Versioning in Fluid Cross-Media Document Format

Paul Otuyalo
Supervisor: Ahmed Tayeh
Supervisor: Dr. Bruno Dumas
Promoter: Prof. Beat Signer

January 11, 2014

Abstract

As document formats evolve over the years with the emergence of ubiquitous computing, where everyday objects are likely to access and process information on the fly. These document formats are enriched additionally with digital features to adapt to the upcoming age of ubiquitous computing. With the introduction of a versioning feature, these objects can keep track of a document revision history. Rather than saving multiple versions of a document which has to be managed by an external versioning system, a document revisions can be considered implicitly as different facets of the same document. This notion allows these devices to process the metainformation to relate the set of document versions instead of an individual version.

Our analysis of some existing document formats and versioning solution systems reveal the document components and aspects that are required for providing this versioning concept. However, the results of these analyses show that majority of the document formats have failed to support versioning. Therefore, this thesis presents several concepts for which future document formats can adopt to support versioning as an implicit feature.

The main objective of this study is to extend the *Fluid Cross-Media Document Format Metamodel* (FCMD) to support versioning. Although FCMD has been developed to address some challenges faced by existing document formats in the upcoming age of the ubiquitous computing, yet it failed to support versioning. Since the metamodel has been built by extending the resource-selector-link (RSL) metamodel, important dimensions for version support like advance linking and user rights management are readily available.

Finally, a prototype application of the extended Fluid Cross-Media Document Format metamodel has been implemented as a proof of concept. The implementation provides a FCMD format text editor, a version space window and a diff viewer. The text editor is used to create a FCMD document. Reviewing a version history of a FCMD document is done by the provision of the version space window. The diff viewer is used to view the FCMD document modification difference between two successive revisions.

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Statement of The Problem . . . 4
- 1.3 Aim of The Study . . . 4
- 1.4 Research Methods . . . 4
- 1.5 Thesis Outline . . . 5

2 Background **6**
- 2.1 Document Formats . . . 6
  - 2.1.1 Document Preparation Family . . . 7
  - 2.1.2 Meta-languages Family . . . 9
  - 2.1.3 Print-oriented Family . . . 9
  - 2.1.4 World Wide Web Family . . . 10
  - 2.1.5 Electronic Digital Publishing Family . . . 10
  - 2.1.6 Document Transformation Family . . . 11
- 2.2 Fluid Cross Media Document Format . . . 11
  - 2.2.1 Resource Selector Link Model . . . 12
  - 2.2.2 Fluid Cross-Media Document Metamodel . . . 15
- 2.3 Versioning . . . 18
  - 2.3.1 Taxonomy of Versioning . . . 18

3 Review of Existing Versioning Solutions **22**
- 3.1 Document Formats and Versioning . . . 22
  - 3.1.1 Document Logical Structure . . . 22
  - 3.1.2 Requirements for Versioning in Document Formats . . . 24
  - 3.1.3 Versioning in Document Formats . . . 28
- 3.2 Third Party Solution . . . 30
  - 3.2.1 GIT . . . 30
  - 3.2.2 SVN . . . 31
  - 3.2.3 CVS . . . 32
  - 3.2.4 Versioning Support in Third Party System . . . 33

4 Versioning Models for Extending FCMD metamodel 35
4.1 Introduction . . . 35
4.2 Versioning Models . . . 35
4.2.1 Delta Property Model . . . 36
4.2.2 Component Versioning Model . . . 37
4.2.3 Link Versioning Model . . . 38
4.2.4 FCMD Versioning Model . . . 39
4.3 Scenario: Accessing Revisions . . . 42
4.3.1 Scenario: Delta Property Versioning Accessing Revisions . 44
4.3.2 Scenario: Component Versioning Revisions . . . 47
4.3.3 Scenario: Link Versioning Accessing Revisions . . . 51
4.3.4 Scenario: FCMD Versioning Revisions . . . 54
4.4 Comparison of The Versioning Models . . . 56

5 Implementation 59
5.1 Objectives of the FCMD GUI Diff Viewer Implementation . . . 59
5.2 The Implementation Architecture . . . 60
5.3 db4o Database . . . 61
5.4 The FCMD GUI Diff Viewer . . . 63
5.4.1 Creating a New FCMD Document . . . 63
5.4.2 Creating a Revision . . . 64
5.4.3 Version Space Visualization . . . 66
5.4.4 Diff Viewer . . . 67
5.5 The Implementation Limitations . . . 67

6 Conclusions and Future Work 69
6.1 Summary . . . 69
6.2 Future Work . . . 70

Chapter 1

Introduction

1.1 Context

It is important to manage documents in such a way that several revisions of the same document can be accessed easily when tracking its history. The evolution of a document history can be used for purpose such as exploration, re-usage and comparison. In legal organizations like courts, tracking the history of legal cases of an individual is imperative, meaning that cases should be persistently stored with respect to an individual along their history. Financial institutions can also benefit significantly from versioning during audits, as auditors can create a version link between their audit papers' content and the company financial statements [10]. Nowadays in software engineering, it is required in large projects to know who did what, when and where. Furthermore, individuals working on small projects can recover from difficult problems such as finding out where and when bugs were introduced. Basic needs like tracking what changed between revisions is imperative in the merging of collaborative works.

Version Control Systems (VCS) have been introduced to tackle this revision management process. For most document formats, their authors must rely on VCS tools for versioning in which only text are supported. However with the emergence of web 2.0 technology and ubiquitous computing, high quality information can be accessed and processed from anywhere, the source of the information could be small devices like smartphones, small sensors on electronic devices, etc. One question that needs to be answered is: *How should these devices address versioning such that the right version to an information is accessed correctly?* While most document formats do not provide an implicit versioning functionality, they rely on external solutions to provide this feature.

To address the issue of versioning as an implicit functionality within a document format, this thesis has analysed existing document formats and other versioning solutions. These analyses will be used to present a versioning model that address the strength and weaknesses of these versioning solutions.

1.2 Statement of The Problem

It is important to keep records of the revisions of a document implicitly as different facets of that document which may have different versions equipped with meta-information that provides authoring, traversal scheme, access right etc., rather than having copies of that document versions that have to be explicitly identified by the document file name.

Some existing document formats do not support versioning functionality as an implicit feature within the format. Therefore they rely on external third party solutions like GIT, Subversion (SVN), Concurrent Versions System(CVS), etc. These solutions are usually mainly capable of managing entities that are text based.

Research from [14] identified the limitations that exist among some existing document format. Features such as advanced linking, transclusion (reuse content), user access right management,etc., were not addressed by these document formats. [14] came up with a document format metamodel to tackle most of these limitations, and the metamodel is called *Fluid Cross-Media Document Format* (FCMD). The FCMD metamodel is based on the hypermedia model *Resource Selector Link* (RSL) [8]. Nonetheless, this document format metamodel also failed to tackle versioning implicitly but versioning was considered as a potential future work.

1.3 Aim of The Study

The main aim of this study is to extend the FCMD metamodel to support versioning. The objectives of this study are to: (i) extend FCMD to support versioning without altering the overall goal of the document format, using the best versioning model from several proposed models. (ii) as a proof of concept, provide a prototype implementation will be provided to support versioning in FCMD based document formats.

1.4 Research Methods

To achieve the aim of this research, the following steps were carried out:

1. a) *Review of Versioning in Existing Document Formats*: an in-depth analysis of versioning support among a range of document formats.
   b) *Review of Third Party Versioning Solutions*: a review on versioning control system such as GIT, SVN, and CVS were studied.
   c) *Review of Versioning in Hyermedia Systems*: some hypermedia systems already provide versioning within their models. An analysis of their versioning methods with respect to versioning dimensions was achieved.
2. Identification of the strengths and weaknesses of existing versioning solutions, considering the best dimensions that are highly relevant to the

FCMD metamodel and choosing the best way to extend the model to support versioning.

3. Implementation, as a proof of concept via a prototype diff GUI display to reveal the revision evolution of the document, where two revisions changes can be compared using a diff veiwer. These revisions are mapped on the extended FCMD metamodel nodes.

1.5 Thesis Outline

The structure of this thesis is presented as below:

Chapter 2: Background
The aim of this chapter is to introduce the FCMD metamodel and RSL model on which FCMD is based. Also we will discuss some representatives among various document formats. Finally, we will introduce versioning and the various versioning classifications.

Chapter 3: Review of Existing Versioning Solutions
Versioning in existing document format, hypermedia systems and third party versioning systems will be reviewed.

Chapter 4: FCMD Versioning Model
In this chapter, several versioning models based on the versioning taxonomy will the described and from these models, we will define the extended model of the FCMD metamodel to support versioning.

Chapter 5: Implementation
This chapter will provide the overview of implementation for the proof of concept of our defined versioning FCMD metamodel.

Chapter 6: Conclusion and Future Work The summary of the overall research will be presented in this chapter and our intended future work.

Chapter 2

Background

This chapter highlights a set of document format families and present the features of some representative document formats for each family. The *Fluid Cross Media Document Format* metamodel (FCMD) and its features are also introduced. Lastly, the general concepts behind versioning will be discussed.

2.1 Document Formats

This thesis focuses on the versioning aspect in document formats; therefore this section is aimed at providing a brief review of some chosen representatives of the document formats. The family of documents we will consider are as follows [14]:

Document Preparation Family: Document formats from this family use some kind of generic coding for marking up the presentation of the document. We will choose GML, Scribe, LATEX and DocBook to represent this family.

Meta-Languages Family: This is a document format family where other document formats can be described. SGML and XML was chosen for this family.

Print-Oriented Family: This family tries to adapt the WYSIWYG in printing and editing. PDF and Open Office XML are chosen as representatives.

World Wide Web Family: The family of web document formats used on the Internet for presenting information. This family is represented by HTML (Version 4 and 5) and XHTML.

Electronic Digital Publishing Family: This family provides solutions that integrate e-publishing with rich digital functionalities. EPUB will be discussed as a representative of this family.

Document Transformation Family: This family provides a standard that supports the interchange of documents. ODA will be used to represent this family.

In the rest of this section, we will present the representing document formats of the above document format families. We will discuss the structure, data elements, syntax and examples. The discussion of versioning support of these different document formats will be present in the next chapter.

2.1.1 Document Preparation Family

GML

Generalized Markup Language (GML) [34] is a markup language. It describes a document based on the relationship among its content parts and their organization structure. GML's goal is to describe what a document represents rather than what they look like on display. In contrast to most markup languages, GML is very complex. Henceforth, in 1974 GML was extended to a simpler industry-developed markup language known as Standard Generalized Markup Language (SGML). With GML, documents are marked up with tags that define the document structure such as paragraphs, headers, lists, tables and so forth. In addition, GML supports profiling so that the visual rendering of the document can differ from one device to another.

LaTeX

LaTeX [25] is a typesetting system aimed at typesetting text and mathematical formulas. LaTeX uses the TeX [36] formatter as its typesetter which was created by Donald E. Knuth. With the high quality of typesetting achievable by TeX, LaTeX is used produce very rich presentation of documents, slide presentations and much more. It separates the layout from content with a mechanism similar to the use of *Cascading Style Sheets* in HTML document.

The LaTeX file contains plain text. This text may be document content or LaTeX commands syntax that tell LaTeX how to typeset the text. The Figure 2.1 depicts a LaTeX plain text file on the left side and its typeset presentation on the right side.

DocBook

DocBook [30] is described as a semantic markup language for writing technical documents using XML. It also creates document content in a presentation-independent form that captures the logical structure of its content, meaning that its content can be published in other document formats. These include formats like HTML, XHTML, EPUB, PDF, etc. As a result, these formats will not require authors to make any changes to the source. Its structure resembles to the general idea of how a book is composed. The document classes such as *Book*, *Articles*, *Reports*, etc. can be written using the docbook document

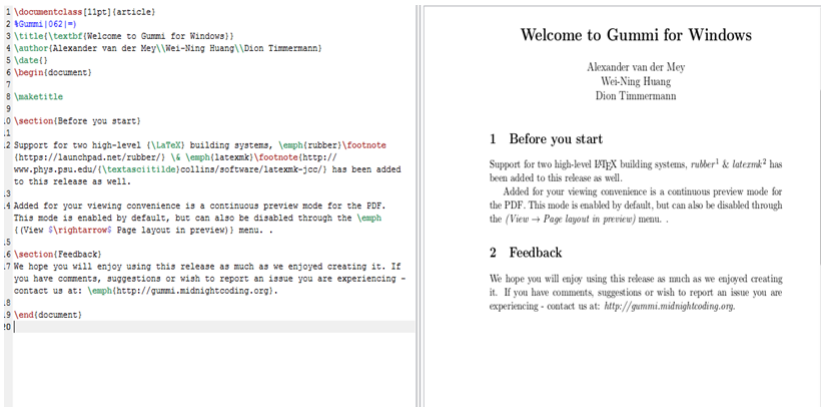

Figure 2.1: Example of a LaTeX article class document and presentation

class [29]. The Figure 2.2 shows the composition of a book document class in Docbook.

```
<!DOCTYPE book [
<!ENTITY chap1 SYSTEM "chap1.xml">
<!ENTITY chap2 SYSTEM "chap2.xml">
<!ENTITY chap3 SYSTEM "chap3.xml">
<!ENTITY appa SYSTEM "appa.xml">
<!ENTITY appb SYSTEM "appb.xml">
]>
<book xmlns="http://docbook.org/ns/docbook">
<title>My First Book</title>
&chap1;
&chap2;
&chap3;
&appa;
&appb;
</book>
```

Figure 2.2: Example of a Docbook composition.

Scribe Document Model

Scribe is a markup language that was developed by Brian Reid. It is referred to as the first markup language that separated the structure and format of a document [27]. Scribe provides commands (markups) which are not commands as in the ordinary sense. These commands provide the semantics that describe the content presentation such as chapter, section, etc. Processing a scribe document is established in two phases: 1) Typing a manuscript file conforming to

the scribe standard. 2) Using a scribe compiler to process and generate an associated document file that can then be printed. The document format definitions of scribe are stored in a database so that the compiler can identify the correct rule to format a given scribe document.

2.1.2 Meta-languages Family

SGML

Standard Generalized Markup Language (SGML) [33] was extended from Generalized Markup Language (GML). SGML's markups are defined to describe the document structure and provide the technique to support external programs to process defined objects. SGML offers to define the semantics for the markups in documents. Thereby, determining which tags are allowed, where and which attributes are supported within the tag. This is achieved using a Document Type Definition (DTD). These DTDs describe the logical structure of a document so that it is easy to associate the different kinds of elements that form the document. With SGML meta-language, other markup languages have evolved from it such as XML and HTML.

XML

Extensible Markup Language (XML) [35] is a data exchange document format, capable of describing arbitrary structures. XML is a simplification of SGML, which is less difficult to parse. In XML, authors can create an unlimited amount of sets of tag names. XML data models are represented by textual representation. This data model can be viewed as a tree structure.

XML starts with the XML declaration which includes processing instructions like its version and encoding character type. This can be followed by pairs of elements start and end tags. Attributes may only be present in the start tags. However there exist common tags and processing instructions as part of the data model. Yet comment tags allow authors to include some meta-information in the XML document, processing instructions will allow them to specify certain commands to specific processors. However, there exists only one root to a XML document; well-formedness can easily be tracked by tools. Special characters can be referenced with character references.

2.1.3 Print-oriented Family

OOXML

Office Open XML is an XML-based file format for representing spreadsheets, charts and presentations. It is also informally called OOXML or OpenXML. However, the main goal behind the OOXML standard is to provide the capability to represent the pre-existing corpus of word processing documents that had been produced by the Microsoft Office applications. OOXML facilitates extensibility and interoperability by supporting implementations via multiple vendors

and on multiple platforms. So that other applications like Simple OOXML and SoftMaker Office 2010, can be used to read and write Open XML format. OOXML, like OpenDocument are formats aimed at producing WYSIWYG office applications.

PDF

Portable Document Format (PDF) was initially released in 1993 by Adobe Systems as a means of document sharing [28]. PDF is represented as a two-dimensional document such that is independent of the software, hardware and operating system. Every PDF file encapsulates the entire description of a fixed-layout two-dimensional document. This includes the text, images, and other information needed to render it.

PDF can be extracted from any document or word processing software. This implies that an accurate and fixed-layout representation of the original document can be rendered. As a result, PDF can be used for long term storage and archiving in distributed systems. PDF mainly focuses on the preservation of the display appearance of a document. In contrast, it does not ensure the preservation of the logical and physical structure of a document.

2.1.4 World Wide Web Family

HyperText Markup Language

HyperText Markup Language (HTML) is a markup language essentially used for creating webpages. It is derived from SGML. It was created by Tim Berners-Lee for the purpose of sharing scientific documents. The latest revision of HTML is HTML 5. However, compared to HTML 4 provides a more support to seperate content and layout. While the XHTML version is a XML well-formed version of HTML 4.

HTML provide means for structuring web documents. The author can inserts HTML markup tags, or commands, before and after words or phrases to indicate the structure, format and location to present them. These markup consists of a number of vital components, such as elements and their attributes, character references and entity references. Like SGML, HTML has a the document type declaration that provide the standards mode rendering.

2.1.5 Electronic Digital Publishing Family

Electronic PUBlication Format

The Electronic PUBlication format (EPUB) [38] is an open eBook format standard developed by The International Digital Publishing (IDPF). It is a document format based on Web Standards like XHTML, CSS, SVG, etc. EPUB 3 is the current version of the EPUB standard. Via EPUB, web content can be enhanced in structure and semantic by its representation, packaging and encoding. Hence, the format is interoperable between software and hardware such

that it can be published as a single reflowable digital book. XHTML or SVG documents may define the readable content within the EPUB document and may reference other media resources such as images, audio and other media resources. Figure 2.3 shows an example of an EPUB document based on XHTML standard.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
        <head>
            <meta http-equiv="Content-Type" content="application/xhtml+xml; charset=utf-8" />
            <title>Versioning</title>
            <link rel="stylesheet" href="css/main.css" type="text/css" />
        </head>
        <body>
          ...
        </body>
    </html>
```

Figure 2.3: An example of an XHTML file for EPUB [38]

2.1.6 Document Transformation Family

ODA

ODA formally known as Open Document Architecture and Interchange Format is referred to as an international standard document format. It defines a complex document format that comprises text, images and other vector graphics. Its goal is to support the interchange of documents such that different kinds of content types can coexist within a document. ODA can transmit the purpose of original documents logical and layout structure to a document recipient. Like cascading style sheets in HTML, the layout mechanism of ODA is similar. This format also supports storage or interchanged in any of three formats: Formatted, Formatted Processable, or Processable [43]

2.2 Fluid Cross Media Document Format

Understanding the various document formats helps provide a clear view of which the problems they pose to the ubiquitous computing world. Research on existing document formats [14] shows that the digital features like advance linking (bidirectional links), content reuse, user rights management, adaptation and versioning are rarely supported. The lack of these features in existing document formats will reduce the possible variations of functionalities within its format. These limitations in existing document formats motivated the researchers from the *WISE lab* to come up with a document metamodel that supports these dimensions. This document metamodel is called *Fluid Cross Media Document Format Metamodel* [14]. Though it lacks the versioning dimension, it provides a strong support for extensibility. This metamodel is based on the *Resource Selec-*

tor Link model (RSL) [8], a powerful hypermedia model that overcame flexibility and navigational issues that are common in most hypermedia systems.

In the following section we will introduce the RSL model and its concepts. Next, we will discuss the FCMD metamodel and analyze its logical structure.

2.2.1 Resource Selector Link Model

RSL metamodel [8] is a hypermedia metamodel that provides a rich and powerful set of navigational functionalities for implementing a hypermedia system. RSL is based on the concept of linking arbitrary resources [8]. Unlike some other hypermedia model, RSL gives a great flexibility in supporting features such as bidirectional links, multi-source and target links and link groups. This is achieved by managing links separately from resources. IServer [20] is a cross-media information platform application based RSL. IServer supports a wide range of different resource types from web pages to the physical objects marked with RFID tags [8].

RSL Core Component

The RSL model is defined to provide a generalized and extensible model that models sevaral features in most hypermedia systems via a link model concept. Such features could include *user context modeling and adaptation*, *content reusability* and so on. The schema of the core RSL model is displayed

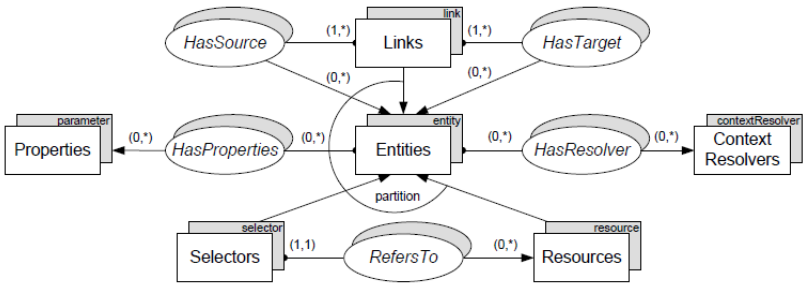

Figure 2.4: RSL core components, based on [14]

in Figure 2.4. The shaded rectangular shapes represent collections of objects (classification) where the name of the collection is shown in the unshaded part and the name of the associated type in the shaded part. The shaded ovals represent associations between entities of two collections. In this section, we will describe the six core collection types and the associations connected between them. An *entity* is an abstract representation of the objects that can exist in any hypermedia system. An Entity is extended by three subtypes; *selector, link and resource* types.

The simplest type of entity type is the *resource type* which describes an entire information unit. The model also supports relating part of a resource to another resource like the anchor in a web page that a link relates to using the href anchor tag. This can be reflected by the *selectors subcollection*. A selector is an abstract concept that defines the link enabling specific parts of a resource to be addressed. For example selecting a sound clip from a song in a hypermedia system and enabling it for referencing. The *ReferTo* association represents that a selector can only be associated with exactly one resource, while a resource can be associated by zero or more selectors. These are shown in Figure 2.4 as cardinality constraints on both the source and target (1,1) and (0,*) respectively.

In RSL, the *link* type in the metamodel provides the mechanism for entities to be referenced. Links in RSL are directed and may be from one or more sources to one or more targets, with sources referring to a resource or part of a resource referenced by a selector or even link. These are provided in the model by making the collection of entities (the supertype) the target of both the *HasSource* and *HasTarget* associations. Both the *HasSource* and *HasTarget* associations have the cardinality constraint on the source collection (1,*) which enforces that a link must have at least one or more source entities and one or more target entities. The effect of the constraint means links in RSL based applications will never have dangling links. Meaning that, every link is constrained to have at least a source entity and a target entity. Links in RSL are also treated as first class citizens. This is reflected in the model, as a link is a subtype of the entities collection. This means we can annotate a link between two entities with another link. This annotated link will provide supplementary information which can be not only textual but also arbitrary entities like a resource or parts of resources.

A set of *context resolvers* are associated with each entity. They provide the context-dependent handling functionality of an entity which is used to determine the visibility of the entity. A *contextResolver* in RSL returns a boolean value that denotes the entity accessibility based on some data and contextual information managed by the hypermedia model. An example will be a resource that adapts the context resolver so that only links from a certain source resource can target it. The association *HasResolver* ensures that multiple context resolvers can be applied on each entity. Since context resolver is an abstract concept, various domain specific context resolvers can be registered with the system.

Finally we will discuss the last core component of the RSL metamodel, the *property*. A property is a key/value tuple and can individually be used to customize the entity's behavior for a specific application domain.

RSL User Model

Since most hypermedia systems do not incorporate users as part of their model to ensure data ownership, RSL authors have introduced user rights management in their metamodel. The *User Model* in RSL provides functionality for managing data ownership and access rights at the entity level. Thus it is possible to define individual or group permissions (access rights) for links, resources and selectors.

Users are classified as either individuals or as a group. Here, a group itself can

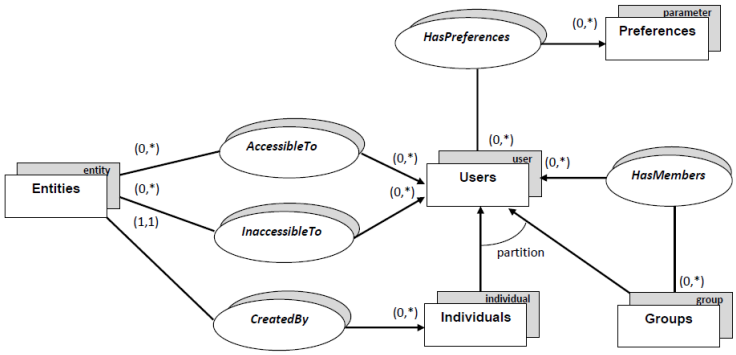

Figure 2.5: User model in RSL, based on [14]

be included in other groups. RSL user model provided the support for user access right by introducing *AccessibleTo* and *InaccessibleTo* associations in the model.

RSL Structural Links

Finally, we will discuss structural links in RSL. As mentioned earlier, links in RSL model are treated as first class objects and with this concept, RSL authors are able to model distinctively *structural* and *navigational* links between resources. *Navigational links* are more like the regular links we find in our

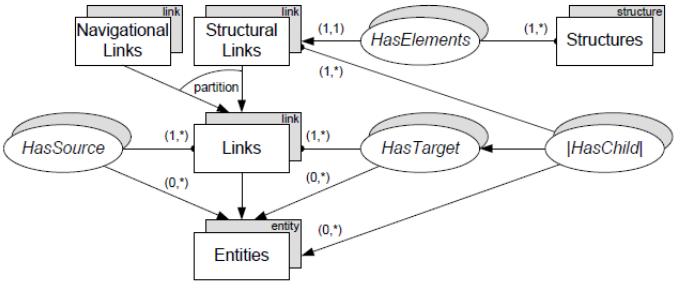

Figure 2.6: Navigational and structural links are subtypes of a link collection, based on [14]

web pages. However, links in RSL support multiple source and target entities. On the other hand, *Structural links* help to define the composition of resouces. For instance, a hypermedia document may be composed of several entities in a composed manner and structural links within the document will keep track of each resource. In Figure 2.6, the association *HasElement* defines the relation

between a single structure and its structural links, where the *structures* collection handles all structures. Structural relationships can be ordered by using the association |HasChild|which is a sub association of the *HasTarget* association. A sub structural relationship of composed resources may be ordered differently which may be based on user access rights or any other context resolver-based variation.

Finally, *structure over structures*, *structure over links* and *structure over data* can also be defined by structural links which makes it possible to reuse the same entities in a different structure.

2.2.2 Fluid Cross-Media Document Metamodel

In the upcoming age of ubiquitous computing, Tayeh [14] researched on how prepared are existing document formats for this age. The result this research described was the lack of support in current document formats for a number of digital features such as transclusion, user right access management, etc. [14] provided a format meta-model that will address these limitation known as Fluid Cross-Media Document format (FCMD). Firstly, FCMD is modeled over the powerful hypermedia model called RSL [8], discussed earlier. As a result, FCMD can represent all document object types that may exist within a document because RSL already provides an abstract concept to address it. As discussed in [14], most document formats have limited support for advanced linking. It is almost impossible for an author of an existing document to be informed implicitly who quotes is work. However, all these limitation and much more have been tackled by the FCMD metamodel. In the following section, we will discuss the logical structure behind the FCMD metamodel.

FCMD Logical Document Structure

Documents can be considered on two different representation levels: the logical representation and the physical representation. At the logical representation level, the document is expressed in terms of an abstract model that defines its components and their relationships. On the other hand, the physical representation expresses how the concrete representation of the document is rendered onto a display medium. This research is mainly concerned with the logical representation of a document and this will be discussed more details in the next chapter.

As observed from most existing document formats, document classes like books, reports, articles, etc. have different logical representations. However, these logical representations are related, yet their physical representations are considered to be completely independent of the logical structure. For instance, a report document class section can have a similar logical representation with a chapter within the same document class but its physical representation may differ.

Consider the logical structure of these document classes: *book*, *report* and *article*. They all have sections, subsectiona and a table of content. However the

book and report classes both have chapters which are not present in the logical structure of an article. Meaning that sections are the highest level within the structure of an article, whereas in books and reports, the higher level can be considered as the chapter. Generally, all the document classes have some kind of hierarchical relationship that provides a unique rendering for its physical representation.

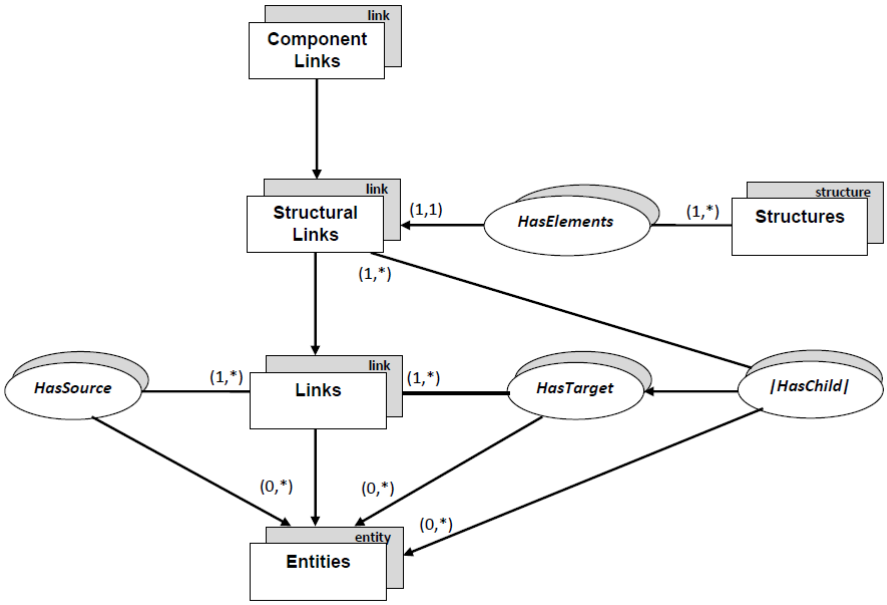

Figure 2.7: Component links in the FCMD metamodel by [14]

Hence in FCMD, the metamodel considers a document class to have at least some atomic structure. Consider a book which contains several chapters, and within the chapter are sections, and within a section are several subsections, etc. FCMD considers this atomic structure of the document to be called *component*. Hence, a component can be regarded as section, chapter, etc., depending on its level in the FCMD document class hierarchy, as in RSL where *structural links* are used to compose a new *resource*. Likewise, FCMD will compose a new document using an extended structural links to relate components as in Figure 2.7.

In Figure 2.8, the composition of a FCMD document of class "book" is shown. Figure 2.8 (a), shows how two atomic structures, component1 and component2 are created from text and image data resources. While in Figure 2.8 (b), component3 and component4 are consider higher level components, as they are composed of the lower level component1 and component 2 respectively. However, in Figure 2.8 (c) component5 is the book representation. As a result, component3 and component4 are considered the highest level of the class book

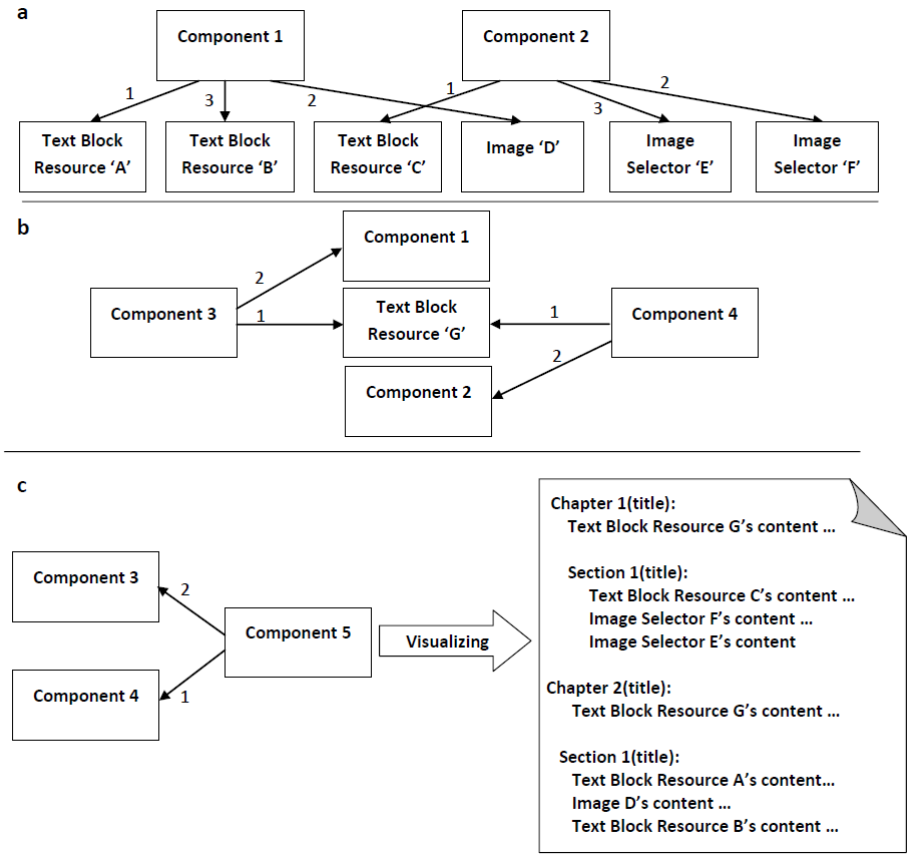

Figure 2.8: Composing a book using the FCMD metamodel by [14]

which is a chapter. The numbering in the Figure 2.8 denotes the order in which the components are rendered in the physical representation.

2.3 Versioning

The idea behind versioning is to manage data so that tracking changes can be possible. Versioning can be used for a variety of reasons such as revision tracking, coordinating and collaborating teams, content reuse and safety. Revision tracking is important in situations where data needs to be compared. For instance, a court is able to track a recurrent case of an individual criminal history via comparison with the collection of the individual's criminal case documents. Versioning also enables the support for coordinating and collaborating teams working on a project. Here, teams may either be co-located or distributed. One of the effects of collaborating teams for a project is content reusability. Other teams or individuals may use the data presented by a team without altering the data. Imagine a situation where someone accidentally overwrites a file, which can never be retrieved again. With versioning, the safety of data can be guaranteed.

In the rest of this section, we will discuss the taxonomy of versioning to understand the concept behind some versioning features.

2.3.1 Taxonomy of Versioning

Different versioning concepts are motivated by the goals they try to achieve. One goal could be to have a collaborative environment where team members could work together. This environment should support tracking team changes, branching of development lines and managing merge conflicts. For this reason, the organization of the modelling entities might have an impact on which features a system can support. Where [4] defines an entity as a concept representation for an abstraction in modelling. Furthermore, another goal could be having a stable reference to data items. Hence, dangling references should never occur in the system. The following section describes the taxonomy among versioning solutions.

Versioning Organization

Many versioning solutions tend to focus more on how data items are organized within its system. A lot of motivations tend to support this argument. Firstly, data item retrieval speed. Hence, the organization will determine how traversing within the system is perceived. Another motivation is how to deal with large data organization. As a result, more organization techniques tend to modularize data better. Versioning organizations are categorized into three techniques as described below.

File-Based Organization: Versioning solutions, mostly version control systems adapt this type of organization technique. Here entities are

organized in a file-directory hierarchy. GIT [3] and SVN [41] are typical examples.

Containment Organization: Dexter model [6] was the first to embrace this organization method. A container-like concept is used to compose the model resources. They refer to the container as *components*. Here, *container* describes an abstract concept that represents a collection of *versioned items*[1] or *versioned* and *unversioned items*[2]. These containers tend to support modularization. They are linkless in structure. Related entities can collectively be contained with a container. One drawback to this method are the overhead operations needed to overcome unstable references. CoVer [16] and NCM model [13] ensured a stable references by introducing membership registration.

Link-Based Organization: Contrary to containment organization, link-based methods use links as a container for the information unit needed at endpoints, as a result, solving the overhead issue of membership registration operation. In RSL [8], the link-based organization technique is adopted, but here both endpoints of a link are equipped with the information unit. This bidirectional capability eases the flow of navigation. In summary, with a link-based technique, modularization of data and flexible navigation can be achieved.

State-based and Change-based Versioning

The states of data items as they evolve can be observed. Records of these states can be persistently stored as snapshots. Versioning based on the recorded state of an item is called *state-based versioning*. [12] referred to state-based versioning to be described in terms of *revisions*[3] and *variants*[4]. On the other hand, a collection of changes can be aggregated based on some underlying baseline. These aggregated changes define how the snapshots within revisions are perceived. Versioning based on a set of changes between states of versioned items is known as change-based versioning [11]. While identifiers are placed on version items in state-based, it is the aggregated changes that have identifiers in change-based versioning. GIT [3], CVS [42] and SVN [41] are typical examples of state-based versioning solutions, whereas change-based versioning has been used by hypertext system like PIE [17]. However CoVer [16] provided an hybrid model that converged both towards versioning orientation.

Versioning Representation

The approach required for pesistently representing revisions of a data item depends on the versioning goal intended. In other words, data needed for version-

[1] Versioned items refer to data items whose state can be altered.

[2] Unversioned items refer to data items whose state and properties can not be changed.

[3] A revision is a version that differs from its predecessors due to an upgrade or modifications.

[4] A variant is an instance of a data item intended to coexist with other revisions and may differ from them by some property changes.

ing are either duplicated or referenced. The most widely used representations depicted in Figure 2.9 are *referential*, *inclusive* and *predecessor/successor relationships* representation.

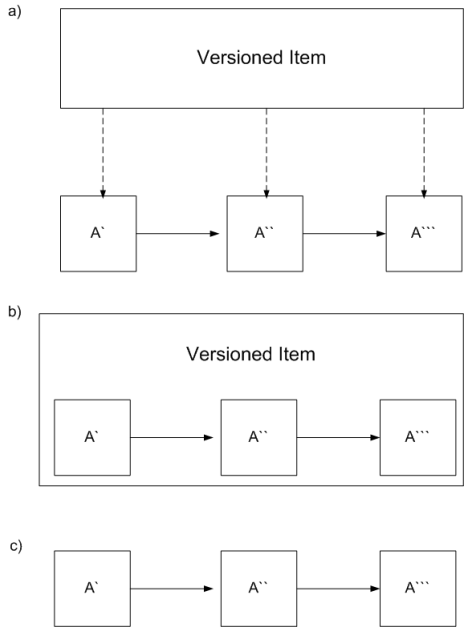

Figure 2.9: Versioning Representation: a) Referential b) Inclusive c)Predecessor/Successor Relationship

In referential representation, the versioned item contains links to individual revisions. Thus each link is encapsulated with the information of the relationship between the version item and the revision. Additionally, this representation approach captures structure by collections of referential links. Furthermore, a single revision can be referenced by several objects. As a result, only a single member of this revision is needed within the document. Many hypertext/hypermedia systems such as CoVer [16], VerSe [22], NCM [13], HyperPro [23], etc., have adopted this representation approach.

In inclusive representation, a single data item instance contains all its revisions. This data instance serves as a container. [4] explained how objects stability of references can be guaranteed based on the availability of endpoints computation. Thus having revisions within the object, the granularity of changes can be used to compute a required revision. System like CVS [42] and SVN [41] define objects that represent inclusive versioning representation. One disadvantage is that reusing a revision's data unit by an object requires the whole revision object to be copied within it.

On the contrary, predecessor/successor relationship representation is an ordered linear sequence of revisions in a predecessor and successor linking. All revisions are stored in a central object pool. This representation orientation

is used by Xanadu [44]. The drawback of this representation is the cost of evaluating revision selection queries.

Chapter 3

Review of Existing Versioning Solutions

This chapter discusses the logical structure of a document format. It outlines the requirements needed to model versioning as an implicit function within a document format. Versioning support among existing document formats was investigated. Also the versioning concepts used by a selection of third party versioning solutions will be introduced.

3.1 Document Formats and Versioning

In order to understand the versioning in document formats, we must first explore how the document formats are logically structured. Although there exist several document format logical structures, the basic concepts are similar. The logical pattern for these document formats can be reduced to a tree form. The nodes in the logical tree represent the entities that exist within the document and their attributes. However, some document models do not have mechanisms to represent all types of logical nodes. For example, SGML [33] does not have a feature to display footnotes.

3.1.1 Document Logical Structure

The logical structure of a document provides the logical organization of the document formats; thereby defining the relationship between the document entities. As mentioned in the previous section, the logical structure can be represented as a tree for most document format. Hence, the role of each node in the tree is defined by the logical structure. A node can be classified as a *data node* or *structural node*. While data nodes represent the information unit of the document, structural nodes define the possible relationship between data nodes or a data node and a structural node or how structural nodes are organized.

Structural nodes in a document (like chapters, sections or paragraphs) reflect the organization of logical nodes. However, the logical structure of every document class (such as report, article or book) determines which structure node exists for the document formatting. The highest level of a node may vary among document classes. For instance, in LaTeX, specifying the document class to be an article will set the highest level node as a section while in a report class, the highest level node is a chapter. In contrast, the lowest level of a node (atomic object) may be similar among document classes. As an example, the atomic nodes for a report and article class may contain atomic objects like a table, an image or a text string. To illustrate, we will consider the LaTeX document in Figure 3.1 and its logical tree in Figure 3.2.

```
\documentclass{report}

\begin{document}
\title{\textbf{Versioning in Fluid Cross-Media Document Format}}
\author{Paul Otuyalo\\Supervisor: Ahmed Tayeh\\Supervisor: Dr. Bruno Dumas\
\Promoter: Prof. Beat Signer}
\maketitle
\tableofcontents{}

\chapter {Introduction}
.......
\chapter {Review of Existing Versioning Solutions}
.............
\section{Hypertext/Hypermedia Systems}
.............
\paragraph {Neptune Model}
.............
\paragraph {VerSe Model}
.............
\chapter{Summary}
...........
\section{Future Work}
.................
\end{document}
```

Figure 3.1: A LaTeX Report Document Class

With the tree like representation that most document models use, we can transform our formalized LaTeX document in Figure 3.1 into a logical tree structure as shown in Figure 3.2. The logical structure of the document format can be a primary logical structure or a secondary logical structure. The primary logical structure can be supported by a secondary one. Thus the relationship that cannot be expressed within the primary structure itself can then be expressed in a secondary structure. According to [14], the secondary structure mainly defines three constructs: *attributes*, *floating objects* and *cross-references*. Attributes are used to express the semantic information that is not provided by the primary structure. The floating objects are objects that can appear at any arbitrary place in the document, while a cross-reference construct will al-

low objects to refer to other objects. Cross-referenced objects can be defined among objects in the same document or among objects in a different document. Figure 3.3 presents the overall logical document structure of Figure 3.2. This shows both the primary and the secondary logical structures which can possibly be augmented.

In our LaTeX document example described above, the *generic structure* of the LaTeX report document class determines the relationship among the document nodes. As a result, the generic structure will define the sequence and order of the tree structure. The generic structure contains the node definitions, node relationship definitions and constraints that refine its logical structure. Given the generic structures applicable to LaTeX documents, our LaTeX document example can map its document class to an appropriate generic layout that formats the document. Generally, using the generic structure, the document formats are able to have knowledge of where a specific kind of information is located in a document.

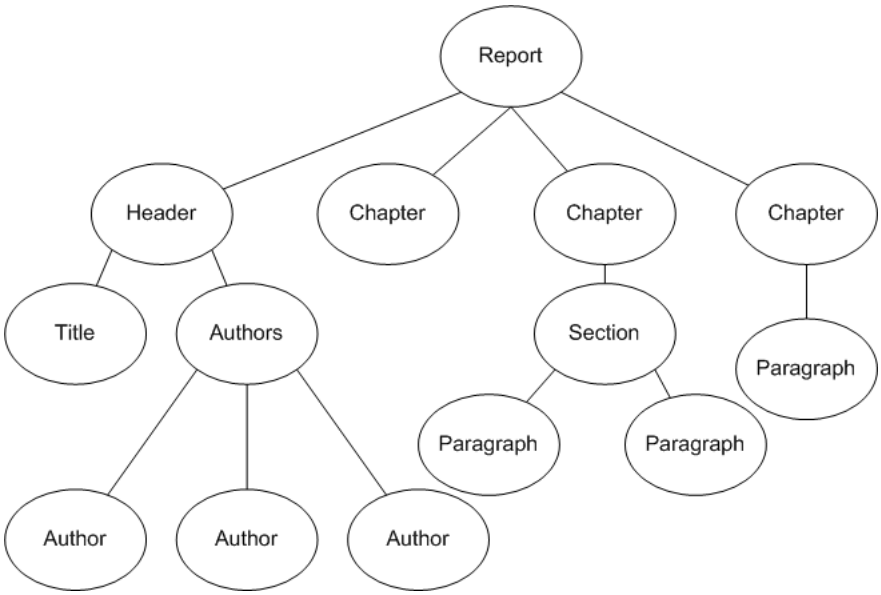

Figure 3.2: The logical tree structure of the presented LATEX document

3.1.2 Requirements for Versioning in Document Formats

After analyzing the general logical structure of document formats, we observed that nodes of the logical tree either contain a structural node or a data node. Despite the fact that cross-reference objects such as HTML links [40] are not treated as first class objects like data and structure in most document structures, they play an important role in the organization of logical nodes. A link node, also known as a navigational element, with two endpoints, connects one resource

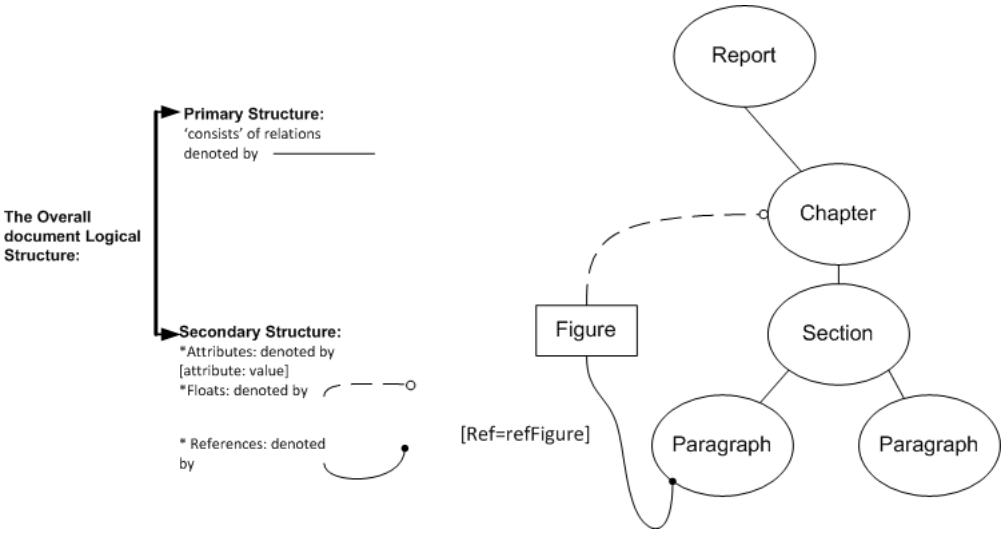

Figure 3.3: The overall logical document structure with a primary structure and enriched by a secondary structure

to another. A link has one direction, and starts from a source endpoint to a target endpoint. This linking concept adds extra advantages to a document format. With links, a document format has the ability to connect with external resources outside the document. One drawback with the link as defined in most document formats is that the target node is never aware of its resource being referenced. One solution to this drawback has been tackled by RSL [8] by providing a bidirectional connection between endpoints.

In order to achieve versioning in a document format such that different facets of the same document can represent its revision history, we have to identify those conceptual objects that need to be addressed in the document model. At this moment we have identified three dimensions within a document format logical structure: *data*, *link* and *structure*.

Data Versioning: The persistent storage of versioned data is significant for the safety of a document data and the exploration of the evolution of a versioned data. However, with the concept of versioning data, we are provided with a mechanism to backtrack data revisions for the sake of re-usage or comparison. Most of the time a revision may contain shared data units and properties with its predecessor. Using Figure 3.4 a) to illustrate data re-usage using versioning, consider the three revisions of a data (*version 1*, *version 2* and *version 3*), whereby each revision contains some similar copy of an information unit as its predecessor. Rather than have *version 2* to copy the information unit shared with *version 1*, we can make a cross reference to the piece of information unit, thereby avoiding redundancy. The cross reference object (link) is equipped with the change aggregation (also known as *delta operation*) information that informs

target endpoint on how the source data should be manipulated to get the up-to-date version of the data piece. This concept of avoiding copying entire data is very important when a new revision is needed after a simple modification or correction is made on a document [4].

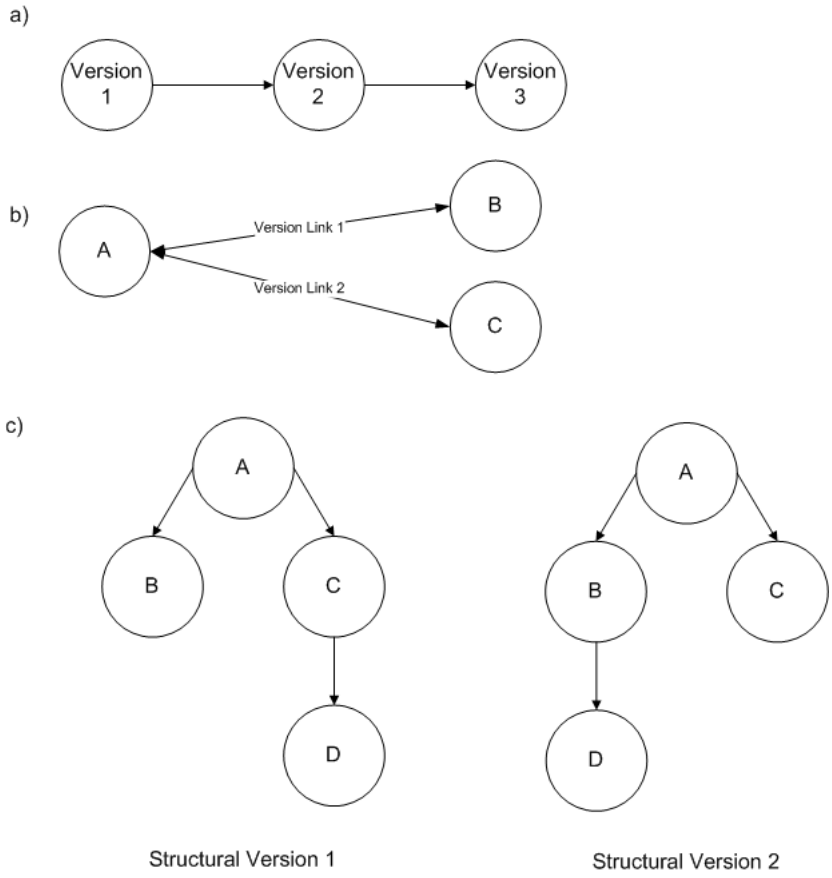

Figure 3.4: Versioning dimension in document format. a) Data Versioning b) Link Versioning: Storing link states for exploration safety c) Structural Versioning.

Link Versioning: Here, we assume the link mechanism is bidirectional and can be versioned. Link versioning provides a recorded link history that enables several techniques for recording the version history of document nodes [11]. For instance, Figure 3.4 (b) illustrates persistent storing the link states between source node A and target node B and C, where node B and C reflect the modification data unit of node A. Let node A represent a paragraph of a document with the content "*This is a version object* ***A***", where the character ***A*** in the paragraph is an anchor (endpoint reference) of node A. Let node B and C rep-

resent a data unit with the characters ***B*** and ***C*** respectively. Thus we can store *version link 1* and *version link 2* such that node A can be modified with the data unit in node B and C and can be used present its two revisions; "*This is a version object* ***B***" and "*This is a version object* ***C***".

In spite of versioning links for data history exploration safety, we can force the stability of each link endpoint reference in a *version space*[1]. The result of removing any data node from a version space will cause unstable link endpoint references also known as *dangling links*. Each connecting link to the lost data node loses their endpoint reference. As a result, the integrity of the version space is lost. Therefore, providing a mechanism of forcing the stability of endpoint references is an important concept in versioning document node objects [8].

Structure Versioning: Like versioning links, versioning a document structure provides data re-usage. Thus the same data nodes can be applied to multiple structures. Consider the example in Figure 3.4 (c), where *structure version 1* and *structure version 2* are composed of the same nodes (A, B, C, D). In a document format, this concept means that we can have revisions of the same content in different document classes. For example, an author of a book may decide to create an article from a chapter in his book, in order to restructure its chapter content from its book class logical structure to its article class logical structure [11].

From the three dimensions presented, we can recognize some additional dimensions that can enhance a versioning model for a document format. In data, link and structure versioning, change aggregation is vital to achieve content reusage. However, these change operations could range from simple changes like alter a character position within a text to more complex image or video manipulations. Futhermore, change aggregation for links can involve the storage of the collection of the changes made on the link metadata so that these metadata can have independent version histories. For example in figure3.4 (b), applying change aggregation on the *version link 1* so that the position of the data unit in node B is placed at a different position in node A ("*This is a version* ***B*** *object*"). This can be considered a new revision using the same node data in A and B. Likewise for structures, change aggregation can store the processing information needed to transform from one structure to another. For instance Figure 3.4 (c), illustrates a change aggregation on *structural version 1* which contains the organization information to change the link endpoint reference from node C to node B to produce *structural version 2*.

In the final analysis, we have identified one further dimension to enhance a versioning model with respect to a document format, namely *Change Aggregation Support*. As a result, our research was based on versioning in existing document format with the following dimensions: i) *Data Versioning Support* ii)

[1] Version space is the organization of data objects and their relationships in form of network spaces with linked nodes.

Structure Versioning Support iii) *Link Versioning Support Change Aggregation Support.*

3.1.3 Versioning in Document Formats

In the review of existing document formats in the previous chapter, only very few of these document formats have incorporated the versioning mechanism implicitly within the format model. However, a few others have an explicit provision of some mechanism that allows external systems to identify revisions within the format. In terms of link versioning and change aggregation, no existing document format that was investigated in this thesis supports them. Therefore, versioning in document formats is mostly based on versioning data and structure.

Among the meta-language family, SGML seems to support versioning in a minimalistic way but not implicitly. This is achieved by marking up version sections of the document. As a result, content within versioned markup sections can be rendered or ignored by explicitly specifying what version should be included or ignored in its DTD. XML, GML, HTML, XHTML and LaTeX do not provide any such mechanism. However, since these documents are plain text files, they can use revision control software to track version history of the document.

EPUB provides versioning support through the *publication identifiers* called *unique identifiers* and *package identifiers*. However, we do not consider them to implicitly support versioning but they provide an external accessibility to version its document content type. The package identifiers are used for minor modifications on the content without changing the identity of the document. In contrast, when a major modification is applied to an EPUB document, its identity is changed via the change in its unique identifier. In summary, we can conclude that the use of both unique identifiers and package identifiers in the EPUB metadata provides external support for document and data (content) versioning respectively.

OOXML support versioning by providing the track changes mechanism for data content, which is inspired by the Microsoft Office suite. The mechanism can track operations like insertion, deletion and even format change. The author performing the modification is also registered by the track change.

In DocBook, one is able to keep track of revision history with a specific element tag called *revhistory*. This mechanism does not explicitly track changes on the document revision evolution, but we considered DocBook as a versioning supported document format.

Table 3.1 presents an overview comparison table depicting the versioning support in existing document formats over the dimensions outlined in the previous section.

Table 3.1: Versioning support comparison of investigated document formats.

Formats	Data	Link	Structure	Change Agreggation
	Document Preparation Family			
GML	✗	✗	✗	✗
LATEX	✗	✗	✗	✗
DocBook	☑	✗	✗	✗
Scribe	✗	✗	✗	✗
	Meta-languages Family			
SGML	☑	✗	✗	✗
XML	✗	✗	✗	✗
	Print-oriented Family			
OOXML	☑	✗	✗	✗
PDF	✗	✗	✗	✗
	World Wide Web Family			
HTML	✗	✗	✗	✗
	Electronic Digital Publishing Family			
EPUB)	☑	✗	✗	✗
	Document Transformation Family			
ODA	✗	✗	✗	✗

3.2 Third Party Solution

After reviewing the requirements of versioning in a document format, we introduce third party versioning systems that provide versioning solutions to document formats. We explored how these systems address data, link and structural versioning within the document format. The versioning solution systems we have chosen (GIT [3], SVN [41] and CVS [42]) are very widely used version control system, used on the largest projects in today's software configuration management systems(SCM). Finally, we will compare the three versioning systems with respect to the four dimensions mentioned in the above section (data, link, structure and change aggregations).

3.2.1 GIT

The Git project was initiated by Linus Torvalds, who was eager to solve the need for a fast, efficient and massively distributed source code management system for the Linux kernel development [2]. In 2005, there was a break down of the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper. This was due to the fact that the tool's free-of-charge status was revoked. For this reason, the Linux development community which Linus Torvalds was a member, developed their own version tool evolving from some of the lessons learned while using BitKeeper. The resulting system yielded the following five characteristics: 1) Speed; 2) Simple design; 3) Strong support for non-linear development (thousands of parallel branches); 4) Fully distributed; 5) Able to handle large projects like the Linux kernel efficiently (speed and data size) [3]. Git is a very good content tracker as it monitors files and directories which are structured in a tree like repository.

Git Data are represented as *Objects*. These Objects are categorized into four main types; *blob*, *tree*, *commit* and *tag*. The first three objects essentially describe the main functionality in Git. The objects are stored in the Git Object Database, which is kept in the Git Directory and they are also compressed. They are referenced by the SHA-1 value of its content which is 40 characters long plus a header. The four main object types in a Git repository are:

Blob: This is the actual file under revision control.

Tree: This represent the directories which can contain others trees and blobs.

Commit: In Git, commit is the way the history parts of a tree are stored. it points to a tree and keeps an author, committer, message and any parent commits that directly preceded it.

Tag: It is a special type of commit object. It gives a permanent name to a commit and contains *object, type, tag, tagger* and a *message*.

Git versions only the data object its represents. Versions of a data object are stored in Git as a snapshot of a mini filesystem, so that, if the data object does change, then it will not store that data object again. This is in contrast to the way SVN and CVS store versioned objects, where every version is a copy or cheap copy of its predecessor. The Git data object storage mechanism claims that the data reuse via the snapshot technique is significant in reducing merge conflicts and the complexity of resolving them. Git system like most version control systems only supports data versioning.

3.2.2 SVN

Subversion (SVN) is an open source software used to maintain the historical version of a file such as source code, web pages and so on. It was developed to succeed CVS. Thus by addressing the limitations that CVS had. SVN was created in 2000 by CollabNet[2]. SVN is not much more different than CVS; it stores the history of files, a user can check out a working copy of the files to be able to work locally, comparison of versions is also possible. SVN can version directories. CVS remembers the history of single files, whereas SVN manages the history of files, meaning that it tracks changes of the whole directory trees. Therefore it manages files and directories [1]. SVN merges revisions by locking the repository while it is in use called the *lock modify unlock*. This locking mechanism is a drawback as locking can lead to deadlock. It also has another merging mechanism called the *copy modify merge*. This technique requires personal working copy of a revision to the updated anytime a modification is made on the revision copy in the repository. The structure of SVN is comprised of three main parts:

Trunk: is the main body of development, originating from the start of the project until the present.

Tag: will be a point in time on the trunk or a branch that you wish to preserve. The two main reasons for preservation would be that either this is a major release of the software or this is the most stable point of the software before major revisions on the trunk were applied.

Branches: will be a copy of main file derived from a certain point in the trunk that is used for applying major changes to the file while preserving the integrity of the original file in the trunk. If the major changes work according to plan, they are usually merged back into the trunk.

SVN manages files and directories the same way, thus cannot distinct between them, so that the same information concerning release history applies to both the directory and file [1]. The fundamental aspect of version control in SVN is the branching and merging of related revisions.

[2]CollabNet is a software company that develops software development and application lifecycle management tools.

Futhermore, we will introduce the basic ideas on how revision control are achieved in a simple operation but yet complex to implement. Each time the repository accepts a commit, this creates a new state of the file system tree, called a *revision*. Revisions are each assigned a unique natural number, usually one greater than the number assigned to the previous revision and each revision number will select an entire tree, not just individual files after some committed change is made on the repository. The method used in SVN to store the files and directories is called *the bubble-up method*. With bubble-up method, we expect that SVN simply copies the entire working node (this means: files or directories) but rather it stores the latest revision as a full text, and previous revisions as a succession of reverse delta of the node. We illustrate with the diagram shown in Figure refsvnreverse.

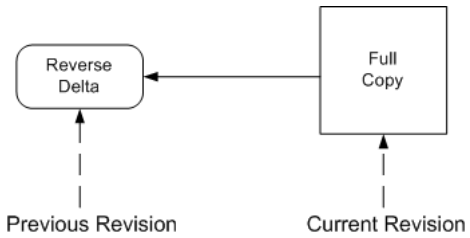

Figure 3.5: The bubble up method using reverse delta.

Branching in SVN solves the problem "isolation", here one tries not to interfere with the server repository in order not to make minor changes that may annoy other members in the working group. It is a usual practice in software development to have peer review and feedback, as a working group member may advance in a wrong direction in their task for weeks before someone notices their mistakes. In SVN, you create your branch in the repository which copies the most recent working files or directories you will be working on. Here, other members of your team can view your progress and make reviews on your work.

Finally, SVN like Git lack the initiative of link and structural versioning. Therefore, only data (mostly plain text files) are versioned and stored persistently.

3.2.3 CVS

Concurrent Versions System (CVS) is one of the most popular and efficient revision control system. CVS is a client-server free software revision control system in the field of software development and was originally created as a series of shell script written by Dick Grune [7]. It's also based on the Revision Control System (RCS) which was first developed by Walter F. Tichy at Purdue University in the early 1980s [48]; and was mainly used to improve performance by storing an entire copy of the most recent version and then stores reverse differences which were mostly text files such as source codes or configuration files.

The main structure of a CVS repository is a tree mapped to the directories in the *working directory*. The CVS repository stores the entire copy of all the files and directories which are under the intended revision control. Note that CVS works on files and not directories. We are expected to store our changes as we proceed and it is most likely we lose files locally. The repository should then be able to help us with a rollback to an earlier committed version. This mean the repository is located on a different machine. When we access the CVS repository, we don't do this directly but rather we copy the files into a *working directory*.

Ordinarily, a file's revision history is a linear series of increments [7]. Usually, in large project environments, it is ideal that the main line of development is splitted into several parallel lines, known as *Branches*. Branches are given numbers to each branch and are attached to the revision where they were branched off. Revisions on a branch are also numbered such that CVS appends the old branch number to the corresponding revision number.

In CVS, any conflicts that are not auto-mergeable will be skipped. This means that the merged branches most have branched off from an existing ancestor in the revision history. Branching in CVS is similar to the way SVN does, but the major difference is that SVN allows the whole repository to get the version number, but not each separate file.

When editing a file under version control, a new revision of that file is made along with a modification report such as log messages or history database for that revision. When reviewing the modification reports of files under version control in more detail, i.e. into commit messages, certain patterns can be found that might be useful in constructing the broken link between versioning and issue tracking. Thus these reports provide textual description of the changes made as well as some additional information. We can use these reports to retrieve more information about the file's history. Every revision is a copy of its parent revision accompanied with the addition of the revision information in the modification report. Ultimately, CVS only supports data versioning.

3.2.4 Versioning Support in Third Party System

This section presents a comparison of the versioning solution systems mentioned in the above sections. The comparison is based on the versioning support requirements needed in versioning for document formats. We mentioned in the previous section the importance to have data, links and structures versioned and the need to store a change aggregation to enable the re-usage of any document's logical structure nodes.

In Git [3] and SVN [41], content re-usage is a factor in their versioning solution. Git uses the idea of making a snapshot of the state of a data object that will never be changed for re-usage of content. SVN approach is different. SVN uses the idea of storing the parent revision as a full text, and successive revision a reverse change aggregation of the parent version. This process is called the bubble-up method, where other revisions are cheap copies of the parent version. In contrast, CVS does not support content re-usage. Revisions

that share data information units from previous revisions must copy the shared data physically to its version environment.

All three versioning systems have failed to version the links between revisions. SVN and CVS did not provide documentation on how a revision structure can be reused. Therefore, we assume both systems lack the innovation to version structures. On the other hand, Git argues that using the snapshot mechanism for revision storage promotes the ability for structures to be versioned. Thus with a snapshot of a structure object, users can replace data node within the structure object with a different node content.

Finally, we present our comparison in Table 3.2 on GIT, SVN and CVS with respect to the support of data versioning, link versioning, structure versioning and change aggregation.

Table 3.2: Comparing Versioning Support in Third Party Solutions

Features	CVS	Subversion	GIT
Data Versioning	Supported	Supported	Supported
Link Versioning	Not Supported	Not Supported	Not Supported
Structure Versioning	Not Supported	Not Supported	Supported
Change Aggregation	Not Supported	Supported	Supported

The result from table shows that GIT, SVN and CVS do not support link versioning like all of the document formats we reviewed. Likewise, structure versioning is not supported by SVN and CVS. The support for change aggregation by GIT and SVN reflects the important providing this concept in the versioning supported document format model. Therefore, in the following chapter, we will introduce the support of these concepts as we design a versioning model to extend the FCMD metamodel.

Chapter 4

Versioning Models for Extending FCMD metamodel

4.1 Introduction

Considering the dimensions (*data versioning*, *link versioning*, *structure versioning* and *change aggregation*) we have defined in chapter 3 that should be supported when versioning a document format, we have extended the FCMD metamodel to adopt these versioning capabilities. In this chapter, four models have been proposed to extend the FCMD model. Three of the models described will be merged to create the fourth. Thus, each of the three models strengths and weaknesses in terms of the four dimensions are identified and used to create the fourth model. This chapter is structured as follows: section 4.2 presents the definition of each model. An illustration of a scenario to help us measure the strengths and weaknesses of each model based on our proposed dimensions is described in section 4.3. Finally, section 4.4 entails what dimensions were supported by each model, and the effect of each model to the overall design of the FCMD metamodel.

4.2 Versioning Models

After reviewing the document formats, their logical structures and versioning solutions, our versioning model aims to address the drawbacks that these solutions failed to tackle. Our versioning model also supports the important features that these solutions presented like persistent data versioning, etc. Our plan for achieving these goals is to present different versioning models, presenting the strength of features they possess. Likewise, we identified the difficulties they introduce. Henceforth, we were able to refine and create a final model that

comprised their strengths and excluded as much as possible their weaknesses. The goal of our versioning model for extending FCMD metamodel is as follows:

1. Support data, link and structure versioning.
2. Support deltas (change aggregation).
3. Achieve a balance between computational and memory performance for storing and retrieving a revision.

In the following sections, we will introduce our four versioning models, namely: *Delta Property Model*, *Component Versioning Model*, *Link Versioning Model* and *FCMD Versioning Model*.

4.2.1 Delta Property Model

The idea behind this versioning concept is similar to the *bubble-up method* in SVN [41]. The concept is to provide a complex RSL type that stores the changeset of operations needed to create a new revision. We have named this new RSL type the *Delta Property*. A delta property type is a collection of change operations that when applied on a RSL *entity* type, creates a new revision of an entity. Thus only the root revision entity is stored in full and for each subsequent revisions, only its delta property is stored. This process is known as *forward diff version control* [49].

We depict the delta property versioning model in Figure 4.1. The association *HasDeltaProperty* cardinality constraints (0,*) on the entity side signifies that an entity may have zero or several delta properties. Thus (1,1) at the delta property end indicates that each delta property is always associated with exactly one entity.

Figure 4.1: Applying Version Delta Property to FCMD

While this model supports the creation of a new revision using content reusage of the root revision, subsequent revisions on a child revision will require the child revision to be computed and stored. In effect, the child revision will need to store its own delta properties for its revision. However, persistently storing all old revisions as diffs against the current revision makes retrieval time faster. This model supports all our versioning dimensions: data versioning, link versioning, structure versioning and change aggregation.

The downside to this model is not only the complex resolver required for collaborative developments to tackle merging conflicts but also it makes storing slower, since we are required to recompute the delta operations each time we save a revision. To solve this issue, SVN [41] proposed the used of *Least Recently*

Used (LRU) algorithm as an optimization concept keeping a LRU cache of the old revisions that have been retrieved. Another drawback in this model is that both revision data and the change operations are stored in the delta property, making the revision data unlikely to be reused.

4.2.2 Component Versioning Model

The component versioning model is a concept normally used by most hypermedia systems that support versioning like CoVer [16] and VerSe [22]. The concept is to have a linkless versioning environment. To support versioning, a container-like type must be defined that holds all the data versions. Furthermore, composed data revisions within a container can also become a container themselves to other revisions.

In Figure 4.2, we depict the component versioning model. The association *HasResource* signifies that a resource is embedded within another resource. The cardinality (1,1) on the resource restricts a composed resource to belong to a single resource container. A resource can comprise from zero to many resources with the cardinality (0,*). The delta property in this model is applied the same way as in the delta property versioning model. However, unlike the delta property versioning model, revision data and change operations are separated. Here the association *HasDeltaProperty* indicated at the cardinality constraints (0,*) on the resource side signifies that a resource may have zero or several delta properties. Each delta property applied defines a new revision. Thus (1,1) at the delta property end indicates that each delta property is always associated with exactly one resource.

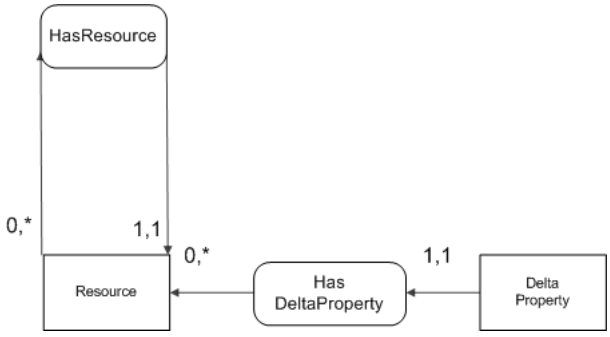

Figure 4.2: Component Versioning Model

To compute a new revision, the revision data (composed resource) and the delta property must be embedded to the parent revision (container resource). The delta property provides the parent revision with the information needed on how to apply the composed resource. To retrieve a given revision, If there is a nested hierarchy of resource composition then from the root container, each child composed resource and its delta property are used to reconstruct the revision recursively to the needed child resource. We illustrate this reconstruction of a

revision from a nested hierarchical resource composition with Figure 4.3. From the figure, in order to retrieve *revision 3*, first we must construct *revision 2* by applying *resource 2* and its delta property to *revision 1*. Note that *resource 3* is the root resource container and is referred as the *revision 1*. Next, we use the same logic to reconstruct *revision 3*, *resource 1* and its delta property are applied to *revision 2*.

The component versioning model lacks the support for both link versioning and structure versioning. However it supports data versioning and change aggregation. Memory performance for this model is better than that of the delta property versioning as complete data re-usage is possible.

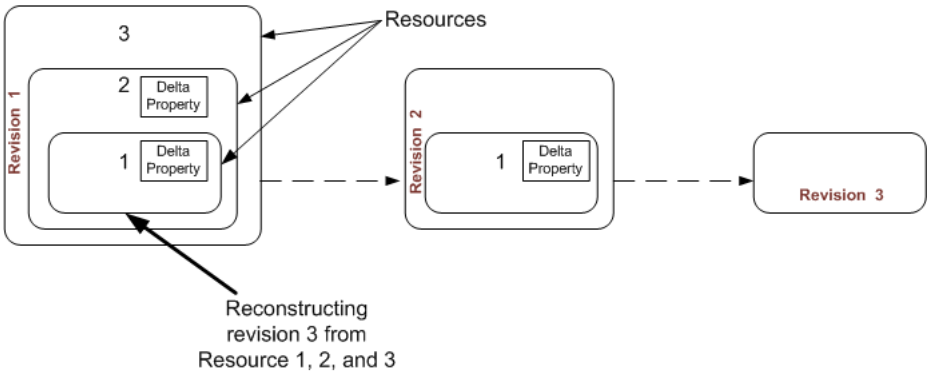

Figure 4.3: Reconstructing a revision with the component versioning model.

4.2.3 Link Versioning Model

The idea of this versioning model is to use a link reference to associate all the entities that are collective revisions of a source entity. The link reference here is called the *Version Link* element which is a subtype of the *Link* entity in RSL model. The main difference between a link type and version link type is that a version link has a single source entity that represents the parent revision to the target revisions.

As shown in Figure 4.4, the association *HasEVersion* is ordered and is a subassociation of the *HasTarget* association in RSL Model, which defines the relationship of the source revision to a target revision. The cardinality (1,*) on the version link side indicates that a revision may be connected to via one to many version links. This enables versioning of the version link itself. A version link may only have a single source entity and we indicate that with the cardinality (1,1) on both side of the *HasESource* association.

This model does not provide content reusability. As a result, new revisions that share information units with their predecessor must copy them. Though it provides the intuition for versioning links for content re-usage, it failed to provide the core mechanism like the introduction of the delta property type defined in the previous models. However, revision storage and retrieval are better than

both previously mentioned models. To illustrate the link version model version space, we depict how logically revisions are organized in Figure 4.5.

Our link model supports only data versioning and does not support the other versioning goals we proposed but has provided us with the foundation needed to support them with respect to better revision storage and retrieval time. To compare this model with the previously defined versioning models, we presented a comparison table in section 4.4 that reflects the strengths and weaknesses of each model. This comparison is based on the proposed versioning requirements and the cost (with respect to time and memory usage) of storing and retrieving a revision.

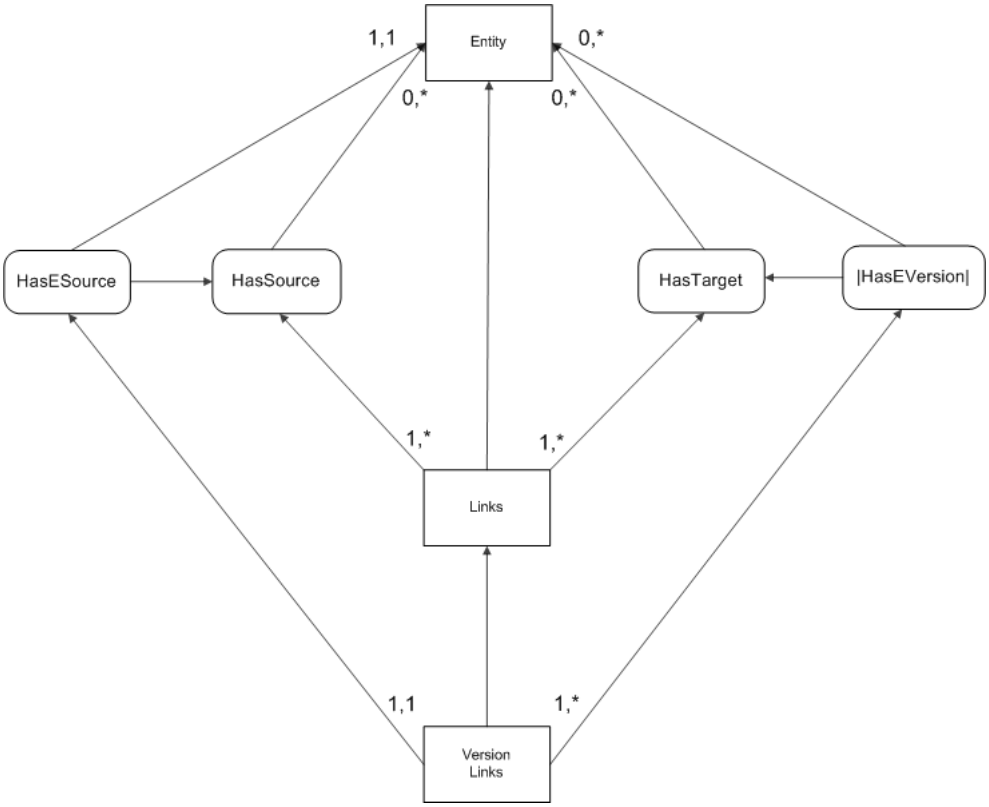

Figure 4.4: Link Versioning Model

4.2.4 FCMD Versioning Model

To address the drawbacks of the previous models described above, we present the final versioning model named *FCMD versioning model*. This model identified the lack of separating revision data from delta operations in the delta property versioning model in order to support the re-usage of this data. Although the component versioning model resolved this issue, it failed to support

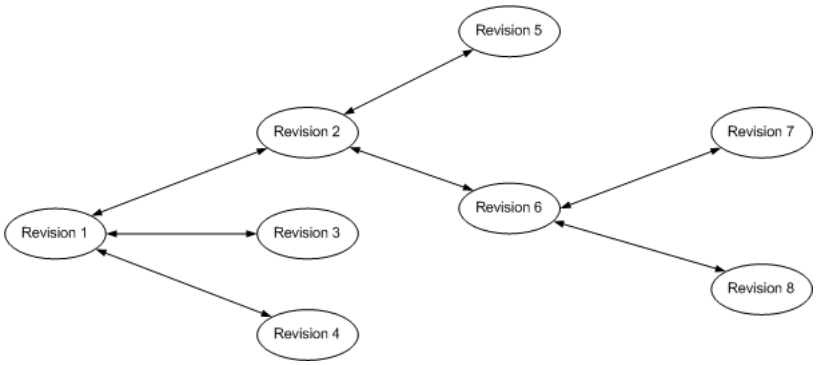

Figure 4.5: A simple version space of link versioning model.

link and structure versioning. The link versioning model provided a better organization of revisions, thus giving the intuition on how we can use these links to carry change aggregation while separating data units from change operations. To summarize, we have used the content re-usage concept from component versioning model, the revision organization concept of the link versioning model and the change aggregation concept of the delta property model.

From Figure 4.6, all element types and association types are the same as the link versioning model except the introduction of the *delta property* element and *HasDeltaProperty* association. Here, (1,1) cardinality on both side of the *HasDeltaProperty* association indicates that every version link has at most one delta property. Likewise, each delta property belongs to at most one version link.

To reconstruct a revision, the FCMD versioning model provides support for both *forward deltas*[1] and *reverse deltas*[2] storage and retrieval method. By keeping these deltas on the version links, we can reconstruct a revision using the forward and reverse deltas technique. As a result, provides our model with a balance between storage requirement and retrieval time due to a revision computation.

The FCMD versioning model supports all our versioning goals: data versioning, link versioning, structure version and change aggregation. To illustrate the support for our goals, Figure 4.7 depict the version situations. In Figure 4.7 (1), *revision 1* and *revision 2* may be data nodes or structural nodes of the FCMD document. The *version link* here depicts the version relationship between both revisions. The relationship could be simple (connecting revisions of different states) or complex (providing complex deltas on how either revision can be computed from the other). Two revisions can be represented with link versioning in FCMD as shown in Figure 4.7 (2). Thus *version link 1* may contain a delta property that structures the modification of *node 1* using *node 2*

[1]The forward delta storage technique defines a method that keeps a full copy of the parent version and just keeps the difference between that parent copy and the current version.

[2]The reverse delta storage technique defines a method that keeps a full copy of the current version and just keep the difference between that copy and the parent version.

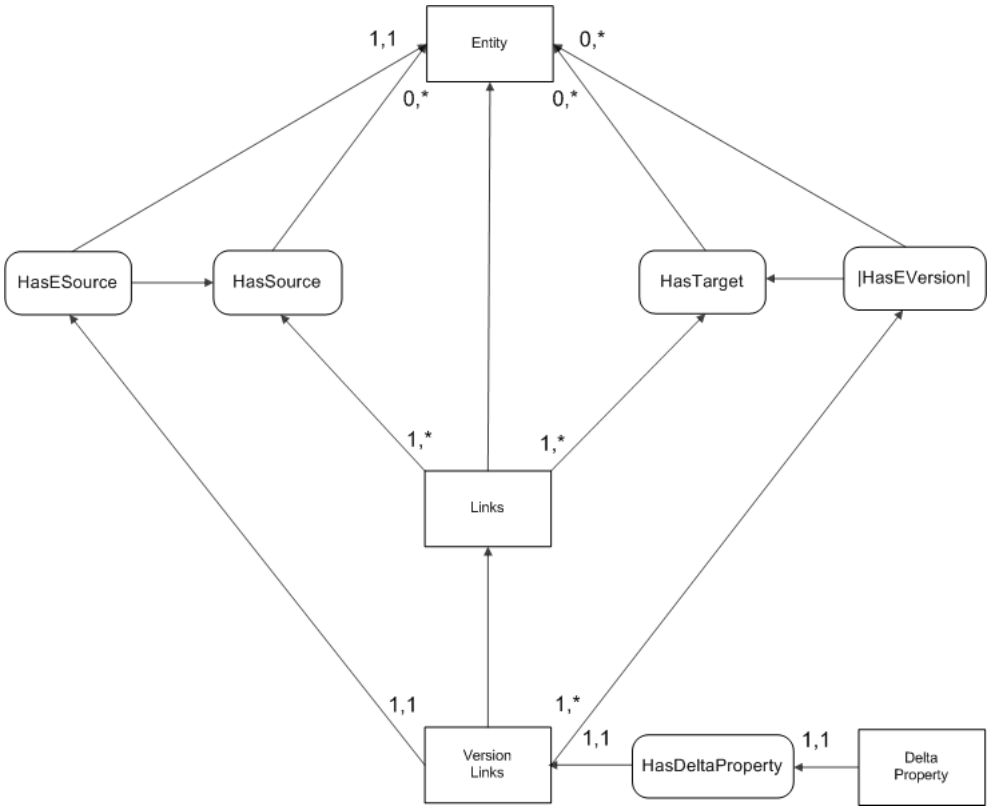

Figure 4.6: FCMD Versioning Model.

data unit which are different from that of the delta property in *version link 2*. For example, say node 1 represents a paragragh with content "Versioning links with strutural node" and node 2 represents a single word with content "two". Thus *version link 1* may have a delta property to insert node 2 content into the last position of node 1 content in this way: "Versioning links with strutural node **two**". Meanwhile *version link 2* may have a delta property to insert node 2 content at the different location; "Versioning links with **two** strutural node".

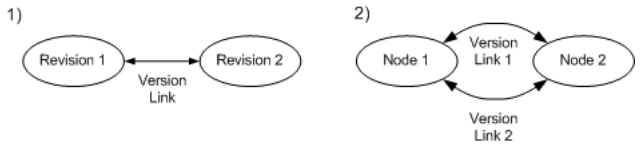

Figure 4.7: FCMD versioning model support scenario: 1) Data or Structure Versioning. 2) Link Versioning.

After describing all four versioning models, we will present a scenario to demonstrate storage and retrieval of revisions for all four models.

4.3 Scenario: Accessing Revisions

To understand the conceptual idea behind each of our versioning models, a scenario has been coined for the purpose of comparing the strengths and weaknesses of these versioning models. We will consider a simple revision storage and retrieval scenario in which all of the versioning goals discussed above can be observed. Figure 4.8 shows the scenario we will apply to all four models.

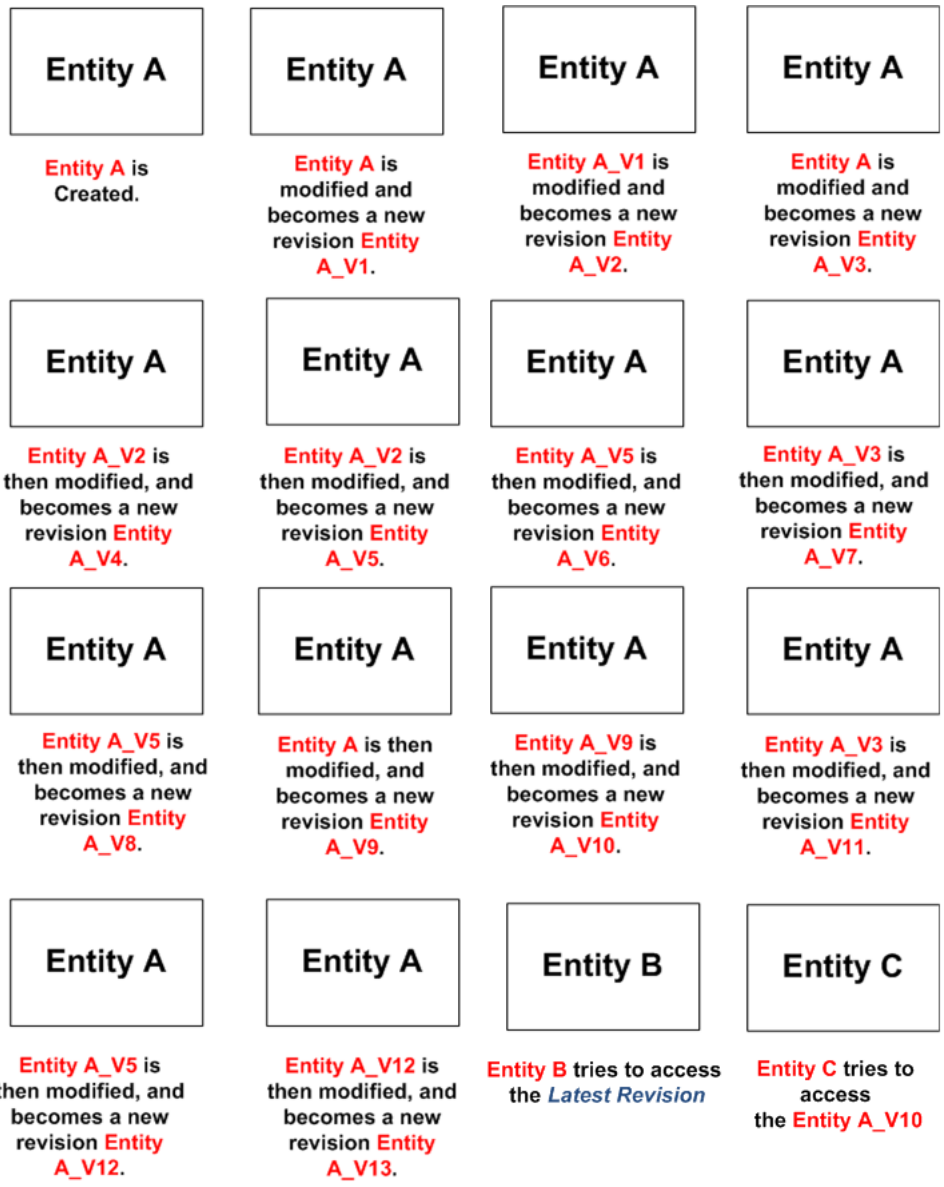

Figure 4.8: Accessing revisions scenario

4.3.1 Scenario: Delta Property Versioning Accessing Revisions

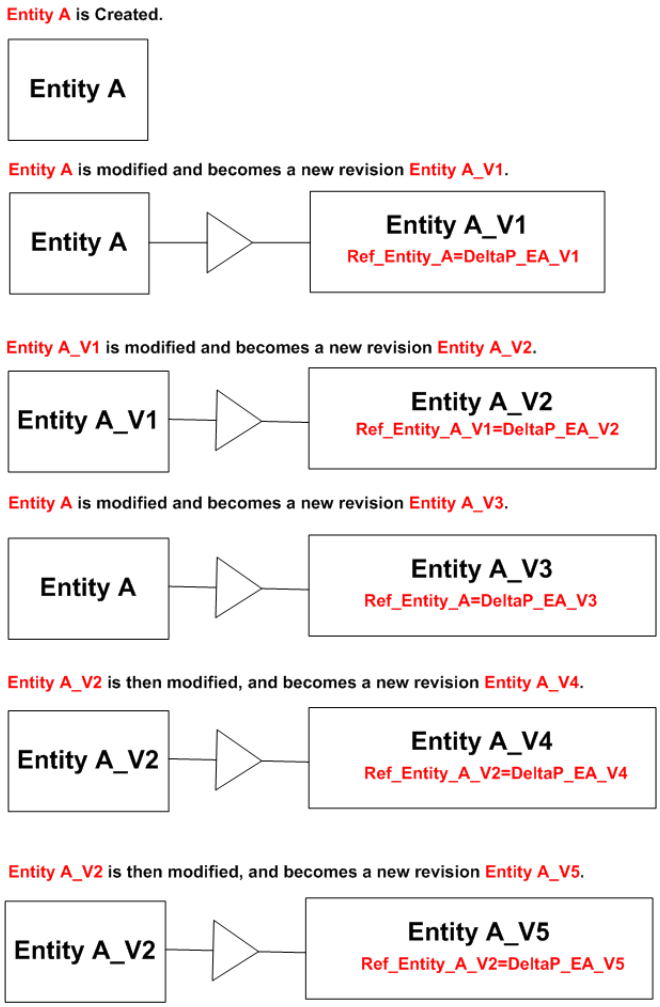

Figure 4.9: Delta property versioning illustration on the access scenario is depicted using *forward delta technique*. Part 1.

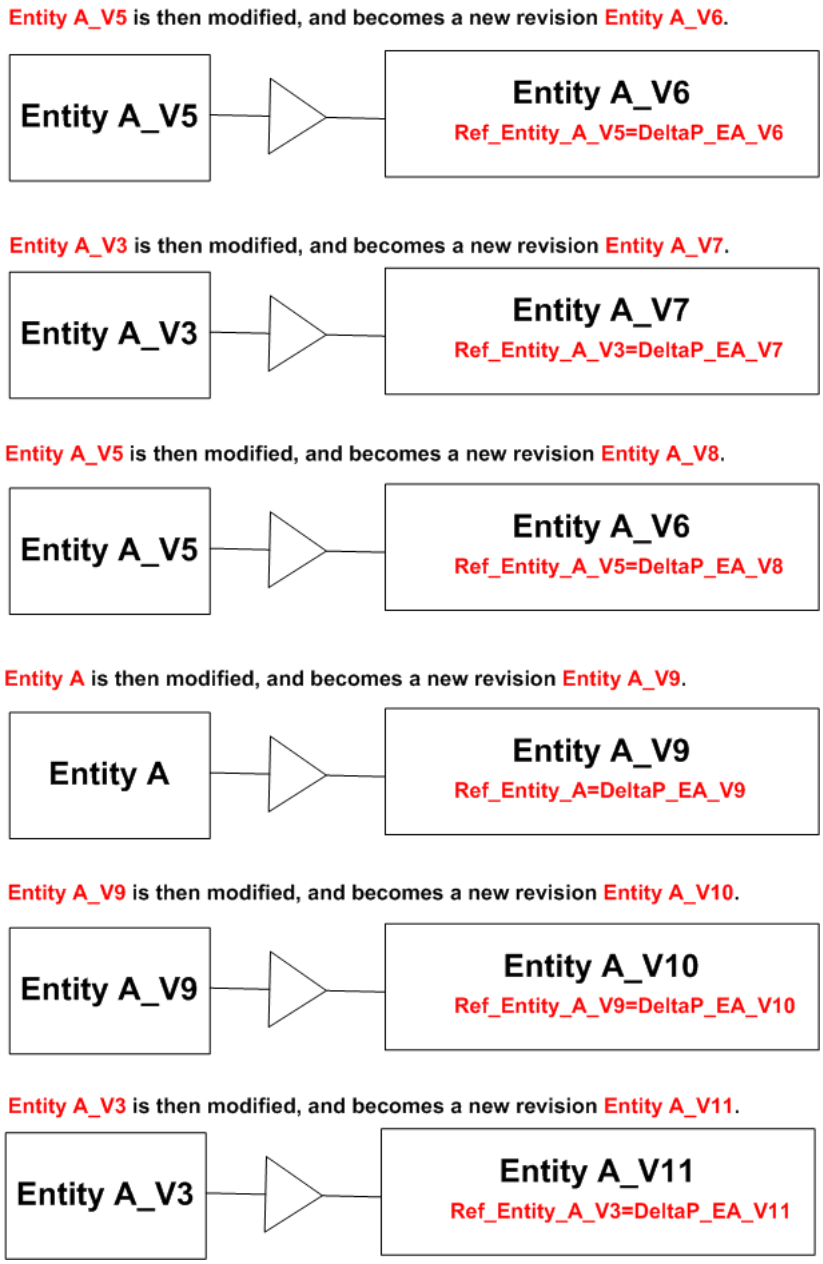

Figure 4.10: Delta property versioning illustration on the access scenario is depicted using *forward delta technique*. Part 2.

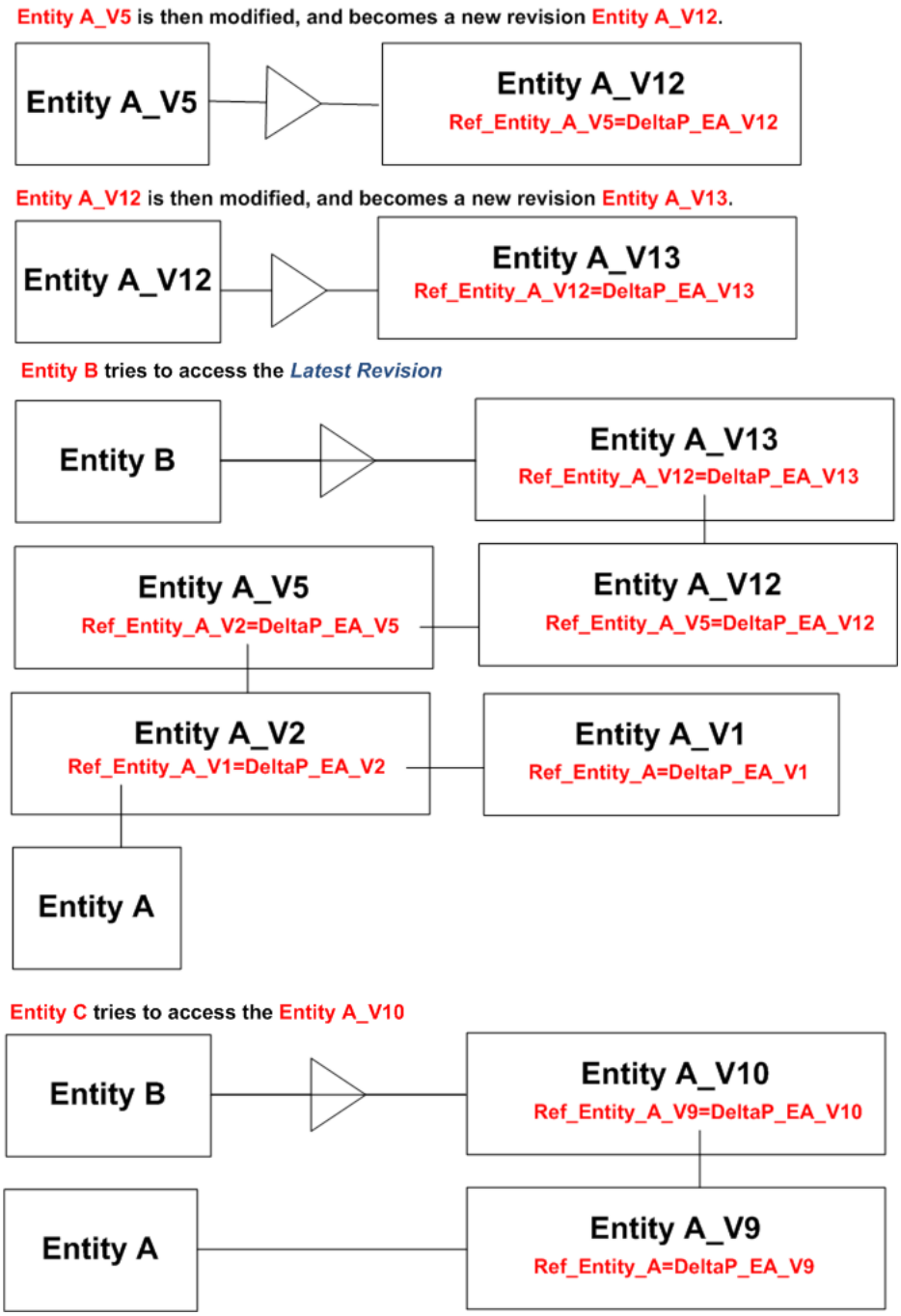

Figure 4.11: Delta property versioning illustration on the access scenario is depicted using *forward delta technique*. Part 3.

4.3.2 Scenario: Component Versioning Revisions

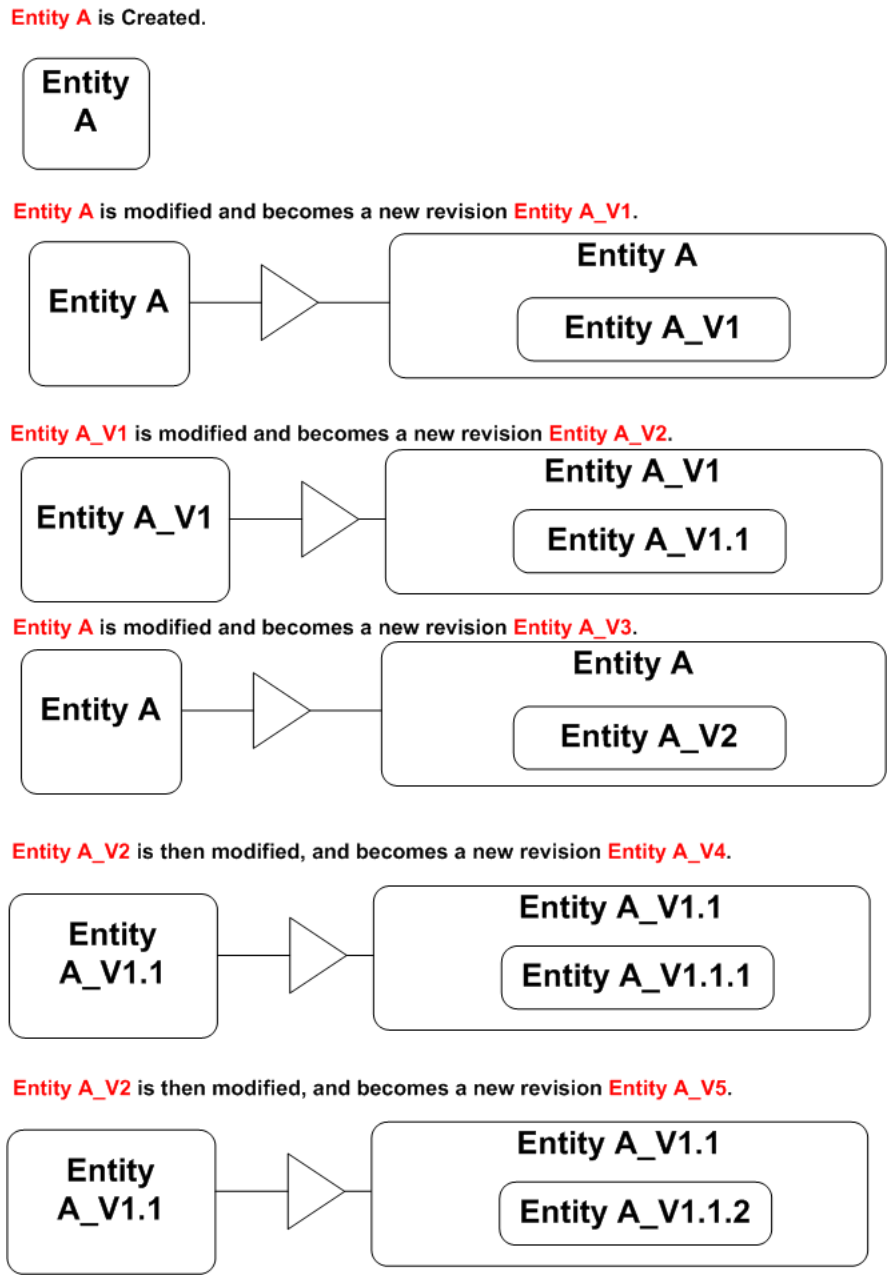

Figure 4.12: Illustration for the access scenario in component versioning model. Part 1.

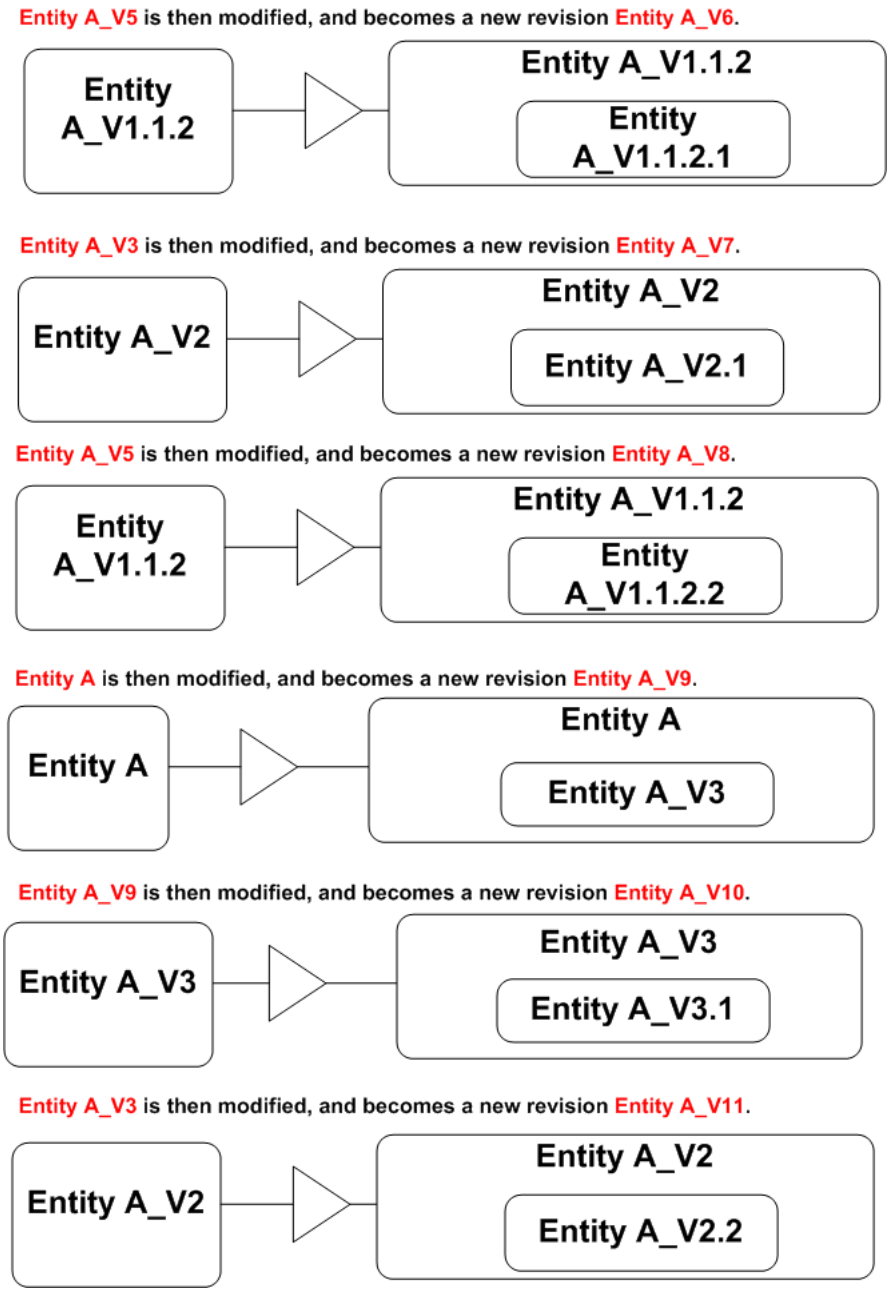

Figure 4.13: Illustration for the access scenario in component versioning model. Part 2.

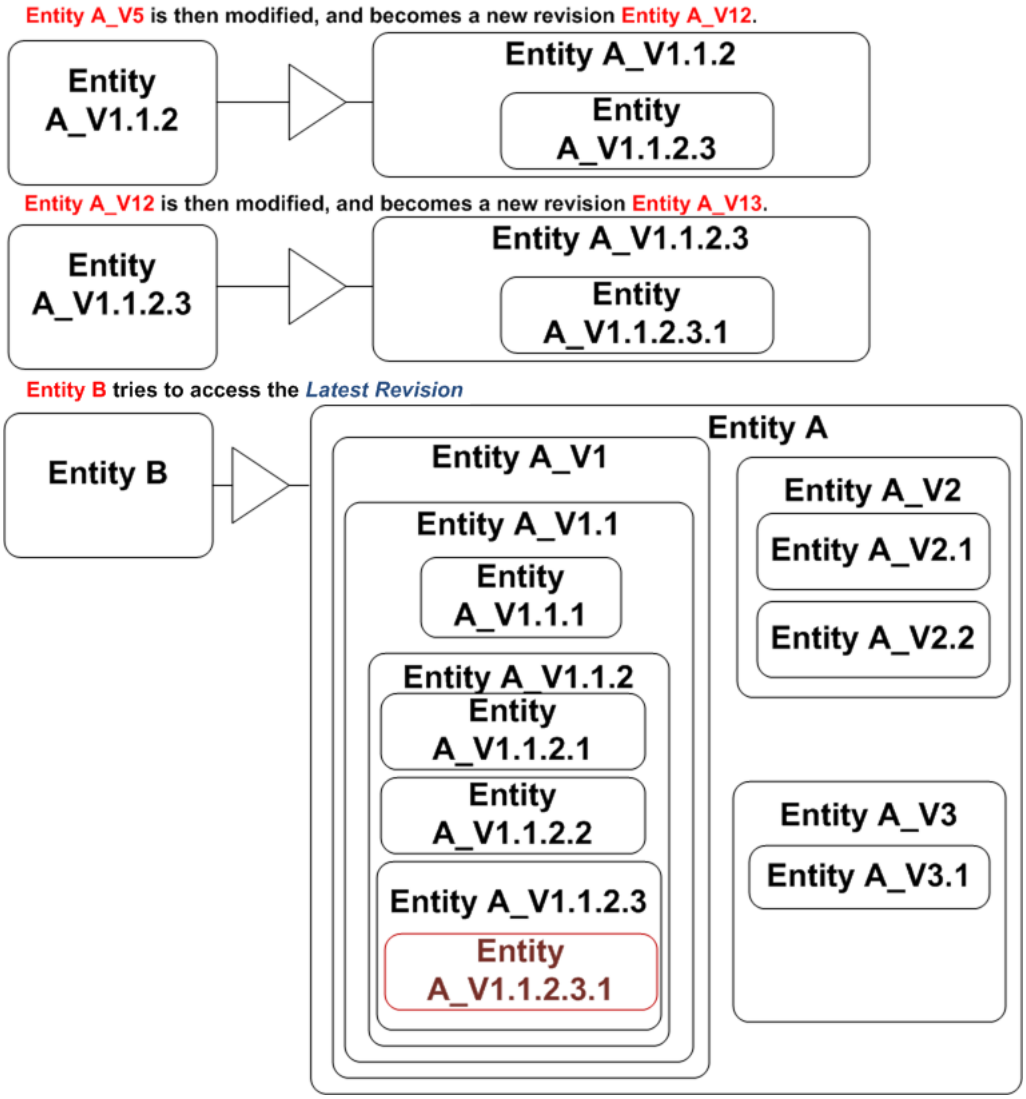

Figure 4.14: Illustration for the access scenario in component versioning model. Part 3.

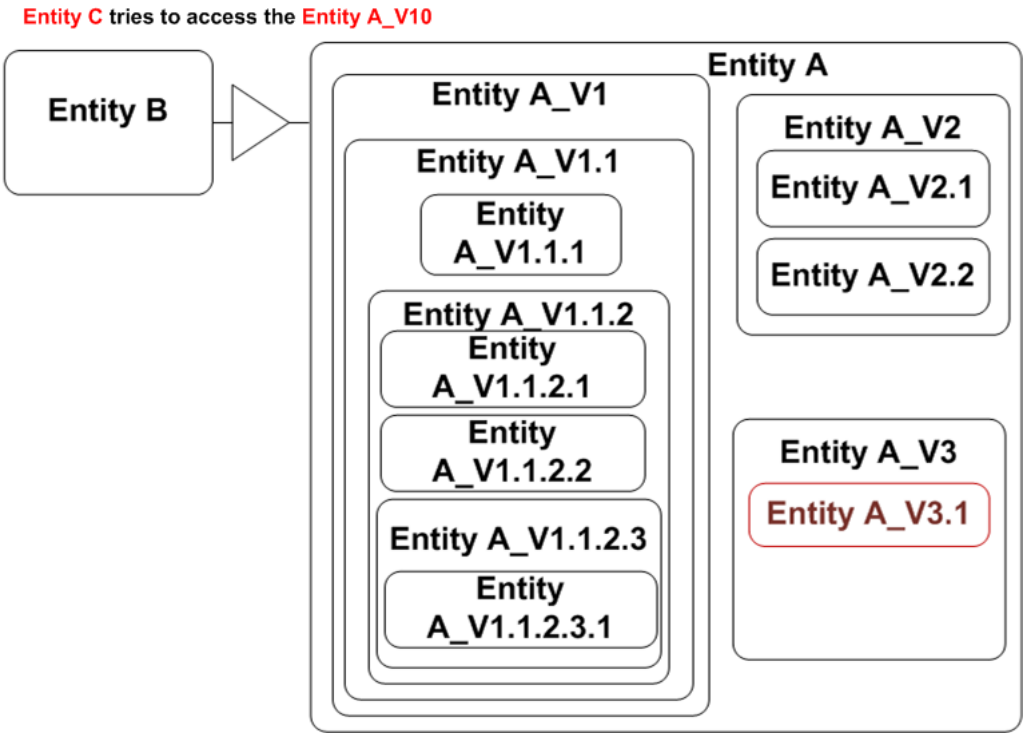

Figure 4.15: Illustration for the access scenario in component versioning model. Part 4.

4.3.3 Scenario: Link Versioning Accessing Revisions

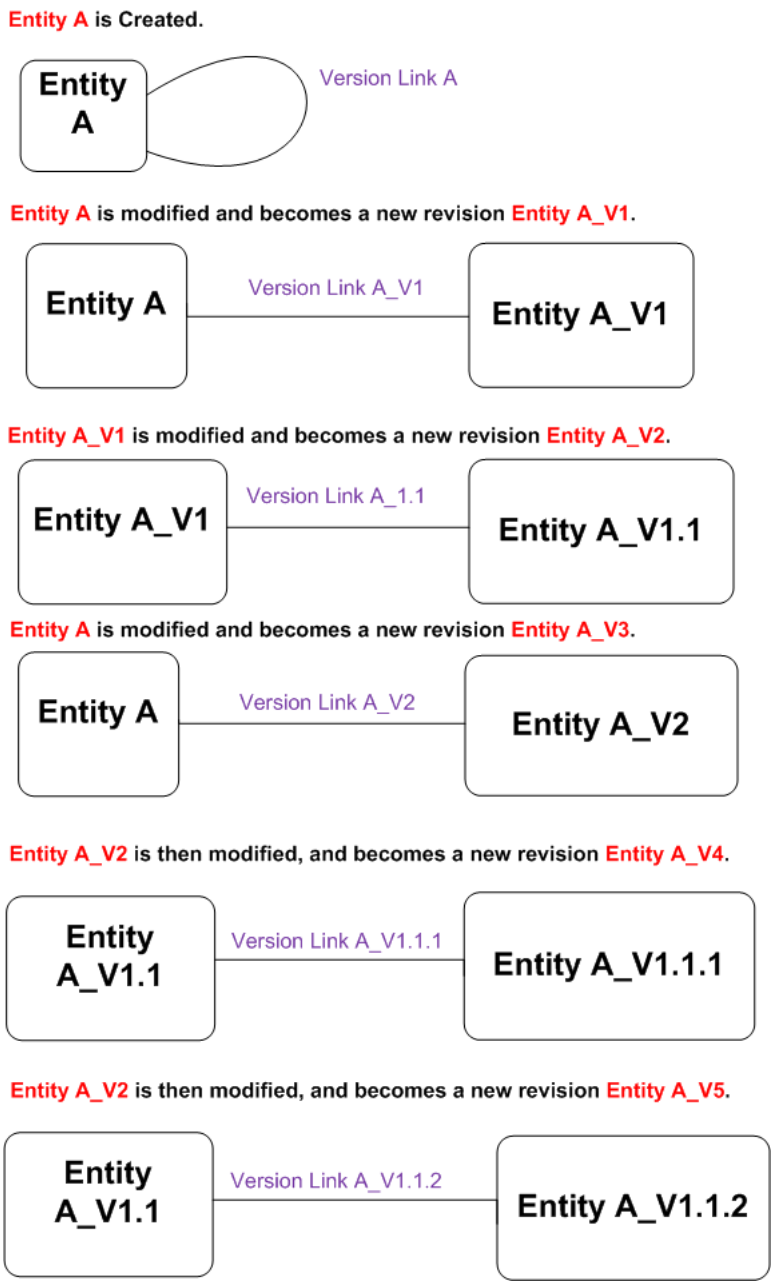

Figure 4.16: Illustration for the access scenario in link versioning model. Part 1.

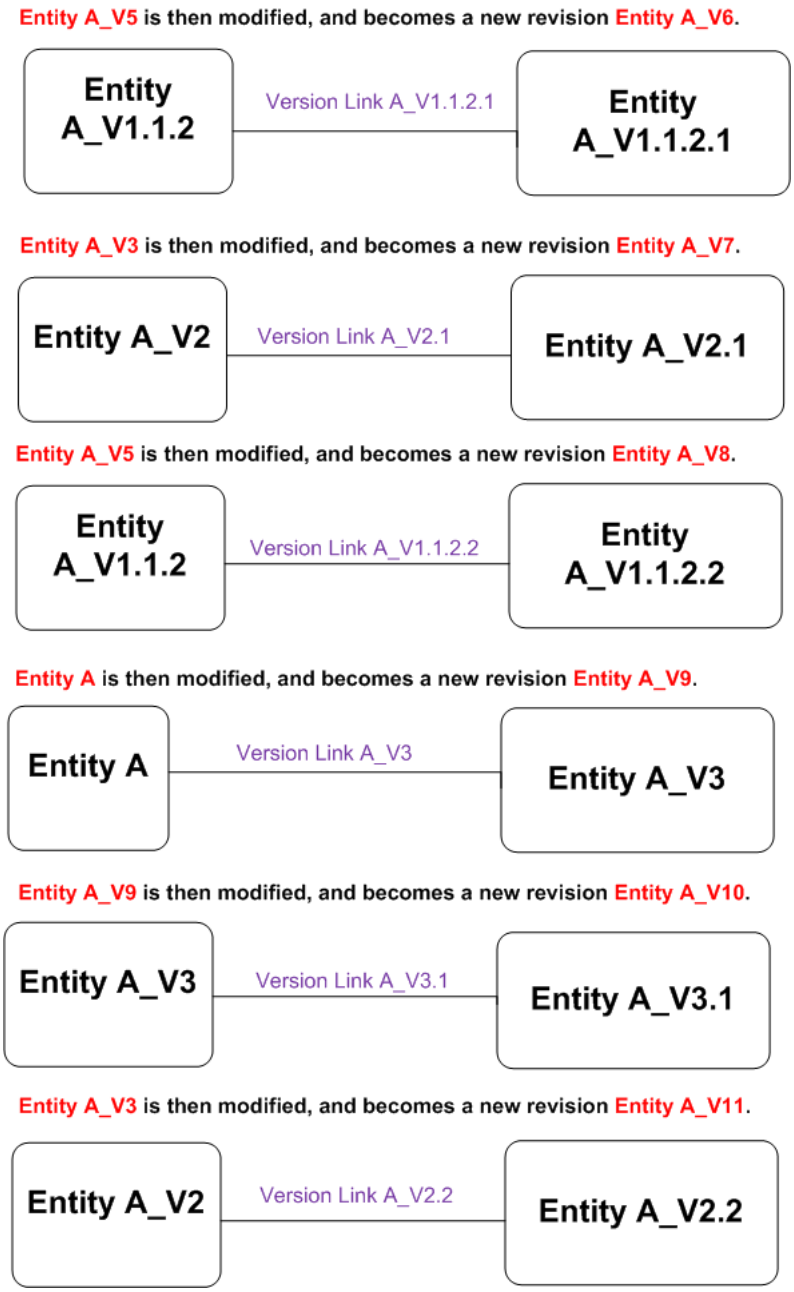

Figure 4.17: Illustration for the access scenario in link versioning model. Part 2.

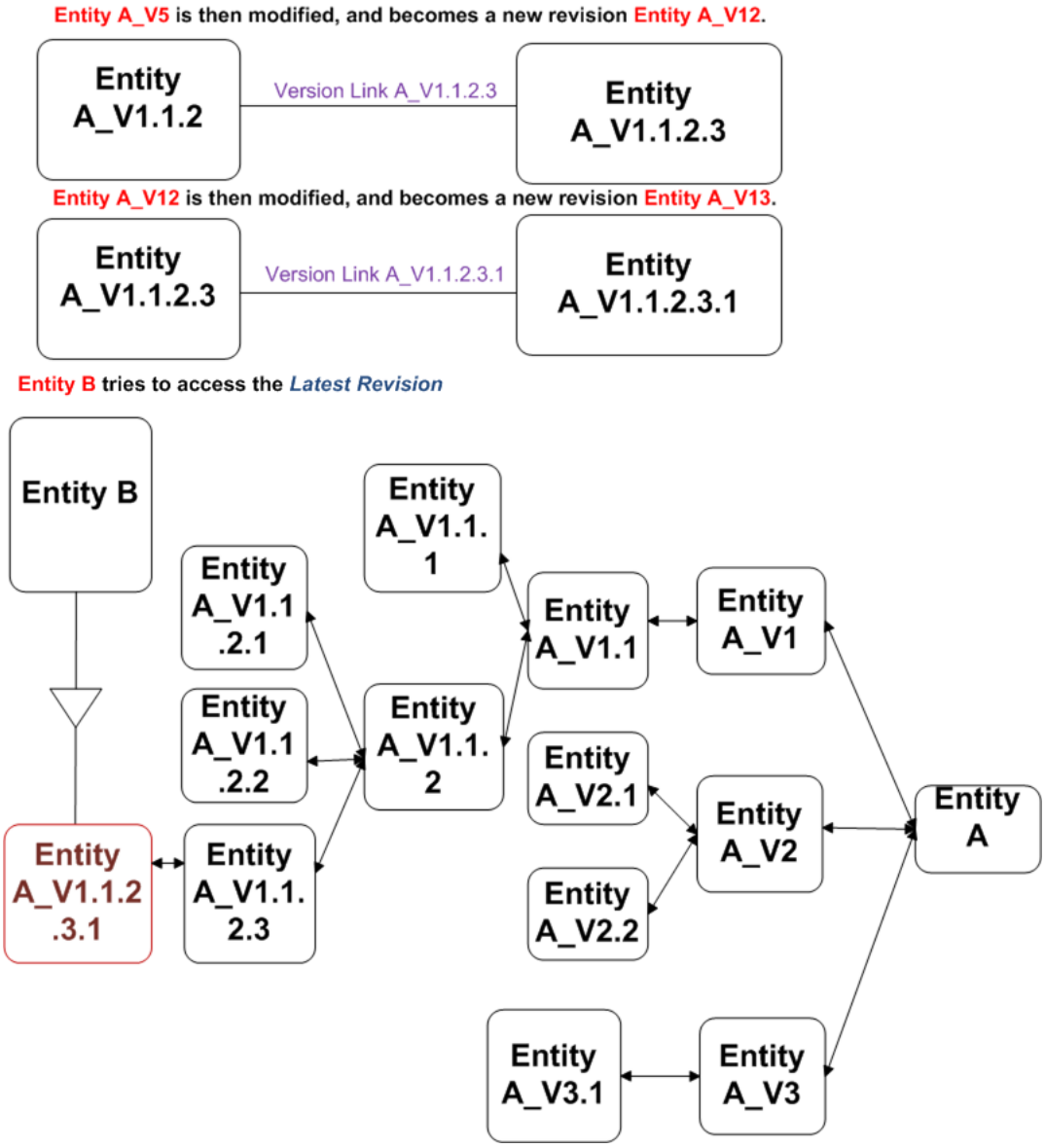

Figure 4.18: Illustration for the access scenario in link versioning model. Part 3.

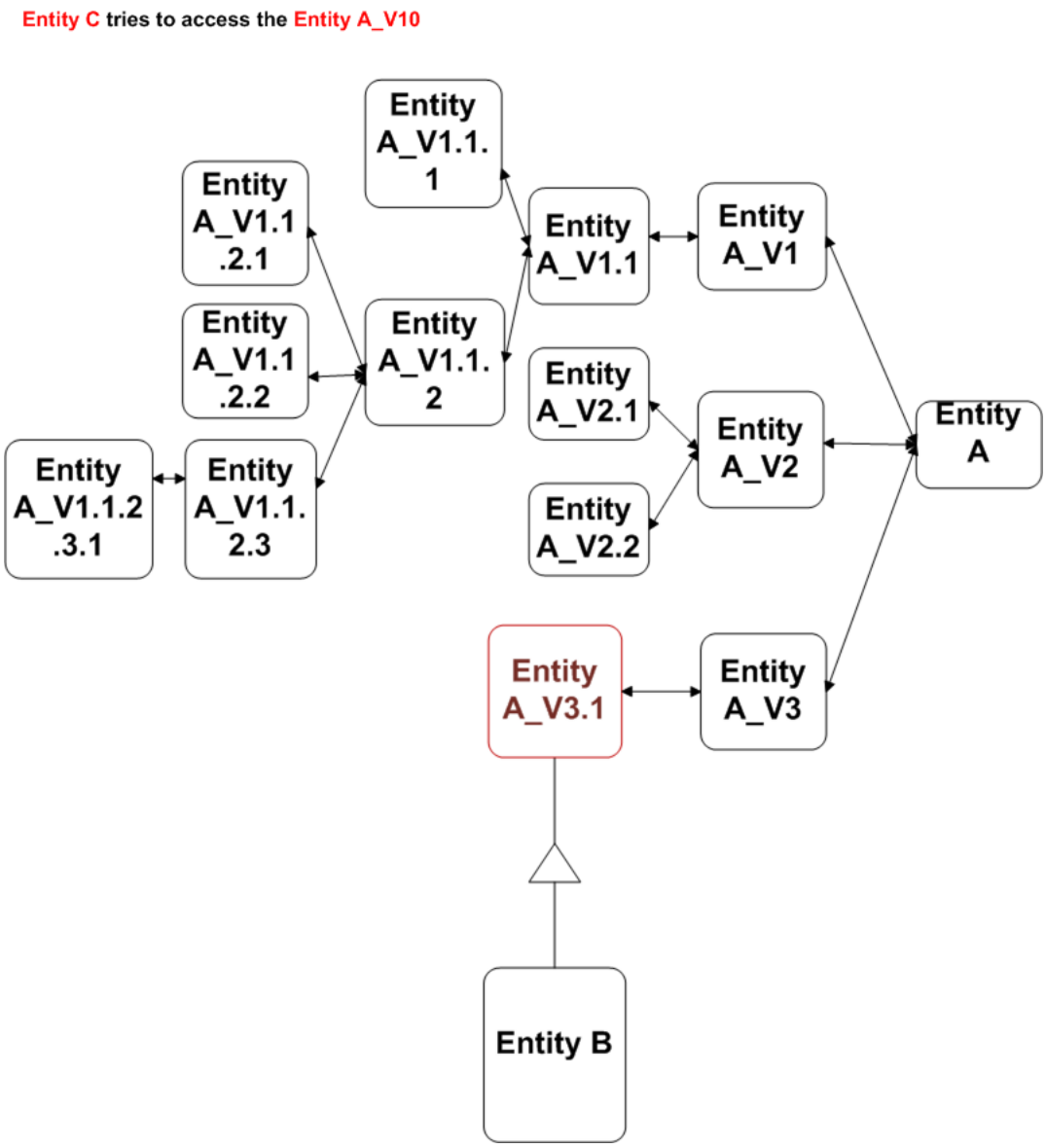

Figure 4.19: Illustration for the access scenario in link versioning model. Part 4.

4.3.4 Scenario: FCMD Versioning Revisions

Since the FCMD versioning model and the link versioning model have a similar version space that depicts the node organization, we will only depict the retrieval scenario. The difference between both versioning models is that the FCMD

versioning model supports delta operations on the versioning link via the delta property element. Figure 4.20 shows the retrieval of the last revision from the scenario in Figure 4.8.

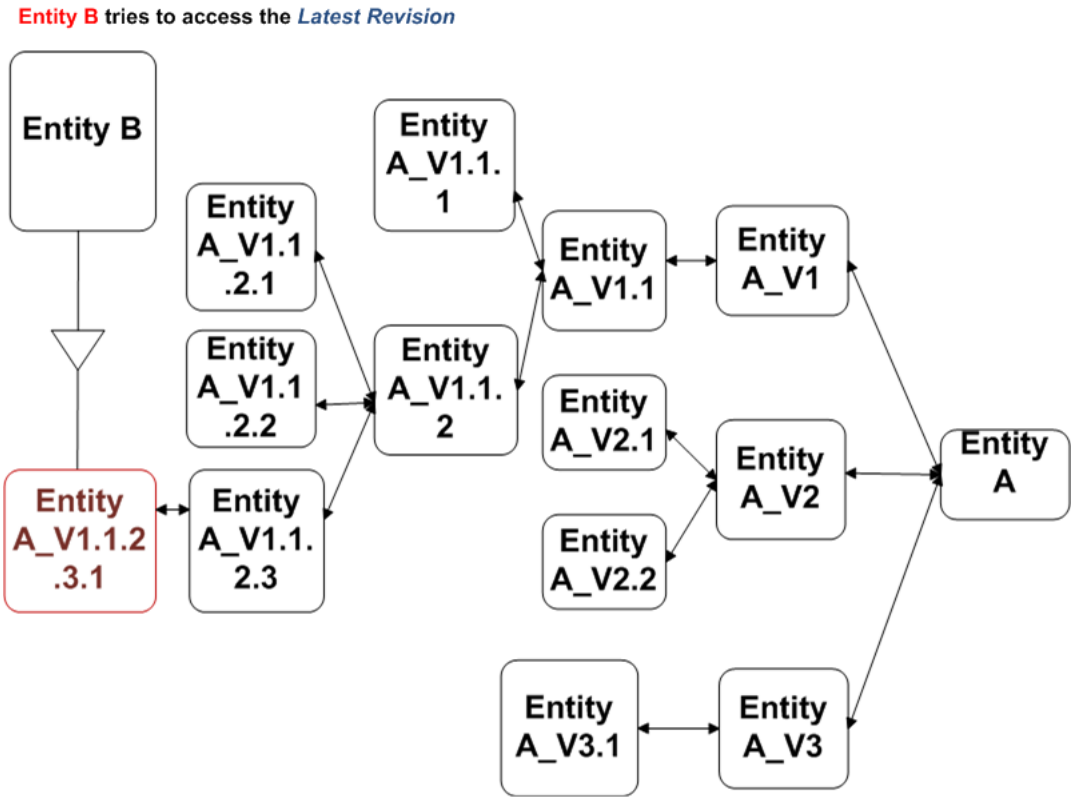

Figure 4.20: Illustration for the access scenario in the FCMD versioning model.

Next we will show the retrieval process using the forward delta technique. Note that the retrieval process using the reverse delta technique for accessing the latest revision means that the revision is already available in storage, thus access is immediate. From Figure 4.21, to retrieve the latest revision using the forward delta technique, we must recompute all the version links along the path to the root version. Furthermore, version nodes in the FCMD versioning model are not revisions themselves, thus they contain the data units that are shared by ancestor nodes. In fact, a revision is the computation of the version link's delta operations and the revision node on the parent revision node. Thus, if you have two version links with the same source (parent) node and target (child) node then two revisions have to be computed. In the FCMD versioning model, these two links are versioned links.

Consider a scenario where you have a large amount of ancestor nodes to be recomputed for a revision. The FCMD versioning model provides an optimal

solution such that at certain nodes on the version space, full copies of the revision are stored and the version links from nodes support forward and reverse deltas. Thus, it is left to the implemention to decide what nodes are fully stored.

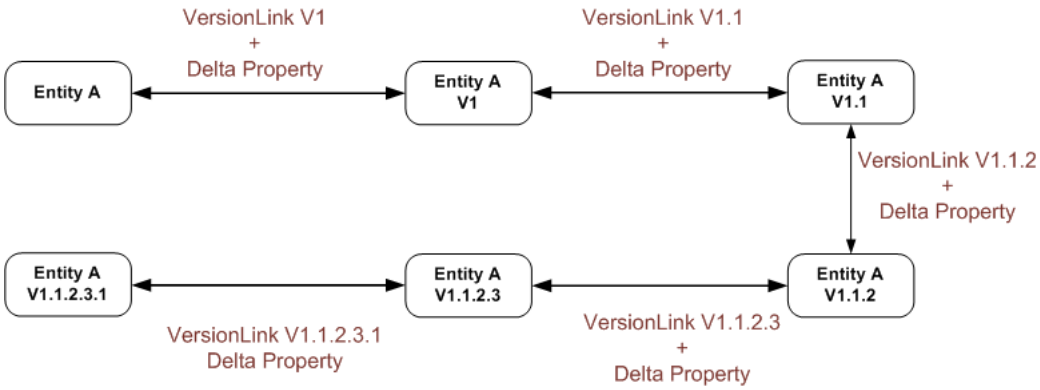

Figure 4.21: Illustration for retrieving the last revision in the FCMD versioning model.

4.4 Comparison of The Versioning Models

In this section, we compare the four versioning models with respect to our versioning goals and some technical dimensions. Next we will conclude this chapter with the overview of our final FCMD versioning model.

Data Versioning All four versioning models support data versioning. Component and FCMD versioning model both support the persistent storing of modification data for versioning data nodes. Link versioning supports the persistent storage of the different states of the data. However, delta property versioning model does not support data versioning explicitly as it persistently store data with the delta operations for each revision.

Link Versioning Link versioning is only supported by the FCMD versioning model. However the model uses the delta property concept to achieve this versioning concept.

Structure Versioning The FCMD versioning model is the only model among the four models that supports structural versioning. By freezing the state of a structural node, its sibling nodes are used to construct a new structure revision using the version link to store the node references.

Change Aggregation The delta property, component and FCMD versioning models support change aggregation. Thus all three models benefit from the high storage memory performance this concept provides.

Creating a New Version In the component versioning model, creating a new version depends on the revision resource depth in the composition hierarchy. This implies that the revision's resource data will access all the resources that will composed it. Therefore creating a revision with resource data at depth n implies that the time complexity of creating the revision is *O(n)*. Where n is the number of container resources for which the revision's resource data is embedded. The delta property and FCMD versioning models require just a single node access to create a new revision. However, a single access in the FCMD versioning model depends on the delta technique adopted at the point of creation. The link versioning model does not need to access a revision node before creating a revision but rather uses the version link to relate to the parent revision.

Retrieving the Most Recent Version In the delta property versioning model, visiting the most recent version needs a recomputation of all the delta operations on all its ancestors using the forward delta technique. The time complexity will be *O(n)*, where n is the number of ancestor revisions needed to be computed. Similarly visiting the most recent version in the component versioning model takes a time complexity of *O(n)*, but in this model, n is the number of containers the revision is composed from the root container. The link versioning model has a different approach. Since the node contains the copy of previous versions then there is no need to compute the node version from its ancestors. Thus accessing the most recent version is an *0(1)* time complexity operation. In the FCMD versioning model, retrieving the most recent version is similar to the delta property versioning model but in an optimal way. Since the RSL model provides bidirectional linking, the FCMD versioning model retrieval can support both forward and reverse delta techniques. Thus the retrieval time depends on the closest fully stored revision on the most recent version path.

Memory Performance Although the delta property, the component and FCMD versioning model have a high memory performance given that revisions are computed using shared contents, they all require a great deal of computation to generate the required version. To tackle this computational strain, FCMD versioning provides the support for fully resolved intermediary nodes that hold the full revision copy. These intermediary nodes can be placed at strategic point on the version space. The link versioning model has the worst memory performance as each version has a copy of the previous version since each revision has a full copy of the shared data unit of its parent revision.

To summarize, Table 4.1 presents the comparision of the versioning models with respect to the dimentions presented above. Among the versioning models described in this chapter, the FCMD versioning model is the best model to extend the FCMD document format metamodel, since the FCMD versioning model supports all our versioning goals. Furthermore the model also provides the ability to optimize some technical requirements like the retrieval time of a

Table 4.1: The comparison of among the models based on above the dimensions.

Dimensions	Versioning Requirement Support			
	Delta Property	Component Versioning	Link Versioning	FCMD Versioning
Data Versioning	☑	☑	☑	☑
Link Versioning	☒	☒	☒	☑
Structure Versioning	☒	☒	☒	☑
Change Aggregation	☑	☑	☒	☑
	Time Complexity (Worst Case)			
Creating a New Version	*O(n)*	*O(n)*	*O(1)*	*O(n)*
Retrieving the Most Recent Version	*O(n)*	*O(n)*	*O(1)*	*O(n)*
	Space Complextiy			
Memory Performance	★★★☆☆	★★★★★	☆☆☆☆☆	★★★★☆

revision and the storage space needed to create a revision.

To demonstrate this versioning concept, we have implemented a GUI diff viewer for versioning FCMD documents. The next chapter will describe our implementation of a FCMD GUI diff viewer.

Chapter 5

Implementation

In this chapter, we briefly introduce the details of the implemented prototype to support versioning in FCMD documents. The prototype provides a FCMD format text editor that create documents and a diff viewer that shows the modification difference between two successive revisions. This prototype is called *FCMD GUI Diff Viewer*. First, we will discuss the objectives of the prototype application. Then, we will present the overall architecture of the application. Finally, this chapter outlines the versioning features supported by the prototype application and its limitations.

5.1 Objectives of the FCMD GUI Diff Viewer Implementation

The main objective of the FCMD GUI diff viewer is to stand as a proof of concept for the versioning support in FCMD document format metamodel. The implementation uses the FCMD versioning model to extend the FCMD metamodel for versioning support. Thus the application is able to persistently store FCMD data with a good performance in storage and retrieval time. Another objective is to provide the revisions of a FCMD document as different facet of the same document. Thus only the first revision is fully stored while successive revisions are stored as meta information that contain deltas and cross-reference objects of each revision. Another objective is to provide a version space that shows the revision history of a FCMD document. The version space also shows different revision branches, so that each modification line can be traced to its revision root. Finally, to provide a diff viewer that highlights the textual difference between two successive revisions. This provides an easy modification tracking mechanism for working teams.

5.2 The Implementation Architecture

The FCMD GUI diff viewer was implemented using Java programming language and an object oriented database *db4o*[1]. The implementation offers a platform that provides separate Java classes for all the FCMD metamodel and the extended FCMD versioning model. In Figure 5.1, we depict the three main part of the application platform: `Model`, `File Handler` and `Data Storage Manager`.

In the `Model`, the FCMD metamodel [14] and the FCMD versioning model are implemented. The overall structure of the Model implementation is shown in Figure 5.2.

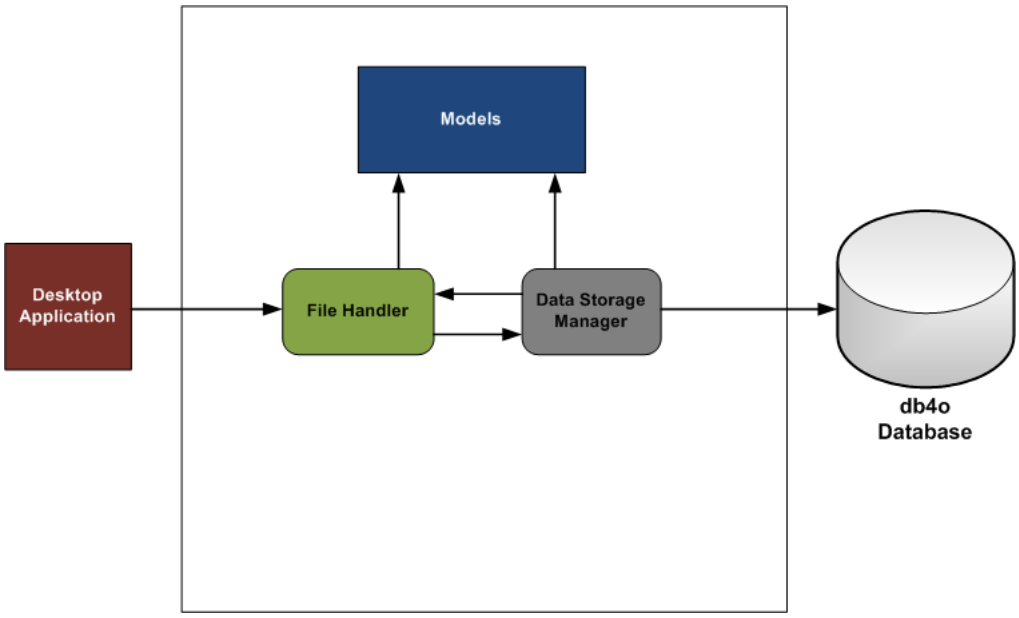

Figure 5.1: Architecture diagram of the FCMD GUI diff viewer implementation

In order to provide a single FCMD document with its revisions as different facet within the file, we introduced a `File Handler`. The `File Handler` serves as a controller that interprets the meta information of the file stored in the database. Once a file is access, the `File Handler` retrieves the objects that are associated with the file via the `Data Storage Manager`; the meta information objects are then used to compute the associated revisions. It is the role of the `Data Storage Manager` to store and retrieve all objects and their relationships. Whenever an update or modification is made to the database (via another running application), the `Data Storage Manager` notifies the File Handler, which in turn updates the application view.

[1] `http://www.db4o.com` (accessed November 1st, 2013)

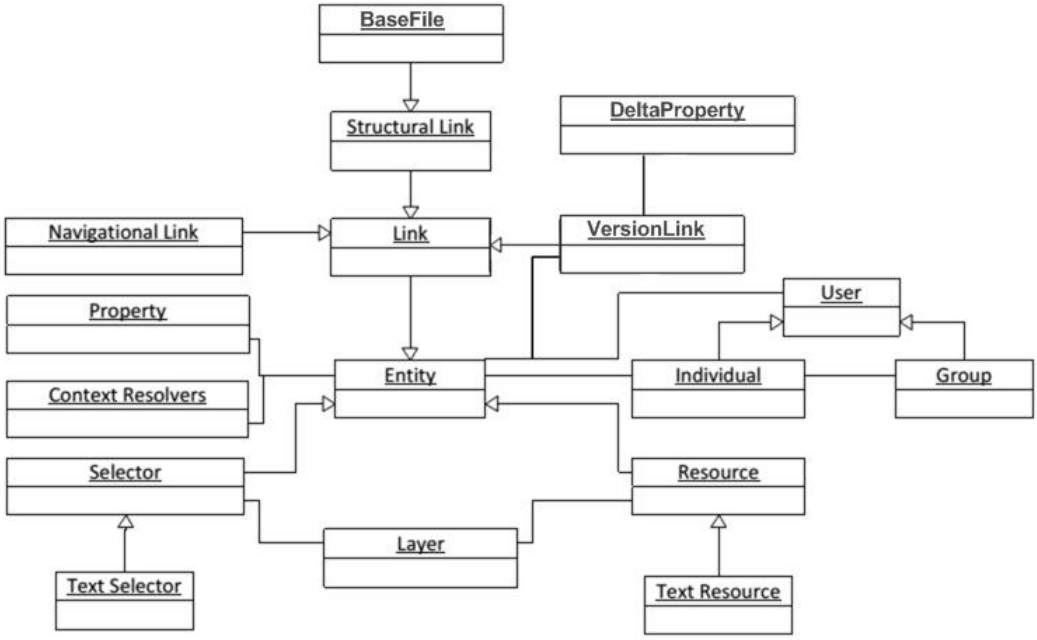

Figure 5.2: General organization of the extended FCMD metamodel implementation

Although the FCMD versioning model provides support for forward and reverse delta operations, our implementation provides only a forward delta operation. In order to avoid the excessive computation on revision retrieval and better retrieval time, each revision that bears a child revision is stored as a full version content. Therefore only the leave nodes in the version space store modification contents. As shown in the version space in Figure 5.3, the base revision and revisions node that bear siblings are denoted as circles with double border lines; these revisions are stored in full. In contrast, the leave revisions denoted in circles with single line border only store the modification content needed to compute the required revision.

Note that opening a FCMD document file without the FCMD GUI diff viewer application, only the base revision will be displayed. In order to access the revisions of the file, the rendering application must be able to interpret the meta information associated with the file.

5.3 db4o Database

Each FCMD document file is associated with meta information that provides its revisions accessibility. All the meta information (resources, links, version links, delta properties, etc.) are stored in a database called `db4o`. db4o (stands for

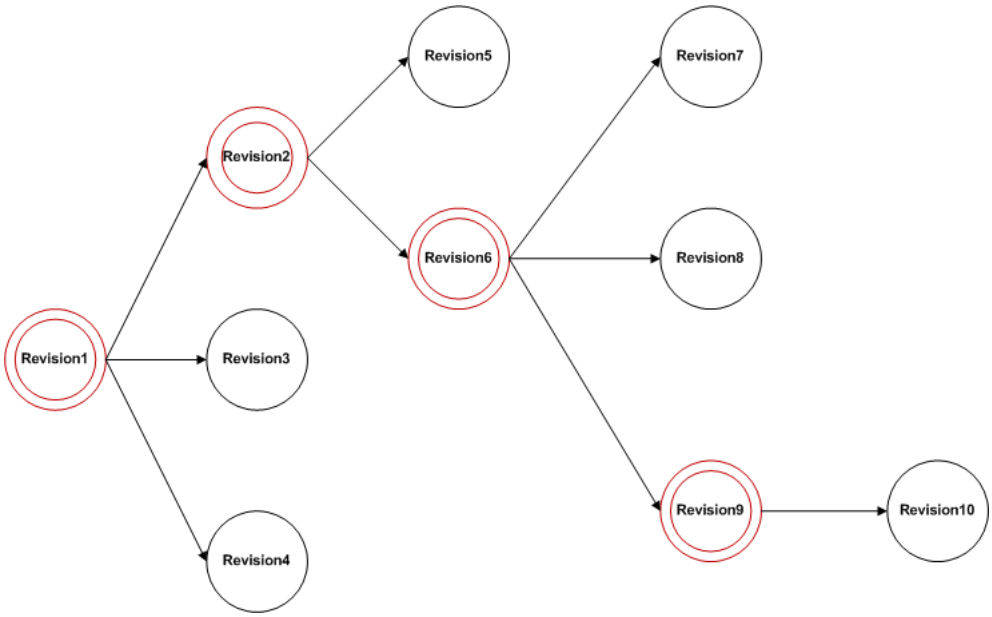

Figure 5.3: A simple FCMD version space

database for objects) database is an object-oriented database system written in C# and Java which uses a one line of code to eliminate the complexity of storing and retrieving objects from a database. The reasons behind the motivation to use the db4o database engine are as follows:

1. It is very simple to integrate in Java applications since it was implemented using Java programming language.
2. It support for many different platforms.
3. It provides the possibility to carry the database file with the actual application: hence the database file can also be used by other applications since it is portable.
4. It provides a lot of in-built security features (role based access rights, encryption, etc.) which are available without the application providing them.
5. Using both the embedded mode and client-server mode in an application guarantees some level of safety: so that synchronizing the database generated from the running the application on client side with the server database provides backup if a failure incident occur on either side.

5.4 The FCMD GUI Diff Viewer

The FCMD GUI diff viewer is a desktop application that runs on a Java platform. When running the application, the users can create a FCMD document, modify the document to create a revision and view the version space for each created document. Furthermore, the users can view the visualized difference between two successive documents. Each document and its revisions must be provided with creation and modification date respectively.

5.4.1 Creating a New FCMD Document

To create a new FCMD document, users are provided with a simple text editor. Figure 5.4 shows the text editor present in FCMD GUI diff viewer application. The document is composed by structural links, which have been described in section 2.2.2. In our implementation, only text resources have been supported when composing a FCMD document. In Figure 5.5, we illustrate how the FCMD document is composed.

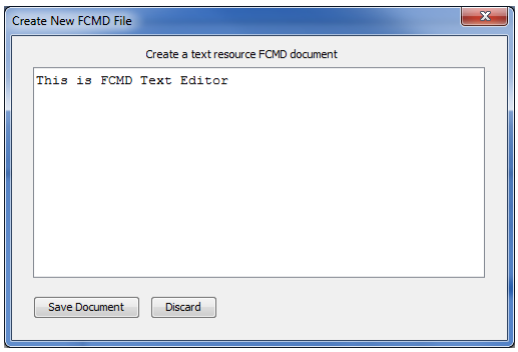

Figure 5.4: The text editor in FCMD GUI diff viewer

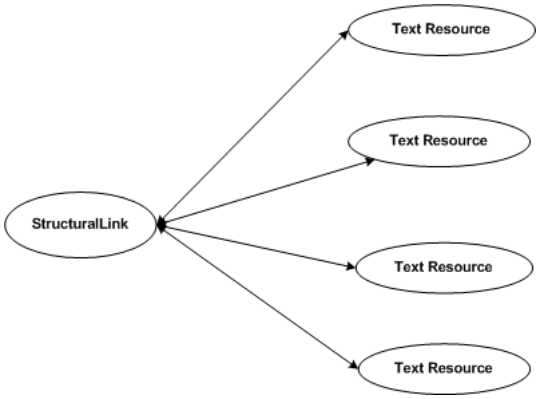

Figure 5.5: The FCMD document composition in the text editor

After creating and saving the FCMD document as shown in Figure 5.6, the author's name and comment for each document should be provided by the user as shown in Figure 5.7.

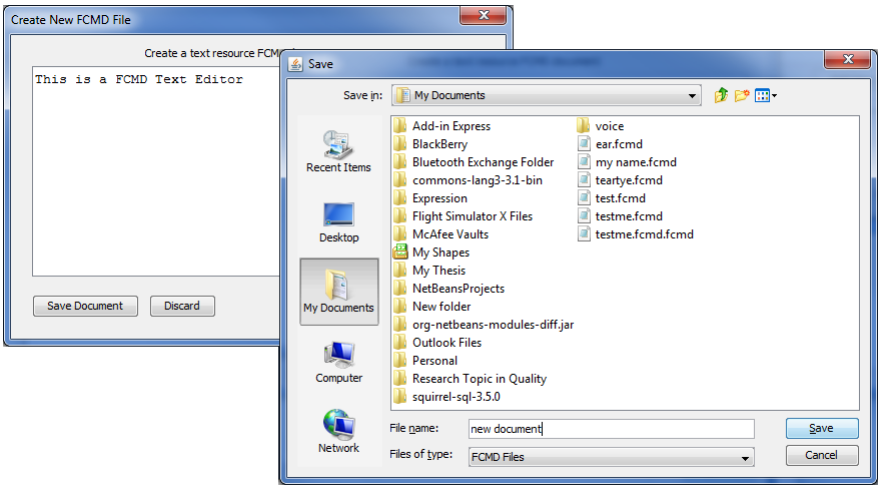

Figure 5.6: Saving the created document as a FCMD document file

5.4.2 Creating a Revision

The FCMD document revisions are stored as deltas using the version link defined in the FCMD versioning model. Therefore the first revision (base file) of the FCMD document includes file version metadata (deltaset) for each revisions created. Thus using the version links to relate each revision to the base file is important for tracking and retrieving a revision.

The deltaset stored in the `VersionLink` class reflects the modification difference between two successive revisions. Hence the shared text resources are stored in the parent revision while the added text resources are kept in the child revision. Note that the metadata contain both delta operations and the added text resources.

Figure 5.8 shows a snapshot of the overall view of the application during the revision process. At the top of the application, a window that shows the list of documents monitored by the application. The user can click on a document to show the revision history of the the document on the version space window. The version space window is located at the lower left side of the application window and displays the evolution of revision history in the form of file-directory organization hierarchy. The modifications are made on the lower right side of the application window. As more revisions are stored, the version space window is updated.

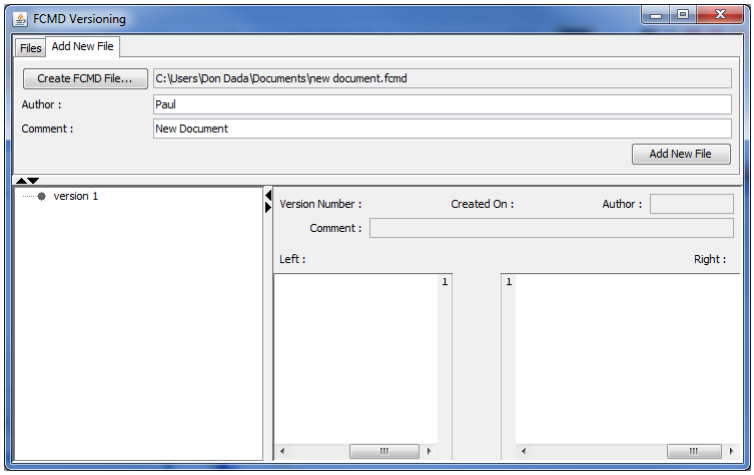

Figure 5.7: Adding the created FCMD file to the monitored list of FCMD documents

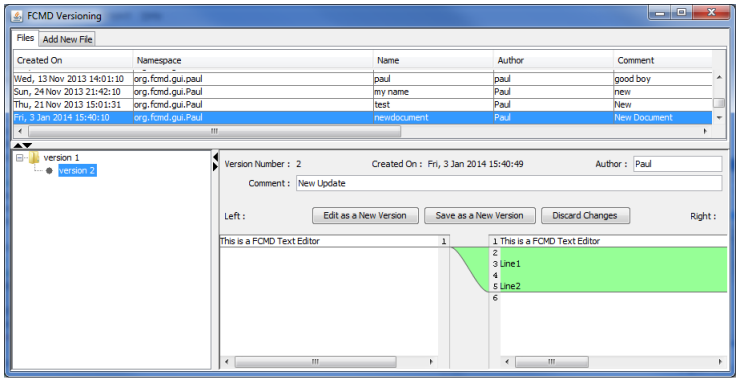

Figure 5.8: Overall view of the FCMD GUI diff view

5.4.3 Version Space Visualization

One of the technical requirements to support versioning in systems outlined by [11]: is to provide a version space visualization for users so that users are able to explore the revision history in a visual display. The FCMD GUI diff viewer application provides this versioning feature to users. Thus users can click on a specific revision on the visualized version space window which then displays the document content of the revision as shown in Figure 5.9. In the implementation

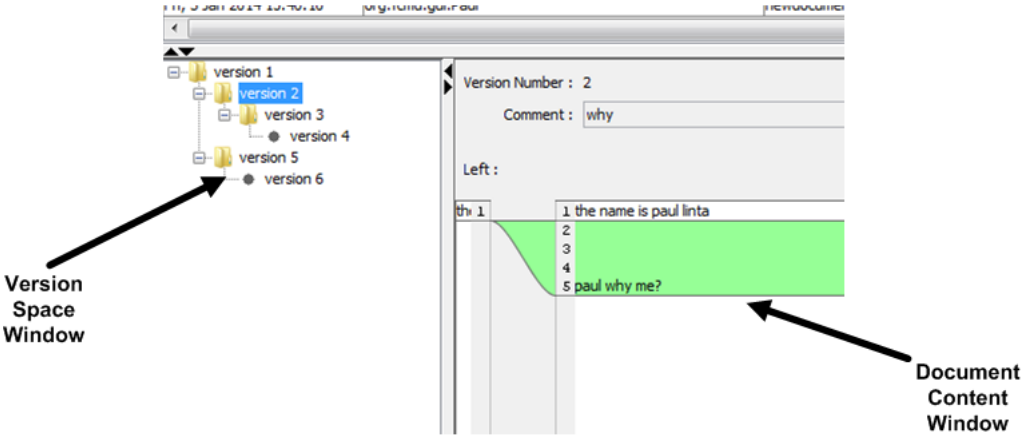

Figure 5.9: The version space window in the application

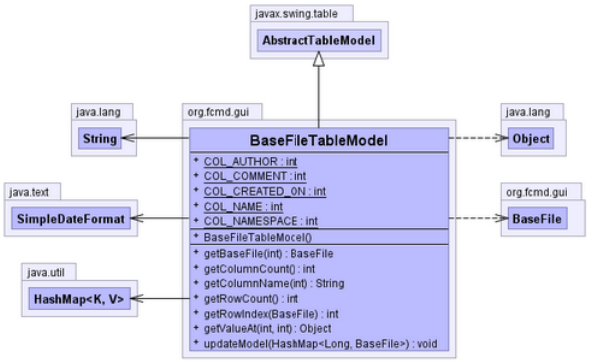

Figure 5.10: The version space visualization controller class depicted using the UML class diagram

of the FCMD GUI diff viewer, a version space controller was provided for the following reasons: 1) To visualize the version space using the metadata from the document, 2) To provide notifications that enable the document content window shown in Figure 5.9 to refresh once another revision node is clicked.

Figure 5.10 reflects the class dependencies of the version space controller class `BaseFileTableModel`, to enable these functionalities.

5.4.4 Diff Viewer

The provision of the diff veiwer in the prototype enables users to track changes between two revisions. The diff viewer provides different colors for the text block background that reflect the different update operations. For instance, using the diff viewer in Figure 5.11 where the right window is the child revision textual contents and the left window is the parent revision textual contents: a text block in a red background means that the text block is not present in the child revision, a text block in a green background means that the text block is not present in the parent revision, a text block in a blue background means that the text block is modified in the other revision and a text block in a white background means that the text block is present in both revisions without any modification.

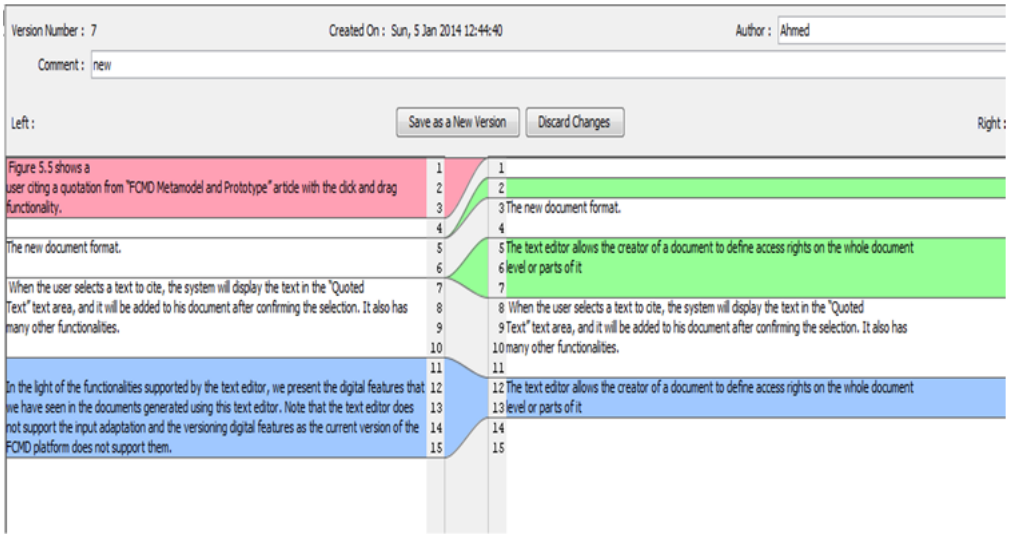

Figure 5.11: The diff viewer window

5.5 The Implementation Limitations

Although the FCMD GUI diff viewer support a lot of the versioning features, due to time limitations, we did not provide support for some technical requirements and a few document versioning requirements outlined in chapter 3. For example, the prototype do not support link and structure versioning.

Although there exist several delta implementation framework for data versioning, these frameworks only support text based deltas. This is the reason

that the prototype application is unable to provide a diff viewer for data objects that represent other content types such as images, videos, etc.

Creating an *alternate revision* in data versioning (for example revisions identified as textual content type of different languages) is achieved using the state-oriented versioning technique, see Section 2.3.1. Even though the FCMD versioning model provides support for state-based versioning using the RSL complex `Property` type to store the version state property of an item, our prototype application has not provided an implementation for this versioning technique. Since change-oriented versioning is a widely used versioning solution technique, the FCMD GUI diff viewer implementation is based on this technique.

In order to support a collaborative environment for teams working in parallel on shared resources, some technical requirements should be considered. For example, the handling merge conflicts during integrating changes in shared resuorces [47] and the identification of those revision branches that are of low benefit, yet causing increased integration[2] time [46]. Although this requirement is beyond the scope of this study, the FCMD versioning model provides a solution for *semantic merge conflict*[3] which the notion of supporting structure versioning. For instance, a programming text editor built using the extended versioning supported FCMD document format metamodel, where packages, classes, methods, etc. are mapped as structure nodes such that the semantic relationship among its composing nodes are well defined. Therefore the modifications on a shared structure node by two authors are semantically bounded, providing easier merging algorithms.

[2] An integration merges the contents of a file at a specific point in time on one branch (source) into another branch (target).

[3] Semantic merge conflicts also known as high-order conflicts [47] are integration bugs from program source code caused by successfully merge process that fail to build or pass certain test suite.

Chapter 6

Conclusions and Future Work

6.1 Summary

The support for versioning in document formats so that document's revisions are related implicitly as different facet of the same document is significant to the upcoming age of ubiquitous computing. Thus any information needed on a document's revision is readily available without requiring external versioning system for support. However, only few document formats provide this versioning functionality in a very limited approach. In this thesis, we tackled this issue by providing a review of several representatives of five document formats families which included: *Document Preparation Family* (GML, Scribe, LaTeX and DocBook), *Meta-Languages Family* (SGML and XML), *Print-Oriented Family* (PDF and Open Office XML), *World Wide Web Family* (HTML and XHTML) and *Electronic Digital Publishing Family* (EPUB). In order to provide versioning support for document formats, we have identified three requirements from the deep analysis of the logical structure of the reviewed document formats. These versioning requirements are *data versioning*, *link versioning* and *structure versioning*. Our analysis also reveals that most of the document formats do not support these requirements. Likewise, the review of third party versioning solutions like GIT, SVN and CVS, led to an additional versioning requirement needed to support versioning in a document format, which we called *change aggregation* (delta operations).

In this thesis, the FCMD document format metamodel was extended to support versioning implicitly by fulfilling the four versioning requirements (*data versioning*, *link versioning*, *struture versioning* and *change aggregation*) embraced from the above analysis. The FCMD metamodel is based on the RSL hypermedia model which provides a powerful navigational feature which enabled the ease to extend the metamodel. This thesis presented four versioning models perceived from the analysis of versioning supported documents and versioning

solution systems. However, the strengths and weaknesses of three of the four models have been used to create the fourth model which is known as the *FCMD versioning Model*.

Finally, a prototype application of the extended Fluid Cross-Media Document Format metamodel has been implemented as a proof of concept. The implementation provides a FCMD format text editor, a version space window and a diff viewer. The text editor is used to create a FCMD document. Reviewing a version history of a FCMD document is done by the provision of the version space window. The diff viewer is used to view the FCMD document modification difference between two successive revisions.

To summarize, we outline the contributions our research has made to the document engineering, software engineering, ubiquitous computing and hypermedia communities.

1. As a result of our analysis on existing document format, We identified three main dimensions that are significant to the modeling of versioning supported document formats: *data versioning*, *link versioning* and *structure versioning*. Thus providing the basis for the concept of having revisions of a document as different facet of same document rather than saving multiple versions of a document which have to be related explicitly.

2. We addressed the time-space-tradeoff involved during storing and retrieving revisions; by introducing some concepts extracted from our analysis in third party versioning solutions in our model, few computing time and low memory consumption is needed to store and retrieve a revision: *change aggregation*, *forward delta operation* and *reverse delta operation*.

3. Providing a versioning model that extends a middleware document format metamodel (FCMD metamodel) on top of which document formats can be enriched with digital features of the upcoming age of ubiquitous computing, this versioning model is known as the *FCMD versioning model*.

4. Finally, A prototype application was provided as a proof of concept for the proposed versioning supported extended FCMD metamodel. The prototype includes features like text editor, diff viewer and version space window.

6.2 Future Work

The FCMD versioning model provided a sufficient model for extending the FCMD document format metamodel and the resulting model forms a robust and well-formed model. The proposed versioning model could, however, be improved or extended within each of the different aspects outlined:

1. Providing some speculative mechanism for reducing merge conflicts during integrating revisions.

2. Providing support for versioning delta operations to increase the flexibility of storing and retrieving revision with respect to time-space-tradeoff.

Bibliography

[1] Dana Marjanovic *Development a Meta Model for Release History Systems 23-29* 2006.

[2] Scott Chacon *Git Internals* 2008.

[3] Scott Chacon *Pro Git book* 2009: Apress.

[4] Emmet James Whitehead, Jr. *An Analysis of the Hypertext Versioning Domain* 2000: Dept. of Information and Computer Science, University of California, Irvine.

[5] Luiz Fernando G. Soares, Noemi L. R. Rodriguez and Marco Antonio Casanova. *Nested Composite Nodes and Version Control in Hypermedia Systems* 1994: Depto. de Informatica and Centro Cientifco Rio, IBM Brasil, Rio de Janeiro, RJ Brazil. SPRINGER-VERLAG.

[6] Jacco van Ossenbruggen, Anton Eliens *The Dexter Hypertext Reference Model in Object-Z* Vrije Universitiet, Department of Mathematics and Computer Science De Boelelaan, Amsterdam, The Netherlands.

[7] User Manual *CVS–Concurrent Versions System.*

[8] Beat Signer, Moira C. Norrie *As We May Link: A General Metamodel for Hypermedia Systems* 2007: Institute for Information Systems, ETH Zurich CH-8092 Zurich, Switzerland.

[9] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato *Version Control with Subversion: For Subversion 1.7* 2011.

[10] Emmet James Whitehead, Jr. *An Analysis of the Hypertext Versioning Domain* 1999:Dept. of Information and Computer Science, University of California, Irvine.

[11] Emmet James Whitehead, Jr. *Versioning in Hypertext Systems* 1999:Dept. of Information and Computer Science, University of California, Irvine.

[12] Reidar Conradi, Bernhard Westfechtel *Version Models for Software Configuration Management* 1998: ACM.

[13] Luiz Fernando G. Soares, Rogerio F. Rodriguez, Guido L. De Souza Filho and Debora C. Muchaluat *Versioning Support in HyperProp System* 1999: Kluwer Academic, The Netherlands.

[14] Ahmed A. O. Tayeh *A Metamodel and Prototype for Fluid Cross-Media Document Formats* 2012: Faculty of Science, Department of Computer Science. Vrije Unversiteit Brussel.

[15] Peter M.D. Gray (Editor), Krishnarao G. Kulkarni and Norman W. Paton *Object-oriented databases: a semantic data model approach* 1992: Prentice-Hall, London.

[16] A. Haake, *Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications*Sept. 18-23, 1994, pp. 81-93: Proc. Sixth ACM Conference on Hypertext (ECHT'94), Edinburgh, Scotland.

[17] I.Goldstein and D,Bobrow. *A Layered Approach to Software Design.* 1984: In: D. Barstow, H. Shrobe, and E. Sandewell (Eds.), *Interactive Programming Environments*, McGraw Hill.

[18] K. Grafek, Jens A. Hem, Ole L. Madsen, and Lennert Sloth *Composites in a Dexter-Based Hypermedia Framework*1994: Proc. 1994 European

[19] Romain Robbes, Michele Lanza. *Versioning Systems for Evolution Research.* 2005: Faculty of Informatics, University of Lugano, Switzerland.

[20] B. Signer. *Fundamental Concepts for Interactive Paper and Cross-Media Information Mangement.*2006: PhD thesis, ETH Zurich, Switzerland.

[21] G. Conboy, M. Garrish, M. Gylling, W. McCoy, M. Makoto, and D. Weck. EPUB 3 Overview, October 2011: Recommended Specification.

[22] Haake A., Hicks D., *VerSe: Towards Hypertext Versioning Styles* 1996; Proceedings of Hypertext 96. Washington, EUA. Setembro de.

[23] AbdelAli Ed-Dbali, Pierre Deransart, Mariza A. S. Bigonha, Jos'e de Siqueira and Roberto da S. Bigonha. *HyperPro: An integrated documentation environment for CLP* 2001.

[24] John B. Smith, Stephen F. Weiss, and Gordon J. Ferguson A. *Hypertext Writing Environment and its Cognitive Basis* 1987: ACM 1989.

[25] Tobias Oetiker, Hubert Partl, Irene Hyna and Elisabeth Schlegl. *The Not So Short Introduction to LATEX 2.* 2011; Free Software Foundation, 2011.

[26] Wikipedia. *Scribe (markup language).* `http://en.wikipedia.org/wiki/Scribe_(markup_language)`. [Online; accessed 25-August-2013]

[27] Brian K. Reid. *Scribe: Introduction to User's Manual*;1978.

[28] Wikipedia. *Portable Document Format* http://en.wikipedia.org/wiki/Portable_Document_Format [Online; accessed 25-August-2013].

[29] N. Walsb. *DocBook 5.1: The Definitive Guide*. OREILLY, 2010.

[30] Wikipedia. *DocBook* http://en.wikipedia.org/wiki/DocBook [Online; accessed 25-August-2013].

[31] Marko Leppnen. *Differences between SGML, XML and HTML* http://www.students.tut.fi/~leppane7/leppanen.html [Online; accessed 30-August-2013].

[32] Wikipedia. *Standard Generalized Markup Language* http://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language [Online; accessed 25-August-2013].

[33] Charles F. Goldfarb. *The SGML HandBook*. Clarendon Press, 1990.

[34] IBM. *GML Starter Set User's Guide*. http://publibfp.boulder.ibm.com/cgi-bin/bookmgr/BOOKS/dsm04m00/CCONTENTS, 1991.

[35] World Wide Web Consortium. *Extensible Markup Language (XML)*. http://www.w3.org/XML/, [Online; accessed 25-August-2013].

[36] Wikipedia. *TEX*. http://en.wikipedia.org/wiki/TeX, [Online; accessed 25-August-2013].

[37] Brian R. Gaines , Mildred L. G. Shaw *A networked, open architecture knowledge management system* 1996;.

[38] Bill Kasdorf *Metadata in EPUB 3: It's what makes it all work*; General Editor, The Columbia Guide to Digital Publishing Metadata Subgroup Lead, IDPF EPUB 3 Working Group.

[39] T. H. Nelson *Literary Machines* 1981., 93.1 ed. Sausalito, CA: Mindful Press.

[40] World Wide Web Consortium. *Hypertext Markup Language : Links(HTML)*. http://www.w3.org/TR/html401/struct/links.html, [Online; accessed 25-August-2013].

[41] Ben Collins-Sussman, Brian W. Fitzpatrick, C. Michael Pilato *Version Control with Subversion* 2011.

[42] Per Cederqvist et al *Version Management with CVS* 2005. Free Software Foundation, Inc.

[43] Wikipedia. *Open Document Architecture*. http://http://en.wikipedia.org/wiki/Open_Document_Architecture, [Online; accessed 25-August-2013].

[44] Nelson, T. *Literary Machines.* 1982; Mindful Press.

[45] whitelink) James Whitehead *Design Spaces for Link and Structure Versioning* 2000.

[46] Christian Bird and Thomas Zimmermann *Assessing the Value of Branches with What-if Analysis.* 2012; Cary, North Carolina, USA.

[47] Yuriy Brun, Reid Holmes, Michael D. Ernst and David Notkin. *Proactive Detection of Collaboration Conflicts.* 2011; Szeged, Hungary.

[48] Wikipedia. *Revision Control System (RCS).* `http://en.wikipedia.org/wiki/Revision_Control_System`, [Online; accessed 25-August-2013].

[49] Wikipedia. *Delta encoding.* `http://en.wikipedia.org/wiki/Delta_encoding`, [Online; accessed 25-August-2013].