/smil.html?page=1 [12-01-10]
2. EISENBERG, J. David, *SVG Essentials*, United States of America, O'Reilley Media, 2002, ISBN: 0-596-00223-8
3. LEAS, Dennis, PERSOON, Emilia et al. *Daisy 3: A Standard for Accessible Multimedia Books*, IEEE Computer Society, 2008, ISSN 1070-986X
4. NIEDERST, Jennifer, *Webdesign in a Nutshell*, O'Reilley and Associates Inc., United States of America, 2001, ISBN: 0-596-00987-9
5. STORY, Derrick, *Essentials Graphics with SVG*, 2000. Retrieved from:http://www.oreillynet.com/pub/a/network/2000/04/28/feature/svg.html [15-01-10]
6. THATCHER, Jim, BURKS, R. Michael & al, *Web Accessibility, Web Standards and Regulatory Compliance*, Apress Company, United States of America, 2006, ISBN: 1-59059-638-2
7. W3C, *Accessibility Features of CSS*. Retrieved from: http://www.w3.org/TR/CSS-access [10-01-10]
8. W3C, *Accessibility Features of SVG*. Retrieved from: http://www.w3.org/TR/SVG-access/ [10-01-10]
9. W3C, *Accessibility Features of SMIL*. Retrieved from: http://www.w3.org/TR/SMIL-access/ [10-01-10]
10. W3C, *Authoring Tool Accessibility Guidelines (ATAG) 2.0*. Retrieved from: http://www.w3.org/TR/ATAG20/ [02-02-10]
11. W3C, *XML Essentials*. Retrieved from: http://www.w3.org/standards/xml/core [30-11-09]
12. W3C, *Extensible Markup Language (XML): W3C Working Draft.* Retrieved from:http://www.w3.org/TR/WD-xml-961114.html#secA [01-12-09]
13. W3C, *Introduction to Web Accessibility*. Retrieved from: http://www.w3.org/WAI/intro/accessibility [27-02-10]
14. W3C, *Specifications for the Digital Talking Book*. Retrieved from: http://www.daisy.org/z3986/2005/Z3986-2005.html [09-10-01]
15. W3C, *SMIL 3.0 DAISY Profile.* Retrieved from: http://www.w3.org/TR/SMIL/smil-daisy-profile.html [27-02-10]
16. W3C, *WAI Mission and Organization*. Retrieved from: http://www.w3.org/WAI/about.html [15-01-10]
17. W3C, *SMIL 3.0 DAISY Profile*. Retrieved from: http://www.w3.org/TR/SMIL/smil-daisy-profile.html [20-01-10]
18. W3C, *User Agent Accessibility Guidelines (UAAG) 2.0*. Retrieved from: http://www.w3.org/TR/UAAG20/ [02-02-10]
19. W3C, *Web Content Accessibility Guidelines (WCAG) 2.0*. Retrieved from: http://www.w3.org/TR/WCAG20/ [15-01-10]
20. W3C, *Synchronized Multimedia Integration Language (SMIL 3.0)*. Retrieved from: http://www.w3.org/TR/SMIL3/ [20-01-10]

A Semantic Representation of Users Emotions when Watching Videos

Eva Oliveira[1], Teresa Chambel[2], Nuno Ribeiro[3]

[1] LaSIGE, University of Lisbon FCUL, 1749-016 Lisbon, Portugal,
IPCA, 4750-117 Arcozelo BCL, Portugal
+351 2175000533
eoliveira@ipca.pt
[2] LaSIGE, University of Lisbon FCUL, 1749-016 Lisbon, Portugal,
+351 2175000533
tc@di.fc.ul.pt
[3] CEREM – Centro de Estudos e Recursos Multimediáticos,
Universidade Fernando Pessoa
Praça 9 de Abril, 349, 4249-004 Porto - Portugal
nribeiro@ufp.edu.pt

Abstract. One of the greatest strengths of video is its power to provoke emotions and induce states of mind, and is an excellent tool for displaying affective information. A need for appropriate tools to allow the characterization of video scenes with information about induced and expressed emotions is becoming more and more evident. Having video emotionally classified will enable video access and search through emotions, and it will also enable users to find interesting affective information in unknown or unseen videos. In this paper, we propose a set of semantic descriptors based on both user physiological signals, captured while watching videos, and on video low-level features extraction. These descriptors based on XML contribute to the creation of automatic affective meta-information that will not only enhance a video recommendation system based in emotional information, but also enhance search retrieval of videos affective content from both users classification and content classification.

Keywords: Video affective classification, user physiological signals, domain specific xml schema, video access, video search, video recommendation.

1 Introduction

The usage of video has been steadily increasing and the tendency is to grow even more due to the increasing number of uploaded videos on popular online video sites, which are added in a daily basis. In fact, video is emotionally rich due to its natural way of expressing affective content and its ability to stimulate emotional changes in its viewers. It is clear that there is a demand for new tools that enable automatic annotation and labeling of videos. In addition, the improvement of new techniques for

gathering emotional information about videos, be it through content analysis or user implicit feedback, through user physiological signals, is revealing a set of new ways for exploring emotional information in videos, films or TV series. In fact, gathering emotional information in this context brings out new perspectives to personalize user information by creating emotional profiles for both users and videos. In a collaborative web environment, the collection of users profiles and video profiles has the potential to: (a) empower the discovery of interesting emotional information in unknown or unseen movies, (b) compare reactions to the same movies among other users, (c) compare directors' intentions with the effective impact on users, and (d) analyze, over time, our own reactions or directors' tendencies. The work described in this paper will first give an overall perspective on video classification techniques from both the content analysis perspective and from the perspective of user implicit classification techniques. Then, we review semantic approaches for creating emotional descriptions and describe some of the major problems regarding the semantic description of emotions. We then propose a semantic description of emotion oriented towards the user experience when watching videos, considering the user implicit assessment (by acquiring physiological signals), the user explicit assessment, and video content analysis. We also consider how such a semantic description may enhance the exchange of emotional knowledge about each video, and improve the recommendation of videos based on emotions. Finally, we conclude by suggesting a domain specific XML schema that will provide information in a collaborative recommendation system that is based on emotional information on videos.

2 Emotional data gathered while watching videos

Video emotional classification based on user assessment has been recently improved with new techniques, especially since the introduction of emotion theories [1; 2] [3] in Human-Computer Interaction by Rosalind Picard [4]. The automatic user emotional assessment for video classification has recently been the focus of Money et al. [5] who report on how user physiological responses vary when elicited by different genres of video content using biometric artifacts (electro dermal response, respiration amplitude, respiration rate, blood volume pulse and heart rate). Another relevant work [6] characterizes movie scenes by emotions felt by users through the analysis of their physiological responses and by user self-assessment as the ground truth: the emotional categorization and representation use the dimensional model of emotions [2], where valence (positive/negative) and arousal (calm/excited) serves as emotional classification. In another work [7], the authors measured physiological signals from users in order to classify the emotional impact of films and tested whether the film experience is different in a group context or in an individual context, and found out that music is correlated with users' emotional highlights. The classification is obtained by comparing the detected biometric peaks with the manually introduced emotional categories for each movie segment and using a categorical perspective, using 21 Salway's [8] type of emotions as tags for emotion classification. These works show that there are significant differences between users' responses to the same movie scene, which reflects that there are personal information items that can be

considered for the creation of personalized profiles based on users emotional impact while watching videos. The development of such profiles requires semantic descriptions to allow the analysis of user preferences and to enable the comparison with other users.

Another emotional gathering approach for videos is focusing its content. Every emotional classification based in video content is also fundamental to the creation of user profiles. In fact, user profiles will be constructed based on emotional features derived from both users and videos. The collection of emotional information from video content is usually achieved by using algorithms that extract video low-level features, such as the analysis of color, motion, lightning or shot cut rates. In this context, [9] it has been proposed an affective representation for video content based in Hidden Markov Models to analyze color, motion and shot cut rate information, and output their results into a two dimensional (2D) emotional space . On the other hand, other authors [10-11] have been concentrating their classification in motion analysis, vocal effects, shot length and sound and rhythm analysis, and use the dimensional approach of a 2D space (valence, arousal) to represent affect in video scenes. A more recent work by [12] correlates multimedia feature extraction from video, self-assessment valence-arousal grades and user physiological signals to determine an affective representation of videos. The following section reviews semantic approaches for the description of emotions and describe some of the major problems regarding the semantic description of emotions.

3 Describing and representing emotions

There are three main models for emotion classification. Dimensional theorists defend that two or more dimensions can define emotions, like the dimension of valence (positive/negative) or arousal (calm/excited) [2]. Others admit a set of basic emotions, like the Ekman's [1] six basic emotions of anger, disgust, fear, happiness, sadness and surprise. The psychologist, Klaus Scherer [3] defined emotional descriptors, and presented the Geneva Affect Label Coder (GALC) with a thirty-six affective categories, which allows exploring ways of gathering emotional user profiles and classification of content. Every work based on emotion classification, be it from perceived emotions or from video content, always follows an emotion recognition process. The main steps begin with the data set selection followed by the use of data extraction methods to acquire the most important features, which then need to be processed. The process might end with a classification method that would transform the extracted features into emotional data or categories. The large number of methods, techniques and classification results from users emotional states [5; 11-15] and video content classification [6; 10; 12; 16; 17], and the open issues that these categorizations still exhibit demand for more coherent, flexible and representative forms of emotion specification.

Regarding the transformation of physiological signals into some representation according to any emotional model representation, there are still a number of open issues. For example, in [15] a group of physiologists and psychiatrics showed that affect representation of users when watching videos within a dimensional

categorization of emotions can have some problems: they measured fear and sadness responses induced by films and found out that a 2D space affective model of emotions, along with this complex patterns of bio measures, turns out to be quite inadequate to express what they call "long term emotion", a pattern observed in film viewing, listening to music or real-life emotion induction. Thus, a 2D space model of emotions seems to be more adequate to short term emotion elicitation and to initial appraisal processes.

The W3C Emotional Markup Group [19] aim was to design a general emotion markup language that can be used in different contexts and applications and to allow developers to work with emotional data and provide means of sharing this information. In [20] there is a suggestion of syntax for an XML-based language to represent and annotate emotions in technological contexts. These efforts were aimed at the creation of a generic structure for the semantic description of emotions in these contexts. Another perspective of semantic description regarding emotional states is shown in a preparatory work described in [21] where it was attempted to create an emotion specification for physiological signals: it analyzed a set of the most dominant physiological signals used to capture emotions - heart rate/pulse (measured through ECG), respiration rate, facial muscle movement (EMG), skin conductance (SC or GSR), and electroencephalogram (EEG). Their main goal was to turn the huge amounts of raw data classified by the mostly used pattern classification methods into one record of information. In fact, there are many classification methods, thus producing very different results, for example as suggested by Scheirer et al. [22] and Picard et al. [13], who explored a feature-based approach to classification, gathering a variety of features from the literature, and proposed other features of physical nature or obtained from statistical and nonlinear combinations thereof. In [23] the authors present a list of the dominant works in physiological signal based emotion recognition describing, for each signal, how emotions were induced, how many features were considered, the methods used to classify them, the selection criteria and the obtained results. These examples reveal that there are specific data that needs to be assembled in order to obtain, both quantitative and qualitative information about users while watching videos. The aggregation of such data items will further improve a collaborative recommendation system due to the wider range of data to relate.

4 Semantic representation of affective data of users while watching videos

In this paper we propose a semantic description of emotions based on an Emotion Annotation and Representation Language (EARL)[20], which aims to represent emotional states in technological environments. Being an XML-based language, it standardizes the representation of emotional data, allowing the re-use and data-exchange with other technological components. Because there is no agreed models of emotion, and there is more than one way to represent an emotion, and also because there are a number of use-cases which demand specific data requirements, EARL leaves freedom for users to manage their preferred emotion representation. For example EARL provides means for encoding the following types of information:

Emotional Data	Description
Emotion descriptor	Any emotional representation set
Intensity	Intensity of an emotion expressed in numeric, discrete values
Regulation types	Which encode a person's attempt to regulate the expression of her emotions
Scope of an emotion label	Link to an external media object, text, or other modality
Combination	Co-occurrence of emotions and it's dominance
Probability	Labeler's degree of confidence

There are a number of other information items that can be crucial in other specific contexts. As the general usefulness of many of these information types is undisputed, they are intentionally not included as part of the EARL specification. The authors clearly state that, when it's needed, it should be specified a domain-specific coding scheme that embeds EARL.
In this context, our proposal attempts to develop a connection between physiological signals and video low-level features extraction with emotions respecting formal standards.

In our case, we wish to represent emotional data felt by users while watching movies, in order to enable the creation of an effective recommendation system. Affective information of users when watching videos has specific data requirements to be represented, namely:

- Physiological representation
- Classification methodology
- Feature analysis
- Unit of measure for each of the above

Hence, for each emotional detection concerning a specific user in a specific time of the scene, we should define two blocks of information: 1) information related with video scenes - concerning specific features of video scenes content classification; 2) items related with users - including specific physiological parameters and user self-assessment of the emotional impact of video scenes, and include in each one the emotional data in accordance to emotion mandatory requirements suggested in [22]. This kind of information is crucial for the creation of recommendation rules for video in a collaborative context.
The following table presents our proposed domain-specific information that describes major emotional elements, which classify users when watching videos. The elements of the first block represent video affect analysis and related classification information,

and the second block contains elements that represent user affective information and related classification methods.

Table 1. XML Elements for Emotion Classification Videos

Video Elements		
Element	**Description**	**Example**
Scenes analysis method	Color Variance, Lightning, Motion, Shot Cut Rates.	Vector of methods
Video features analyzed	Saturation Luminosity.	Vector of features
Classification techniques	Video content classification techniques.	HMM
Unit measure	Percentage or other	%
Affective Result	Depend on the Emotional model adopted.	Happiness
Value	Depend on the unit measure	78%
Accuracy of the result	Success Rate	95%
Type of phenomenon that is being presented	In a movie it can be an intense episode triggered by a concrete event.	A surprise event can trigger a commemoration
Action tendencies	In movies, it corresponds to the prediction of scenes;	Low saturation, slow movements
Affect Interval	time interval of the extracted feature.	2 minutes

Table 2. XML Elements for Emotion Classification of Users While Watching Videos

User Elements		
Element	**Description**	**Example**
Date	Acquisition Date in timestamp	2802201018300030
Physiological signal	Physiological signal (Cardiovascular, GSR, Heart-rate, ...)	Cardiovascular
Physiological feature	heart rate, T-wave amplitude, low- and high-frequency heart rate variability, ...	Heart-rate
Multiple signal analysis	Physiological signal (Cardiovascular, GSR,Heart-rate, ...)	Vector of methods
Signals features analyzed	heart rate, T-wave amplitude, low- and high-frequency heart rate variability, ...	Vector of features
Classification method	Classification techniques	Fisher Projection
Unit measure	Percentage or other	%
Affective Result	An emotion, described by any model. If not such output, this value is void	Happiness
Value	Depend on the unit measure and on classification methods	78%
Type of phenomenon that is being presented	It can be an emotion, a mood, an attitude.	Fun
Action tendencies	It may be (positive mood/negative mood).	Positive
Affect Interval	Physiological signal interval time	2 minutes
User self-assessment	User manual classification	Joy, amusement

In the following paragraph we present a summary presentation of the table above in an XML structure:

```
<earl xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns="http://emotion-research.net/earl/040/aibolabels"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://emotion-
research.net/earl/040/aibolabels file:earl-aibolabels-
0.4.0.xsd">
```

```
<earl:emotion start="0.4" end="1.3">
<earl:complex-emotion xlink:href="videoXZV.avi">
  <emotion category="pleasure" simulate="0.8"/>
  <emotion category="annoyance" suppress="0.5"/>
</earl:complex-emotion>

<video id="V00001239123">
      <MultipleScenesAnalysisMethod method="Color">
       Low Satured </MultipleScenesAnalysisMethod>
      ...
      <ClassMethAccuracy>80%</ClassMethAccuracy>
</video>

<user id="U00001239123">

      <PhysiologicalSignal>Cardiovascular
      </PhysiologicalSignal>
      ...
      < User self-assessment >80%</User self-assessment >

   </user>
</earl:emotion>
```

Hence, this structure will represent the emotional information of users and videos regarding users when watching videos, for an affective and effective characterization of both videos and users. EARL specification covers a list of use cases for emotional representations such as 1) manual annotation of emotional content of (multimodal) databases, 2) affect recognition systems and 3) affective generation systems such as speech synthesizers or embodied conversational agents. Our purpose with this specification is to add crucial information to represent emotional recognition systems that use physiological signal processing. Emotion recognition by physiological analysis is about feature analysis and their classification into emotional categories after classification procedure. Feature selection depends on the purpose of the recognition, and there is no agreement which are the best features to analyze neither the best methods to classify, but all recognition process based in physiological signal define them. In order to compare different classification processes and to distinguish feature extraction we need a structure that can inform for instance that, the emotion "sad" was achieved using the heart-rate variation below 0.2. This representation will be embedded in EARL as a specific-domain coding scheme to help solving the problem we addressed. This XML schema also allows the creation of automatic affective information that will not only enhance a video recommendation system based in emotional information, from the user side and from the content side, but also enhance search retrieval of videos affective content from both users classification and content classification.

6 Conclusion

Recent studies reveal that there are specific data that needs to be assembled in order to obtain, not only quantitative but also qualitative information about users while watching videos. The aggregation of such data items will further improve a collaborative recommendation system due to the wider range of data to relate. In one hand, physiological signals results can show significant differences between users responses to the same movie scene, other conclusions reveals that some arousal result from users, watching the same scene are captured by different physiological signals, which shows one more time the existence of important information that collected in a user profile can enhance search and recommendation in a collaborative context. The development of such profiles requires semantic descriptions to allow the analysis of user preferences.
We adopted EARL (Emotion Annotation and Representation Language) an XML-based language for representation of emotional data, given that it supports the representation of the three main models of emotion, respects all the characteristics indicated above, and allows the integration of new elements in its specification. In our case we want to represent emotional data felt by users while watching movies, in order to enable the creation of an effective recommendation system. Affective information of users when watching videos has specific data requirements to be represented. Having video emotionally classified will further allow finding interesting emotional information in unknown or unseen movies, and it will also allow the comparison among other users' reactions to the same movies, as well as comparing directors' intentions with the effective emotional impact on users and analyze users' reactions over time or even detect directors' tendencies. This classification can contribute to the specification of a software tool, which enables the personalization of TV guides according to individual emotional preferences and states, and recommend programs or movies.

Acknowledgements

This work was partially supported by LaSIGE through the FCT Pluriannual Funding Programme. This work was partially supported by PROTEC 2009

References

1 Ekman P., Friesen, W. (1975). Unmasking the face: a guide to recognizing emotions from facial clues. Prentice-Hall, Englewood Cliffs, NJ..

2 Russell, J. A. (1980). A circumplex model of affect. Journal of Personality and Social Psychology, 39, 1161–1178.

3 Scherer, K. R. (1999). Appraisal theories. In T. Dalgleish, & M. Power (Eds.). Handbook of Cognition and Emotion (pp. 637–663). Chichester: Wiley.

4 Picard, R. W. (1997). Affective computing. The MIT Press.

5 Money, A. G., & Agius, H. (2009). Analysing user physiological responses for affective video summarisation. Displays , 30 (2), 59-70.

6 Soleymani, M., Chanel, G., Kierkels, J. J. M., & Pun, T. (2008). Affective ranking of movie scenes using physiological signals and content analysis. In MS '08: Proceeding of the 2nd ACM workshop on Multimedia semantics , (pp. 32-39). New York, NY, USA: ACM.

7 Alan F. Smeaton , Sandra Rothwell, Biometric Responses to Music-Rich Segments in Films: The CDVPlex, Proceedings of the 2009 Seventh International Workshop on Content-Based Multimedia Indexing, p.162-168, June 03-05, 2009

8 Salway, A. and Graham, M. 2003. Extracting information about emotions in films. In Proceedings of the Eleventh ACM international Conference on Multimedia (Berkeley, CA, USA, November 02 - 08, 2003). MULTIMEDIA '03. ACM, New York, NY, 299-302. DOI= http://doi.acm.org/10.1145/957013.957076

9 Kang, H. (2003). Affective content detection using HMMs. In Proceedings of the Eleventh ACM international Conference on Multimedia (Berkeley, CA, USA, November 02 - 08, 2003). MULTIMEDIA '03. ACM, New York, NY, 259-262. DOI= http://doi.acm.org/10.1145/957013.95706612]

10 Hanjalic, A., & Xu, L.-Q. (2005). Affective video content representation and modeling. Multimedia, IEEE Transactions on , 7 (1), 143-154.

11 Healey, J. A. 2000 Wearable and Automotive Systems for Affect Recognition from Physiology. Doctoral Thesis. UMI Order Number: AAI0801928., Massachusetts Institute of Technology.

12 Soleymani M.S., Chanel, C. G., Kierkels, J. K., Pun, T. P. (2009). Affective characterization of movie scenes based on content analysis and physiological changes. International Journal Of Semantic Computing. Vol.3 No.2, 235-254

13 Picard, R. W., Vyzas, E., & Healey, J. (2001). Toward machine emotional intelligence: Analysis of affective physiological state. IEEE Transactions on Pattern Analysis and Machine Intelligence , 23 (10), 1175-1191.

14 Nasoz, F., Alvarez, K., Lisetti, C. L., & Finkelstein, N. (2004). Emotion recognition from physiological signals using wireless sensors for presence technologies. Cognition, Technology & Work , 6 (1), 4-14.

15 Kreibig, Sylvia, D., Wilhelm, Frank, H., Roth, Walton, T., Gross, & James, J. (2007). Cardiovascular, electrodermal, and respiratory response patterns to fear- and sadness-inducing films. Psychophysiology , 44 (5), 787-806.

16 Wei, C. Y., Dimitrova, N., & Chang, S. F. (2004). Color-mood analysis of films based on syntactic and psychological models. In IEEE International Conference on Multimedia and Expo (ICME) . Taipei, Taiwan.

17 H. L. Wang, L. F. Cheong, Affective understanding in film, IEEE Transactions on Circuits and Systems for Video Technology, 16(6),:689--704, 2006.

18 Fahrenberg.J., Forster, F. (1982). Covariation and consistency of activation parameters. Biological Psychology, 15151-169

19 Schröder, M., Devillers, L., Karpouzis, K., Martin, J., Pelachaud, C., Peter, C., Pirker, H., Schuller, B., Tao, J., and Wilson, I. 2007. What Should a Generic Emotion Markup Language Be Able to Represent?. In Proceedings of the 2nd international Conference on Affective Computing and intelligent interaction (Lisbon, Portugal, September 12 - 14, 2007). A. C. Paiva, R. Prada, and R. W. Picard, Eds. Lecture Notes In Computer Science, vol. 4738. Springer-Verlag, Berlin, Heidelberg, 440-451.

20 Schröder, M., Pirker, H., & Lamolle, M. (2006). First suggestions for an emotion annotation and representation language. In L. Deviller et al. (Ed.), Proceedings of LREC'06 Workshop on Corpora for Research on Emotion and Affect (pp. 88-92). Genoa, Italy.

21 Luneski, A. and Bamidis, P. D. 2007. Towards an Emotion Specification Method: Representing Emotional Physiological Signals. In Proceedings of the Twentieth IEEE international Symposium on Computer-Based Medical Systems (June 20 - 22, 2007). CBMS. IEEE Computer Society, Washington, DC, 363-370.

22 Scheirer, J., Klein, J., Fernandez, R., & Picard, R. (2001). Frustrating the user on purpose: A step toward building an affective computer.

23 J. Wagner, J. Kim, E. Andre (2005). From Physiological Signals to Emotions: Implementing and Comparing Selected Methods for Feature Extraction and Classification .

CardioML: Integrating Personal Cardiac Information for Ubiquous Diagnosis and Analysis

Luis Coelho[1], Ricardo Queirós[1],

[1] ESEIG - IPP, Rua D. Sancho I, 981,
4480-876 Vila do Conde, Portugal
{lcoelho, rqueiros}@eu.ipp.pt

Abstract. The latest medical diagnosis devices enable the performance of e-diagnosis making the access to these services easier, faster and available in remote areas. However this imposes new communications and data interchange challenges. In this paper a new XML based format for storing cardiac signals and related information is presented. The proposed structure encompasses data acquisition devices, patient information, data description, pathological diagnosis and waveform annotation. When compared with similar purpose formats several advantages arise. Besides the full integrated data model it may also be noted the available geographical references for e-diagnosis, the multi stream data description, the ability to handle several simultaneous devices, the possibility of independent waveform annotation and a HL7 compliant structure for common contents. These features represent an enhanced integration with existent systems and an improved flexibility for cardiac data representation.

Keywords: XML, e-diagnosis, cardiopathology.

1 Introduction

In the last few years information technologies and electronic devices acquired an increasingly important role on clinical practice and medical services. Most activities, from patient's clinical history management to assisted diagnosis, have benefited with the latest technological developments. Cardiology, a highly requested area in modern societies, is not an exception. The electrocardiogram (EKG), the primary diagnostic instrument for cardiopathology detection, can now be obtained with portable and accurate equipments which can provide reports in the classical graph paper or in a digital format. Phonocardiograms (PCG) and others are also important complements for accurate diagnosis. However the structure and definition of a file format for integrating these informations still finds no agreements and will be the subject of discussion in this paper.

In an effort to improve organization efficiency and eliminate paper as a support for medical exams many healthcare institutions adopted medical image examination and management environments with Picture Archiving and Communication (PAC) system. In PAC systems paper based electrocardiograms are digitalized and stored in an image format which is restrictive and inefficient for a waveform type signal. The

main disadvantages are the inability to directly perform signal processing operations, the difficulty in waveform segmentation/annotation, the absence of complementary diagnosis information and, not less important, the higher database storage requirements. In some cases image processing algorithms are used to extract waveform information but the process in highly inaccurate. In 2000, the National Electrical Manufactures Association (NEMA) extended their Digital Imaging and Communications in Medicine (DICOM) format [1] in order to include ECG waveform information, the DICOM-ECG. This format uses two possible file types, Standard Communications Protocol ECG (SCP-ECG) in binary format and Extensible Markup Language (XML-ECG), that are encoded in a new file containing patient information and other examination related data. Additionally the International Standards Organization (ISO) released his own SCP-ECG [2] while the American Food and Drug Administration (FDA) announced the FDA-XML format [3], different from the first. The FDA-XML format is based on the results of the openECG project [4], an European Commission sponsored project for enhancing the interoperability between medical devices and information systems which gathered the interest of several organizations and companies of the medical area. However there is still no consensus between equipment manufacturers which continue to make their own interpretations of the available file formats and provide proprietary data encoding. In addition to the information storage mode, binary or text, the main differences between these formats are related with the quantity and type of content that complements the waveform and how it is structured. Some authors [5,6] have proposed new extensions to the existing data descriptions structure but they still present issues like lack of flexibility, insufficient number of parameters and/or non-compliance with current standards.

In this paper we propose a new file/data structure designed to bring together the best features of each of the above mentioned formats and simultaneously provide an extended and wide range of possibilities in order to make it suitable for most applications. The proposed format uses an XML structure and encoding in order to provide flexibility, portability and an enhanced integration with other existent architectures and technologies such as Health Level Seven (HL7) (Health Level Seven International) [7] recommendations (which evolved, in his last version 3, also to an XML based document format).

The rest of this paper is organized as follows. In the next section we describe the range of applications for which the proposed file format can be used and a possible usage architecture. In section 3 we thoroughly describe the proposed EKG format giving special details of the included features. Finally we provide the main conclusions and some envisioned work.

2 System Architecture

In this section we describe an architecture to acquire cardiac related data and how to display it in multiple targets for later interpretation of specialized medical personnel. We formalize this data proposing a new XML language, called CardioML, an XML based electrocardiograph language for ubiquous diagnosis and analysis.

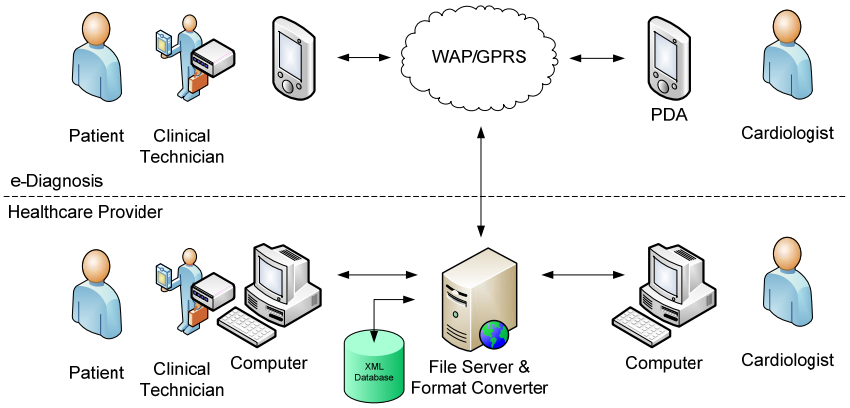


Fig. 1. Overall system's architecture.

2.1 Architecture

Latest cardiac signal monitors are highly portable and with good power autonomy which allows the execution of exams in remote areas either for routine exams on villages which are far from medical centres or in emergency situations where fast action is paramount. However this data requires interpretation from specialized medical personnel which is not always available. In this case the acquired data can be transmitted to a server that can forward it to a personal digital assistant (PDA) or mobile phone where the physician's professional can interpret and suggest an adequate procedure. The overall system architecture is illustrated in Fig.1.

The flow execution sequence for this system is summarized in 5 steps:

1. The Clinical Technician makes the examination to the Patient using one or several equipments (with or without web access);
2. The cardio-physiological signal and related data is passed from the equipment that made the examination to the server; When the equipment does not have this functionality the acquired data is sent to an equipment (computer, PDA or smartphone) with web access (local or remote) which will then forward it to the web server.
3. The acquired data is then converted to the language CardioML (this format conversion will depend on the source file type and data description structure). A first-in first-out document queue is created for managing the conversion process. Server side conversion allows an always updated source file format database and a faster conversion time than client side conversions;
4. The server keeps all the XML files in a XML Database. (Since one of the motivations for the development of this new format was system integration and data portability the database should be supported by a native XML structure);

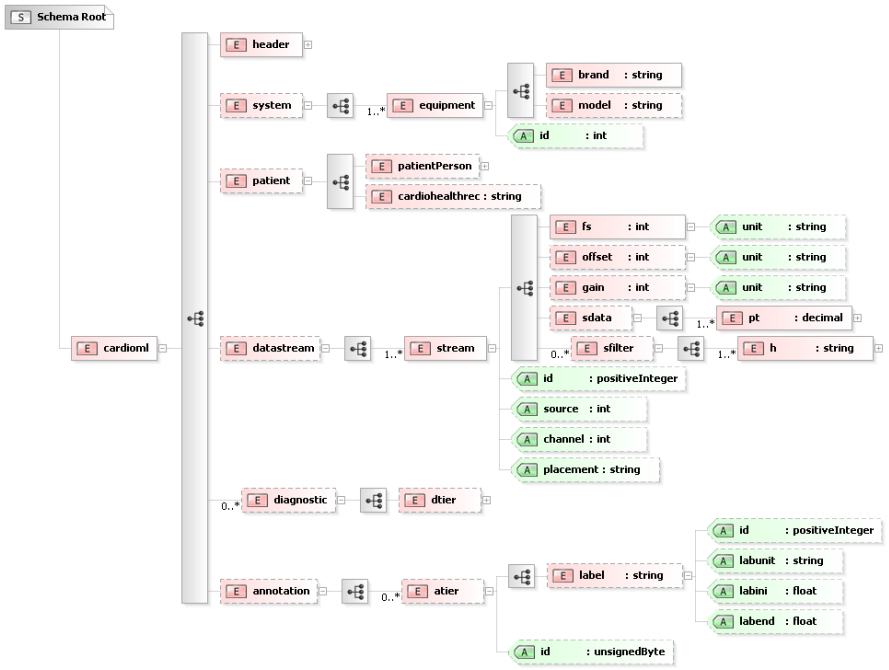


Fig. 2. Partial view of the CardioML schema.

5. The cardioML encoded data can be viewed or annotated using a web based server-side application or downloaded and analysed on the client side. For devices with low computational resources the server can transform the XML file in a specific graphic format (HTML, SVG, Flash, etc.) for viewing purposes only;

2.2 The CardioML language

As we have seen in the previous subsection, the acquired data can be transmitted to servers which can then forward it to a mobile phone where the medical professional can interpret and suggest an adequate procedure. This data transmission over several equipments demands a portable file format and a common description language.

Figure 2 exposes the CardioML schema language. It consists of an element root `cardioml` composed by six elements: `Header`, `System`, `Patient`, `Datastream`, `Diagnostic` and `Annotation`. Elements such as `patientPersonType` or `diagnosticType` are not fully expanded in figure 2 due to space constraints but their content will be thoroughly covered in the next paragraph.

`Header` element contains file/record related information such as date of creation and owner. For e-diagnosis purposes a set of novel tags are included for registering the geographical information where the data acquisition took place. A document status element was also included for enabling the integration of cardioML files in

workflow management systems. The element `system` is used to store all the information related with the equipments that were used to perform data acquisition. One novel feature of cardioML is that its structure allows the registration of several devices on an unique file. This is an important feature since the diagnosis of several cardiopathotologies can be enhanced by simultaneously acquiring and analysing several cardiac related signals (for example electrocardiography and phonocardiography) that require independent specific devices. Child elements of `system` are not fully represented on figure 2 but informations such as next calibration date, accuracy, stability or environmental operating conditions can be stored and are paramount to assess the confidence of the measurements. Patient information, in the `patient` element, is stored in two distinct child elements, one for the patient's personal information and other for relevant clinical history. CardioML only stores cardiac related information but it allows possible observations or references to extended health records. Both sub-elements are based on the HL7 representation since this format encompasses patient information, most pathologies and medical acts and its usage is quickly being adopted on most healthcare information systems. Cardiac signals are stored as independent data streams that can be provided by any of the existent channel in the given equipments. Each stream is linked to his related source by their unique `id` numbers while the related acquisition conditions are described by elements gain, offset and fundamental frequency (`fs`), all with a related measurement unit. Data from each acquired signal is organized in the sub-element `sdata`, composed by a sequence of `pt` elements, an optional attribute allows the time indexation of samples. Additionally, the frequency response of the several data acquisition chain components (active or passive electrodes, connectors, filters, amplifiers, etc.) can be included in `sfilter` elements. The described schema for cardiac signal description extends existent formats [5] by providing a flexible structure which enables the inclusion of several data streams with distinct sources, each independently characterized. A new important feature is also the delineation of the data acquisition chain. The `diagnostic` element is used to store diagnosis information which can be added by a physician after signal visualization and analysis. Several diagnostics are allowed, always dated and signed by an author, and, for pathology description, the HL7 structure should be used. Finally, an annotation element is included for enabling the inclusion of waveform metadata. Each annotation tier, composed by several `label` elements, can be related with a specific data stream, using an id reference, or can have a global scope. Besides their intrinsic description, the labels are associated with a time frame defined by `labini` and `labend` attributes. The separation between annotation and waveform elements is a novel feature that presents advantages such as multi-authoring and local or global scopes.

3 Application

A software infrastructure that supports data interchange using the cardioML format has already been implemented and is being tested. Some screenshots of the client application running on a smartphone are presented in figure 3. The system's full description is out of the scope of this work and will be presented in a future article.

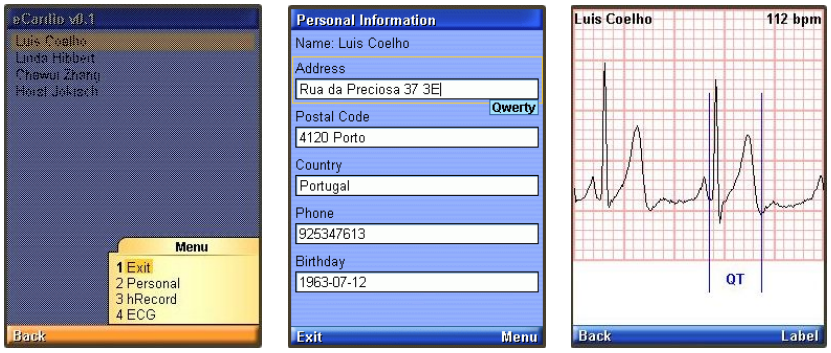


Fig. 3. Screenshots of a mobile application making use of the cardioML structure.

4 Conclusions

In this paper a new XML based format for storing cardiac signals and related information was presented. The evolution of management and archiving techniques for health records in this scope was thoroughly covered and the main challenges identified. An application architecture for e-diagnosis services brings additional motivations for the development of the CardioML language. CardioML was then presented giving special attention to the main elements and the advantages they provide when compared with distinct formats recently proposed by other authors. The presentation is complemented with the related schema and an application example. The proposed format can provide an enhanced integration with existent systems and an extended flexibility for cardiac data structuring. Among others, the main advantages are the independent description of elements in the signal acquisition chain, the support of several data streams of different types and from distinct sources and the multitier annotations.

References

1. DICOM Homepage, http://medical.nema.org/
2. Health informatics—Standard communication protocol - Computer-assisted electrocardiography. ICS: 35.240.80 IT applications in health care technology; reference number EN 1064:2005+A1, (2007)
3. Brown B, Kohls M, Stockbridge N: FDA-XML data format design specification, revision C. www.openecg.net, 1–27, (2003)
4. OpenECG Portal, http://www.openecg.net/
5. Xudong, L., Huilong, D., Zheng, H.: XML-ECG: An XML-Based ECG Presentation for Data Exchanging. In: Proceedings of the 1st International Conference on Bioinformatics and Biomedical Engineering ICBBE 2007, pp. 1141-1144, China, (2007)
6. Badilini, F., Isola, L.: Freeware ECG Viewer for the XML FDA Format. In: 2nd Open OpenECG Workshop, Germany, Berlin (2004)
7. Health Level Seven International, http://www.hl7.org/

Author Index