University of Wisconsin Milwaukee
UWM Digital Commons

Theses and Dissertations

May 2014

Version Aware LibreOffice Documents

Meenu Pandey
University of Wisconsin-Milwaukee

Follow this and additional works at: http://dc.uwm.edu/etd

 Part of the Computer Sciences Commons

Recommended Citation
Pandey, Meenu, "Version Aware LibreOffice Documents" (2014). *Theses and Dissertations.* Paper 828.

This Thesis is brought to you for free and open access by UWM Digital Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of UWM Digital Commons. For more information, please contact kristinw@uwm.edu.

VERSION AWARE LIBREOFFICE DOCUMENTS

by

Meenu Pandey

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in Computer Science

at
The University of Wisconsin-Milwaukee
May 2014

ABSTRACT

VERSION AWARE LIBREOFFICE DOCUMENTS

by

Meenu Pandey

The University of Wisconsin-Milwaukee, 2014
Under the Supervision of Professor Ethan V. Munson

Version control systems provide a methodology for maintaining changes in a document over its lifetime and provide better management and control with evolving document collections, such as source code for large software systems. However, no version control system currently supports such functionality for the office documents.

An office document can go through different modifications during its lifetime and can be developed by multiple technical or non-technical users. It might be desirable to know how the document came to its final stage and to sometime retrieve older versions of the document or merge two different versions of a document without manual effort.

This thesis work explains how we could implement versioning support for LibreOffice documents without using additional infrastructure for version repositories. Since embedding versioning data within the office document can indeed make version control a seamless part of the writing process. Such a modified document with embedded versioning data is called a *version aware document*.

A versioning framework has been developed previously at UWM that provides this versioning functionality for version aware XML documents by calculating the reverse deltas between revisions. A Version Aware XML document integrates full versioning functionality into an XML document type, using XML namespaces to avoid document type errors. Version aware XML documents contain a preamble with versions stored in reverse delta format, plus unique ID attributes attached to the nodes of the documents. They support the full branching and merging functionalities familiar to software engineers, in contrast to the constrained versioning models typical of Office applications.

LibreOffice is a free open source office suite that is widely used for document creation and branched off from OpenOffice in 2010. It is managed by "The Document Foundation" and includes application for text documents, spreadsheets, presentations, drawings and database. Each document is represented in the Open Office Document Format (ODF), which is a collection of XML files.

The current project is an endeavor to show the practicality of the version aware XML documents approach by modifying the LibreOffice document suite to support version awareness. It is necessary to understand the architecture of LibreOffice application as well as the document load and save cycles, the XML element and attribute processing, the class hierarchies and the internal data structures. We have modified the source code of the LibreOffice Writer application to accept and preserve the required changes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. Ethan Munson for his guidance and support during the course of my research. I would like to extend my gratitude towards Cheng Thao and Brandon Nielsen, for assistance on this project. I would also like to thank the committee members for taking time to assist by providing feedback for this thesis.

Chapter 1

Introduction

1.1 Version control in office documents

Version control systems provide a methodology for maintaining changes on a document over its lifetime as a collaborative team goes about developing it. Typical version control systems, like Subversion[1], Mercurial[2], and Git[3] , provide the functionality for version repository creation, storage and retrieval of versions from a repository and creation of a graph of versions via branch and merge operations. Experience has shown that version control systems provide useful services for a large technical user base. These tools often require access to a central repository or a shared file system to store the versioned data. To make effective use of the storage space, different revisions of a file are stored in the form of deltas. A delta is a difference between two revisions of a file.

Traditional version control systems use the line based differencing approach for calculating the deltas between 2 revisions of a document and it works well with text

based files. However, most of the office documents use XML files to store their data and the XML files are inherently different than text files. The XML[4] files form a tree structure where only one root element exists that stores the user data in its child elements. Typically, an XML file has only two lines - encoding information creates the first line and the root element with its child elements creates the second line. The line based differencing approach does not work effectively for the XML files because every time, for any changes in the file, the whole second line of the XML file will be changed and the delta calculated will have almost the same size as the previous revision. Thus, the storage space cannot be used effectively for XML files using line based differencing methods.

Also, the user base for the office documents is typically non-technical and while the documents often go through many revisions, they are often standalone objects or are part of small collections. Thus, the overhead of creating and managing a repository is hard to justify.

One approach to track different revisions of a document is to save them by different names that suggest the evolution of the document. In this approach, manual branching and merging can achieve collaboration, but this can be a cumbersome and confusing task.

Office document software does provide simple version control, in the form of current version/past-version (Microsoft Office) or linear document histories (LibreOffice). While this support for versions is helpful, it is insufficient for collaboration

in large teams because parallel editing is not supported. This can force users to perform manual merges in order to integrate changes from multiple sources. Some cloud storage systems offer versioning and collaboration support for stored documents but they do not provide the systematic support for collaboration that a version control system does.

Branch and merge functions for office documents can aid users by keeping track of multiple revisions of a document within the document itself, maintaining branch information for the document when multiple authors work on it simultaneously, and later by merging those parallel changes into a unified version when needed. The ability to track changes on a document is important for many official and professional documents (user manuals, regulatory documents, technical design documents, etc.) as well as for personal documents.

1.2 Proposed Approach

The purpose of this thesis is to show that it is possible to add branch and merge functionality to the LibreOffice software suite, with the goal of facilitating collaboration with significantly less manual effort. The first step towards this goal is to convert LibreOffice ODF files into Version Aware Documents (VAD) and provide the basics of version control support. A VAD is a special document that stores complete change history of the document. As a VAD contains its entire document history, users do not need to interact with any version repository. Thus users will

gain the ability to access past versions, along with the ability to recover content that was deleted from multiple revisions in the past. Also, the VAD approach will allow support for authors to work simultaneously on the same sections of a document and later merge their changes. Non-conflicting changes can be merged automatically while conflicting changes will require some manual effort.

A document is a version aware, when it contains a namespace referred as *molhado* from the UWM versioning framework, a preamble that includes the reverse deltas, an XML signature to enforce the data integrity and Unique Identifiers for every XML elements that will be used by the versioning framework to calculate reverse deltas. We will show that LibreOffice documents can support these version awareness properties with few extensions to the software.

1.3 Thesis Organization

In the rest of the thesis we introduce version aware approach[5] for XML documents and how it can be implemented for LibreOffice Writer documents. Chapter 2 describes the concept of version awareness and importance of required changes. Chapter 3 gives necessary details about the LibreOffice architecture, text document processing, extensions made to add preamble and extensions made to the individual document components. Following chapters describe the scope of changes, scope for future work and conclusion.

Chapter 2

Background

This chapter describes the current status of versioning support provided by the conventional office applications. It explains the Version Aware Document Framework in detail and its importance for implementing version control functionality in office documents. Finally, it describes the motivation behind selecting LibreOffice for version awareness implementation.

2.1 Current Versioning Support for Office Documents

Versioning support is not new to conventional office document programs. They already support simple forms of version control.

Microsoft Word has a "Track Changes" feature that can be viewed as a two-version system. When changes are being tracked, there is a notion of the current version and of a single previous version. Differences between these versions are

tracked automatically and are shown to the user in the document margins. The changes are represented in terms of editing operations. A user can review the changes and can manually choose which edits should be preserved. The Word application can perform a three-way merge of the parallel edits to a document, but this only works correctly if the changes in the documents have never been accepted by the users.

LibreOffice stores text documents in a compressed archive that holds a series of files representing a linear document history. Each document is represented in the Open Office Document Format, which is a zipped collection of XML files. Users can choose to view or edit different versions if this is needed. Changes can be recorded and authors can accept or reject the changes between versions but LibreOffice does not provide three-way merge functionality and thus does not support simultaneous editing by multiple authors. Thus, neither Word nor LibreOffice applications supports true collaboration because neither one provides adequate services for merging parallel edits of the same base document. Whether existing LibreOffice version control functions can be used for better change detection is left for future work.

2.2 Version Aware Documents

Conventional source code version systems support branch and merge operations with tools like diff3[6] which assumes that the source material is raw text and that line breaks represent frequent and meaningful delimiters within files. In fact, modern

office document systems often store all their content in XML files with exactly two lines: one for the XML declaration and the second for the rest of the content. Line based differencing does not provide an efficient way to model XML tree structure. So researchers have developed versioning approaches based on XML elements.

Meaningful merging of XML content is challenging, because it is difficult to be certain how to match XML element content between two versions. Thao and Munson[7] showed that using unique IDs (UID) allow for an efficient merging algorithm. Based on this work, Thao[8] proposed a new Version Aware Document Framework which allows any XML document to contain its complete version history. The framework supports a full tree-based version history and can model changes to document content to document tree structure, and to the attribute values for the elements.

The Version Aware Document Framework uses four simple extensions to an application's native XML format:

1. a special namespace (called "molhado") allows co-existence of version data with the native XML content of the application ;

2. a revision history element in a preamble location that holds the version history information in reverse delta format;

3. XML signature elements to prevent users from altering the version data without detection; and

4. a unique identifier attribute for every element of the document content so that changes between different versions can be identified easily.

2.2.1 Revision History

Version control systems do not store entire files if there exist multiple revisions of a document. These systems use the concept of deltas to effectively manage disk storage. A delta is a sequence of edit operations that transforms a document from one revision to the other. Only one of the revision stores the full content of the document and all the other revisions are constructed by applying the deltas to the fully stored revision. There are two schemes to store the delta information in a repository.

- Forward deltas - The first revision of a file is stored as a complete file in the repository and the subsequent revisions are saved in the form of deltas. A chain of deltas is applied to the base revision to retrieve a forward revision.

- Reverse deltas - The latest revision of a file is stored as a complete file in the repository but the previous revisions are calculated by applying chain of reverse deltas to the latest revision.

The existing VAD system makes use of reverse deltas where the latest revision contains the complete document content, and applying a chain of deltas to the latest revision can retrieve the previous revisions.

The revision history element is responsible for storing the change information between revisions. Each sub-element of the revision history element specifies the edit operations performed from one revision to another. The main edit operations are: attribute value update within any element, changes in node sequences, node deletion, node addition, and node name update. Also, signature information to the XML is required to prevent users from modifying the existing versioning data. Thus, extension of revision-history in a document is essential to support effective version control.

2.2.2 Unique Identifiers

This versioning framework includes an efficient 3-way merge algorithm that requires each XML node to have a UID. UIDs are important for efficient matching of nodes between versions. If correctly maintained by an editing system, they allow the versioning system to match nodes between versions even when some nodes have undergone substantial transformations. UIDs also help to identify conflicts between two versions, which are currently expected to be resolved manually by the authors.

2.2.3 Version Aware Document Prototype

A first application of this framework was made using the Inkscape SVG editor. A wrapper application was designed that manages the maintenance of versioning information in Inkscape saved files.

When a wrapper application reads an XML file for the first time, it adds versioning information into the XML file. For the next changed revision of the XML, the wrapper application compares current revision with the previously saved revision according to the element node identifiers and saves reverse deltas with in the file.

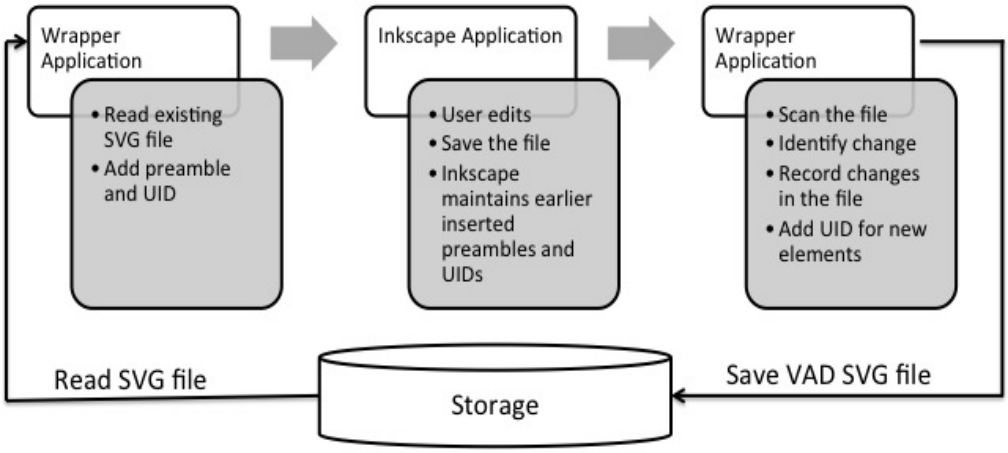

Figure 2.1: A wrapper application to create a VAD for Inkscape SVG editor

The reverse deltas include edit operations performed between two versions. Wrapper application also adds UIDs to the newly created elements. This approach works well with SVG editors as they are designed with the expectation that other applications might add namespace-protected content that should be preserved.

2.3 Research Motivation

We started with the idea of including version awareness features on to word documents. We looked at MS Office and LibreOffice, both of which are widely used office applications that internally use XML file formats. It is important to note that SVG XML files can be transformed to a Version Aware Document (VAD). With this in mind, it is natural to think that this approach will work with other XML based Office applications as well.

After experimenting with both these applications, we concluded that MS Office does not tolerate versioning information in its XML files. LibreOffice on the other hand does tolerate versioning information, but does not preserve it during the load-edit-save interaction cycle. This problem forced us to analyze options to make changes to LibreOffice so that it can support the version aware preamble and element UIDs.

This research work is an endeavor to prove the practicality of the Versioning Framework by implementing version control functionality in LibreOffice documents. For this purpose, LibreOffice documents need to be converted into a VAD and later a wrapper application would provide version control functionality to the LibreOffice VAD, as discussed earlier.

Few interesting facts about LibreOffice are - it is a widely used open source office document application. It supports Text Documents, SpreadSheets, Drawing and

others. Versioning support to the LibreOffice application will facilitate better collaboration among document users. Additionally, LibreOffice is open source software that provides free support for source code in C++ and other languages. Thus, LibreOffice provides an excellent test bed to experiment with office applications and our version awareness objectives. A LibreOffice ODF document is a compressed archive that contains four XML files: meta.xml, settings.xml, content.xml, styles.xml. The "content.xml" file stores user created content such as text, pictures and tables. This is the primary file, which can assist in maintaining version awareness in a LibreOffice document.

Only when the versioning information is preserved through load-save cycle in this file, we would achieve a real Version Aware Document (VAD).

Thus, if the content.xml can accept the versioning information, then a LibreOffice document will be a true VAD. Currently, VAD support is being implemented in the LibreOffice Writer application. Once the approach is successful, VAD support can be easily extended to other applications as well.

Chapter 3

Introduction to LibreOffice

3.1 LibreOffice History

LibreOffice[9] originally known as StarOffice, was created in 1985 as a proprietary software. In 1999, Sun Microsystems acquired StarOffice and renamed it OpenOffice. In the year 2000, Sun released OpenOffice source code and created an open source community. Oracle acquired Sun Microsystems and continued support for OpenOffice. In Sept 2010, the open source community branched off a new version named LibreOffice. The Document Foundation, which is a non-profit organization, now manages LibreOffice and supports open source document software. LibreOffice document suite includes following components - Writer for word processing, Calc for spreadsheet processing, Impress for presentation creation, Math for math functions, Draw for graphic documents and Base for database support. In short, LibreOffice provides a free alternative to Microsoft Office and other commercial office applications.

Our focus for VAD is on the LibreOffice Writer application and as Writer is part of LibreOffice suite, we will first discuss the general architecture of LibreOffice. As LibreOffice recently branched off from OpenOffice, there is lack of documentation for the application source code. Because, OpenOffice has been open source software for a long time, there is detailed documentation on most aspects of OpenOffice.

The Following sections describe the basic component technology used in LibreOffice called Universal Network Objects (UNO), the word processing artifacts used in Writer and the details of XML file handling in Writer.

3.2 Universal Network Objects

Universal Network Object(UNO)[10] is the base component technology used in the OpenOffice and LibreOffice application suites. A software component could be a software package or a module that encapsulates set of functions and data. UNO components can interact across platforms, languages and networks. UNO components can be accessed or implemented in any programming language for which language bindings exist. Language bindings are provided for C++, Java, OpenOffice Basic and other languages. UNOs operate within the UNO Runtime Environment (URE) and they can be used independently from LibreOffice.

UNO objects are specified in an abstract meta language, called UNOIDL. UNOIDL specifications are used to generate language dependent header files and libraries to implement UNO objects in the target language. UNO provides bridges to

send method calls and receive return values between processes and between objects written in different implementation languages. The remote bridges use a special UNO remote protocol (URP) for this purpose.

3.3 OpenOffice API

The OpenOffice API is a comprehensive specification and a language independent approach to describe the shared functionality used by OpenOffice applications. OpenOffice is made up of multiple smaller components, which are integrated with each other using OpenOffice APIs. The OpenOffice APIs can also be used by custom software applications. This enables external applications to make use of all the functionality provided by the OpenOffice.

Developers can lookup the details of these LibreOffice APIs on the API reference, which is a part of the SDK. API reference also include UNOIDL data types(UNO types). These data types would be mapped to the native data types of a programming language used for implementation. Some of the main UNO types are services, interfaces and properties.

1. Services: Services are units of functionality that are self contained. Every UNO component is registered as a service to the ServiceManager. Each service implements some actions. LibreOffice objects can inherit services. A Service consists of one or more interfaces and services are used through interface methods and through properties.

2. Interfaces: UNO objects can communicate based on object interfaces. Interfaces are abstract specifications. Interfaces provide access to objects by publishing a set of operations that cover a certain aspect of an object without telling anything about its internals. UNO uses the interface type to describe such aspects of UNO objects. By convention, all interface names start with the letter X. All interface types must inherit the XInterface root interface. An object can have more than one aspect and UNO uses multiple inheritance to define all those aspects.

3. Properties: An object in office environment can have large numbers of properties that may not appear to be a part of the structure of the objects, rather they are superficial changes to the underlying objects. So the properties are name-value pairs belonging to a service and determine the characteristics of an object in a service instance. XPropertySet and XPropertyAccess are some of the interfaces that are used to access the properties. These interfaces define methods as getPropertyValue and setPropertyValue.

3.4 The ServiceManager

The ServiceManager object is the main factory object for UNO components. In every UNO based application, a service manager is used during UNO component instantiation. The service manager maintains a database of registered components

that are known by their names and can be created using those names. ServiceManager supports the XMultiServiceFactory interface, which offers methods to create a service instance. E.g createInstance method returns a reference to XInterface. As a service defines more than one interface, the returned reference to XInterface can be further queried to access other interfaces specified by the service.

3.5 C++ Implementation of LibreOffice

For the current thesis work, we are working with the C++ implementation of LibreOffice. As described earlier, UNO provides language bindings for C++, Java and others programming languages. These language bindings are also called UNO runtime environment. Language bindings provide a mapping of all UNO types to the specific programming language types, a mapping of UNO exception handling to the language, and other language specific mappings.

3.6 LibreOffice Text Document Processing

LibreOffice document processing starts by a simple bootstrap mechanism. The bootstrap process asks the service manager to create an instance of a remote Desktop object which handles application windows and loaded documents. The desktop object creates a new text document or a spreadsheet document by creating a component context. Desktop object queries this component context to secure an appropriate

interface. With the help of these interfaces, LibreOffice application can access the in-memory document containers where the user data is saved.

The LibreOffice Writer application depends heavily on a UNO service called Text[1]. The Text service[11] supports the XText and XEnumerationAccess interfaces. These interfaces provide text editing and text iteration functionality respectively. During text document creation, a document interface is queried that supports the getText method. The getText method provides the document via the XText interface.

Text editing: The basic building blocks of a text editing process are

- Text contents[2] as paragraphs, textfields, and text tables,
- Text ranges[3] that keeps track of the start and end position of a text and finally,
- Text cursors[4] that provide movement between start and end of the text.

The service manager of the document using the factory design pattern can create all the text contents. Once a text content object is created, the application attaches it to the text range using the attach method. Similarly text position can also be retrieved by getAnchor method.

Text Iteration: The Text enumeration interface is used to iterate over the main

[1] com.sun.star.text.Text

[2] XTextContent

[3] XTextRange

[4] XTextCursor

components of the document. The component objects also support their own enumeration interfaces for iterating over their text portions. A text portion is a continuous text range whose contents are all formatted in the same way. A simple paragraph, formatted in a uniform manner and containing only strings, is equivalent to a single text portion. But paragraphs can have text fields and text spans with different formatting styles and these are handled as text portions.

Text Formatting: The Text service also supports many character and paragraph properties that define the formatting of the text. Because TextCursor, Paragraph and TextPortion are all sub services of Text, they also support these properties. Character and paragraph properties are handled by Style services. All objects supporting these properties support another important interface named XPropertySet that provides support for storing and accessing object properties.

Text Navigation: Text cursors allow easy navigation of the text by characters, words, sentences or paragraphs. A text document provides supplier factories for creating text contents and storing them in a collection. For example, to find a bookmark or text field, XBookMarksSupplier or XTextFieldsSupplier interfaces are used.

3.7 LibreOffice Load-Edit-Save Process

Introducing versioning information in the content.xml and preserving it through the load-edit-save cycle of a document is the main goal of this research paper. One of

the objectives of this research work is to preserve the UIDs of all the XML elements through load-edit-save cycle of the document. In order to achieve this, it is required to further understand the current XML file processing approach used in LibreOffice Writer application.

The zipped XML files undergo through changes during three main operations - load, edit and save.

First, during document load process, an XMLReader object reads each XML element and creates a corresponding import context class object that holds information about the element's attributes. Second, with the help of UNO objects, the saved element information is transferred to the in-memory document model and is saved in the respective data structures. During the load process, program control flows back and forth between context classes and document model classes with the help of UNO Runtime Environment (URE). This XML load process is also referred as the XML import process.

By the end of the load process, the document is available for user edits. A LibreOffice document is represented by a SwDoc class in the in-memory document model. This is a container class that can hold other container classes as well. These containers save the user content in the memory. Input data to these containers is provided by UNO objects, which help to initialize the specific sub containers. The new edits made in the document are also saved in the in-memory data structures.

Finally, during the save operation, the application creates a new set of class objects called "export context objects". These context objects use UNO objects to

retrieve the saved in-memory data. The export context methods then write the data back to the output XML file. The XML save process is also referred as the XML export process.

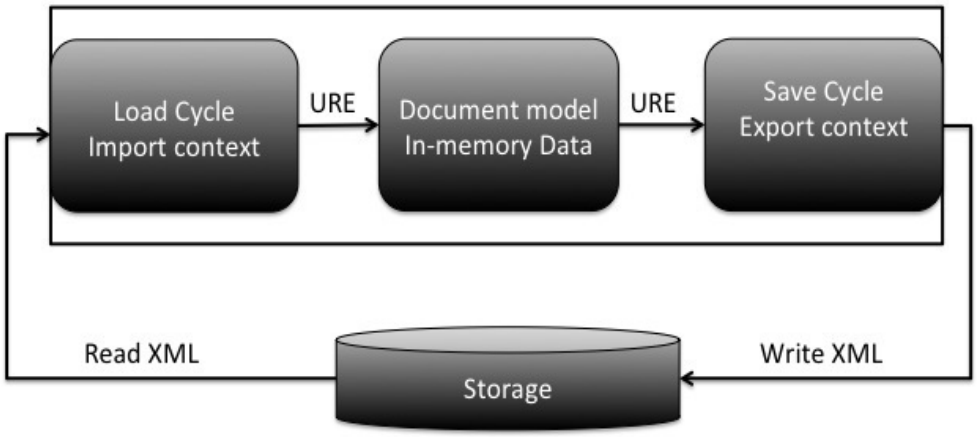

Figure 3.1: XML load-edit-save cycle

The LibreOffice source code corresponding to the load-edit-save process is contained in two modules named xmloff and sw. The "xmloff"[12] module implements most of the XML import and export functionality while the "sw"[13] module implements the UNO interfaces and their methods as well as the in-memory document model container definitions.

3.7.1 Detailed Description of the Import Process

This section describes the specific details of XML import process. The SvXMLImport class implements XML element import processing methods for load operation. This class defines methods such as startElement and endElement. The start method call is invoked when the start tag of an XML element is encountered. Similarly, the end method is invoked when the end tag is encountered. This class also has a vector data structure that holds import context class objects created during the import process.

During XML load operation, when an XML element is read and processed by the startElement method, an import context class object is created for every XML element. The base import context class is named as SvXMLImportContext[14].

A context class has three main responsibilities:

- Save the XML element attributes in its data variables.
- Initialize another context class object for its nested elements, if needed.
- Initialize the required UNO objects[5] and invoke their methods to transfer the XML element attribute information to the in-memory data structures.

In general, these responsibilities should be performed by the three methods of a context class- StartElement, CreateChildContext and EndElement respectively. But, all the context classes do not always follow this process uniformly. We will discuss the anomalies in detail in the following sections.

[5]XTextCursor, XTextRange, XTextField

Depending on the context class, the XML element attributes are processed either during context class object initialization or after context object creation, using StartElement method. Similarly, UNO object creation and transfer of saved attributes to the in-memory data structures, can be performed during context construction call, context EndElement method call or during context destruction call.

This non-uniformity in the import process makes it difficult to perform required changes in one key control point.

3.7.2 Document Model(In-memory Data Structure)

A document model implements various containers for every document component and also implements efficient logic to store and access the data in these containers. For example: The paragraph element information is stored in a SwNode object in a document model and every SwNode object then is stored in a vector SwNodes for rest of the document life cycle. Similarly, the text fields are stored in a SwAttrPool object, which is a large container that stores SfxPoolItem objects.

The SwDoc class is the main container class that stores all the text objects in the memory. Some of the text objects are paragraph, text fields, character and paragraph properties (formatting properties) and bookmarks.

Depending on an import context object, a corresponding UNO service instance is invoked. Each UNO object in a service instance supports certain interfaces and

properties that are implemented by LibreOffice sw module[6]. To set an XML attribute, a UNO object method such as setPropertyValue is invoked with attribute name and value.

An XML element attribute is represented as a property in the document-model. The setPropertyValue method passes on this information to the document model classes only if the property corresponding to the attribute is supported by the UNO object.

The input attribute name should map to a supported UNO property name. To make sure that an UNO object supports an input attribute, a Property map is defined where all the UNO object properties are listed. Unknown attributes are rejected during this process and their values are lost. This is one of the reasons why the versioning information is not processed during load cycle of the LibreOffice document. The property list is an important place to introduce a new property for any UNO object.

Once a property name is verified, the document model classes then store this input data to the appropriate data structures.

The program control comes back to import context classes, where multiple calls are invoked for setting the rest of the XML element attributes in the in-memory data structures. Once all the attributes of an XML element are set in the memory, the import context object is deleted. It marks the end of the loading process of an XML element. Similarly, the rest of the XML elements are processed as well.

[6]The code for UNO interface methods is present in unocore sw/source/core/unocore folder

3.7.3 Detailed Description of the Export Process

During the save operation, the saved in-memory data is exported back to XML files. The export process retrieves the saved document data with the help of UNO objects and writes XML elements with the help of export context class methods.

A content.xml file has a certain XML element pattern - namespaces followed by font-styles, auto-styles and text data. The export context objects and their methods are invoked accordingly.

The Export process is implemented by both SvXMLExport[7] and SwXMLExport[8] class methods and the control flows between their methods. The SvXMLExport class uses other helper classes to export font-styles[9], automatic-styles[10] and text[11] data.

In general, these helper classes write the XML elements to the output XML file in three steps.

- First, all the attributes of an XML element are retrieved from the document model with the help of UNO objects. These attribute values are stored in a attribute vector[12] by AddAttribute method. It is mandatory to add all the attributes to the vector before writing the XML element itself to the output file.

[7] xmloff/source/core/xmlexp.cxx
[8] sw/source/filter/xml/xmlexp.cxx
[9] XMLFontAutoSylePool
[10] SvXMLAutoStylePoolP
[11] XMLTextParagraphExport
[12] SvXMLAttributeList

- Next, a SvXMLElementExport object is instantiated with the attribute vector that was saved earlier. This object calls sax::XDocumentHandler method to write the XML element and its attributes to the output file.

3.8 UNO Properties in LibreOffice

In the earlier section, we elucidated that UNO properties define the characteristics of UNO objects. Let's understand the concept of properties with an example and how it is relevant to version awareness properties.

In the LibreOffice Writer application, users can insert current date using date field option. To process this field, in the LibreOffice environment, a DateTime (UNO) text field service object is instantiated by invoking the ServiceManager factory method createInstance (service name).

According to the DateTime service specification, it can support up to seven optional properties such as DateTimeValue, DateTimeFormat or IsDate. The DateTime service also supports XPropertySet interface that makes it possible to access these properties.

A property is defined as a Struct that has a name, a handle, a type and a property attribute.

- Property name is a string value which also corresponds to an attribute name of an XML element.

- Handle is a numerical identifier, used for multiple purposes.

- Type of a property identifies the data type of the property value.

- Property attribute specifies the behavior of a property(void, read-only).

In the LibreOffice environment, the seven DateTime properties are listed in a property map defined for DateTime object in unomap.cxx[13]. The XPropertySet interface is implemented by SfxItemPropertySet[14] class. A property is defined by a SfxItemPropertyMapEntry struct.

To process a DateTime property value, the XPropertySet object first ensures that the DateTime object supports this property name. So it iterates over the objects property map until the property is found. If the property exists then the property values are transferred to a SwFieldDateTime class object. A UNO DateTime object is equivalent to a SwDateTimeField[15] object of LibreOffice document model. A property entry for a DateTimeValue is listed as below

```
SW_PROP_NMID(UNO_NAME_DATE_TIME_VALUE), FIELD_PROP_DATE_TIME,
CPPU_E2T(CPPUTYPE_DATETIME), PROPERTY_NONE, 0
```

The handle value of a Property plays a very important role and is used through out the document model data structure creation because:

- Numerical comparisons are faster than string comparisons, so handle variable is used in place of property names, wherever required.

[13] sw/source/core/unocore/unomap.cxx

[14] svl/source/item/itemprop.cxx

[15] SwField is a base class for all text field classes

- Many of the text object properties are represented by a class called SfxPoolItem, in the memory. The SfxPoolItem is a base class and inherited by many of the sub classes that represent different properties. The LibreOffice defines a mapping[16] between the property handle values and the SfxPoolItem sub classes. So, a handle value is used to identify the PoolItems through out the document processing.

- A container called SfxItemSet holds an array of PoolItems. As a handle value is mapped to a PoolItem class, it is also used to specify a range of PoolItems that an ItemSet object can store.

One of the objectives of this research work is to preserve the UID attribute of all the XML elements. After understanding the concept of properties, we can say that we want to add a UID property to the property map of all the text objects. This is a first step towards preservation of UID through the load and save cycle.

An item to watch for - Every property of an object is identified by a unique name. For our purposes, we need to add the UID to every XML element. This can cause a naming conflict in a property map of some objects like paragraphs or tables because these elements nest many sub-elements inside. To avoid any naming conflict, each UID attribute name should be different in the property map.

[16] init.cxx

3.9 Important In-Memory Data Structures

The SwDoc class is the main container class for a LibreOffice document. It holds other containers to store different type of text contents.

The SwDoc class holds a vector container SwNodes to save all the SwNode objects, a vector that stores all the IMark objects and a Swphint array to hold text field objects and so on. Many of the character and paragraph property objects are stored in SfxItemPool container, which is a vector of SfxPoolItem vectors.

The store and access process of SfxPoolItem objects is complex. Here is a basic description of PoolItems and containers used to store the PoolItems.

1. SfxPoolItem: A character or a paragraph property, a reference mark, hyperlink and some other elements and attributes are represented by a SfxPoolItem object. It is a smallest unit that represents a property attribute in the document memory model. SfxPoolItem is a base class that is inherited by many sub classes representing specific properties. Most of the time, XML element attributes are stored as a collection of PoolItems. But some elements such as footnote, hyperlink and reference mark, create only one PoolItem object and store all the element attributes in their data members. We have used this idea to set UID in the PoolItem objects for such cases.

2. SfxItemSet[17]: The SfxItemSet container holds a set of PoolItems. This container has multiple usages:

[17] svl/source/items/itemset.cxx

- During load operation, it helps to hold the PoolItems temporarily and also assists in saving them to the larger Item Pool container
- ItemSets are also used to create a map data structure for style properties.
- They are also used to retrieve SfxPoolItems from the item-pool during document export process.

We have identified the important document model classes that are used to represent text content in the memory.

Table 3.1 provides a list of these classes.

Text Document Component	Document model Class
Text Paragraphs	SwNode
Text-Table	SwTable, SwTableLine, SwTableRow
Text Fields	SwField
Text-Field Master	SwFieldType
BookMarks	IMark
Footnote, EndNote	SwFmtFtn
Indexes	SwToxBase, SwForm
List item	SwNumberTree
Paragraph-Character properties	SfxPoolItem

Table 3.1: Document model classes for different text content types

Chapter 4

LibreOffice Version Aware Extensions

The version awareness extensions to the LibreOffice Writer application are provided in two phases. First phase was implemented by a UWM student B Nielson[15] where he was successful to preserve preamble (molhado namespace and revision-history element) information through the load-edit-save cycle of the content.xml. The second phase implements the preservation of UIDs for every XML element type of content.xml through load-edit-save cycle. This chapter describes the modification implemented for namespace, revision-history and UID preservation.

4.1 LibreOffice Modifications for Preamble

A Preamble includes a namespace and revision-history element. These are necessary version awareness properties for an XML document.

4.1.1 Molhado Namespace Addition

The first milestone towards getting LibreOffice to support version-awareness is simply getting the molhado namespace declaration to survive a document load and save cycle. This unique namespace is required to prevent the collisions with element and attribute names of the existing XML file.

Our goal is to add `xmlns:molhado="http://www.cs.uwm.edu/molhado"` as a new namespace entry in the content.xml and preserve it in the saved document.

As discussed earlier, in the load operation, the XML elements are processed in SvXMLImport::startElement method. Also, each namespace entry is saved in a map data structure. Further, based on the namespace entries, respective export flags are set. Finally, during save operation, the saved namespaces are exported back to XML document if a corresponding export flag is set to true for a namespace.

A simple approach for molhado namespace is to save it in the namespace map during load, initialize an export flag that should maintain its status till save operation. This way it can be ensured that the molhado namespace only gets exported to XML if the namespace entry is present in the input file.

It is easier to add a namespace into the existing map and can be performed during startElement method processing. But first, the xmltoken.cxx is updated for molhado token entries.

```
TOKEN( "molhado", XML_NP_MOLHADO ),
TOKEN( "http://www.cs.uwm.edu/molhado", XML_N_MOLHADO ),
```

Second, a uniqe namespace identifier is added for molhado namespace in xmlnmspe.hxx `XML_NAMESPACE( MOLHADO, 104U )`

We can utilize document properties and save options to set an export flag that indicates the presence of molhado namespace in the input file. Document property values are set during file load operation. Additionally, document properties can be used to change the value of export flags. Document properties do not necessarily persist through a load-save cycle as some properties only serve to change behavior of the file importer. But properties related to document saving are persisted through the use of save options.

To achieve the objective, UseMolhado property is added in the document property list and a corresponding save option variable. As the save options persist throughout the document lifecycle, it is possible to retrieve the saved values back during the export process.

Save options are implemented by SvtSaveOptions_Impl[1] class. Every save option has a corresponding Boolean variable in this class. The save option for the UseMolhado property is implemented by adding a bUseMolhado variable to the save option class with access methods.

Additionally, there is a SvtSaveOptions class defined in the same file, with the same methods as described above. The reason for the two matching classes is to enforce the singleton design pattern. The SvtSaveOptions_Impl defines the actual

[1] unotools/source/config/saveopt.cxx

implementation of the class responsible for handling save options, and the SvtSaveOptions class is a singleton, which contains an instantiation of the SvtSaveOptions_Impl class, as well as the necessary code for ensuring only one implementation class is instantiated. Corresponding methods for the UseMolhado save option must be added to both classes.

Next, a document property for molhado namespace is added in a property map - PropertyMapEntry in both swxml.cxx and wrtxml.cxx files. These file perform XML load and save operations respectively.

During load operation, this document property and save options are set to true only when molhado namespace is read. An XPropertySet object is instantiated that can access all the document properties for the document currently being imported.

During the export process, the UseMolhado property value is set by retrieving the value from the save option. The last step is performed in SvXMLExport::initialize() where the molhado namespace is added to the mpNamespaceMap if the UseMolhado property value is set to true. The EXPORT_MOLHADO flag is set to true and is used for revision history element export process as well.

4.1.2 Revision History and Sub-element Addition

The revision history stores the change information among multiple revisions in the form of reverse deltas. After the collection of revisions, a signature is attached to

preserve the data integrity. The revision information is represented by revision-history element. A “revision-history” element nests multiple “revision” elements and a single “signature” element. A “revision” element includes the sub-elements to hold the edit operation details like add, update, delete etc.

In general, LibreOffice application generates a SvXMLImportContext class object for every xml element. The context object invokes CreateChildContext method to create sub-element class objects. Later the element and attribute details are saved in the in-memory data structures. During edit operations, the in-memory objects can be modified by the application.

As a revision-history element does not need to be changed by the LibreOffice Writer application, so the objective is to just preserve this element during load and save. Thus, a revision-history element is not implemented in a similar way as other native XML elements are.

To handle the revision-history element, a new faux context class MolhadoHelper_Impl is defined. It does not create any other class objects for its sub-elements, rather it uses a list data structure to hold the subsequent elements. The class declares variables for revision-history element attributes and for signature values. All the getter and setter methods are defined as well. To store the nested “revision” elements, a list variable, named “seqRevisions” is declared. A revision element is represented by a struct, named Revision. This struct holds revision element attributes and a list of edit operation elements. An edit operation element is represented by a struct, named Operation. The Operation struct stores the operation name and two string

lists to store the attributes as name-value pairs.

AddRevision, AddOperation and AddOperationAttribute methods initialize the Revision, Operation and Operation attribute struct objects and save them in their respective containers.

The revision-history element processing starts as usual in the startElement method of SvXMLImport class. As the revision-history element is read, the attribute values are set in the MolhadoHelper object. When a nested revision element is read, a Revision struct is initialized and attribute values are set in the struct and an empty Operation list is initialized. This newly created Revision is inserted in the Revision list. Similarly when edit operation elements are read, an Operation struct is initialized and its attributes are set by AddOperationAttribute method. This method finds the last saved operations in the Operation list of last saved Revision and set the keys and values lists using these method calls.

```
seqRevisions.back().seqOperations.back().seqKeys.push_back(key);
seqRevisions.back().seqOperations.back().seqValues.push_back(value);
```

This completes the import process of revision-history element.

The MolhadoHelper_Impl is implemented as a singleton in the similar manner as described for the save options. Thus, during the export, the earlier created single instance of this class is accessed and the saved information is exported to the output XML file.

The export process takes place in the 'exportDoc' method of the SvXMLExport[2]

[2]xmlexp.cxx

class and the 'EXPORT_MOLHADO' flag, which is simply a bit flag toggled depending on UseMolhado value, is added. Using this new flag, the 'exportDoc' method is modified to call the ImplExportMolhado method to write the revision-history in the output XML file.

In the ImplExportMolhado method, the revision-history attribute values are retrieved from MolhadoHelper object and written to the output XML file. Then the revision list is iterated over to retrieve its attributes and edit operations, and these are written to the output XML file. A similar process is repeated for all the operations within a revision and their attributes are written to the output XML file.

4.2 Modifications for Molhado ID

The UIDs on elements are essential for identifying the XML nodes that have undergone substantial transformations and can still be matched with their original version, so the main focus of this thesis project is to preserve the UIDs through the load and save cycle of LibreOffice text documents.

A text document contains a large number of XML elements to represent the structure of the document. Some of the XML elements are paragraphs, fonts, textfields, index marks and tables. The wrapper application inserts UIDs for every XML element in content.xml. The LibreOffice Writer application must be modified to preserve the UID attributes which are otherwise unknown to the application.

Although, most Writer document components follow the general import-export

process as discussed earlier, there are many underlying non-uniformities in their implementation that make the code modification process challenging. Each document component has a complex life cycle because they are represented differently in their XML representation, the document model class representation, and finally, the export class representation.

Also, the UNO objects and understanding their working process adds to the challenge. The enormity of the source code along with a lack of comments makes the code change process difficult.

But the silver lining is that the al overl import-export process for all document component types is similar and with the knowledge of the existing XML import-export process and how the other XML element attributes are processed by the application, we can also process the UID attributes in a similar fashion with a small amount of code modifications.

In general, to preserve a UID of an element, code modifications should be made at 5 main key locations in the source code.

- During the XML load process, the UID attribute should be retrieved from the attribute list and saved in the import context object before further processing of the context.

- The UID name and value must be transferred from the import context objects to the document model data structures using UNO objects.

- The unique ID property name must be included in the property map of an

object.

- The document model base class representing each type of XML element must be identified and have support added for the ID property.

- During the XML save process, the respective export classes and methods must be identified for the elements and attributes, and we must modify the export code to support saving of UIDs.

4.2.1 Paragraph Molhado ID

Paragraphs are one of the most important building blocks of the LibreOffice Writer documents. In the XML document representation, paragraph elements can include many sub-elements such as text-fields, spans, bookmarks, frames, hyperlinks and sections. During the import process, each paragraph is represented by a XMLParaContext[3] class object. The attributes of the paragraph elements are processed during construction of the context object and the transfer of the saved attribute values to the in-memory document model is performed during destruction of the object.

A paragraph context object is stored in a context vector until all the sub-elements inside a paragraph are processed. When a nested element of a paragraph is read, the saved paragraph context object invokes the CreateChildContext method to create a corresponding child context object.

[3] txtparai.cxx

There are two sets of sub-elements inside paragraph, which are processed in different ways from each other. One set of sub-elements includes hyperlinks, spans, reference marks and bookmarks. The other set includes text-fields, sections and others.

For the first set of sub-elements, the child import context construction call creates a hint[4] object and stores all the sub-element attribute values in the hint object. This hint object is then pushed inside the hint vector[5] of the paragraph context object. During the paragraph context destruction call, first, paragraph attributes are transferred to the in-memory document model using UNO object methods. Second, all the saved hint objects are processed one by one and further method calls are made to save their attributes in the in-memory data structures. We will discuss individual sub-element processing in detail in next sections.

For the second set of sub-elements, such as text-fields, the child import context objects are created as usual and those context objects transfer the attribute values to the document-model first. Thus, these elements and their attributes are saved in the document model first, before paragraph element attributes are transferred to the in-memory data structures.

To support ID attributes in paragraph objects, the base context class SvXMLImportContext[6] is modified to hold a variable for IDs. The base hint class[7] is also

[4] XMLHint_Impl

[5] XMLHints_Impl

[6] xmlictxt.hxx

[7] txtparaimphint.hxx

modified to store the ID in the hint objects.

Import Context	Hint Vector	UNO Interface Class	In-memory Data Structure
XMLParaContext	XMLHints_Impl	SwXTextCursor	SwNode

Table 4.1: Paragraph import details

To process the saved paragraph ID attribute value, an XPropertySet reference is created by issuing a query to the XTextCursor interface. Next, the property set object is queried to see if the text cursors property map contains a property entry corresponding to the paragraph ID. Then, one more setPropertyValue call is invoked to transfer the paragraph ID attribute to the document model class objects. The UNO runtime environment helps the method call to reach its correct implementation defined in the "sw" module. The paragraph context destructor[8] method is modified as below.

In the document model, a paragraph is represented by a SwTxtNode class which is a derived class of SwNode[9]. All the node objects are stored inside a SwNodes vector of SwDoc class. The XTextCursor interface is implemented by SwXTextCursor[10] class.

For most of the cursor properties, a PoolItem object is created based on the handle value of the property entry. These PoolItem objects are saved inside an

[8] txtparai.cxx

[9] node.hxx

[10] unoobj.cxx

array of a SfxItemSet container defined for every SwNode class. But few of the cursor properties are saved directly in the node object itself. For UID property, we can use this approach and save the ID value directly in the node object itself. The base class SwNode is modified to add an ID variable and its access methods.

During the save process, the text export class object is initialized and in the exportParagraph method, the paragraph element attributes are retrieved from the in-memory data structures with the help of UNO objects[11]. The XPropertySet object invokes getPropertyValue (attribute name) method call to get a saved property value. With the help of URE, this call is directed to the method implementation provided by the SwXParagraph class. This call further invokes a call to getCrsrPropertyValue[12] helper method and retrieves the specific property value. For UIDs, the same process is followed and the code modifications are done to retrieve the saved ID value from the SwNode object.

Export Context	Export Interface	Export method
XMLTextParagraphExport	SwXParagraph	exportParagraph

Table 4.2: Paragraph export details

The molhado ID property is exported to the output XML file in the similar fashion as discussed in the XML export section.

[11] txtparae.cxx

[12] unocrsrhelper.cxx

4.2.2 HyperLink Molhado ID

In the XML document representation, a hyperlink element is always a sub-element of a paragraph element. So, before a hyperlink object is read, a paragraph context object is already present in the context vector. To process the hyperlink element, the paragraph context creates a child context object for the hyperlink first.

During the construction of a hyperlink context object, a hint[13] object is initialized that stores all the hyperlink element attributes and is inserted into the hint vector of the parent paragraph context object.

During destruction of the paragraph context object, first, paragraph properties are set in the document model and then each saved hint object is processed. Depending on the hint type, a corresponding method is invoked with the hint attribute values as arguments. This method processes the attributes with the help of UNO objects and saves them in the in-memory class objects.

For a hyperlink, a SetHyperLink[14] method is invoked with all the attribute values as arguments.

The SetHyperLink method creates an XPropertySet reference by querying the cursor. This object sets the attributes using the setPropertyValue (attr name, attr value)) method.

For the ID attribute processing, changes have been introduced into the hyperlink context constructor to save the ID in the hint object, in the paragraph destructor

[13] XMLHyperlinkHint_Impl

[14] txtimp.cxx

Import Context	Hint Class	UNO Interface Class	In-memory Data Structure
XMLImpHyperlink-Context_Impl	XMLHyperlink-Hint_Impl	SwXTextCursor, SfxItemSet	SwFmtURL

Table 4.3: Hyperlink import details

to process the hint ID and in SetHyperlink to call the setPropertyValue for ID attribute value. The hyperlink ID property is added in the cursor property map and is identified as A_Id. The cursor interface is implemented by the SwXTextCursor class in the LibreOffice environment[15].

The setPropertyValue call goes through following steps to save the property values in the in-memory data structures.

1. It create an empty ItemSet to hold the PoolItems corresponding to the attributes of a hyperlink element.

2. It calls GetCrsrAttr method to get any existing PoolItem for the current hyperlink object, at this cursor location. Based on the cursor position, a saved SwTxtNode object is accessed. This object represents the current paragraph object. This paragraph object is queried to find any hyperlink PoolItems saved in its ItemSet container. If any PoolItem is found, then the PoolItem is retrieved and saved in the new ItemSet container initialized in the previous step.

[15] unoobj.cxx

3. The actual implementation of setPropertyValue method for hyperlink properties is found in the SfxItemPropertySet class[16]. Then a PoolItem of SwFmtURL[17] type is created when the first hyperlink property value is set. For the subsequent hyperlink properties, the saved object will be retrieved and the property value will be set in the same object.

4. Finally, the SetCrsrAttr method saves the created PoolItems of the aItemSet in the SwDoc's item-pool. Subsequently, it calls the InsertItemSet and then the lcl_InsAttr methods of SwDoc class to store the PoolItems.

To handle molhado ID for the hyperlink element, the base PoolItem class SfxPoolItem is modified for a new ID variable and its access methods. Also, the SfxItemPropertySet methods are modified to set and get the ID values.

During the export process, the addHyperlinkAttributes method processes all the hyperlink attributes. With the help of the getPropertyValue method, the ID attributes can be retrieved too. This method retrieves the properties from a text portions interface. Then ID attribute value is exported to the output XML.

Export Context	Export Interface	Export Method
XMLTextParagraphExport	SwXTextPortion	exportTextRange

Table 4.4: Hyperlink export details

[16] svl/source/items/itemprop.cxx

[17] atrfrm.cxx

4.2.3 Text-Field Molhado ID

In the LibreOffice Writer application, a text-field object adds additional information to the surrounding text. There are a large number of text-fields supported by the Writer application such as: author, date, page number, cross-reference, bibliography fields and database fields.

Broadly, there are two groups of text-fields. The first group of text-fields contains their own data and are attached to a text range object. The second group of text-fields does not have their own data and are dependent on their text-field-master objects for their data. For example a bibliography text-field gets its contents from a bibliography field-master that fetches the data from the bibliography index. We will discuss the first set of text fields in this section.

In the XML document representation, a text field is a sub-element of a paragraph element. A base import context class XMLTextFieldImportContext represents all the text field elements. This context class implements the CreateTextFieldImport-Context[18] method that creates the individual context class objects for each type of field element.

During the import process, first the base import context object is instantiated, which creates the specific context objects. Then the text-field element attributes are processed and saved in the field context object.

A TextField service supports the XTextField interface which is implemented by

[18] txtfldi.cxx

the SwXTextField[19] class in the LibreOffice environment. The TextField service also supports the XPropertySet interface. The ServiceManager supports a UNO service for every text-field object, e.g. for the chapter text-field a chapter[20] service is supported.

The attribute transfer to the in-memory data structures is performed in two steps inside the EndElement method of the context class. First, the PrepareField method invokes the setPropertyValue method calls to set all the attribute values in the memory. This process saves all the attribute values in a temporary struct SwFieldProperties.

Next, the text-field is attached to a text range object. In this process, a SwField object is created by retrieving all the saved property values from the earlier created struct. Finally, a SwFmtFld PoolItem object is initialized using the field object as an argument. Finally, this PoolItem is saved in the documents item-pool by the InsertPoolItem method.

The following code changes are performed for processing text-field molhado ID.

- The molhado ID value is set to a text-field import context object.
- During the EndElement method processing, a setPropertyValue method call is invoked to set ID value in SwFieldProperties struct. This struct is modified to store the molhado ID.

[19] unofield.cxx

[20] com.sun.star.text.textfield.chapter

- The attach method is modified to add the ID value in the SwField object. SwField[21] class is updated for an ID variable and its access methods.

- In the constructor of SwFmtFld, a SwField::CopyField() method is called to copy the input field attributes to a new field object. So CopyField is updated as well to copy the field ID value.

Import Context	UNO Interface	In-memory Data Structure
XMLTextFieldImportContext	SwXTextField	SwField, SwFmtFld

Table 4.5: TextField import details

The property map for all the text-fields should also be modified to include the ID property. In unomap.cxx, a macro COMMON_FLDTYP_PROPERTIES identifies all the common properties of all the text-field objects. So a molhado ID property is added in this macro which also adds the ID property for every text-field object.

The export process starts in the ExportField method of XMLTextFieldExport class that further calls tje ExportFieldHelper and exports all the text field elements and attributes. The getPropertyValue method of the SwXTextField interface is modified to retrieve field molhado ID as well.

[21] sw/source/core/fields/fldbas.cxx

Export Context	Export Interface	Export Method
XMLTextFieldExport	SwXTextField	ExportField , ExportFieldHelper

Table 4.6: Textfield export details

4.2.4 Text-Field-Master Molhado ID

The text-field-master components provide data for dependent text-fields such as User, Database, SetExpression, Bibliography and DDE. A dependent text-field should be attached to a field-master before adding the text-field to the document. In this section we will discuss dependent text-fields and text-field-master element processing through the load-edit-save cycle.

During the import process, XMLVarFieldImportContext class represents the dependent text-field elements, which is a base import context class for all the variable related fields. The import process is similar to that of the other text-fields processing except, in the EndElement method, the dependent text-field is added to its field-master by the attachTextFieldMaster method and then the text-field object is attached to the document. At the end, the text-field properties are set in the field objects that are stored in the memory.

During the import process, the XMLVariableDeclImportContext[22] class represents a text-field-master element. The field-master import process starts during the construction of this context class object. First, the common attributes of field masters are set in the data members of this context object. Second, a UNO object

[22]txtvfldi.cxx

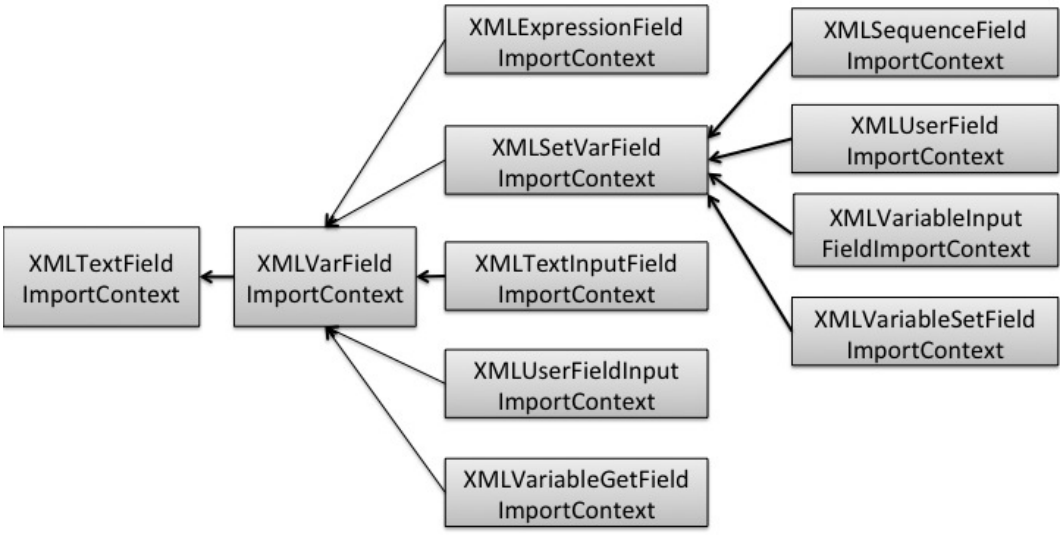

Figure 4.1: Field master class hierarchy

corresponding to the field-master is created in FindFieldMaster method by invoking the respective field-master service. Once the master object is created, the attribute values are set in the document model by the setPropertyValue method.

The entire sequence of attribute processing and setting the attribute values in the document model is performed in the constructor call. For the molhado ID, the ID attribute is read from the XML attribute list and saved in a local variable. Once the field master object is created, the ID value is transferred to the document model by setPropertyValue method.

Because a related text-field is also created for a field-master, we have to save the

ID of the dependent text-field element as well. This is performed in EndElement method of XMLSetVarFieldImportContext class.

Import Context	UNO Interface Class	Data Structure
XMLVarFieldImportContext	SwXTextFieldMaster	SwFieldType

Table 4.7: TextFieldMaster import details

The SwXFieldMaster class implements a text-field-master interface. A field-master element is represented by the SwFieldType class in the document model and holds all the master properties. The SwFieldType class is thus a good place to save the molhado ID value as well and is modified to hold an integer variable and its access methods. All the field masters are inserted in the document by the InsertFldType method that stores all the field types in a field type vector.

The field master export process starts in the ExportFieldDeclarations method of the XMLTextFieldExport class. The molhado ID for each master is retrieved by the getPropertyValue method and is added in every exported master XML element.

4.2.5 ReferenceMark and BookMark Molhado ID

The reference marks and the bookmarks allow a user to jump to a label in a document. The LibreOffice Writer application supports two methods of using a reference mark or a bookmark. In the first case, a user inserts a mark at some location without selecting any text in the document. In the second case, a user selects a text range and inserts a mark in the document. Based on these methods, marks are

represented in two ways in an output XML file.

For the first case, a “referencemark” element is created which only has a name attribute. For the second case, two elements, a “reference-start” and a “reference-end”, are created that hold the selected text in between the two elements. The bookmarks use the same approach.

The import and export processes are similar for these elements. During the import process - XMLTextMarkImportContext class represents reference mark, bookmark and field mark elements. These elements support only a name attribute. Because the reference mark and the bookmark services don’t support the XPropertySet interface thus they don’t provide the implementation of the setPropertyValue and the getPropertyValue methods. For our purposes, the XNamed interface method setName is used to combine the ID value with the mark name, the ID is later separated from the name in the document model.

The document model representation is different for both the marks. A reference mark is saved as a PoolItem, while a bookmark is saved as an IMark object in a mark vector of a document.

Reference Mark and Molhado ID

The two types of reference mark elements discussed above have slightly different import process. For a “referencemark” element, an import context class object is created that processes the attribute in StartElement and EndElement methods.

The start method saves the reference name in the context object and the end

method invokes the CreateAndInsertMark method to transfer the name and attribute value to the document model class.

For the "reference-start" and "reference-end" elements, a corresponding context object is created first, where the context object initializes a hint[23] object and sets the name attribute value in the hint object. The hint object in inserted in the paragraph hint vector. During the paragraph destructor call, the reference hint object properties are passed on to the CreateAndInsertMark method of the context class for further processing of the attribute.

In the first case, the molhado ID is saved in the context object as usual after object creation. In the second case, code modifications are done to handle the ID value in the respective context classes[24]. The CreateAndInsertMark and EndElement methods of XMLTextMarkImportContext class are modified to process the ID attribute value.

Inside the CreateAndInsertMark method, a ReferenceMark service object is created and queried to create an XNamed interface object. The saved ID value is concatenated with the reference name and passed as an argument to the setName method. Once the mark object is created in the memory and the attributes are set, then the mark object is attached to the text object by invoking the attach method.

The reference mark interface is implemented by the SwXReferenceMark[25] class. In the setName method, the ID value is separated from the name value and saved

[23] XMLReferenceHint_Impl

[24] txtparai.cxx

[25] unorefmk.cxx

Element	Import Context
reference-mark	XMLTextMarkImportContext
reference-mark-start	XMLStartReferenceContext_Impl
reference-mark-end	XMLEndReferenceContext_Impl

Table 4.8: Reference mark import details

in a new data variable added to this interface class. After the attribute values are set, the attach method is invoked to save the reference object in the memory.

During this process, a SwFmtRefMark PoolItem is initialized and the previously saved attributes are transferred to this object. This PoolItem is then inserted in the document by InsertPoolItem method, which internally saves the PoolItem in the SwpHintsArray, the in-memory data structure.

The reference mark ID property is also added in the text portion's property map. The ID property does not play any role during the import process but it is useful during the export process to identify the ID property of a text portion created for the reference mark.

During the export process, the reference mark is returned as a text portion after iterating over all the saved references. The exportParagraph method invokes the createEnumeration method of the SwXParagraph class, which creates a container[26] that holds all the text portion[27] objects.

[26] unoport.cxx

[27] SwXTextPortion

The lcl_CreatePortions[28] method creates different portions of a document. Further, lcl_CreateRefMarkPortion method is invoked to retrieve the saved "SwFmtRefMark" object, which is then stored in a new text portion object. The SwXTextPortion class is modified to save the ID value and also the getPropertyValue method is modified to retrieve the ID value from the PoolItem.

In the exportTextMark method, a getPropertyValue call is invoked to retreive the ID value from the text portion object and this value is exported to the output XML file.

BookMarks

The import process for bookmarks is similar to the process used for reference marks except that all three of the elements (bookmark, bookmark-start and bookmark-end) are processed in a manner similar to that used for the XMLTextMarkImportContext context class object. The ID value is concatenated with the bookmark name attribute and passed on to the document model using the setName method. Once a bookmark object is created in the memory and the attributes are set, then the mark object is attached to the text object.

In the in-memory document model, the bookmark interface is implemented by the SwXBookmark[29] class. In the setName method, the ID value is separated from the bookmark name and is stored in a new data variable of the SwXBookMark class.

[28] unoportenum.cxx

[29] unobkm.cxx

Later, the insertTextContent method calls the attach method of the SwXBookMark class, which attaches the bookmark to the document.

The IMark class is the base class to save bookmarks in the document model. The IMark class is modified to support the ID value and its access methods. During the attach method call, the bookmark ID is saved in the IMark object. The IMark objects are stored in a vector container of the SwDoc class. A bookmark ID property is also added in the text portion's property map.

The export process is the same as that for reference marks. The lcl_CreatePortions method creates the bookmark portions by invoking lcl_ FillBookmarkArray method. This method iterates over the saved IMark vector to retrieve the bookmarks.

In the exportTextMark method, the getPropertyValue of the SwXTextPortion class is invoked to get the ID value and is then exported to the output XML file.

4.2.6 Indexes and Index Mark Molhado ID

Indexes

LibreOffice indexes are the text contents that pull together information that is dispersed all over the document. They can contain chapter headings, locations of arbitrary index marks, or locations of text objects, such as illustrations or bibliography.

LibreOffice indexes include 7 main indexes - the alphabetical index, the table of contents index, the user defined index, the illustration index, the object index,

the text table index and the bibliographical index. These indexes are structured by levels and the number of levels are different for each type. For example a TOC index has 10 levels that correspond to the chapter numbering levels, a bibliography has 22 levels, and so on.

All of these indexes have similar XML element structure and the load-edit-save operations follow a similar process, too. This section describes the code modifications performed for all the indexes with the help of TOC index element structure. Figure 4.2 shows the TOC index element structure and will help us to understand how the LibreOffice Writer application processes these elements during the load-edit-save cycle.

Every index element has two sub-elements "source" and "index-body". The source element has an 'index-tittle-template' sub-element that stores the tittle of the index. Also, there are ten entries of 'table-of-content-entry-template' sub-element which correspond to the ten levels of the TOC index. Every level entry element should contain at least one of the sub-element entries mentioned in the example.

During the import process, every index XML element is represented by a context class object. Table 4.9 lists all the TOC index elements and their corresponding import context classes.

Most of the index elements follow a similar import process. First, a context object is created for an element and then the attribute values are set in the object. Next, a corresponding UNO service object is instantiated and an interface object is queried. Finally, with the help of the setPropertyValue method, the attribute

```
<text:table-of-content .... >
    <text:table-of-content-source .... >
        <text:index-title-template .... >
            Table of Contents
        </text:index-title-template >
        <text:table-of-content-entry-template .... >
            <text:index-entry-link-start ...../>
            <text:index-entry-link-start ...../>
            <text:index-entry-chapter ...../>
            <text:index-entry-tab-stop ...../>
            <text:index-entry-page-number ...../>
            <text:index-entry-link-end ...../>
        </text:table-of-content-entry-template>
    </text:table-of-content-source>
    <text:index-body .....>
        <text:index-title ..... />
    </text:index-body>
</text:table-of-content >
```

Figure 4.2: TOC Index element structure

values are transferred to the in-memory data structures. The code modifications are performed at multiple locations to preserve the ID values.

Some of the index sub-elements follow specific operations to process their attributes and are listed below.

- The context classes corresponding to the "source" element of all the index types, are derived from the base class XMLIndexSourceBaseContext. Once the attribute values are saved in the context object, the EndElement method transfers the attribute values to the in-memory data structures. Towards the end of this method call, the base class EndElement method is also invoked to

Element	Import Context
table-of-content	XMLIndexTOCContext
table-of-content-source	XMLIndexTOCSourceContext
index-tittle-template	XMLIndexTitleTemplateContext
table-of-content-entry-template	XMLIndexTemplateContext
index-entry-link-start	XMLIndexSimpleEntryContext
index-body	XMLIndexBodyContext

Table 4.9: TOC index import details

transfer the common attributes among all the index source elements. We can make use of this step to transfer the ID values of the source element of all the index type, so that the code changes at one location will handle the ID attribute of the source element of all the 7 indexes.

- A content-entry-template element is represented by the XMLIndexTemplate-Context class which holds a vector of a PropertyValues type. As per the earlier example, a content-entry-template element can have multiple index-entry sub-elements that are represented either by a XMLIndexSimpleEntryContext or its sub classes. For every attribute name-value pair of an index-entry element, a PropertyValue type object is created and is stored in a PropertyValues object, which is a vector of a PropertyValue type. Finally this PropertyValues object, corresponding to every index-entry element, is inserted in the PropertyValues vector of the parent template context object.

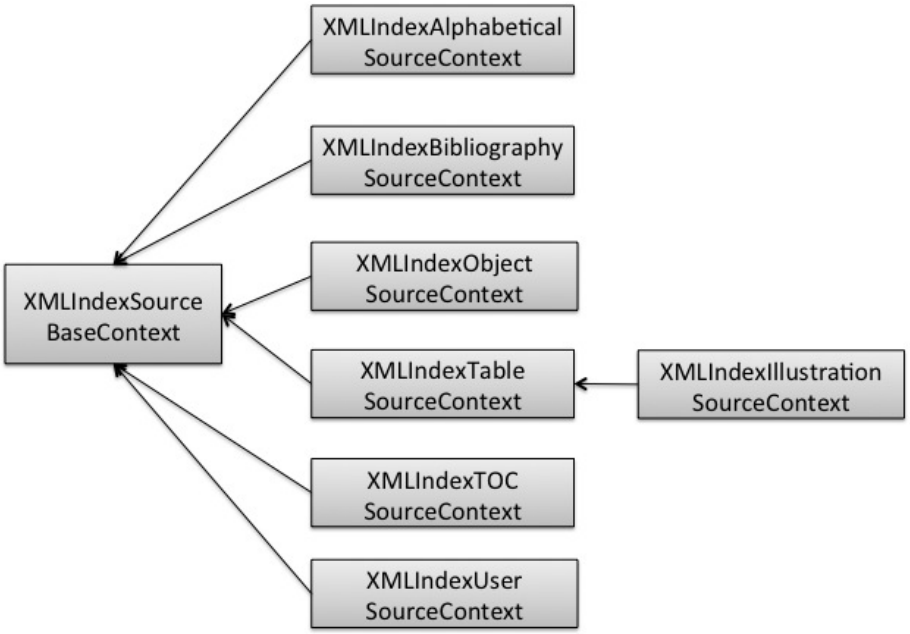

Figure 4.3: Index content source class hierarchy

In the document model, the index interface is implemented by the SwXDocumentIndex[30] class and the indexes are represented by an SwTOXBase class object. The setPropertyValue method saves the index element attributes in this object. For every index table supported by the LibreOffice Writer, the XML element structure is static and have fixed number of sub-elements, thus the ID attribute values of such elements can be stored in a fixed sized array inside the SwTOXBase[31] class. We have created an array of size four that includes the ID values of the main index, source-content, index-tittle and index-template elements. The rest of the elements

[30] unoidx.cxx

[31] tox.cxx

IDs are stored differently in the memory.

During the import process, all the attribute information of the index-entry elements is stored in a PropertyValues vector and is processed by the replaceByIndex method of the index interface. The vector size depends on the indexing levels supported by an index table e.g. 10 for TOC, 4 for alphabetical indexes etc.

An index-entry element is represented by an SwFormToken object in the document model. During the import process, the index-entry element attributes were stored in a PropertyValues object. Now, the saved data is extracted from each PropertyValue object and is stored in the data variables of the SwFormToken class. This class is modified to save the molhado ID as well.

All the SwFormToken objects are stored in a vector data structure inside an SwForm class. Once all the PropertyValue entries are processed, the SwForm object is set inside the SwToxBase object.

The index ID properties are also added to the property map of every index object.

The above process describes the creation of index object in the memory, which is represented by a SwToxBase class object. Next, this index object is attached to the document by the attach method, which inserts this base object into the document containers.

The export process starts in the XMLSectionExport class methods such as ExportTableOfContentStart, ExportObjectIndexStart and so on. These methods invoke the getPropertyValue method to retrieve the saved attribute values from the

in-memory data structures and then export them to the output XML file. The getByIndex and the getPropertyValue methods of the SwXDocumentIndex interface are modified to retrieve the saved index IDs. Similarly, the ExportBaseIndexSource method is modified to export the source element ID value and the ExportIndexTemplateElement method is modified to export all the other template element IDs. Modifications are done individually for every type of index table to export the ID attribute values.

Index-Marks

Index marks are the text contents whose positions and contents are collected and displayed in the indexes. The LibreOffice Writer application supports three types of index marks: DocumentIndexMark, UserIndexMark and ContentIndexMark.

During the import process, the XMLIndexMarkImportContext_Impl context class represents all the index mark elements. All the attributes are set in the context object and later transferred to the document model classes with the help of the setPropertyValue method.

The index mark interface is implemented by the SwXDocumentIndexMark class. An index mark element is represented by an SwTOXMark PoolItem in memory. The setPropertyValue method saves all the properties in this object. Later, the attach method inserts this object in the document containers.

The export process starts in the ExportIndexMark method of the XMLIndexMarkExport class. Here the getPropertyValue method is invoked to retrieve the

saved attribute values from the in-memory data structures.

Code modifications are performed at multiple locations to preserve the index mark element IDs.

4.2.7 Text Table Molhado ID

LibreOffice Writer text tables consist of rows where every row consists of one or more cells, and a cell can contain more text or rows.

During the XML import process of the main table element, a table context object is created first. A table context object has a vector that stores the row context objects and every row context object has a vector to store the cell context objects[32]. During the table context object construction, the following steps are performed by the application.

1. It processes the table element attributes and stores them in the local variables.

2. It creates a TextTable service instance and initializes an XTextTable reference. The XTextTable interface is implemented by the SwXTextTable[33] class.

3. It initializes the default number of rows and columns of a table.

4. A text table is represented by a SwTableNode class in the document model. This class object holds a pointer to a SwTable class object which stores all the table information. The attachToRange method of the SwXTextTable class

[32]import file /sw/source/filter/xml/xmltbli.cxx file

[33]unotbl.cxx

inserts the SwTableNode object in the SwNodes vector of the document object. A SwTable class has two vectors: SwTableLines and SwTableBoxes to store the rows and cells. The SwTable also holds a pointer back to the SwTableNode object. The main table element molhado ID can be stored in the SwTableNode object as the base class SwNode is already modified for saving the ID.

The table 4.10 lists all the table elements, their context classes and the in-memory classes.

Element	Import Context	In-memory Data Structure
table	SwXMLTableColContext_Impl	SwTable
table-row	SwXMLTableRowContext_Imp	SwTableLine
table-cell	SwXMLTableCellContext_Impl	SwTableBox
table-column	SwXMLTableColContext_Impl	ColumnWidthInfo

Table 4.10: Text table details

After the table element is processed and the respective in-memory data structures are instantiated, the respective import context classes are created for its sub-elements and the element attributes are saved. The table context class methods, InsertRow and InsertCell, insert the saved row and cell context attribute values to a new set of implementer class objects named as SwXMLTableRow_Impl, SwXMLTableCell_Impl respectively.

To preserve the molhado IDs for every row and cell element, first, the import

context classes are modified to save the ID values, then the InsertRow and the InsertCell method definitions are changed to pass the molhado ID to a new set of class objects for the rows and the cells. Similarly, the previously mentioned two new set of implementer classes are modified as well to save the molhado IDs.

When the end marker of the table element is encountered, the EndElement method of the table context class is invoked, which further invokes different methods to fill the in-memory data structures for the table. The methods such as MakeTable, MakeTableLine and MakeTableBox are invoked to save the table, table-row and table-cell attributes into the earlier initialized classes - the SwTable, the SwTableLine and the SwTableBox. The implementer class objects provide the input data for these in-memory data structures. The SwTableLine and the SwTableBox classes are modified to hold the molhado ID values for the table-row and the table-cell.

This marks the end of import process and in-memory data structure creation for all the table elements. The export process is carried out by the exportTable[34] method which invokes the ExportTable, ExportTableLine and ExportTableBox methods to export different attribute values. These methods are modified to retrieve the stored molhado IDs from the respective in-memory data structure.

[34] /sw/source/filter/xml/xmltble.cxx

Chapter 5

Scope of Changes

The objective of this project was to preserve the version-aware information through the load-edit-save cycle of the LibreOffice Writer document. To achieve this objective, we have analyzed the Writer application source code and made many changes to the source code.

The "molhado" namespace and the "preamble" information were successfully preserved in the earlier project by Nielson[15]. This thesis work has shown that the "unique identifiers" can be preserved for the important text components that account for the most of the user data. The LibreOffice Writer source code has been modified for the import context classes, the internal document model classes and the export context classes. Although, the code modification process is mechanical the identification of the locations for the code modification was a very time consuming and tedious process.

- To support the VAD properties, a total 128 files were modified out of 3354 files. The main modified modules are xmloff, sw and svl.

- There are 1,064,700 loc present in the three affected modules out of which only 3200 loc was added for VAD support. These changes account for only .3 percent of additional source code.

Component	C++ Interface Class	Implementation Files
Text Paragraphs	SwXTextCursor SwXParagraph	unoobj.cxx unoparagraph.cxx
Text Field	SwXTextField	unofield.cxx
Field Master	SwXTextFieldMaster	unofield.cxx
Reference mark	SwXReferenceMark SwXTextPortion	unorefmk.cxx unoport.cxx
BookMarks	SwXBookMark SwXTextPortion	unobkm.cxx unoport.cxx
Indexes	SwXDocumentIndex	unoidx.cxx
index-marks	SwXDocumentIndexMark	unoidx.cxx
Para-text properties	SwXTextCursor	unoobj.cxx
Text-Table	SwXTextTable	unotbl.cxx

Table 5.1: Modified interface classes

The identification of the UNO interface implementation classes was one of the challenges of this project. The affected interfaces are listed above inTable 5.1.

As our aim was to support version awareness in LibreOffice Writer with the least amount of source code modification, thus it was important to identify the core

classes that were responsible for document content handling in the memory. Most of the text elements are now able to save the molhado IDs. Table 5.2 lists the modified base context classes and base document model classes.

Document Components	Modified Base Class	# Of Sub Classes
Import Context	SvXMLImportContext	190
Text Paragraphs, Text-Tables	SwNode	10
Paragraph-Character properties	SfxPoolItem	66
Text Fields	SwField	35
Text-Field Master	SwFieldType	33
BookMarks	IMark	12
Indexes	SwToxBase, SwForm	None
Text-table-row, Text-table-cell	SwTableLine, SwTableRow	None

Table 5.2: Modified in-memory base class details

The downside of VAD is the increased file size. The content.xml file size increases with the addition of the revision-history and the unique IDs for the text elements. Also, the file size will keep on increasing with every additional revision, as the incremental revision-history will be calculated and stored in the file itself.

Chapter 6

Future work

Because of the work described in this thesis, most text components of a LibreOffice text document are able to preserve the molhado unique IDs during the load and save cycle of the document. But there are other important elements such as font styles and automatic styles that will need ID preservation as well. An effort was made to understand the working process of these elements but it was not completely successful and leaves scope for future work.

6.1 Automatic Style Molhado ID

This section provides a glimpse of the load and save opertions of the “office:automatic-styles” element, its sub-elements and their attributes. Automatic style have a number of sub-elements such as “style:style” and “number:style”.

During the XML Import process, automatic-styles are represented by a SwXMLStylesContext_Impl[1] class which inherits from the base style class SvXMLStylesContext.

The style:style sub-element of the auto-styles element, is represented by a base class SvXMLStyleContext object and these are stored in a vector inside the parent auto-style context object. Some of the style families are text, table, number and page. We can save the molhado IDs in these import contexts easily.

6.1.1 Text and Paragraph Styles

The sub-elements of a style element are text and paragraph family styles such as "text-properties", "paragraph-properties" and "section-properties". During the import process, a SvXMLImportPropertyMapper object is created for these elements, where all the element attribute values are saved.

Then all the parent style objects are stored in a style vector as mentioned above. Once the parent automatic-styles element's end tag is read, the EndElement method of the auto-style context is invoked. All the saved text and paragraph properties are then transferred to the in-memory data structures.

Some of the style interfaces are XAutoStyle, XAutoStyles, and XAutoFamily, which are implemented by SwXAutoStyle, SwXAutoStyles and SwXAutoStyleFamily[2] classes, respectively.

[1] sw/source/filter/xmlfmt.cxx

[2] unostyle.cxx

The insertStyle method of the SwXAutoStyleFamily class initializes a SfxItemPropertySet object depending on the style family name and creates a SwAttrSet object and fills this ItemSet object with the help of the setPropertyValue method. Every property has a corresponding SfxPoolItem object that is stored inside the ItemSet.

To save the molhado IDs for paragraph and text property elements, a "SwStyleMolhadoId" class is defined. Also a property handle value RES_MOLATR_MID(57) is defined in the hintids.cxx file that maps to this ID property. The molhado ID PoolItem object can be initialized inside an ItemSet corresponding to the paragraph or character properties, by adding the ID property handle value as an argument in the ItemSet constructor call.

Once all the PoolItems are initialized, the ItemSet object is inserted in the document. This insertion process is complicated and uses complex data structures and algorithms to access the properties. Further understanding of the SwStyleManager, the StylePool and the Node classes is required to correctly save and retrieve the ID property values for the style elements.

The Export process takes place inside the _ExportAutoStyles method of the SwXMLExport class, which calls collectTextAutoStylesOptimized method of the XMLTextParagraphExport class to collect the saved styles by iterating over all of the saved styles. The createEnumeration method of the SwXAutoStyleFamily class invokes the getAllStyles method of the SwStyleManager class, which retrieves all the saved ItemSet styles from the StylePool objects. These methods need more

investigation to correctly retrieve the ID property values.

6.2 Span Molhado ID

A span element is a sub-element of a paragraph element and it marks the changes in the formatting of the text. A span element holds only a style name attribute. The span elements are handled differently for plain text style formatting and text-fields style formatting. some of the cases are:

1. A paragraph may have only plain text with one type of formatting, thus a single span covers the complete paragraph text.

2. A paragraph has split spans covering either the plain text or the text fields.

These two cases are handled differently at the document model level. In the first case, a StylePool class object is used to store the data in memory, while in the second case a SwpHints array is used to store the data. But, both cases use the same import process.

The import process of text:span element is similar to that for the other paragraph sub-elements. For a span element, a XMLImpSpanContext_Impl context class object is created that internally initializes a XMLStyleHint_Impl object, which is stored in the paragraph context hint vector.

In the paragraph context destructor, a call to the SetStyleAndAttrs method is invoked to set the style name attribute of the span element. The cursor's setPropertyValue method sets the properties in the in-memory data structures.

During the import process, the auto-styles and its sub-elements are always processed before any text element so, the auto-styles information is already saved in the in-memory data structures. During text span element processing, the lcl_setAutoStyle method retrieves the saved style ItemSet object from the auto-style pool. Next, a new SwFmtAutoFmt PoolItem object is created using this ItemSet. Again, this new PoolItem is inserted back in the document.

When multiple spans exist in a paragraph, a SwpHints array object is used to store these PoolItems. But, for a single span element, the earlier created PoolItem is unboxed and the style ItemSet is retrieved from it and saved back in the paragraph auto-style pool again.

The molhado IDs for multi-spans can be saved easily by setting the ID in the SwFmtAutoFmt PoolItem. But, for the single span element, the saved PoolItem ID value is lost because only the SfxItemSet part of PoolItem is processed by the current process.

One approach is to create a new container similar to SwpHints and store every span element (single or multiple) directly there. This approach can only work when the users don't change the existing spans. The other concern is, lack of understanding of the StylePool class object processing and how this new approach will affect the existing process. So, further investigation is required to preserve the molhado

IDs for single span elements.

6.3 Other Elements

The Font styles, Shapes, Redline and Ruby are other text components where uniqe IDs are not yet being preserved. The table style IDs are working correctly when the table has no merge cells, but for merged cells, the ID export process exhibits some anomalies and needs further investigation. Some of the other text elements such as “text:s” “text:h” and “table-column” also need some efforts to preserve the ID values.

Chapter 7

Conclusion

A Version Aware Document has the potential to implement full version control functionality in office applications. A version aware LibreOffice document will contain a complete change history and will be able to undergo 3-way XML merging and conflict resolution so that document collaboration and management will be possible without the use of a conventional version control repository.

With this purpose in mind, we have worked in this project to convert LibreOffice Writer documents into Version Aware Documents (VAD). To convert a native XML document to a VAD, three key features are inserted in the native XML file: a molhado namespace; a preamble element that stores the secure change history of a document; and unique identifier attributes for every XML element.

Unfortunately, LibreOffice applications do not automatically preserve these additions to their XML files through a complete load and save cycle so, further analysis of the XML import and export processing was performed to understand the working of the existing system. Based on this analysis, the source code of the Writer

application was modified at multiple locations to preserve the VAD properties.

This thesis work has shown that the version aware properties can be preserved in the LibreOffice Writer documents with relatively few modifications of the LibreOffice source code. A general code change pattern has been identified for different document component processing that provides a roadmap for the code modification for the rest of the LibreOffice applications. It has also been established that, although the modification process is mechanical, the identification of the import, the document-model and export classes that must be modified, is time consuming and tedious process.

Finally, this research work shows that it is possible to create a version aware LibreOffice document and it is possible to provide many of the sophisticated features of modern software version control systems in a context designed for less sophisticated users. The version aware document approach integrates easily with office document systems because it is designed to work with the XML representation that those systems have already accepted. Thus, full-blown branching and merging can be accessible to non-technical users working on everyday documents.

Bibliography

[1] "Subversion." http://subversion.tigris.org.

[2] "Mercurial scm." http://mercurial.selenic.com/.

[3] "Git version control." http://git-scm.com/.

[4] "Extensible markup language." http://www.w3.org/XML.

[5] M. Pandey and E. V. Munson, "Version aware libreoffice documents," in *in Proceedings of the 13th ACM symposium on Document engineering*, DocEng 13, (Florence, Italy), ACM, 2013.

[6] "Gnu diff3." http://www.gnu.org/software/diffutils.

[7] C. Thao and E. V. Munson, "Version-aware xml documents," in *in Proceedings of the 11th ACM symposium on Document engineering*, DocEng 11, (New York, NY, USA), pp. 97–100, ACM, 2011.

[8] C. Thao and E. V. Munson, "Using versioned tree data structure, change detection and node identity for three-way xml merging," in *Proceedings of the 10th ACM symposium on Document engineering*, DocEng '10, (New York, NY, USA), pp. 77–89, ACM, 2010.

[9] "Wikipedia." http://en.wikipedia.org/wiki/LibreOffice.

[10] "Apache openoffice developer's guide." https://wiki.openoffice.org/wiki/Documentation/DevGuide/OpenOffice.org_Developers_Guide.

[11] "Openoffice text services." http://www.openoffice.org/api/docs/common/ref/com/sun/star/text/module-ix.html.

[12] "Libreoffice xmloff module details." http://docs.libreoffice.org/xmloff/html/index.html.

[13] "Libreoffice sw module details." http://docs.libreoffice.org/sw/html/index.html.

[14] "Libreoffice import base class." http://docs.libreoffice.org/xmloff/html/classSvXMLImportContext.html.

[15] B. Neilson, "Version-aware libreoffice." Masters Capstone Project, University Of Wisconsin Milwaukee, May 2012.