SUPPORTING LEARNING OBJECT VERSIONING

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Masters of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Christopher Arthur Hansen Brooks

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

S7N 5C9

Abstract

A current popular paradigm in e-learning is that of the "learning object". Broadly defined, a learning object is a reusable piece of educational material intended to be strung together with other learning objects to form larger educational units such as activities, lessons, or whole courses. This aggregating of learning objects together is a recursive process – small objects can be combined to form medium sized objects, medium sized objects can be combined to form large objects, and so on. Once objects have been combined appropriately, they are generally serialized into content packages, and deployed into an online course for delivery to learners.

Learning objects are often stored in distributed and decentralized repositories throughout the Internet. This provides unique challenges when managing the history of such an object, as traditional versioning techniques (e.g. CVS, RCS, etc.) rely on centralized management. These challenges have been largely ignored by the educational technology community, but are becoming more important as sharing of learning objects increases.

This thesis explores these issues by providing a formal version model for learning objects, a set of data bindings for this model, and a prototype authoring environment which implements these bindings. In addition, the work explores the potential benefits of version control by implementing a visualization of a learning object revision tree. This visualization includes the relationship between objects and their aggregates, the structural history of an object, and the semantic changes that an object has undergone.

Acknowledgements

I owe much thanks to a countless number of people who helped to make this work a reality. It is through social relationships that we build knowledge, and I feel privileged to have had an opportunity to meet and work with some great people.

Firstly, I would like to thank my supervisors John Cooke and Julita Vassileva. Their guidance and support have helped me grow both as a scholar, and as an individual. In particular I would like to thank John for taking a chance on me as an undergraduate student – his teaching style encouraged students to explore open ended questions was a motivating factor for me when deciding to whether to enter into graduate school or not.

Secondly, I would like to thank my thesis committee members, Ralph Deters, Kevin Schneider, and the external Mohammed Ally. Their insight and comments into this work proved both interesting and useful. I would also like to thank Permanand Mohan, a scholar from the University of the West Indies who spent a year visiting our laboratory. His support, both through discussion on the field of learning technology as well as financial support to present our joint works, was invaluable.

Everything is harder when you have to go at it alone, and I've been lucky to have great friends and colleagues to help along the way. Special thanks go to Lori Kettel, Alan Fedoruk, Brian Richardson, Collene Hansen, Mike Winter, and all the members of the ARIES laboratory I've interacted with over the years. In addition, I am deeply indebted to the administrative, technical, and instructional support teams within our department. They rarely receive the thanks they so much deserve.

Finally, I would like to thank the University of Saskatchewan and the College of Graduate Studies and Research for financial support in the form of scholarship and bursaries.

for my wife

Table of Contents

List of Tables

List of Figures

List of Code Listings

List of Acronyms and Abbreviations

API	Application Programming Interface
CVS	Concurrent Versions System
DOM	Document Object Model
DTD	Document Type Definition
EBNF	Extended Backus-Naur Form
HTML	HyperText Markup Language
IMS	IMS Global Learning Consortium Inc.
IMSCP	IMS Global Learning Consortium Inc. Content Package Specification
IMSSS	IMS Global Learning Consortium Inc. Simple Sequencing Specification
LCMS	Learning Content Management System
LMS	Learning Management System
OID	Object Identifier
RCS	Revision Control System
RDF	Resource Description Framework
SCM	Software Configuration Management
SVG	Scalable Vector Graphics
SVN	Subversion
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VID	Version Identifier
XML	Extensible Markup Language
XHTML	Extensible HyperText Markup Language

Chapter 1
Introduction

The rise in use of widely distributed networks like the Internet has increased the accessibility and affordability of education for the masses. Educational institutions are better able to distribute their courses and programs to both national and international markets. People who are unable to participate in traditional learning environments due to the timing of classes, inability to relocate to a traditional classroom, or costs involved with leaving employment, are turning to electronic learning (e-learning) as a viable alternative. The simplicity of publishing on the web has decreased the costs for some publishers of educational material as well. While there is no consensus on the effects of the Internet on publishing costs for traditional vendors, a variety of independent communities built around specific topics have emerged. These online communities usually provide member-created content for little or no cost in the form of tutorials, white papers, or discussion forums. This has introduced literally millions of new, highly accessible, educational resources.

With the learning, education, and training industry market estimated at more than one and a half trillion dollars per annum (2002), a number of universities, government entities, and corporations have shown a great deal of interest in capitalizing on electronic learning technologies. Principle amongst these technologies are learning objects – pieces of educational content meant to be reused to provide for customized, yet cost effective, learning.

At a very coarse grain level, learning objects are simply digital resources that are annotated with metadata and deposited in electronic learning repositories. Instructors or learners can search or browse these repositories to find objects of interest. In and of it-

self this is not an overly novel idea – traditional learning materials such as textbooks have a long history of being categorized in library systems with a variety of metadata, including topic based classification schemes like the Dewey Decimal system or the Library of Congress Catalog. This metadata was then stored in physical card catalogs, which are centralized repositories that describe every holding a library has. Learners could then choose to either browse the shelves of the library based on topic, or search for known titles in the card catalog by author name. While most current library systems have replaced these catalogs with electronic versions allowing for more diverse searching options, the resulting process followed is relatively unchanged.

As digital resources which are produced in a decentralized manner for a global market, a number of unique issues arise when using learning objects:

- How can learners find the educational material that is personalized for them and the situation they are learning in? Traditional library and card cataloging systems depend on users of common interests and abilities to be situated geographically close to one another. The Internet has removed this restriction and now communities of interest and practice are often distributed in culture, language, and location.
- How can a content management system present educational material developed by different authors and communities in a consistent and comprehendible manner? Traditional educational resources such as books are decidedly immutable and their content is presented as the publisher intends it. For instance, to view a book in another format such as a large print edition requires buying another copy of the book (if it is even available in a large print version). Electronic materials are much more mutable, and can see content rendered based on the format the user desires, instead of the format the publisher desires. When mixed with communities of practice, derivative works become the norm rather than the exception.
- Given the web's strongly decentralized nature, how can the duplication of work be prevented? Duplicating educational works to offer competing textbooks for a particular course is common between publishers, and is seen as a way to main-

tain interests in lucrative markets. As more course material is offered over the web in electronic form for no cost (e.g. the open courseware initiative (Massachusetts Institute of Technology, 2003)) the focus changes, and it becomes important to reduce the amount of duplication of work, and instead provide easy means for creating new derivative works that are easily customized and support individualized learning.

This work explores these concerns by studying how changes to learning objects can be captured and expressed both within a traditional structural sense, as well as in a semantic sense.

1.1 Hypothesis and Research Objectives

A learning object is made up of at least two parts; the content of the object which adheres to a well defined structure, and the semantics of the object, which adheres to well defined ontologies. The lack of metadata specifically for structural versioning of a learning object makes it impossible for a learning object authoring environment to provide common version control functions. I hypothesize that the addition of versioning related metadata and an understanding of the structure of a learning object can be used to perform automatic regression of a learning object (typically called rollbacks). Further, when semantic changes are also captured, an authoring environment can support the learning object selection process by differentiating between the meaning of related learning objects.

The research objectives of this thesis are as follows:

- To provide a model for expressing the structure of learning objects
- To provide a model for expressing both structural and semantic changes that can be applied to a learning object
- To provide a vocabulary for expressing the ways in which a learning object can change structurally, in particular those that conform to the Extensible Markup Language (XML) and the IMS Simple Sequencing Specification

- To provide a vocabulary for expressing the ways in which a learning object can change semantically
- To demonstrate by way of an implementation how collecting structural versioning information can be used to perform automatic learning object regression (rollbacks)
- To demonstrate by way of an implementation how collecting semantic versioning information can be used to visualize differences between related learning objects.

1.2 Organization of Thesis

This thesis is divided into five chapters. Chapter 2 introduces the contemporary concept of learning objects, and identifies the major areas of current research. An overview of the state of the art in software versioning is also discussed. Chapter 3 provides a model for capturing both structural and semantic changes as applied to learning objects. Integral in expressing how a learning object can change at a structural level is an understanding of the kinds of learning objects that exist, and their aggregate parts. This is also discussed in chapter three, which ends with a description of a formal schema for a learning object version model. To demonstrate the abilities of the version model, Chapter 4 provides an overview of a prototype authoring environment. This authoring environment, the Learning Object Versioning Environment (LOVE), provides a learning object author with advanced editing features that allow for structural rollbacks (shown in section 4.3) and semantic visualization (shown in section 4.4). Finally, the work concludes in Chapter 5 with a discussion of the research contributions made and avenues for further exploration.

Chapter 2
Background

This chapter discusses the current state of the art of reusable education materials often coined as "learning objects". It focuses more on the technical aspects of learning objects, such as composition, distribution, and description, instead of the pedagogy behind their use. In addition, this chapter briefly summarizes the field of version control, and indicates how version control principles are currently being applied to learning objects.

2.1 State of the Art in Learning Objects

Learning objects are reusable pieces of educational material meant to be strung together to form larger educational units such as activities, lessons, or whole courses. This definition is purposefully vague as contemporary literature often disagrees on the more precise characteristics of learning objects. While the learning object design field is young, it stems from over twenty years of research in instructional design, educational technology, and computer science. This diversity of origin leads to a variety of questions and debates, ranging from the more abstract notions of pedagogy, methodology, and instructional design, to the more technical aspects of structural syntax, technological specifications, and interoperability. Should the field of learning objects include traditional physical materials, or are digital artifacts the only ones of interest? Are learning objects related to the concept of objects as described in the object-oriented programming paradigm? How much educational material should a learning object contain?

Despite the lack of consensus on a single definition, the field of learning objects has seen a surge of research from information technology companies, academic institutions, and government organizations. Section 2.1.1 provides an overview of the most recog-

nized learning object definitions and identifies both the commonalities and the differences in those definitions. Like many technology fields, the field of learning objects is filled with analogies and metaphors to describe the structure of the items being discussed. Section 2.1.2 gives an overview of the various metaphors that have been used to describe learning objects, and identifies a number of models used to describe their structure. The ability to search for and retrieve appropriate learning objects is a core interest for learning object practitioners and has seen much work within specification and standards communities. Two aspects of this interest, categorizational metadata and data repositories, have seen a large amount of development and are described in sections 2.1.3 and 2.1.4 respectively. Finally, section 2.1.5 outlines how learning objects can be combined together to help satisfy more coarse grain educational goals.

2.1.1 Learning Object Definition

It is not clear when reusable pieces of educational material were first labeled "learning objects", though Wiley attributes this term as a derivative from the title of the Computer Education Management Association working group called "Learning Architectures, API's, and Learning Objects"[1], which was chartered in 1994 (Wiley, 2001). Perhaps the broadest definition of a learning object is that which is given by the Institute of Electrical and Electronics Engineers, Inc. (IEEE) Learning Technologies Standards Committee as "any entity, digital or non-digital, that may be used for learning, education or training" (IEEE, Inc., 2002). This definition includes anything that can be used for learning, ranging from traditional print material and online tutorials, to the more abstract notions of an instructional institution, department, or individual. It has been criticized as being overly vague, and most practitioners introduce their own definitions. Wiley argues that, in addition to being vague, this definition is incorrect in including non-digital artifacts as learning objects, and is not useful when learning technologies are primarily digital in nature. He provides a subset definition as "any digital resource that can be reused to support learning" (Wiley, 2001). This definition contains two important differences from the IEEE definition: resources that are not digital in nature and cannot be easily

[1] At the time of writing this group had been renamed as the "Learning Architecture, Learning Objects: Learning API Task Force". The interested reader is directed to (Hodgins,) for more information about the working group.

reused are excluded, and only those resource that are actively being used to support the learning process are included.

Robson takes an object-oriented programming approach to learning objects, and defines them (using the term "learning resource") as "any resource that an instructor makes available to a student for a pedagogic purpose and that can be realized in some type of learning environment" (Robson, n.d.). He suggests that learning objects should contain functionality through methods and properties. He identifies the ability to render a learning object based on the context of the learner and the media available as a typical method. This allows for a single learning object to be easily displayed in both interactive and non-interactive environments. Properties are used to encapsulate both the relationships between learning objects, and content related to the pedagogical purpose itself. For example, a learning object that helps teach database modeling through entity-relationship diagrams could have methods to display examples using the Chen, Crow-foot or Unified Modeling Language notation depending on the tool the students can use, and would include blocks of text and images as properties of the various different elements available.

Merrill (Merrill, n.d.) describes a similar instructional design component as a "knowledge object". These knowledge objects are made up of five main components:

- *A name or associated information that identifies the object*, whether it is a device, person, creature, place, symbol, object, or thing
- *an aggregation of parts of the entity*, which can be knowledge objects themselves
- *activities* that learners can complete
- *processes*, which are methods of instruction that execute based on some conditional value
- *properties*, which are data values relating to the object, its activities, or its processes

This description is functionally similar to Robson's object-oriented approach, and allows for a varying degree of granularity by reusing knowledge objects as parts of other knowledge objects.

While not arguing for or against the object-oriented paradigm, Wu (Wu, 2002) introduces the concept of inheritance as an important factor when reusing learning objects. He suggests that by substituting the content of similar learning objects, one can build different versions of the same material faster. He identifies that in addition to substitution, learning objects should be able to reuse one another by concatenating inherited pieces of information together with new information for derivative objects.

Downes suggests that modeling learning objects using the object-oriented paradigm will lead to the rise of Rapid Application Design (RAD) for educators. Educators will view online courses as "a piece of software, [which] may be seen as a collection of re-usable subroutines and applications" (Downes, 2002). He goes on to suggest that a learning object could also be thought of as "a small computer program that is aware of and can interact with its environment" (Downes, n.d.). This definition expands the notion of an interactive learning object by not just adding functions or methods, but by adding some intelligence or ability to reason. Indeed, he likens the difference between this definition and the previous definitions to the difference between passengers and luggage, with luggage being the epitome of inadaptable entities that often get lost, and passengers being intelligent and adaptable entities that get where they intend to go. This notion of intelligent learning objects has recently been embraced by other researchers as an significant direction, and is a prominent research goal of the University of Saskatchewan within the Learning Object Repositories Network (LORNET) project (Learning Object Repositories Network, n.d.).

Sosteric and Hesemeier (Sosteric and Hesemeier, 2002) strongly criticize the object-oriented programming paradigm for learning objects as "marginal, at best, and absolutely counterproductive at worst". They suggest that the primary beneficiary (the "consumer") of object-oriented programming is the programmer who is trying to read and write code, while the primary beneficiary of learning objects is the individual who is gaining instruction from the object, not the instructional designer. Further to this they

argue that it is not so much the structure of a learning object that makes it useful, but the contextual information that describes how the object can be used. Building on this argument, they suggest the following definition:

> "A learning object is a digital file (image, movie, etc.,) intended to be used for pedagogical purposes, which includes, either internally or via association, suggestions on the appropriate context within which to utilize the object."(Sosteric and Hesemeier, 2002)

Regardless of the exact definition of a learning object, there is some consensus that reusability is the most significant property instructional designers hope to achieve. Friesen outlines three facets of learning object reusability: discoverability, modularity, and interoperability (Friesen, 2001). He further identifies that building learning objects using component-based software architectures is common, and many data repositories treat learning objects as "black boxes". He argues that descriptive metadata are needed to allow for a learning object to be discoverable, and that industry standards are required to support interoperability.

2.1.2 Learning Object Metaphors, Models, and Structure

Definitions aside, the learning object literature uses metaphors and analogies that serve to illustrate the problems and benefits of reusability. Perhaps the earliest relevant analogy is one from Ralph Gerard in an article published in a 1969 book on computer-assisted instruction (Gerard, 1969). As cited by Gibbons et al., he identifies that "curricular units can be made smaller and combined, like standardized Meccano [mechanical building set] parts, into a great variety of particular programs custom-made for each learner" (Gibbons, Nelson, and Richards, 2001). This statement suggests that a core issue when using learning objects as reusable elements, particularly within adaptable systems, is the granularity at which they are created. Smaller learning objects can be combined in much more precise ways, adapting a learning system more to a user's interest. Larger learning objects, much like partially built Meccano structures, are less effective in adaptable systems where the chances of different users needing individually generated courses is high.

This line of thinking is also behind the LEGO metaphor which Wiley describes as a method in which "[we] create small pieces of instruction (LEGOs) that can be assembled (stacked together) into some larger instructional structure (castle or spaceship)" (Wiley, n.d.). Wiley, however, attacks this metaphor as one that, while useful for describing the structure of learning objects to the public at large, has restricted the way instructional designers can create learning objects. He further identifies two main deficiencies of using the LEGO metaphor for learning objects as:

- Combinability: Unlike LEGO pieces which can all interconnect, not all learning objects can be combined with other learning objects to form more complex structures.
- Sequencing: While LEGO pieces can be assembled by anyone and in any manner, creating courses of learning objects may require trained instructional designers.

Wiley further suggests that these oversimplifications come from a failure to realize learning objects are highly complex collections of smaller content, called information objects. Unlike a learning object an information object is created in an instructionally design-neutral way. The act of aggregating information objects together using instructional design theories and philosophies creates a learning object. To help capture this understanding of a learning object, Wiley suggests using the metaphor of an atom where information objects (neutrons, protons, and electrons) are carefully crafted together to form learning objects (atoms) which, because of their structure, can only be sequenced with other specific learning objects (atoms) and form units (molecules) which may form larger structures, like courses (crystals).

The concept of learning objects as an aggregation of smaller pieces of content is accepted by many other practitioners. Wu (Wu, 2002) describes an e-learning system that uses a set of "knowledge bits" to create learning objects. Knowledge bits are combined together by an instructional designer, with a keen eye at attempting to use the inheritance features described in section 2.1.1, to create the larger and more publicly reusable learning objects. Barritt et al (Barritt and Lewis, 2002) describe a model of a learning object as a set of five to nine information objects that includes specific overview, sum-

mary, and assessment mechanisms. Each information object is based on a single objective, and contains content, questions, and an indication of the user's ability in the area. These objects can be classified as a concept, fact, process, principle, or procedure, and may contain metadata describing the object.

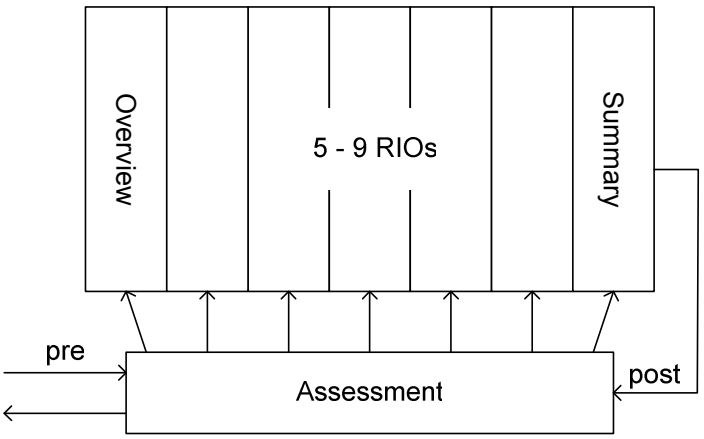

Figure 2-1: Reusable Learning Object (RLO) model adapted from (Barritt and Lewis, 2002)

NETg Inc., a producer of learning management software and an early adopter of learning objects, also mandates that assessment is a core component of a learning object's structure. As cited by Ip et al. (Ip, Morrison, and Currie, 2001), a learning object is a triple including a "learning objective, a unit of instruction that teaches the objective and a unit of assessment that measure[s] the objective.". They further limit their model of a learning object to non-interactive material to exclude the requirement that learning management systems need to make available methods to perform computational support.

Despite the differences between these learning object models, the majority of large open learning object repositories consider any file at the end of a resource indicator, usually a Uniform Resource Locator (URL), to be a learning object. URLs identify only the location and method by which a learning object should be obtained, and do not explicitly dictate the internal structure of a learning object. An analysis of the file extension used within the URL, or an inspection of the first few bytes of the learning object against known file types, may deduce the file structure at a coarse grain level. For example, one

might determine from a URL that ended in ".txt" that the learning object being referenced had the same structure as an ASCII text file. This assumption effectively overloads the function of a resource identifier, and is a dangerous (yet well practiced) procedure. The document might have an incorrect extension, or the extension might be used for many different kinds of documents. For instance, a file with the ".txt" extension might be a rich text file, or a text file that is marked up in Unicode instead of ASCII.

A number of smaller courseware projects have alleviated these problems by creating learning objects with predefined structures. One of the Technology Enhanced Learning projects completed at the University of Saskatchewan uses the Hypertext Markup Language (HTML) to create learning objects for online tutorials (Cooke et al., n.d.). These tutorials focused on teaching various computer science concepts, and included a number of elements of interactivity. The tutorials were written by third year computer science students, and typically contained Java applets, JavaScript, and Macromedia Flash. In particular, many of the learning objects utilize Java applets to display and evaluate multiple choice exams and user driven examples. Similarly, the Educational Object Economy (EOE Foundation, n.d.) is a learning object repository that is made up of more than 2,000 educational simulation resources. Each learning object is referenced by a URL, and is made up of a single HTML page that contains a java applet. These resources focus on interactivity, and often do not include any evaluation mechanisms.

While standardizing a particular structural format for learning objects helps improve reusability and interoperability, it is not enough to support the automatic use of learning objects by Learning Management Systems (LMS). Consider an applet designed to test a learner's knowledge of an HTML tutorial. How can a LMS instruct the applet to adapt questions based on the learner's previous answers? How can a LMS obtain the results of the learner's assessment and adapt further courses to the students needs? Without a description of the internal semantics of a learning object, or at the very least a description of the external interfaces of a learning object, an LMS is unable to reason intelligently about a learning object.

The Extensible Markup Language (XML) is seen as a key technology in helping to solve these semantic problems. XML facilitates vendor and platform neutral interoperability

by specifying an open data model for representing and manipulating semi-structured content called the Document Object Model (DOM), as well as a syntax for serializing this content for interoperability. It has been adopted by a large number of e-learning vendors as the preferred way to model learning objects, testing and assessment schemas, and learner profiles. Wu's thesis work (Wu, 2002) describes an online learning environment built around XML in which learning objects include and reuse assessment mechanisms from other specifications. The Department of Computer Science at the University of Saskatchewan developed a similar Learning Management System (Department of Computer Science, University of Saskatchewan, n.d.), that contains the ability to reuse XML based learning objects in different environments depending on the media available. In particular, learning objects can be rendered into HTML with Java Applets for use on the web, and are re-rendered (without interactive elements) into Adobe's Portable Document Format (PDF) for use in print.

2.1.3 Learning Object Metadata

Most standard and specification development in the area of learning objects has focused on developing metadata schemas to provide categorization and corresponding annotations for learning resources. Instances of a metadata schema are interrogated when a learning object is being searched for, either by software components or by learners/instructional designers. Metadata is also used when learning objects are being compared with one another to determine their suitability for a learner.

The Dublin Core Metadata Initiative (DCMI) (Dublin Core Metadata Initiative, n.d.) has a long history of creating metadata specifications for the World Wide Web. A number of these specifications have seen wide adoption not only among web page creators, but by corporate and government entities as well. Notable among these are the United Kingdom's e-Government initiative (Office of the e-Envoy, n.d.), the Canadian government's Treasury Board Information Management Standard (Treasury Board of Canada, n.d.), and the Irish Public Service Metadata Standard (Government of Ireland, n.d.). In August of 1999 the Dublin Core Education Working Group was formed within the DCMI with the charter of taking existing metadata specifications and making them more appropriate for educational materials. To date, a number of drafts have been released,

the first published in May of 2000 (Mason and Sutton, n.d.). This specification is made up of only fifteen elements used to describe the resource, though each element can be further refined through "Canberra Qualifiers" (Sutton and Mason, 2001) which are attributes that can be assigned to a given element to contextualize it to a specific domain. Many have argued that the DCMI specifications are too coarse grained to provide for effective education resource discovery, and a number of other organizations have created competing specifications. Principal among these organizations are the IEEE, the IMS Global Learning Consortium, Inc. (IMS), and the Alliance of Remote Instructional Authoring and Distribution Networks for Europe (ARIADNE).

ARIADNE was a European project supported by the European Union Commission, and the Swiss Federal Office for Education and Science. It ran from 1996-2000, and released a number of metadata documents that were used by both the IMS and the IEEE. Since December of 1997 the members of the ARIADNE project have worked closely with the IMS to develop an international metadata standard. The IMS is a consortium of commercial, governmental, academic, and other entities whose mandate is to promote open specifications for distributed learning. It is made up of predominantly British and American commercial entities, and was one of the first industry consortiums to define specifications for e-learning, including question and testing frameworks, course sequencing, and metadata models. The metadata model (IMS Global Learning Consortium Inc., 2003) developed by IMS was used as a base for the IEEE Learning Object Metadata (LOM) standard, and differs only in the naming and description of a few elements.

As an official standard, the LOM (IEEE, Inc., 2002) is perhaps the most authoritative learning object metadata specification. It describes a conceptual framework for educational metadata that is separate from any specific implementation. In addition to the conceptual framework, the IEEE provides a number of "bindings", or mappings from the conceptual model into specific markup languages such as XML, or the Resource Description Framework (RDF). This allows for semantically identical descriptions of a learning resource that are syntactically different, allowing for the meaning of the resource to be embedded in environments where one particular syntax is more appropriate than another. For instance, the XML syntax is extremely verbose and may not be ap-

propriate when transmitting large pieces of data over low bandwidth networks, while RDF documents can potentially have a much smaller syntax (e.g. (Grant, Beckett, and McBride, 2002)).

The LOM standard outlines nine categories each of which holds a set of data elements. Data elements can contain other data elements that form a hierarchical structure. Leaf nodes in this structure must contain data values, some of which use a closed and predefined vocabulary. Every data element defined by the LOM is optional with the exception that if child nodes are used, their appropriate parent nodes must also appear. In addition, authors are allowed to extend the model as they see fit as long as they use bindings that provide for interoperability with other author's extensions. To help facilitate interoperability efforts with the Dublin Core Metadata Initiative, the LOM includes an appendix that outlines how the two data models relate to one another. Table 2-1 outlines the structure of both the LOM and the DCMI educational specification.

This specification suffers from the opposite problem that hampers adoption of the Dublin Core, namely it describes an overly fine grained and complex model. In an attempt to fully define a model for educational metadata, vocabularies which are specific to a few ethnic or educational backgrounds have been created. In addition, the sheer size of the model makes full implementation very difficult. Even the best practices guide for the metadata specification from the IMS notes that:

> "Many vendors expressed little or no interest in developing products that were required to support a set of meta-data with over 80 elements…[and the] burden to support 80+ meta-data elements on the first iteration of a product is too great for most vendors to choose to bear" (IMS Global Learning Consortium Inc., 2003)

This realization that application developers may wish to pick and choose some elements of a model and combine them with elements of other models has resulted in the creation of application profiles. Broadly defined, by authors from both the DCMI and the IEEE, an application profile is:

> "An assemblage of metadata elements selected from one or more metadata schemas and combined in a compound stream… [whose purpose] is to adapt or combine existing schemas into a package that is tailored to the functional requirements of a particular application, while retaining interoperability with the base schemas. Part of such an adaptation may include the elaboration of local meta-

> data elements which have importance in a given community or organization, but which are not expected to be important in a wider context." (Duval et al., 2002)

While the original intention of application profiles was to compile a set of metadata elements from different schemas based on the application being used, the majority of application profiles are really being created to meet cultural and domain specific needs.

Table 2-1: IEEE Learning Object Metadata (LOM) outline

- 1 General
  - 1.1 Identifier
    - 1.1.1 Catalog
    - 1.1.2 Entry
  - 1.2 Title
  - 1.3 Language
  - 1.4 Description
  - 1.5 Keyword
  - 1.6 Coverage
  - 1.7 Structure
  - 1.8 Aggregation Level
- 2 Life Cycle
  - 2.1 Version
  - 2.2 Status
  - 2.3 Contribute
    - 2.3.1 Role
    - 2.3.2 Entity
    - 2.3.3 Date
- 3 Meta-Metadata
  - 3.1 Identifier
    - 3.1.1 Catalog
    - 3.1.2 Entry
  - 3.2 Contribute
    - 3.2.1 Role
    - 3.2.2 Entity
    - 3.2.3 Date
  - 3.3 Metadata Scheme
  - 3.4 Language
- 4 Technical
  - 4.1 Format
  - 4.2 Size
  - 4.3 Location
  - 4.4 Requirement
    - 4.4.1 OrComposite
      - 4.4.1.1 Type
      - 4.4.1.2 Name
      - 4.4.1.3 Minimum Version
      - 4.4.1.4 Maximum Version
  - 4.5 Installation Remarks
  - 4.6 Other Platform Requirements
  - 4.7 Duration
- 5 Educational
  - 5.1 Interactivity Type
  - 5.2 Learning Resource Type
  - 5.3 Interactivity Level
  - 5.4 Semantic Density
  - 5.5 Intended End User Role
  - 5.6 Context
  - 5.7 Typical Age Range
  - 5.8 Difficulty
  - 5.9 Typical Learning Time
  - 5.10 Description
  - 5.11 Language
- 6 Rights
  - 6.1 Cost
  - 6.2 Copyright and Other Restrictions
  - 6.3 Description
- 7 Relation
  - 7.1 Kind
  - 7.2 Resource
    - 7.2.1 Identifier
      - 7.2.1.1 Catalog
      - 7.2.1.2 Entry
    - 7.2.2 Description
- 8 Annotation
  - 8.1 Entity
  - 8.2 Date
  - 8.3 Description
- 9 Classification
  - 9.1 Purpose
  - 9.2 Taxon Path
    - 9.2.1 Source
    - 9.2.2 Taxon
      - 9.2.2.1 Id
      - 9.2.2.2 Entry
  - 9.3 Description
  - 9.4 Keyword

Notable profiles include the:

- Canadian Core Learning Object Metadata Application Profile (CanCore) (CanCore Initiative, 2002), which is a subset of the LOM and provides vocabularies specific to the Canadian educational system,
- Le@rning Federation Profile (Friesen, Mason, and Ward, 2002), which has taken a cross section of metadata elements from the Dublin Core Metadata Element Set, the Dublin Core Qualifiers, the EdNA Metadata Standard, and the LOM to create a localized educational schema for the Australian educational system
- Gateway to Educational Materials (Gateway to Educational Materials Consortium, n.d.), which has extended the Dublin Core Metadata Element Set with a number of American specific educational vocabularies, and the Health Educational Assets Library (HEAL) (Dennis et al., n.d.), which specializes upon elements found in the IMS metadata specification.

2.1.4 Learning Object Repositories

A learning object repository is a centralized collection of learning object metadata descriptions, and a search service allowing access to these descriptions. This collection is meant to be accessed by learners and educators, and can be embedded within a learning content management system. Many learning object repositories are also meant to be used without the aid of a content management system, and are available for open access over the web. The majority of learning object repositories do not include learning objects themselves; instead, they link to learning objects through URLs and provide indexing, categorization, and searching services.

The Multimedia Educational Resource for Learning and Online Teaching (MERLOT) is one of the oldest and most well-known learning object repository. It was started in 1997 by the California State University and links to thousands of learning resources including presentations, course notes, examples, and interactive applications. Membership for non-commercial entities to use MERLOT is free, and all of the educational materials that are submitted are peer reviewed before they are accepted. MERLOT stores only the metadata associated with the learning object being entered, and not the learning object

content itself. The metadata used by MERLOT is a subset of the LOM, and the only way to interact with the system is through a series of web pages.

The Campus Alberta Repository of Educational Objects (CAREO) is an Alberta based learning object repository created by the University of Alberta, University of Calgary, and Athabasca University. Unlike MERLOT, CAREO allows users to upload both the metadata and the learning object itself. While the majority of learning objects in CAREO are categorized and indexed using the IMS metadata specification, the repository architecture allows for any XML schema based metadata to be used. Like MERLOT, CAREO allows for peer reviewing of resources that have been submitted, and users can both browse and search through the repository using a set of web pages or interact through third party applications. Materials can be uploaded to CAREO using the Advanced Learning Object Hub Application (ALOHA) client tool. This tool interacts with the CAREO server through a XML-RPC based API, and allows for the creation of arbitrary metadata based on an XML Schema file. Further, it can automatically extract IMS based metadata from over 200 different file types (Magee et al., 2002).

Repositories such as MERLOT and CAREO are susceptible to the traditional centralized management problems that arise in distributed computing including central point of failure, limited scalability, and monolithic access rights. To deal with this, the IMS has created a Digital Repository Interoperability Information Model (2003) that builds upon XML based standards such as XQuery and the Simple Object Access Protocol (SOAP). This model defines operations for searching, downloading, submitting, and alerting users when new learning objects appear.

The recent popularity of peer-to-peer file sharing systems such as Napster and Gnutella has encouraged a number of similar approaches for e-learning. Edutella (Nejdl et al., 2002) is a peer-to-peer based network that provides for resource searching based on the Resource Description Framework (RDF). It is built on top of Project JXTA, a peer-to-peer infrastructure implemented in Java by Sun Microsystems. Peers send and receive requests using any metadata schema they need to, and a number of prototype applications have been developed that use the LOM and DMCI specifications.

Richards and Hatala (Richards and Hatala, 2002) (Hatala and Richards, 2003) have expanded on the idea of decentralized repositories for learning, and created a set of three hierarchically arranged distributed applications for the search and retrieval of learning objects. The first of these tools, called SPLASH, is a desktop client that stores a user's learning objects as local files, and supports the notion of optimal storage as being "close to the creator and close to the user" (Richards and Hatala, 2002). These learning objects may be files that were created by the user, or files that have been used recently by the user. As users search for more learning objects, their search queries are propagated to other SPLASH clients. Search and response queries are both encoded with the CanCore metadata application profile. Larger repositories of similar learning objects can be set up to help support the notion of communities of interest. These repositories are set up using PONDs – a special SPLASH implementation that can provide an interface to third party repositories (such as CAREO or MERLOT). A POND is specialized to perform fast searching and maintain robust database support. PONDs are connected to one another using POOL Central nodes. A POOL Central node replicates queries throughout broadband networks (such as CA*Net3) and may collate results for POND clients. It is important to realize that this is an architectural model, and any specific client implementation could contain the functionality of all three. Figure 2-2 describes this architecture.

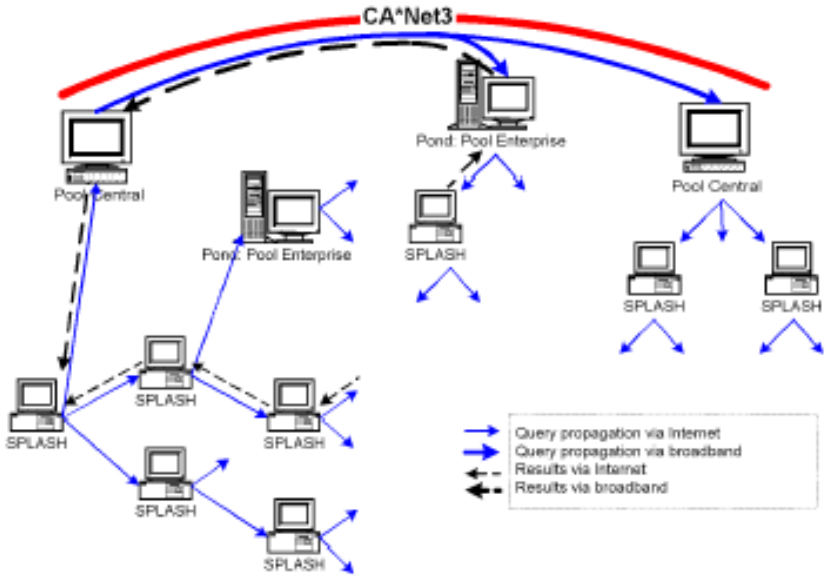

Figure 2-2: POOL architecture, taken from (Richards and Hatala, 2002)

Finally, the eduSource project (McGreal et al., 2002) was a collaboration of Canadian industrial, governmental, and academic institutions responsible for creating a set of interoperable learning object repositories. Perhaps one of the most interesting results of this project was the creation of the eduSource Communications Layer (ECL) (Hatala et al., 2004) which is both a specification and a publicly available implementation library that facilitates communication between repositories. Using this approach, any number of different repositories can be easily included in a larger federated network allowing for more diverse searching.

2.1.5 Learning Object Aggregation

Learning objects can be further broken up into structures of varying granularity. While the simplest view of granularity splits a learning object into only two parts (information objects and their larger learning structures, as discussed in section 2.1.1), more complex levels of aggregation can be imagined. Support for multi-level granularity is included in some metadata taxonomies (for instance, the LOM includes element "1.8 aggregation level").

Aggregation is the process of combining learning objects to form larger (in scope, depth, or size) learning objects. While there have been a number of models suggested for describing aggregation, such as the organic aggregation model (Paquette and Rosca, 2002) and the layered design model (Wiley et al., 2000), there are two principle specifications in widespread use today; the IMS Content Packaging specification (IMS Global Learning Consortium Inc., 2004), and the IMS Simple Sequencing specification (IMS Global Learning Consortium Inc., 2003).

The IMS Content Packaging specification outlines a method for organizing files into logical learning units called a content package. The package is made up of a manifest file which outlines metadata for the package as a whole (usually using the LOM format as described previously), a set of organization hierarchies outlining how smaller units of learning should be arranged, a collection of references to digital assets (e.g. particular files in the package) which in turn include descriptive metadata, and potential sub-manifests which effectively allow for the nesting of content packages within one another.

A content package fully encompasses a learning object in the sense that it includes all digital resources a Learning Content Management System (LCMS) would require to display that object to a learner. This specification is widely supported by a number of different authoring tools and environments.

Organization hierarchies or portions of organization hierarchies can be further sequenced using the IMS Simple Sequencing specification. This specification provides a more general data model for learning activities, where each activity is hierarchically arranged and annotated with rules describing when the activity should be made available to a learner. These rules provide simple tracking and adaptation abilities, by mapping possible activity outcomes (e.g. satisfied, status changed, etc.) to actions to be taken by the learning content management system. Figure 2-3 depicts this data model.

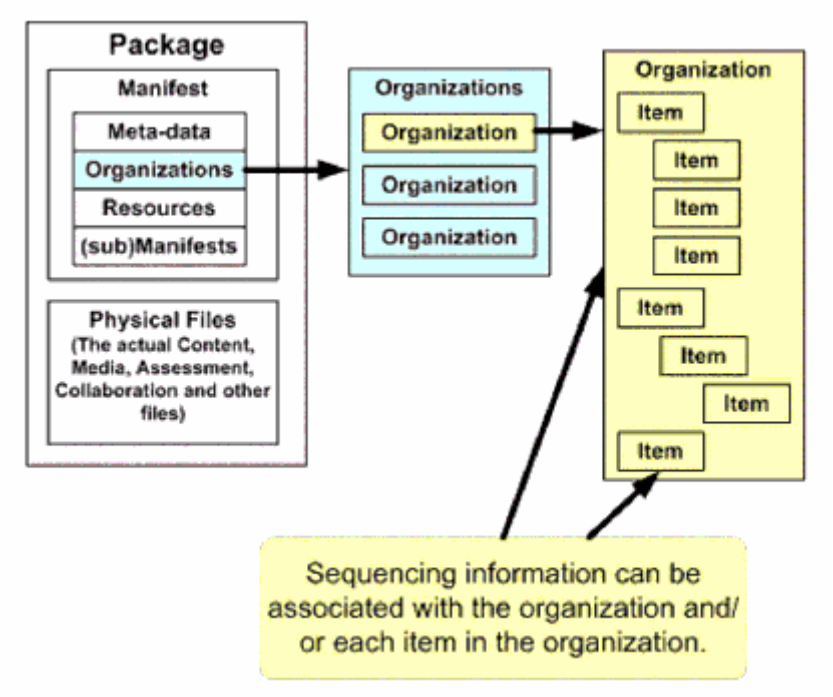

Figure 2-3: Structure of IMS Content Package (IMS CP) with sequencing information, taken from (IMS Global Learning Consortium Inc., 2003)

While tool support for the simple sequencing specification is sparse at present, it is being aggressively pursued by a number of LCMS vendors.

2.1.6 Summary

Learning objects are, in a general sense, an aggregation of ordered content meant to be reused in different educational situations to form larger units of learning. This content is described with metadata that is used to aid in the search and retrieval of learning objects. While a number of metadata specifications are available for use, the trend is towards creating application profiles – collections of elements found in a variety of schemas which are more suitable for a given culture or domain. Finally, learning objects are most often stored in some form of repository that may or may not be integrated into learning management systems.

As instructional designers find and integrate learning objects into their courses, they adapt and modify these objects to better target their audience. These adaptations are not only useful for the authors who create them, but may be used by software components or

learners who are browsing for information on a particular topic, or other authors who are looking for a starting point on which to base their derivative works.

Capturing version changes for electronic documents is not trivial, but has been highly researched. The following section briefly summarizes the state of the art in version control generally.

2.2 State of the Art in Version Control

Documents, both electronic and non-electronic, change over time. While the nature of change is often domain specific, it is common for a document to be derived from others with the intent of either updating the information being presented, or providing different views on that same information. The field of versioning is concerned with the management of electronic documents as they change over time. In the most general sense of the word, a version is a "form or variant of a type or original" (Merriam-Webster Inc., 2004). The act of creating and maintaining new versions is typically known as versioning, and has been a core concept explored in a number of computer science application areas including databases, knowledge representation, and Software Configuration Management (SCM).

Section 2.2.1 briefly introduces three core principles that relate to version control. This is followed with a survey of the commonly accepted version control terminology in section 2.2.2. Section 2.2.3 briefly summarizes the core functions that exist in version control implementations. Finally, section 2.2.4 examines the overlap between the field of learning objects and version control, and identifies how current e-learning standards address the need for versioning.

2.2.1 Version Control Principles

Versioning has been a central focus of research within the Software Configuration Management (SCM) community for over twenty years. The research is mature, and has been applied successfully in a number of other communities including databases (Roddick, 1995), hypermedia design (Bendix, Dattolo, and Vitali, 2001), and knowledge representation (Klein and Fensel, 2001). Version control relies on a *version model* which identifies the kinds of artifacts to be versioned, the properties associated with those artifacts,

and the changes that can be applied to artifacts to result in a version change (Conradi and Westfechtel, 1998). Further, version control is guided by two central principles:

- Sameness principle: Versioning requires a method by which two artifacts can be examined to see if they are of the same version (Conradi and Westfechtel, 1998). Being syntactically the same is often not enough to identify that two artifacts are of the same version; sometimes the ancestry of the artifact is important in determining the context that it will be used in. In addition to this, syntactic comparison is often unachievable because it requires knowledge of the syntax being used.
- Immutability of artifacts: Artifacts in a versioning model are considered immutable, in the way that changes to an artifact force the creation of new version of that artifact. This ensures that the history and traceability of the artifact is never lost (Bendix, Dattolo, and Vitali, 2001).

These lead to a derivative third principle:

- Uniquely identifiable: Given the sameness and immutability principles, there must be some way of identifying each artifact, and each version of each artifact, uniquely. This is usually implemented by giving each software artifact a unique *object identifier* (OID), and then giving each version of that artifact a unique *version identifier* (VID) (Conradi and Westfechtel, 1998).

2.2.2 Version Control Terminology

A change to an artifact under version control creates a new version of that artifact, and each version of an artifact is given a VID. The structure and values of a VID are often derived from other VIDs and OIDs to encode an overview of the history of the artifact being versioned. In addition, a VID may also be made up of symbolic names or statements commonly referred to as *tags* (Cederqvist, n.d.), which are given by users. The properties common to all versions of an artifact are referred to as *invariants* (Conradi and Westfechtel, 1998). Often the only invariant for a set of versions is the reference to the artifact they are a version of. Nonetheless, it is useful in some versioning models to

enforce a given set of invariants. Individual versions that disagree on an invariant property are considered to be unrelated versions (versions of different artifacts altogether).

The set of changes between two versions is referred to as a *delta*, and can be captured either as the set of differences between the two artifacts (called a *symmetric delta*), or a set of change operations that can be applied to one version of the artifact to create the new version (called a *directed delta*) (Conradi and Westfechtel, 1998). Deltas are usually accompanied by human readable comments in the form of *logs* which outline the semantics or significance of the changes that have been made. Further metadata, such as the date of the change and the author of the change, are attached to deltas in an implementation specific manner.

Versions that are meant to replace previous versions are called *revisions*, while versions that are meant to coexist with previous versions are called *variants* (Conradi and Westfechtel, 1998). The set of all revisions and variants of an artifact is known as a *revision group*. The reason for changes in an artifact is domain dependant; Conradi and Westfechtel cite bug fixes, extending functionality, and changes in dependencies as common reasons for creating new versions when dealing with software artifacts (Conradi and Westfechtel, 1998), while Noy and Klien identify changes in a domain, conceptualization of a domain, or specification of a domain as reasons for change in ontology based knowledge representation systems (Noy and Klein, 2002).

Version control systems typically represent the relationships within revision groups using linear sequences, trees, graphs, and grids. The most commonly used form of representation is a hierarchical directed acyclic graph. Nodes in the graph represent different versions of an artifact, with the root nodes representing initial artifacts. Nodes further down the hierarchy represent versions created over time, and each node has a name identifying that node as a revision or a variant. Edges in the graph represent deltas and, while tags and logs are rarely represented, they are sometimes attached as textual notes to nodes and edges respectively. An artifact that is connected with two or more incoming edges is a *merged* artifact – a version that is a partial subset of the union of the versions that are its parents in the revision graph.

Names of the nodes in the reversion graph are version identifiers VIDs, and are usually derived from the VID of previous versions. The root nodes of the graph are either OIDs, or simple VIDs that are unique to the system. The format of a VID is often two numbers separated by a period. The first number is known as the *major number* while the second is referred to as a *minor number*. Significant changes to the artifact that may or may not be backwards compatible with previous versions result in the major number being incremented. Changes that are less significant and are backwards compatible with previous versions result in the minor number being incremented. The set of revisions descending from the root of the graph are referred to as the *trunk*, while variants are represented as *branches* off of this trunk. To differentiate between a branch node and a trunk node, branch nodes generally append a period and a new VID to the VID of the node they are derived from.

Figure 2-4 shows an annotated example of a revision graph. In this revision group there is one artifact that has undergone a number of minor revisions, followed by a single major revision. Several branches are created to allow for parallel development of the artifact by other people, and are later merged back together. Note that the VID of a merged version can be chosen as a branch off of either of the versions it is derived from, depending on the context of the newly created version. The example shown indicates that, while the merged version derives from versions 1.3 and 1.2.1.1, the author has identified it as superseding the work done in 1.3 and has added it to the trunk of the development line.

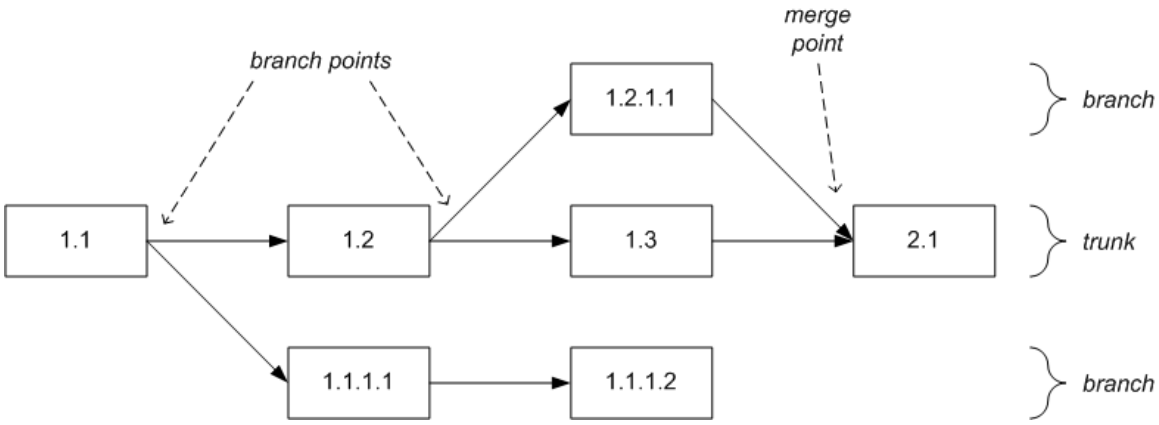

Figure 2-4: Annotated revision graph

2.2.3 Version Control Functions

A number of different implementations for version control exist, including RCS (Tichy, 1985), CVS (Berliner, 1990), and SVN (Collins-Sussman et al., n.d.). While differing in the specifics, these implementations share a number of features:

1. They are all logically centralized. While many version control repositories can be physically distributed on the back end to improve performance, most use a centralization OID and VID naming scheme to ensure consistency. At the time of writing the author could find no examples of logically decentralized version control systems in the SCM literature.

2. They use low level directed deltas. These deltas may be either forward deltas, where the artifact is stored in its initial form, and new versions are recreated when needed, or reverse deltas, where the newest version is stored in its entirety, and deltas are applied to retrieve older versions. Deltas are encoded either at a byte level, for binary artifacts (e.g. images), or at a character level for text objects (e.g. source code). Reverse deltas are the most popular approach, as they allow for the immediate access to the latest revision, and O(n) access to older versions.

3. They allow for rollbacks of an artifact to an earlier version by applying the appropriate deltas. These rollbacks can then be committed as new versions to the software repository.

4. They support the author-assisted merging of two versions into some new version. While most tools attempt to do this merging automatically, the lack of knowledge of the underlying semantics of the item being version requires that often an author must be included in the process.

2.2.4 Versioning of Learning Objects

Learning objects are in constant evolution and are often associated with versioning information. Consider the case of a traditional textbook as a learning object. Textbooks have edition number and printing dates, roughly corresponding to major and minor numbers. They also often contain a preface that represents the logs which document the changes (deltas) that have occurred. Often several volumes on related topics are merged together for special edition texts that are more comprehensive. Learners use this versioning information when assessing a textbook's suitability for a particular course. A previous edition of a textbook may be a poor helper for a course, though sometimes the differences are minor enough that a previous edition will work just as well as a current edition. Previous editions may carry other benefits, such as being easier to find, or cheaper to purchase.

Current metadata standards have little or no support for the versioning of learning objects. The Dublin Core contains only two useful elements, the *source* element and the *relation* element. The *source* element identifies the resource that the new learning object was derived from, using some uniquely identifying index (usually in the form of a Uniformed Resource Identifier (URI)). There are no options to further annotate this element to capture how the current learning object differs from the previous one. The *relation* element allows for a more general form of expressing the relationships between two resources. It allows for a resource to indicate that the current learning object is either a variant of another learning object, or a revision of another learning object by using the modifiers *IsVersionOf* and *Replaces* respectively. Two other modifiers, *HasVersion* and *IsReplacedBy* allow original learning objects to identify the same relationship with respect to their derivative works.

The creation of the LOM was influenced by a number of different metadata specifications, and thus it contains many mechanisms that are similar in name and function to

those in the Dublin Core. One of these mechanisms is the *relation* element that allows authors to describe relations using a simple vocabulary. This vocabulary corresponds closely to that of the Dublin Core, and allows for using the *hasversion* and *isversionof* keywords. The *relation* element also allows for identifying a derivative object similar to the source element, by using the *isbasedon* and *isbasisfor* modifiers keywords.

Also included in the LOM is the *lifecycle* category, which contains a number of data elements that describe the history of a learning object. The *version* element describes the edition of the object using human readable plain text, while the *status* element describes the completion status of the object using a small, predefined vocabulary (draft, final, revised, or unavailable). The *contribute* element identifies what entities have influenced the history of this learning object (such as other authors, organizations, etc).

While both the Dublin Core and the LOM provide solid foundations for describing a simple revision tree, they fail to offer full support of a version model. Indeed, in an analysis of the LOM done by CanCore, it is stated as only generally addressing the concerns of versioning, and even then it does "not do so in a way that is sufficient for the requirements of many projects" (CanCore Initiative, 2002). Instructional designers or software components who are interested in determining the significance of a given change to a learning object are forced to obtain versions of both learning objects, and attempt to reason about the metadata. Comparing metadata records is not a trivial task – the majority of elements in both the Dublin Core and the LOM are meant for human consumption and would be difficult or impossible for a software agent to interpret. Vocabularies are often left open ended, and force implementers to either come up with their own terminology or adopt a given application profile. This leads to data that is difficult even for intelligent software components to understand when trying to dynamically obtain and compose learning objects into larger educational units.

The remainder of this thesis will address these issues by presenting a formal model for a learning object, identifying the artifacts of this model that are important when supporting versioning, and outlining a process for capturing and codifying the nature of change associated with versioning.

Chapter 3
A Model for Learning Object Versioning

Conradi and Westfechtel identify that a core component of versioning is the version model which identifies the kinds of artifacts to be versioned, the properties associated with those artifacts, and the changes that can be applied to artifacts to result in a version change (Conradi and Westfechtel, 1998).

In practice, most version control systems also capture the *nature of changes* as they are applied to artifacts under version control. These changes can be captured both as a collection of structural operations (how the artifact has changed with respect to its internal structure) and as a collection of semantic operations (how the artifact has changed with respect to its meaning). Semantic operations can be further broken down into computer understandable semantic changes, and human understandable changes. Section 3.1 outlines a version model for learning objects. Section 3.2 provides bindings of this model to current web specifications used for learning objects, in particular the Extensible Markup Language (XML) and the IMS Simple Sequencing Specification (IMSSS). Section 3.3 identifies how machine-readable semantics can be attached to learning object change sets, and provides two example vocabularies for this purpose. Finally, section 3.4 concludes the chapter with a general discussion of the benefits and limitations of this approach.

3.1 Learning Object Version Model

3.1.1 Artifacts and their properties

There is rough consensus that a learning object is an ordered aggregation of content, often referred to as reusable information objects (Barritt and Lewis, 2002). This aggrega-

tion is annotated with descriptive metadata that allows for discovery by both humans and software components. The order of content included in a learning object can take a number of forms, including linear (e.g. that implied in (Barritt and Lewis, 2002)), hierarchical, and completely unstructured. At the simplest level, a learning object could contain no sequencing information and only one information object (referred to by (IEEE, Inc., 2002) as being atomic). From a purely structural viewpoint, both the granularity and the breadth of information being shared is unrelated to the number of learning objects being created. For example, an instructional designer could create an atomic learning object that represented a whole course, while another designer could take just one lesson in that course and break it up into several smaller learning objects that are hierarchically arranged. Further one designer may decide to break an overview of a topic into a number of different learning objects, while another designer may create a single object that goes into the content in great depth.

Similar to the object oriented or procedural programming paradigms, learning objects can aggregate one another either by value or by reference. Typically, several smaller learning objects are packaged up into a single file along with a manifest outlining the structure of the objects, and is delivered as itself as a learning object. This example of content packaging is similar to "passing by value" or "deep copying" available in traditional programming languages. To contrast, learning objects may also be arranged in a manifest and referenced (typically through URLs, though other schemes are available) remotely, similar to "passing by reference". Both approaches have their benefits and detriments – formulating a learning object by value ensures to the instructional designer that the learning object will not be impacted by negative changes to the original content. However, new changes (which may be beneficial) are not reflected in the work, increasing the amount of time an instructional designer must spend on updating a course.

The notion of whether an instructional designer should choose to reference or deep copy a learning object when designing a course depends on a number of factors, including the technology available, the license for the content being copied, and the trust the designer holds in the third party to update the content appropriately for the context being taught. While this is an important issue for learning object design teams, it is beyond the scope of this thesis. Instead, this work makes the assumption that aggregated learning objects

follow a referential model, where aggregates are defined by a single piece of content (or a content packaging file), and a version identifier which shows which version of that content should be used.

To ensure clarity the following terms and their definitions will be used to describe both the artifacts and properties of those artifacts in the version model:

> **Definition 1.** Primitive learning object: A learning object that contains a single piece of content with a descriptive set of metadata. This object has no dependencies with other learning objects (e.g. sequencing information), and may be made up of an arbitrary number of digital resources. This object has one single entry point (e.g. a content packaging manifest).
>
> **Definition 2.** Sequenced learning object: A learning object that contains an aggregation of either other sequenced learning objects or primitive learning objects. Aggregation is accomplished by holding a reference in the form of a URL to the aggregated objects. A sequence learning object also contains a single set of metadata which describes it as a whole. Sequenced learning objects contain no content of their own.

Each learning object, sequenced or primitive, can also include annotations in the form of metadata. Metadata is typically added as key/value tuples where the key is related to some ontology of terms (such as those given in (IEEE, Inc., 2002) or (DCMI Usage Board, n.d.)) and the values are either a selection given from a constrained vocabulary (if defined in the ontology) or free form text. While some metadata profiles further arrange key/value pairs into a directed tree, this can be represented as a set of tuples assuming the original tree does not contain duplicate paths. Thus a third definition for the product space includes:

> **Definition 3.** Learning object metadata: A set of key/value tuples that describes a given learning object. Instead of prescribing a specific metadata profile to use, each key is prefixed with a unique namespace identifier linking that key to the ontology which defines it. For instance, the following set contains two metadata entries describing the title and author of a learning object using the LOM and Dublin Core metadata vocabularies respectively:

{ (lom:title , “Introduction to Java”) , (dc:author , “Chris Brooks”) }

3.1.2 Version model change set and the immutability principle

The immutability principle described in section 2.2.2 states that all artifacts under version control remain unchangeable, and that attempted modifications to an artifact result in a new version of that artifact. Thus a change to any portion of the learning object (content or metadata) results in a derivative learning object being created. The difference between an initial learning object and a derivative of that learning object is referred to as a *change set*.

Unlike traditional software content repositories, learning objects that may be used by one instructional designer can be distributed into many different repositories. These repositories generally do not have a way of synchronizing the identification of, searching for, and publishing of learning objects. It then makes sense to attach the change set for a learning object with that learning object, so authoring environments and delivery tools can inspect the history of the object regardless of where it originated from. Thus the definition of a sequenced and primitive learning object can be expanded to indicate this coupling:

> **Definition 4.** Learning object change set: A learning object change set is a set of repeatable operations that can be applied to create a particular version of a learning object. This change set includes structural changes (e.g. the addition of content for a primitive learning object, or the addition of sequencing information for a sequenced learning object) as well as semantic (metadata) changes. This is a forward delta.

The structure of a change set for content is highly dependent on the format of the content that has been changed. For instance, content in textual form could contain a change set which identifies the lines and characters that have changed and what their new values should be. This technique does not work well for binary files which may have to include byte offsets and new byte values to change. In addition to these low level data file format change sets, some kinds of content may be able to be versioned at multiple levels of granularity. Consider the case of a learning object which contains an image encoded in the Scalable Vector Graphics (SVG) format. The SVG specification bases its data

model on the XML Document Object Model (DOM) (Ferraiolo, Fujisawa, and Jackson, 2003). In doing so, an SVG document can be described using the XML DOM. The XML DOM in turn uses Unicode characters to serialize the data model to disk. Thus, an SVG image could be versioned at at least three different layers of granularity; the SVG model layer, the XML data model layer, or the XML serialization layer.

The diversity of data models for learning objects as presented in Chapter 2 makes it difficult to describe what form the content of a learning object should take. This in turn makes coming up with a single versioning change set model for all learning objects difficult. Instead, the approach taken is to come up with a lightweight abstract change set model, then provide data type specific vocabularies to use with this model.

A brief survey of version control software, as well as learning object data file formats suggests the following three items should be captured for learning objects as artificats under version control:

- Human readable description of the change (logs)
- A path to the item that changed (e.g. character position, byte offset, line number, xml element, node in a data model graph)
- A transformation describing how the item was changed, or what the new value of the item is and how to revert to the old value (if reverse deltas are desired)

Of these three elements, the logs are optional and may apply to any number of specific changes that an author has done. The path and transformation items, however, are required and are intended to be consumed directly by a software component responsible for displaying various versions of a learning object.

As a property associated with both sequenced and primitive learning objects, metadata entries also need to have change sets associated with them. Metadata keys are constrained to a specific vocabulary, and items within a vocabulary often share a relationship with one another based on some given data type. For instance, entry 5.4 in the LOM metadata profile contains the key “semantic density” which is made up of five levels; very low, low, medium, high, and very high. An understanding of the structure of this data type (an ordered enumeration) can be useful for an authoring tool – entries

can then be searched or visualized taking into account bounds on individual metadata keys. For instance, a vocabulary aware repository could visually display learning objects with a low semantic density level of orange, and deepen the colour to red for higher semantic density levels. This allows a user of the repository to quickly locate those items of relative high or low semantic density, while holding other search terms constant.

A change in a metadata entry then produces two measurable changes, a change in the value of that entry (the content), and how the entry has been transformed with respect to its data type. Continuing with the example of semantic density, if there exists a learning object with a semantic density of "low", and a derivative is made where the semantic density is "high", the change set could be codified as "increased". Section 3.2 outlines two possible data vocabulary options.

The small size of metadata entries and their semantic transformations (usually a single word) makes it reasonable to just store the current and previous values of those entries instead of using directed deltas. Determining the value of the semantic transformation can be done in two different ways:

1. The learning object software, potentially an editing environment, repository, or delivery tool, can derive this value automatically. This requires that the software has knowledge of the underlying data types that exist within the metadata profiles being used. For instance, an authoring environment that understands the LOM schema could automatically fill in the transformation value of "increased" when the author provides higher values of the semantic density.[2]

2. Require that the learning object author provides an explicit representation of the change in the learning object version history. This increases the size of the history (and thus the size of the learning object), but removes the requirement that the software being used by the author needs to understand metadata profile data types.

[2] Strictly speaking, the published machine-readable LOM schema is not enough to provide for this level of reasoning. This schema uses the XML Schema enumeration data type in an unordered fashion. An application wanting to provide this level of functionality would have to augment the formal LOM schemas with further knowledge.

Adopting the latter approach does not restrict software that understands the metadata profiles being used from filling in appropriate change notations in the version history. Thus a model which supports the explicit representation of change semantics within it is the most flexible approach. Taking this into account, the following three items must be captured for each semantic change:

- The change set transformation of the metadata entry (e.g. “increased”, or “subset of previous value”)
- The perspective of this transformation (the ontology key to which it pertains)
- The new value of the entry (e.g. "low")

Table 3-1: Change set data model

Nr	Name	Explanation	Cardinality	Ordered	Datatype
1	History	The set of changes that have occurred to this learning object	1	n/a	-
1.1	Change	A container for revision information. Changes are temporally ordered	*	ordered	-
1.1.1	Structural	A change in the structure of the learning object (e.g. XML, sequencing for complex objects, etc)	*	ordered	-
1.1.1.1	Path	Identifier which references that portion of the content which has changed (content specific)	1	n/a	String
1.1.1.2	Transformation	Transformation which changes the content given by the path to the new content (content specific)	1	n/a	String
1.1.2	Semantic	A change in the metadata associated with a learning object	*	ordered	-
1.1.2.1	Value	The new value of the metadata entry	1	n/a	Vocabulary (State)
1.1.2.2	Transformation	The manner by which the metadata has changed	1	n/a	String
1.1.2.3	Perspective	The key for the metadata entry (e.g. lom:semantic-density, dc:author)	1	n/a	String
1.1.3	Log	Human readable description of the changes	1	n/a	String
1.1.4	Vid	A version identifier for this change	1	n/a	String
1.2	Identifier	A unique identifier for this learning object	1	n/a	String

Table 3-1 provides an overview of the change set data model. In this model some elements may appear multiple times as indicated by a cardinality level greater than one.

Further, some elements must be presented in an ordered fashion, such as the structural change values. Appendix A defines a normative binding specification for this model using XML Schema.

The uniquely identifiable principle described in section 2.2.1 is supported both at the object level and at the version level by use of the "Identifier" and "Vid" elements respectively. These elements must be represented by globally unique strings such as a Uniformed Resource Indicator (URI). While not prohibited, a Uniformed Resource Locator (URL) is a poor choice for a learning object identifier, as objects are expected to be able to move between repositories.

3.1.3 Sameness Principle

The notion of version control requires that a method exist to determine whether two artifacts are of the same version, called the *sameness principle*. For this method of version control, the sameness method can be defined as follows:

> **Definition 5.** Sameness method: Ensuring that the set of change operations, excluding the human readable log, for two or more learning objects result in both structurally and semantically identical states after each application.

This allows for different implementations to capture structural changes in different ways, but ensures that once each set of structural changes is applied, the resulting learning objects are identical.

3.1.4 Conclusions

By keeping the transformation values for both structural and semantic changes abstract, this model is domain, vocabulary, and granularity neutral. By considering the change set of a learning object to be an integral part of that object, the version model can be used in distributed environments. Finally, by providing a binding of the version model to an XML schema that supports namespaces, it makes including version information with other XML based e-learning specifications (such as the IMS Content Packaging or Simple Sequencing specifications) easy.

3.2 Vocabularies for Expressing Learning Object Structure

3.2.1 Primitive Learning Objects

Learning object data formats range widely depending on the intended use of the object. For instance, the CAREO learning object repository includes learning objects of various graphic types, archived collections of files, URL's to external resources, and XML documents.

The Extensible Markup Language (XML) is a mature and reasonable technology for describing the structure of primitive learning objects. XML was born out of the desire to come up with a generalized markup language based on experiences with the Hypertext Markup Language (HTML) (Pemberton et al., 2000), and the Standard Generalized Markup Language (SGML) (International Organization for Standardization (ISO), 1986). It has seen rapid adoption in the areas of content creation, and a number of typographical formats (e.g. the Extensible HyperText Markup Language (XHTML) (Steven Pemberton et al., 2000), and the DocBook document format (OASIS DocBook Technical Committee, 2002) exist. Anecdotal evidence suggests that, while new learning objects are being created in a variety of formats, well-formed XML documents are a popular choice (e.g. (Cooke et al., n.d.)). This section will deal with how the version model can be applied to primitive learning objects that use XML as their underlying data model.

Every well-formed XML document conforms to the Object Model (DOM) (Wood et al., 2000). This model provides a platform and language-neutral API for interaction with the underlying document. This interaction includes navigation, retrieval, and modification of portions of the document. Instantiations of the XML DOM form an in-memory hierarchical tree of a series of seven kinds of data structures; an XML declaration, character data, elements, attributes, entity references, comments, or processing instructions.

Using the API, the DOM can be serialized to and from character data stored in a variety of character encoding formats. When serialized to the XML data file format, only state information about the document is kept, not a history of how the document was created. This lack of historical record makes version control impractical, as a serialized snapshot of the DOM would need to be taken after invoking any method on the DOM

API that may modify the underlying document. For small changes to a document, the resulting history would include a huge amount of data that would be impractical for even the most ambitious repositories to keep.

Instead, it is useful to describe a set of bindings for invocations against the DOM API. By evaluating portions of this change set, an authoring tool can recreate an XML document into any previous state it contained. In addition, since a given method invocation on the DOM results in only the input parameters for completing that invocation, change sets are typically smaller than serializing the whole document for each change.

3.2.1.1 A DOM Change Set Vocabulary

The DOM API is made up of a number of interfaces which contain methods and constants defined for navigation, retrieval, and modification of one or more DOM instances. By invoking operations on these interfaces, a directed acyclic graph of nodes is formed. Only those methods which actually change the structure or values of nodes within a DOM tree need be captured to maintain reproducible version control. In addition, even though the DOM is only a set of interfaces, models are considered structurally isomorphic in that different implementations may build the model differently, but the result will be the same (Le Hors et al., 2004). Figure 3-1 provides a UML class diagram of the DOM including only those interfaces and methods that perform structural changes to the DOM.

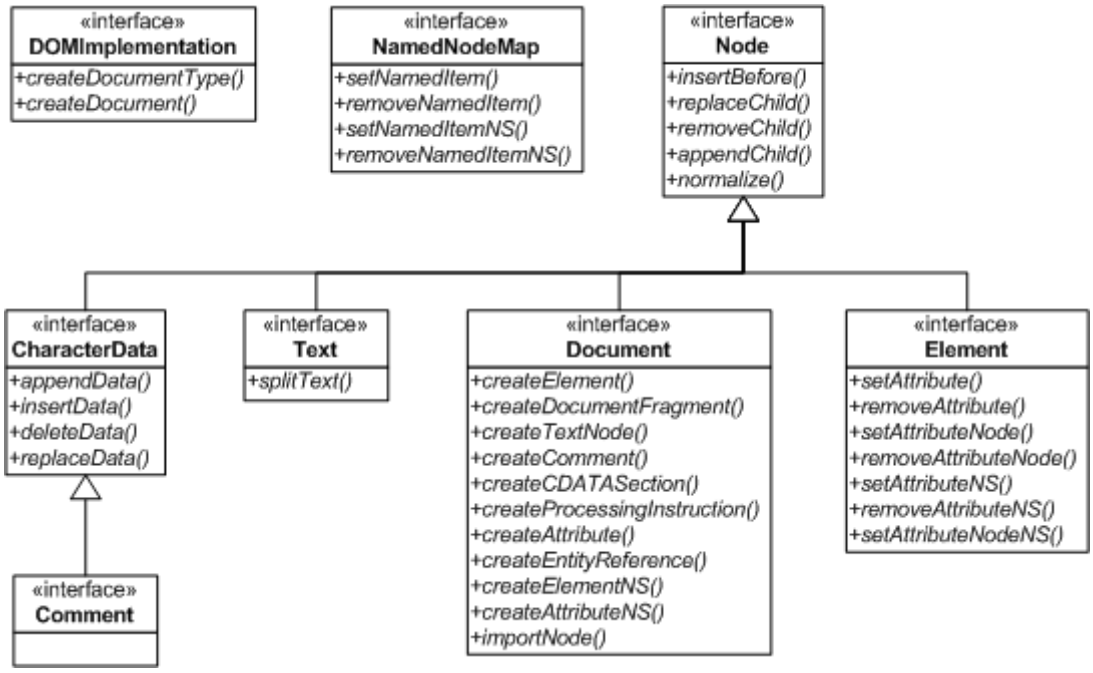

Figure 3-1: UML class diagram of the DOM interfaces

While the DOM API allows the creation of and modification of more than one document at a time, we have considered only the case of interactions with a single document. Because of this, the methods in the DOMImplementation interface create both the default document and the default document type declaration.

The DOM API is meant to be used programmatically, where nodes can be retrieved by reference from a document using the navigation methods and then modified. Serializing successive calls to Node objects to traverse the DOM tree would be extremely verbose. An alternative approach is to use the XML Path Language (XPath) (Clark and DeRose, 1999) to denote which node is in current scope. This language represents paths through the DOM tree structure as strings, similar Uniformed Naming Convention (UNC) paths on a filesystem. It allows for the addressing of nodes by type, and has semantics for distinguishing between element nodes, text nodes, attributes, processing instructions, and entities. XPath has become a de facto standard for querying a DOM tree, and most XML parsers have extended the DOM interfaces to include XPath querying options.

To fit into the version model, both the transformation and the path portions of the change set must be able to be serialized into string values suitable for including within

XML attributes. While the choice of syntax for this is somewhat arbitrary, the Extended Backus-Naur Form (EBNF) is a well-used simple notation for capturing data semantics. Appendix B provides EBNF for describing those methods shown in Figure 3-1, while (Clark and DeRose, 1999) provides EBNF for describing XPath statements. The Xpath statements and method operations are bound to elements 1.1.1.1 (Path) and 1.1.1.2 (Transformation) of the version model respectively.

3.2.1.2 Example

Consider the two primitive learning objects shown in Code Listing 3-1 and Code Listing 3-2 which show an HTML based learning object written in well-formed XML.

Code Listing 3-1: An example HTML document in well-formed XML

```
[01] <?xml version="1.0" encoding="UTF-8"?>
[02] <html>
[03]   <body>
[04]   <p>The quick brown fox.</p>
[05]   </body>
[06] </html>
```

Code Listing 3-2: A possible derivative version of the document described in Code Listing 3-1

```
[01] <?xml version="1.0" encoding="UTF-8"?>
[02] <html>
[03]   <head>
[04]     <title>All about foxes</title>
[05]   </head>
[06]   <body>
[07]     <p align="center">The quick brown fox.</p>
[08]   </body>
[09] </html>
```

One could imagine the set of DOM interactions that would need to take place to convert the original object into the revised form. First, a couple of new elements would have to be created to represent the <head> and <title> tag blocks. Next, a new text node would have to be created as a child of the title tag, and have its CDATA section set to "All about foxes". Finally, a new paragraph tag would need to be introduced as a child of the body tag, and have an attribute "align" added. The next section describes a vocabulary for capturing these kinds of invocations.

Code Listing 3-3 provides an example of how these changes could be bound to the data model described in section 3.1. This listing begins by defining a unique identifier for the object as a whole on line 02, and two change fragments on lines 03 and 12 respectively. The first change fragment creates the initial document given in Code Listing 3-1, and populates it with the elements and attributes corresponding to the first HTML document. This fragment is given a unique VID on line 03, as well as a short log file intended for human consumption on line 10. The second change fragment contains a number of statements which select nodes from the already instantiated XML document. These statements then modify this document using the change set vocabulary described in Appendix B. The result is an in-memory copy of the document described in Code Listing 3-2.

Code Listing 3-3: Example structural change set

```
[01] <history xmlns="http://cs.usask.ca/~cab938/love/xml/11.01.2004"
[02]          identifier="http://cs.usask.ca/~cab938/love/examples/2">
[03]   <change vid="urn:love:vid:049dd7a0-4c78-11d9-9669-0800200c9a66">
[04]     <structural path="" transformation="createDocument" />
[05]     <structural path="/" transformation="createElement;0;html" />
[06]     <structural path="/html" transformation="createElement;0;body" />
[07]     <structural path="/html/body" transformation="createElement;0;p" />
[08]     <structural path="/html/body/p"
[09]                 transformation="createTextNode;0;The quick brown fox." />
[10]     <log>Created a file about the fox.</log>
[11]   </change>
[12]   <change vid="urn:love:vid:75bbbfe0-4c7a-11d9-9669-0800200c9a66">
[13]     <structural path="/html" transformation="createElement;0;Head" />
[14]     <structural path="/html/head"
[15]                 transformation="createElement;0;Title" />
[16]     <structural path="/html/head/title"
[17]                 transformation="createTextNode;0;All about foxes" />
[18]     <structural path="/html/body/p"
[19]                 transformation="createAttribute;align" />
[20]     <structural path="/html/body/p/@align"
[21]                 transformation="createTextNode;;center" />
[22]     <log>Added a title.</log>
[23]   </change>
[24] </history>
```

3.2.2 Sequenced Learning Objects

The XML DOM binding created for primitive learning objects is useful for any learning object which follows an XML specification. For instance, the IMS Simple Sequencing specification provides explicit bindings to the XML data model. However, this specification also provides its own well defined data model. By expressing the version

changes in terms of the IMSSS data model, the semantics of the changes are preserved. This decouples the version history of the learning object from the underlying syntax that the object is stored in. This decoupling provides the ability to maintain a version history even when the underlying storage format changes.

An instance of this sequencing data model can be represented as a tree of activity nodes. Each node represents a single learning activity, and child nodes represent aggregate learning activities. All activities, whether leaf nodes or not, can reference content (usually as IMS Content Package Item elements) directly. Finally, each activity node is associated with two sets of rules. The first set of rules, called *sequencing rules*, is further broken down into three groups; preconditions, exit actions, and post conditions. These rules are evaluated by the learning management system before, while, and after the activity has been delivered respectively. These rules determine the flow of control through the activities child activities (or learning objects). The second set of rules, called *rollup rules*, is used to set values in learning management system tracking model for a given activity. The values are then read by the system when evaluating the sequencing rules of parent activities.

It is important to note that this specification is wide ranging in scope, and also includes models for tracking information, flow of control, end-user navigation requests, and delivery. While versioned copies of this information may be useful (e.g. to rebuild the state of a delivered activity at any moment in time, or the "walk-through" the specific actions of a learner as they progressed through an activity), they are considered outside of the scope of the authoring process, and will not be addressed in this work.

A vocabulary for describing changes to a sequenced learning object can then be thought of as an API for modifying a tree. Similar to the primitive learning objects, a full API may include navigation and retrieval functions, but only those functions which modify the structure or values in the tree need be captured for version control. Methods available on the tree structure itself control the layout of activities and aggregate activities, while methods available on the nodes control modifications to rule behaviors.

Code Listing 3-4: EBNF describing activity tree operations

```
[01] <separator>          ::= ";"
[02] <path>               ::= <NONNEGATIVE INTEGER> | <path> <separator>
[03]                          <NONNEGATIVE INTEGER> | ""
[04] <vid>                ::= <NONNEGATIVE INTEGER>
[05] <title>              ::= <STRING>
[06] <child position>     ::= <NONNEGATIVE INTEGER>
[07] <new activity>       ::= "NewActivity" <separator> <title> <separator>
[08]                          [<child position>]
[09] <delete activity>    ::= "DeleteActivity" <separator>
[10] <insert object>      ::= "InsertLearningObject" <separator> <url>
[11]                          <separator> <vid> <separator> <child position>
[12] <delete object>      ::= "DeleteLearningObject" <separator>
```

Out of the statements described, only a few of them are bound to nodes within the change set XML document. In particular, the <path> statements can be bound to element 1.1.1.1, while the <insert activity>, <delete activity>, <insert object> and <delete object> statements can be bounds to element 1.1.1.2.

Consider the XML document fragment in Code Listing 3-5 which outlines a portion of a version control record for an activity tree.

Code Listing 3-5: Version control record for activity tree

```
[01] <history xmlns="http://cs.usask.ca/~cab938/love/xml/11.01.2004"
[02]          identifier="http://cs.usask.ca/~cab938/love/examples/1>
[03]   <change>
[04]     <structural path="" transformation="NewActivity;General;" />
[05]     <structural path="0" transformation="NewActivity;Navigating 100;0" />
[06]     <structural path="0" transformation="NewActivity;Navigating
[07]      Windows;1" />
[08]     <structural path="0" transformation="NewActivity;Using the Web;2" />
[09]     <log>Created initial activities.</log>
[10]   </change>
[11]   <change>
[12]     <structural path="0,0" transformation="InsertLearningObject;
[13]       file:///c:/module1/imsmanifest.xml;0;0" />
[14]     <structural path="0,0" transformation="InsertLearningObject;
[15]       file:///c:/module2/imsmanifest.xml;34;1" />
[16]      <structural path="0,1" transformation="InsertLearningObject;
[17]       file:///c:/module3/imsmanifest.xml;21;0" />
[18]     <log>Added three learning objects</log>
[19]   </change>
[20]   <change>
[21]     <structural path="0,1" transformation="DeleteActivity;" />
[22]     <log>Removed Navigating 100 from activity list</log>
[23]   </change>
[24] </history>
```

This change history describes three separate versions of a learning object sequence. The first version, identified by lines [03] through [10], creates a new activity tree (line [04]),

then adds two child nodes (line [05] and [06]). The second version modifies this activity tree by adding references to primitive learning objects (lines [12] through [17]). Finally, the third version removes one of the activities in the tree, which has the effect of removing all activities and primitive objects associated with that activity. Figure 3-2 shows the sequencing document progression over time.

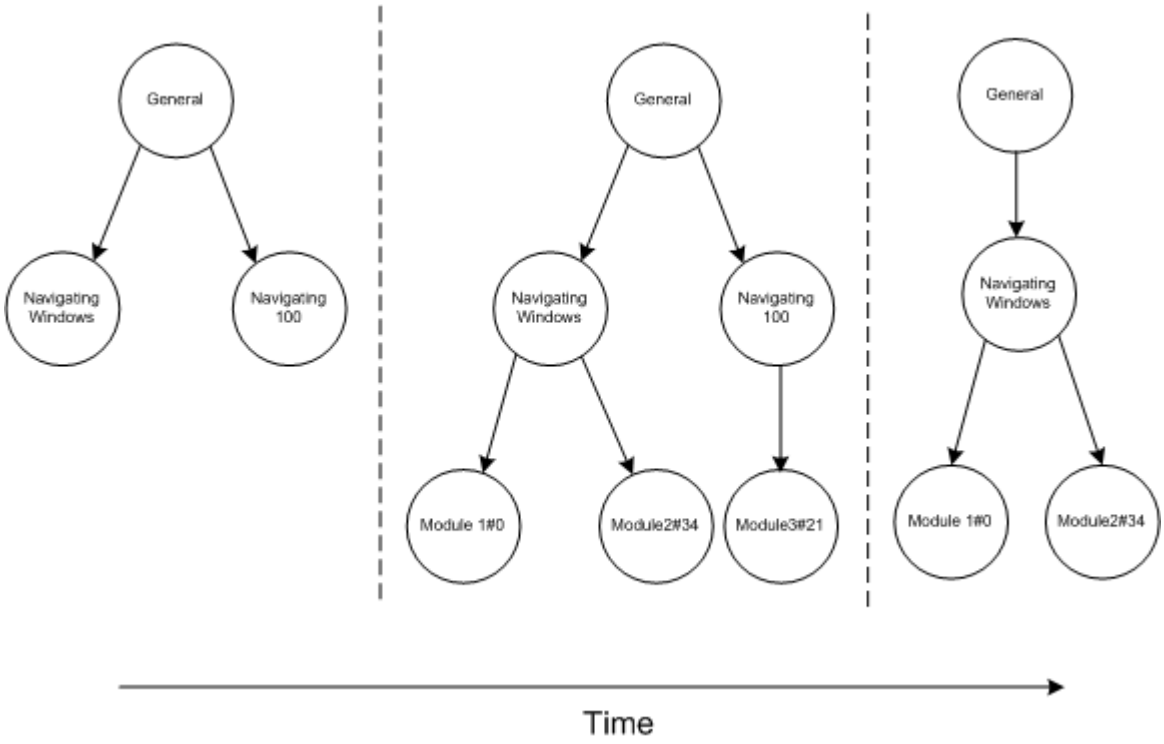

Figure 3-2: History of an example IMSSS activity tree

Sequencing rules in simple sequencing documents are attached to activity nodes and are of the general form:

If [conditions] then [action]

where both conditions and action take their form from the IMS Simple Sequencing Sequencing Rule Description vocabulary (IMS Global Learning Consortium Inc., 2003). Code Listing 3-6 provides a comprehensive change set definition for sequencing rules using the vocabulary defined by the IMS. Similar to the change set definition for activity trees, the <path> element is bound to element 1.1.1.1 of the abstract data model, while the <insert rule>, <delete rule>, and <modify rule> definitions are bound to ele-

ment 1.1.1.2 of the data model. It is important to indicate that designs for this vocabulary were limited by the sequencing specification, which indicates that rules are in an unordered set and do not have unique identifiers. Thus to modify or delete a rule an offset can not be used (as was done in the activity tree example). Instead, the full content of the rule must be evaluated.

Code Listing 3-6: EBNF describing sequencing rule changes

```
[01] <separator>          ::= ";"
[02] <path>               ::= <NONNEGATIVE INTEGER> | <path> <separator>
[03]                          <NONNEGATIVEINTEGER> | ""
[04] <condition>          ::= "All" | "Any"
[05] <rule conditions>    ::= <rule condition> <separator> <operator>
[06]                          <separator> <objective> | <rule conditions>
[07]                          <separator> <rule condition> <separator>
[08]                          <operator> <separator> <objective>
[09] <objective>          ::= <STRING>
[10] <operator>           ::= "Not" | "NO-OP"
[11] <rule condition>     ::= "Satisfied" | "Objective Status Known" |
[12]                          "Objective Measure Known" | "Objective Measure
[13]                          Greater Than" | "Objective Measure Less Than" |
[14]                          "Completed" | "Activity Progress Known" |
[15]                          "Attempted" | "Attempt Limit Exceeded" | "Time
[16]                          Limit Exceeded" | "Outside Available Time Range"
[17]                          | "Always"
[18] <action>             ::= <pre action> | <post action> | <exit action>
[19] <pre action>         ::= "Skip" | "Disabled" | "Hidden from Choice" |
[20]                          "Stop Forward Traversal" | "Ignore"
[21] <post action>        ::= "Exit Parent" | "Exit All" | "Retry" | "Retry
[22]                          All" | "Continue" | "Previous" | "Ignore"
[23] <exit action>        ::= "Exit" | "Ignore"
[24] <seqrule>            ::= <condition> <separator> <rule conditions>
[25]                          <separator> <action>
[26] <insert rule>        ::= "InsertRule" <separator> <seqrule>
[27] <delete rule>        ::= "DeleteRule" <separator> <seqrule>
[28] <modify rule>        ::= "ModifyRule" <separator> <seqrule> <separator>
[29]                          <seqrule>
```

Rollup rules are similar in form to sequencing rules, and are also attached to activity nodes in the activity tree. While sequencing rules are used by the learning management system to inspect values in the tracking and objectives models and order child activities as appropriate, rollup rules are used to set values in the tracking and objective modules for a given activity. Consider again the example given in Figure 3-2. When the learning management system goes to evaluate the sequencing rules for the "General" activity, it inspects the tracking and objective models to determine appropriate values. If the values for either "Navigation Windows" or "Navigating 100" can not be found, then the rollup rules for those activities are executed.

The general form of a rollup rule is:

If [child-activity] [conditions] then [action]

The child-activity is optional and denotes that the rollup rule for this activity requires that child rollup-rules be evaluated first. This allows for the chaining of rollup rules in a hierarchical up manner. Child activity rules are not referenced directly, and instead are referenced using a vocabulary (e.g. "All", "Any", "At Least Count", etc.). Both conditions and actions are Boolean values, and an optional negation operator ("Not") exists for conditions. Code Listing 3-7 provides a formal change set vocabulary for rollup rules.

Code Listing 3-7: EBNF describing rollup rule changes

```
[01] <separator>           ::= ";"
[02] <path>                ::= <NONNEGATIVE INTEGER> | <path> <separator>
[03]                           <NONNEGATIVE INTEGER> | ""
[04] <rollrule>            ::= <activities> <separator> <rule conditions>
[05]                           <separator> <action>
[06] <insert rule>         ::= "InsertRule" <separator> <rollrule>
[07] <delete rule>         ::= "DeleteRule" <separator> <rollrule>
[08] <modify rule>         ::= "ModifyRule" <separator> <rollrule> <separator>
[09]                           <rollrule>
[10] <activity_type>       ::= "All" | "Any" | "None" | "At Least Count"
[11]                           <separator> <INT> | "At Least Percent"
[12]                           <separator> <INT>
[13] <combination_option>  ::= "All" | "Any"
[14] <activity>            ::= <combination_option> <separator> <actitivy_type>
[15]
[16]
[17] <rule conditions>     ::= <rule condition> <separator> <operator>
[18]                           <separator> <objective> | <rule conditions>
[19]                           <separator> <rule condition> <separator>
[20]                           <operator> <separator> <objective>
[21] <objective>           ::= <STRING>
[22] <operator>            ::= "Not" | "NO-OP"
[23] <rule condition>      ::= "Satisfied" | "Objective Status Known" |
[24]                           "Objective Measure Known" |
[25]                           "Completed" | "Activity Progress Known" |
[26]                           "Attempted" | "Attempt Limit Exceeded" | "Time
[27]                           Limit Exceeded" | "Outside Available Time Range"
[28]                           | "Never"
[29] <action>              ::= "Satisfied" | "Not Satisfied" | "Completed" |
[30]                           "Incomplete"
```

3.3 Vocabularies for Expressing Learning Object Semantics

The need to determine how closely related two learning objects are to one another is constantly an issue. It exists for instructional designers searching for materials, for

learners looking to supplement their learning with more information, as well as for educational systems which may attempt to automatically adapt course content by introducing new learning objects. Metadata schemes such as the IEEE LOM provide a semantic description of a learning object that can be represented using key/value tuples. This description can then be interrogated to determine how the learning object has changed. This requires that appropriate data types be set in the metadata schema, and that the authoring tools, repository interfaces, and adaptive systems understand them. An alternative this is to include the author of the learning object in the process, and acquire both the new value of the metadata key as well as the way in which is has been changed from them.

Capturing the new value of an entry, and the key that is associated with it, is a fairly straight forward process. However, capturing the manner in which it changes requires a state change vocabulary. The intrinsic datatyping of metadata keys suggests that different vocabularies are needed to express the relationship between various keys.

Consider two learning objects, *lo1*, and *lo2* where *lo2* is a derivative of *lo1*. Given a metadata key (for instance, the *typical age range*), the result of a change to this key can fall into a number of different states:

- *lo2* and *lo1* are functionally equivalent: The meaning of the key is unchanged between the two versions. In the case of typical age range, this would require that the value for lo1 and the value for lo2 are identical.
- *lo2* is a subset of *lo1*. The meaning of the key for *lo1* would then be a superset of the key for *lo2*. For instance if the age range in *lo2* is 7-8 and the typical age range in *lo1* is 6-9 then *lo2* is a subset of *lo1* with respect to age range.
- *lo2* is a superset of *lo1*. The meaning of the key for *lo1* is a subset of that for *lo2*. For instance, if the age range in *lo2* is 7-8 and the age range of *lo1* is 7 then *lo2* is a superset of *lo1*.
- There exists some non null intersection between *lo1* and *lo2*. This indicates that the ranges for *lo1* and *lo2* contain some common elements, but are not identical. For instance, if *lo1* had an age range of 7-9 while *lo2* had an age range of 8-10.

Set theory based change operations are only one of many different kinds of possible changes available for metadata keys. For instance, consider the same objects, *lo1* and a derivative *lo2*, with the metadata key s*emantic density*. Semantic density is an ordered enumeration ranging from "low" to "high", denoting the degree of conciseness of a learning object. A change in semantic density could then describe learning objects in the following states:

- *lo2* is greater than *lo1*. With respect to semantic density this would indicate that *lo1* is a more simplified description of *lo2*.
- *lo2* is less than *lo1*. This would indicate that *lo2* holds a value earlier within the enumeration.

In addition to these two kinds of changes, there are also two primitive vocabulary values that are suitable for all metadata keys. These values can be used to indicate that either an entry has been newly added, or that the resulting change modified the entry but in an unknown manner. A formal description of these three vocabularies is provided in Code Listing 3-8.

Code Listing 3-8: EBNF vocabulary for capturing metadata semantic changes

```
[01] <separator>          ::= ";"
[02] <transformation>     ::= <primitive> | <set> | <enum> | <extension>
[03] <primitive>          ::= "general:" ("Addition" | "Unknown")
[04] <set>                ::= "set:" ("Superset" | "Subset" | "Equivalent" |
[05]                          "Intersection")
[06] <enum gt>            ::= "GreaterThan" <separator> <INTEGER>
[07] <enum lt>            ::= "LessThan" <separator> <INTEGER>
[08] <enum>               ::= "enum:" (<enum gt> | <enum lt> | "Equivalent")
[09] <extension>          ::= <STRING> ":" <STRING>
```

Each new state is preceded by a namespace. The namespace identifies the vocabulary which identifies the possible options for a given metadata key. For instance, the set transformation states "Superset", "Subset", "Equivalent", and "Intersection" are within the namespace "set". Implementations may extend this vocabulary by specifying a unique namespace value, and following it with a new change operation value.

State changes are stored only in a forward delta format – only the derivative learning objects hold the nature of metadata changes. Bindings from the vocabulary to the model

are straight forward; the transformation state vocabulary entry is bound to the transformation element (entity 1.1.2.2), while the metadata key and value strings (entities 1.1.2.3 and 1.1.2.1 respectfully) are determined by the metadata schema chosen. For example, the version control record fragment depicted in Code Listing 3-9 demonstrates changes in the semantic density metadata entry using the ordered enumeration vocabulary.

Code Listing 3-9: Example version control record for semantic changes

```
[01] <history xmlns="http://cs.usask.ca/~cab938/love/xml/11.01.2004"
[02]          identifier="http://cs.usask.ca/~cab938/love/examples/1>
[03]   <change>
[04]     <semantic value="very low" transformation="general:Addition"
[05]      perspective="semantic-density"/>
[06]     <log>Initial semantic density level set.</log>
[07]   </change>
[08]   <change>
[09]     <semantic value="low" transformation="enum:GreaterThan"
[10]      perspective="semantic-density"/>
[11]     <log>Increased the level of semantic density.</log>
[12]   </change>
[13] </history>
```

It is important to note that a learning object can change with respect to several different perspectives in several different ways at once. For instance, by changing a learning objects content to be more detailed and use more concise language, an author may both be changing the *semantic density* and the *typical age range* that object is appropriate for. This is supported by the metadata model in that any number of semantic changes can be captured and correlated with structural changes within a given change.

3.4 Conclusions

This chapter has provided a metadata model which defines a product space for the versioning of learning objects. This product space encompasses popular learning object specifications and languages, including the Extensible Markup Language, the IMS Simple Sequencing Specification, and any metadata language which can be reduced to a tuples list, such as the IEEE Learning Object Metadata standard, and the Dublin Core Educational specification.

This model builds upon other metadata specifications in a conforming fashion, in that it does not introduce new metadata elements to replace those which already exist (IEEE, Inc., 2002). Instead, it introduces new elements only to capture those versioning principles which are absent from other metadata specifications. Since the branching of learning objects is covered in both the Dublin Core and the IEE LOM, it recommended that branching be handled by use of the *relation* element, as described in section 2.2.3.

While it may be tempting to use other elements from the LOM lifecycle, such as the *version* element, we caution against this. The *version* element is meant to capture the edition of the current learning object. Examples within the LOM specification lead the reader to believe that using a major.minor numbering scheme is an appropriate way to do this. There are two problems with this – first, the major.minor numbering scheme is used within traditional software configuration management to indicate whether this artifact is a revision or a variant and what development branch this artifact was created on. This case is more appropriately handled by the *relation* element as it is easier for software components to understand. Secondly, the major.minor number scheme is only useful when there is centralized control over the artifacts being versioned. Much of the learning object repository literature suggests that learning objects are going to be used in decentralized environments (e.g. (Hatala and Richards, 2003)), with related versions sitting in different repositories. This makes it unreasonable to try and keep track of development branches with a simple incremental numbering scheme, as it would require centralized, or at least synchronized, control of all of the learning objects. Instead, following the approach suggested here couples the version history with the object being versioned. By setting the VID to a globally unique string, no centralized repository control is required.

In addition, the *annotation* element from the LOM is used to add human readable comments to learning object metadata to facilitate the discovery and evaluation of a learning object for a given problem. This element should not be used for versioning specific comments (e.g. change logs) as there is no way to link individual annotations to the syntactic and semantic changes that have occurred. Instead, the *annotation* element should only be used for comments that are pedagogical or functional in nature (for example,

those comments that outline the pedagogical uses of the learning object, or other technical requirements that cannot be captured through other metadata elements).

3.4.1 Benefits

Capturing versioning information for learning objects has a number of benefits for authors, learners, and software components trying to provide for adaptivity in e-learning. Firstly, the ability to regress, or rollback a learning object to a previous point in its history allows an author to retrieve older versions that may be more suited for their purpose than a new version. Secondly, by including semantic descriptions, enhanced searching can be accomplished either through direct request (e.g. "Find me a learning object that has a typical age range which is a superset of this one") or via visualization. Examples of both roll-backs and learning object visualization are provided in section 4.3 and section 4.4 respectively.

The approach suggested in this chapter describes a binding for learning object change histories into XML. The use of XML is rampant throughout the learning object community, with only a few fringe specifications using any other binding syntax. XML, however, tends to be a very verbose syntax. Since version histories also tend to be quite verbose, the choice was made to formalize change histories into a more compact format denoted by the EBNF vocabularies above. This format is appropriate for the data model as the EBNF presented uses only string values, and can be easily serialized into XML attribute values.

3.4.2 Limitations

The definition of metadata as a list of key/value pairs is reasonable for the current state of the art, but leaves little room for accommodating the next generation of learning object metadata. In particular, the efforts within the learning object community to adopt semantic web style RDF bindings for metadata make refinements to the approach suggested necessary. The RDF takes keys and values and turns them into more complicated statements made up of a subject, predicate, and an object clause. This allows for more flexible metametadata, as the object clause (which, in the tuple view of metadata intrin-

sically points to the learning object itself) and can refer to any entity – a learning object, digital resource, other metadata clauses, etc.

In addition, a number of researchers have been questioning the value of coupling metadata so tightly to a learning object. There may be more subjective metadata, such as a learner opinion, which could be held external to the object itself. How this data could be versioned appropriately is in question, as the actor doing the versioning is generally the content author and may not be the appropriate person to indicate what learner opinion data is still appropriate.

Finally, while section 3.1 suggested that it is more practical to get an author to enter metadata change values than it is to get all authoring tools to be fully schema aware, there are recent reports of authors as well as learning object tool developers being overwhelmed by the number of metadata tags already available (IMS Global Learning Consortium Inc., 2003).

Chapter 4
Implementation Prototype

This chapter describes an implementation prototype, the Learning Object Versioning Environment (LOVE), developed to refine portions of the version model described in Chapter 3. It is implemented as a native Windows application, and stores both primitive and sequenced learning objects in a simple file based learning object repository using the IMS Content Packaging format. As it was built during the development of the version model to help define the version model, only portions of the schemas and vocabularies described previously were used. Details of the simplifications between the schemas and the prototype are described in Appendix C.

This prototype is an integrated authoring environment where an instructional designer can build, sequence, and visualize learning materials. It includes primitive learning object creation facilities in the form of an integrated XML editor, and sequenced learning object creation facilities using a directory tree metaphor, where items in the directories are primitive learning objects, and the directories themselves are IMSSS activities. Both of these components also have the ability to modify metadata associated with an object using a built in table interface.

This chapter is organized as follows; section 4.1 lays the groundwork for the chapter by outlining an example scenario based on the anecdotal observation of the needs of university sessional lecturer. Next, section 4.2 outlines both the primitive and sequenced learning object editing facilities. This is followed by a descriptive example of how an author can use the environment to roll back a primitive learning object to an older version in section 4.3. Finally, the chapter concludes with a demonstration of the semantic

visualization features added to the metadata of both primitive and sequenced learning objects in section 4.4, and a list of limitations in section 4.5.

4.1 Example Scenario

A sessional lecturer is an employee hired for the expressed purpose of teaching a single session of a class. While these lecturers sometimes teach later versions of the class, they are often replaced with new sessional lecturers or faculty for future offerings of the course. Creating content in the form of assignments, exams, and lecture notes for a class results in the bulk of the work a sessional must do. It thus typically remains the policy of an academic department to maintain old copies of course content by passing them on to new lecturers. These lecturers can then adapt this content to their particular teaching needs.

Adaptation of content is generally done manually – the lecturer obtains the content from the department, opens it with their editor of choice, and makes changes as they see fit. If the lecturer desires to undo their changes and go back to a previous version of the material they are forced to compare their new content with that of the previous instructor. Changes in between (e.g. drafts of lecture notes) are generally not available. Further, the lecturer is generally limited to only the previous years content – content going back to the first offering of the course is rarely available. This is thus a problem of structural versioning.

A key requirement of the adaptation process is for the lecturer to be able to identify which content should be kept, and which should be revised. This is especially difficult when there is a large amount of content – for example, a class that is taught in three sections may have three different sets of lecture notes for a given topic. Sifting through all of these to determine the differences (and similarities) can be an enormous task. If the lecturer could easily identify the connections between these pieces of content (e.g. which came first, or how they differ with respect to a certain piece of metadata), they could more quickly modify and integrate the content into their course. This is thus a problem of semantic versioning.

The rest of this chapter will consider the case of a sessional lecturer preparing part of a course for a third year computer science databases class.

4.2 Learning Object Creation and Editing

The principle feature of any authoring environment is the creation and modification of content. Figure 4-1 is a screen shot of the general purpose XML and metadata editor built into LOVE. The editor displays learning objects using a custom built XML syntax highlighting component, which maps interactions with the text directly to the underlying DOM model. Further, the table below the content displays the metadata entries that exist for that learning object. This table is editable, and allows the author to introduce new metadata keys (using the "a" button), remove keys that already exist (using the "d" button), change the value of a key (using the "New Version" column), and record the way in which a key has changed semantically (using the "Change" column, and one of the vocabularies presented in section 3.3). Human readable logs can be entered in free form at the bottom of the editor.

The content editing component differs from ones that are typically used in that the context of available actions at any given point in time (e.g. can an element be added, or can a text value be changed) is determined by the location of the cursor in the content. This is done by enabling or disabling the buttons to add new elements ("ae"), delete elements ("de"), add attributes ("aa"), delete attributes ("da"), and either enabling or disabling live editing to allow the modification of character data (such as attribute values). This ensures that the XML created will always be well formed, and provides an easy mechanism for the editor to map changes to the underlying DOM.

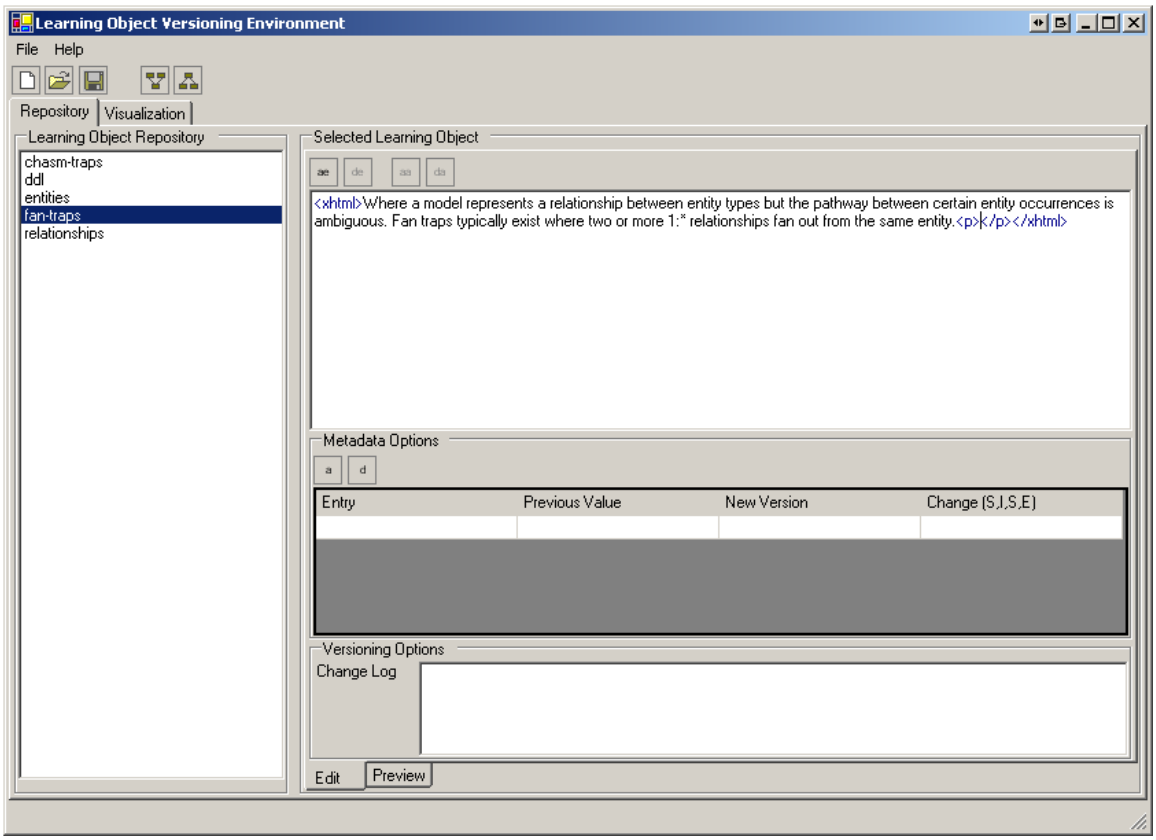

Figure 4-1: The LOVE primitive learning object editing facilities

Creating a new version of a learning object is completely transparent to the author. When the author saves the object or opens a different object for editing the changes are serialized into change sets as discussed, and stored in the IMS manifest file. This change set includes a new version identifier value, as well as mappings to the DOM structure. The prototype stores each keystroke as a separate structural change. While this allows for very fine grained version control, it also makes for large change sets. The merits of this approach and alternatives are discussed in section 5.3.2.

The learning object repository is depicted by the list of objects to the left of the editor. This repository is simply a collection of directories which hold IMS Content Package files with versioning information. In addition to primitive learning object editing facilities, LOVE provides a method of editing sequences of objects. Sequenced learning objects are stored in the same file based repository as primitive learning objects, but include IMS Simple Sequencing information in their manifest files. This sequencing in-

formation is rendered in the editor as a tree of folders, where each folder corresponds to a learning activity. Specific primitive learning objects are represented using a text file icon underneath an activity, and are named with their location, followed by an automatically generated VID. Metadata and log editing facilities are also provided, and are similar to those available for primitive learning objects. To simplify the interface, VIDs are generated using a simple incrementing integer scheme. This scheme would not be acceptable for wide scale deployment as it does not guarantee global uniqueness. Figure 4-2 shows a screenshot of the sequenced learning object editing facilities.

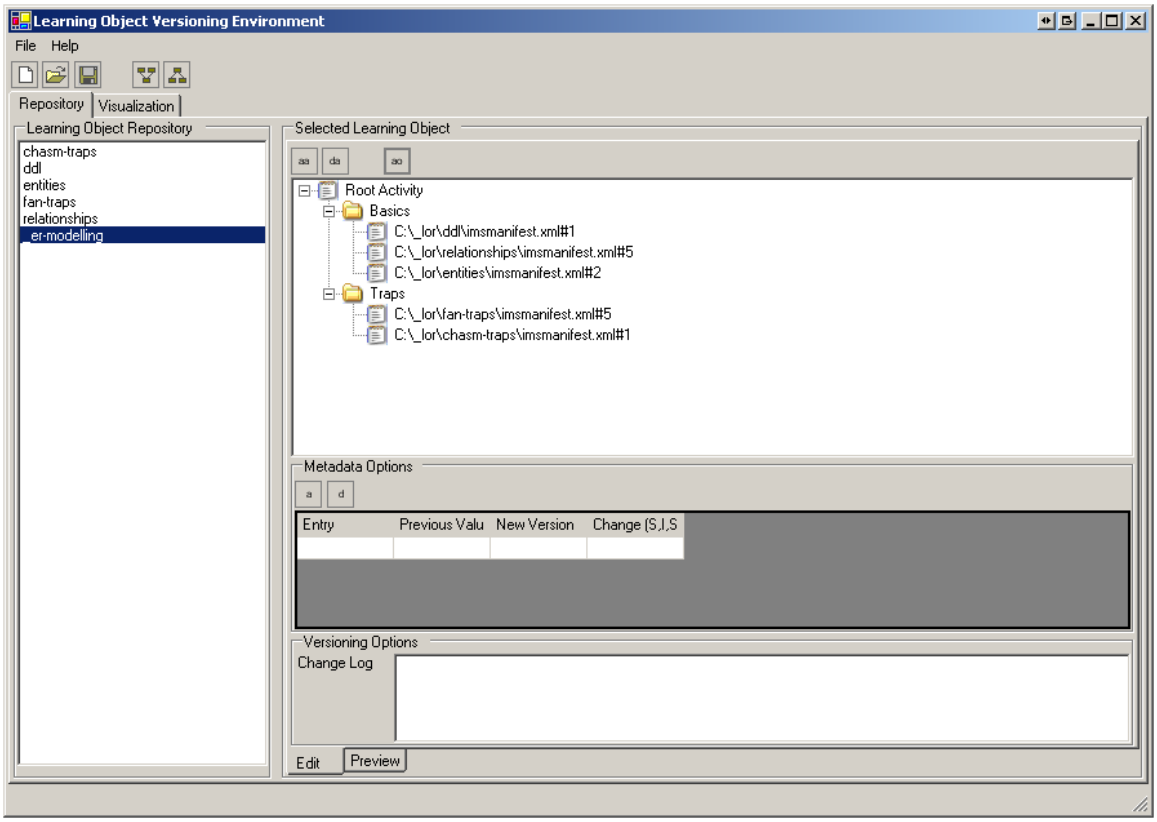

Figure 4-2: The LOVE sequenced learning object editing facilities

In the above sequencing example, and author has chosen to create a new course called "_er-modelling". This course is made up of two activities, "Basics" and "Traps". These activities contain a number of learning objects identified by their manifest location followed by a version offset. For instance, when the "Basics" activity is rendered to a learner it would first show version 1 of the "ddl" primitive object, followed by version 5 of the "relationships" object, and ending with version 2 of the "entities" object. These

objects would be displayed in accordance with any sequencing rules an author might add.

It should be noted that the rollup rules and sequencing rules as provided by the version model in section 3.2.2 are not implemented in this prototype, and that only activity tree modification is captured.

4.3 Supporting Roll-backs

One of the key features in supporting learning object versioning is to allow course designers to pick a particular version of a learning object when creating a course. The prototype implementation supports this when the author edits a sequence of learning objects. When adding a piece of content to a learning activity, the author is able to see all of the primitive learning objects in the repository, and choose a version which should be inserted using a drop down box. Changes in the value of the drop down box causes a particular object version to be rendered immediately in the lower right frame as a live preview. Figure 4-3 demonstrates this ability.

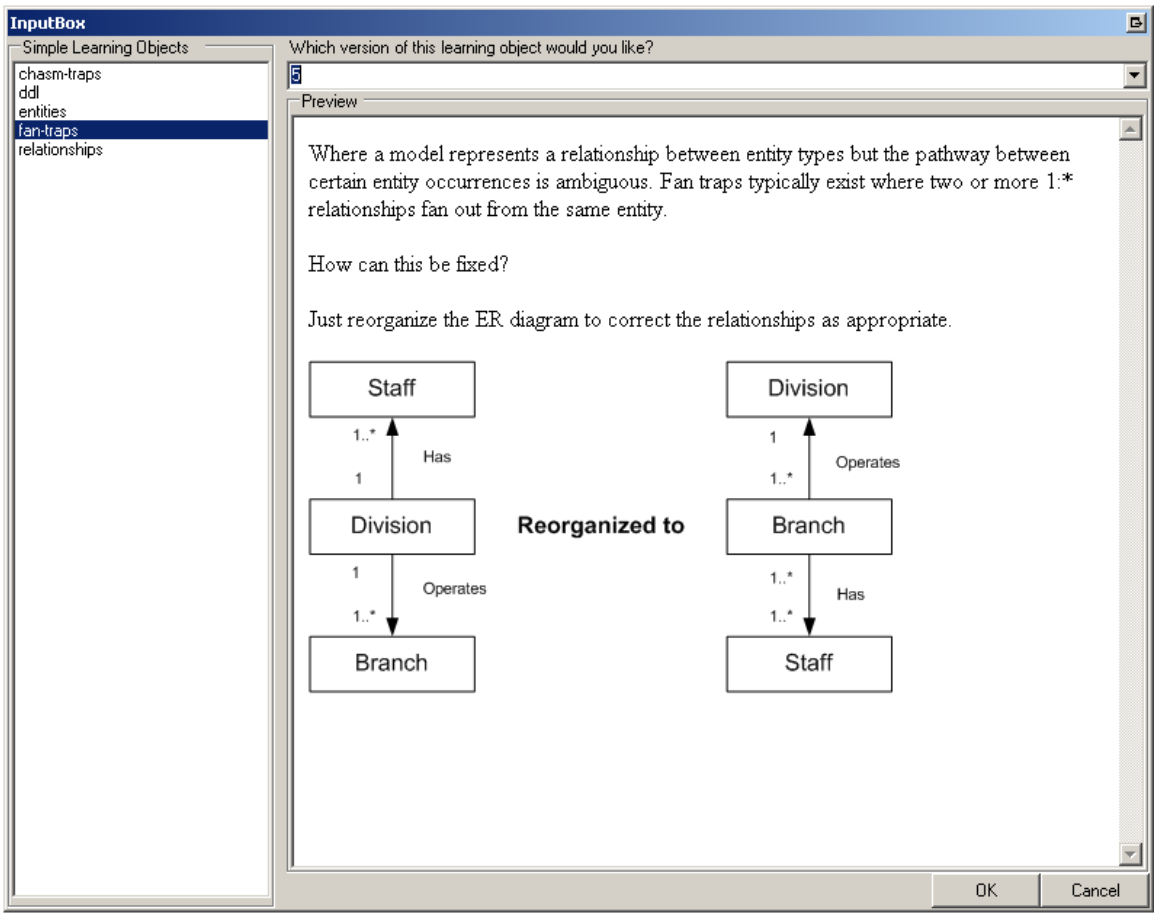

Figure 4-3: Selecting a primitive object to add to a course

In this instance, the author has decided to include the primitive object "fan-traps". Though the initial version of this object was shown in Figure 4-1, the author has since modified the object to include more details and a pictorial description of the problem and its solution. After determining this is the most suitable version, the author presses the "OK" button and the version is inserted into the course as per Figure 4-2.

The rendering shown in the right hand window is achieved by applying all of those structural changes in the manifest which exist between the beginning of the manifest and the particular VID the author has chosen.

4.4 Semantic Visualization

In addition to providing a live preview of what primitive learning objects look like in a web browser, the prototype also provides a visualization of the whole repository of

learning objects[3]. This visualization lays out both primitive and sequenced learning objects in a horizontal revision graph similar to that shown in Figure 2-4. The visualization allows for author interaction with the revisions graph through the use of the "-" and "+" handles attached to the top left of each node. These handles either collapse or expand the corresponding version and allow the author to see a list of the metadata keys, their values, and the change set value for that version. In addition, dependencies between versions of a sequenced learning object and primitive objects are presented to the user using red arrows which can be expanded or collapsed using the "-" and "+" handles attached to the bottom left of each sequencing node. Figure 4-4 shows an example of this visualization.

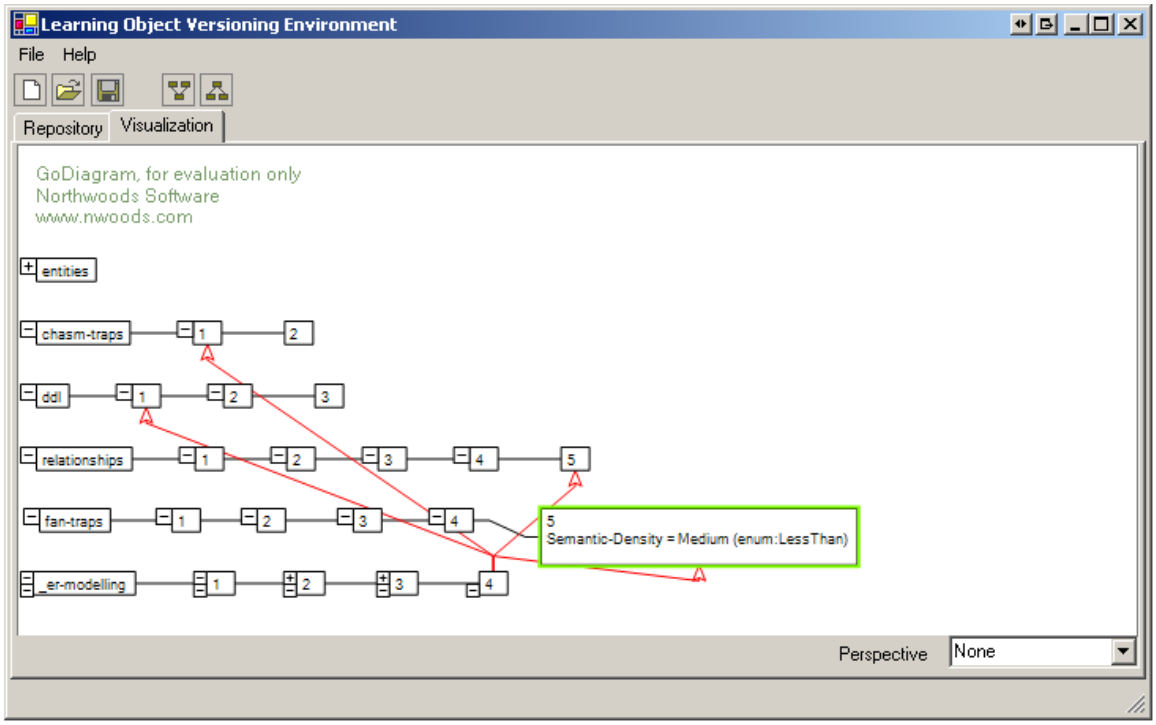

Figure 4-4: Learning object visualization in the LOVE

In this figure, the author has chosen to collapse the “entities” timeline to reduce screen clutter, while keeping the timelines for all of the other learning objects open. The author has expanded revision 5 of the fan-traps learning object to inspect the metadata (which contains only one new key). The extra handles to the bottom left of the nodes in the se-

[3] The graphing library used was an evaluation of GoDiagram, from Northwoods Software. More informa-

quenced learning object timeline allow the author to toggle dependencies on or off. In this example the author has chosen to see only those versions that the latest version of the "_er-modelling" object requires, as shown by the arrows emanating from that node.

In addition to visualizing the sequencing relationships between learning objects, the environment allows for visualizing semantic differences using the vocabularies defined in section 3.3. The author must pick a single metadata key by which to view the repository. Versions that do not have this key associated with them appear as white, while versions that are associated with this key are shown in shades of gray. Each vocabulary also has its own visualization characteristics. The ordered enumeration vocabulary is represented using shades of gray, where darker values indicate that a given version is greater than the previous version, and lighter values indicate the version is less than the previous version. Figure 4-5 shows a rendering of this using the metadata associated with the "fan-traps" learning object. In this example, only the fan-traps learning object versions three through five are associated with the metadata key "Semantic-Density". Version three has a value of "low", which increases to "medium" in version four, and finally to "high" in version five. By glancing at the diagram, the sessional can prune down the learning objects he or she wants to consider based on the depth of color for each node. Once this is done, the sessional lecturer can obtain more metadata for a particular version by double clicking on its node.

When visualizing the repository from the perspective of a metadata key that is annotated with the set vocabulary, a size metaphor is used. A version which is subset of a previous version is rendered as a smaller node in the revision graph, while versions that a supersets of a previous version are represented with larger node. Equivalent and intersection values are shown as similarly sized nodes.

Figure 4-6 shows an example of visualization using the set vocabulary and the "entities" timeline with the "Age-Range" metadata key, which implicitly represents a set of appropriate ages for a learning object. It is important to note that the size of nodes is rendered using only the previous version values, and not by inspecting all version values. Thus comparing the size of the first version and the third version provides no meaning-

---

tion on this library is available at the vendors' website at http://www.nwoods.com.

ful knowledge about the changes if the second version has also changed (as in this example).

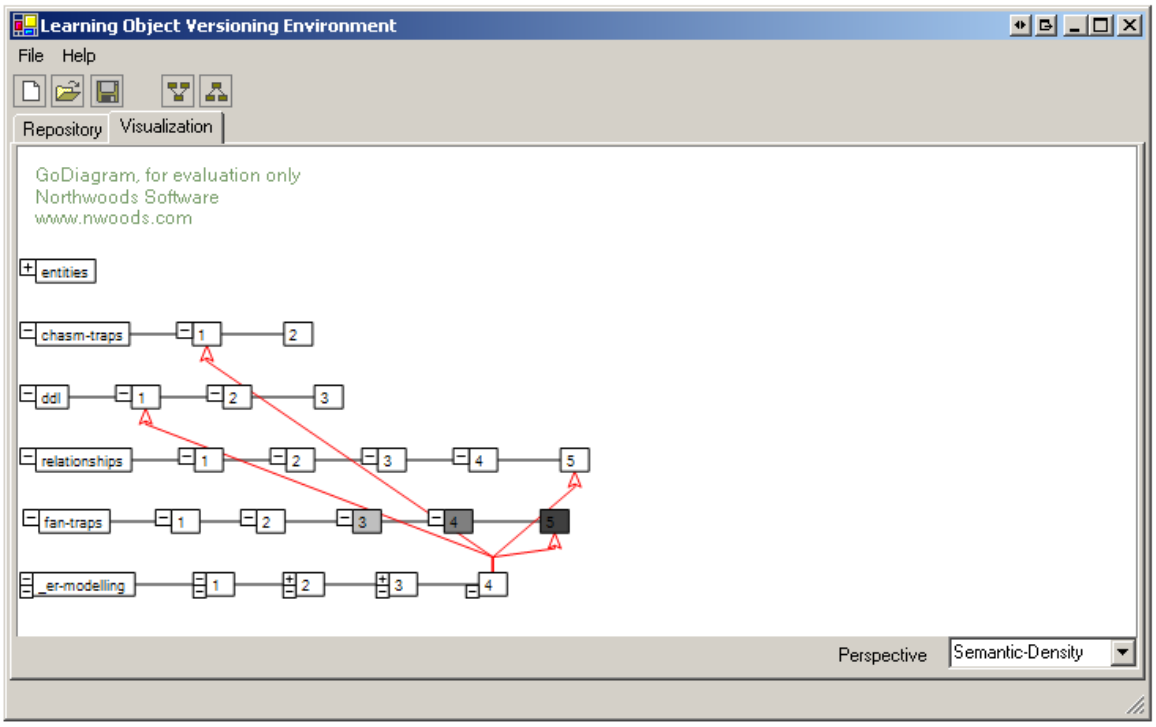

Figure 4-5: Enumeration visualization in LOVE

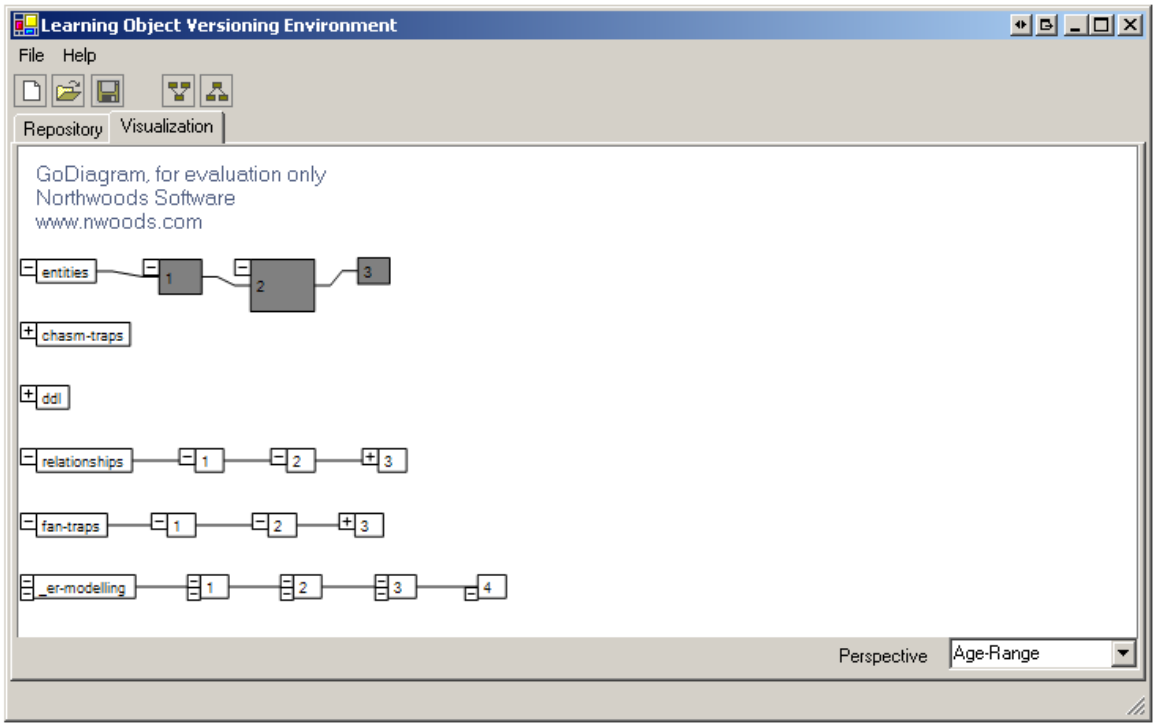

Figure 4-6: Set visualization in LOVE

4.5 Prototype Limitations

Typical versioning environments support the notion of the branching and merging of revision trees. A distributed environment is somewhat different in that the lack of centralized control makes these operations difficult. For instance, this prototype maintains centralized control by assuming that the content package holding the change sets is the authoritative source for that learning object. In this way, no branching or merge facilities are supported. If a more complex repository system that allowed for parallel revision trees to exist for each learning object (potentially by storing separate content packages with identical object identifiers), an author could then do branch based editing of a learning object. After modifying a particular branch, the author could then commit it to the repository and have it stored as another content package. The sameness principle described in section 3.1.3 makes it possible to construct a single visualization of such a repository using branch points where learning objects with the same OIDs differ in their version history. The traditional software configuration management notion of a trunk branch which represents the main development of a learning object makes little sense in this kind of environment. While some branches may be intended to be more stable or suitable than others for a given pedagogical purpose, it is anticipated that many branches will see active deployment to learning management systems, as they address the different needs of instructional designers.

Once branching of learning objects is available at a visualization level, authors are likely to want to merge different versions of an object together. While this feature doesn't exist in the prototype, this is directly due to the lack of branching support. Section 5.3.5 outlines some of the issues involved with implementing merging facilities.

Chapter 5
Conclusions

This chapter concludes the work by providing a summary of the approach taken and relating it to the hypothesis proposed in Chapter 1. General research contributions and future directions for this research are also discussed.

5.1 Summary

Current learning object authoring systems and repositories contain limited, if any, support for the evolutionary nature of a learning object. Despite the strong anecdotal evidence behind the need for such systems, there are no known learning object versioning systems described in the educational technology literature.

This thesis has addressed this issue in three ways. Firstly, it has identified the need for such systems and contextualized this need within the state of the art of learning object research. In particular, it has identified the goal of the educational technology community to focus on reuse. When learning objects become easily discoverable by authors, they become easy to change in order to adapt to localized needs. These changes can be further captured, visualized, and shared with the community as a whole. Further, the thesis outlined the nature of learning objects as entities that couple content to metadata, and exist in a distributed and decentralized fashion.

Secondly, this thesis has provided a model for capturing the changes associated with learning objects. This model is built on the abstract notions of a version model as presented by Conradi and Westfechtel in (Conradi and Westfechtel, 1998), and outlines a product space for learning objects. This product space is then formally described using an XML Schema. While generalized, the schema contains elements which can then be

bound to specific data formats for different kinds of learning objects. Two such data formats are introduced, building on the widely accepted XML DOM and the IMS Simple Sequencing specifications. In addition, the notion of semantic versioning is introduced and a change set vocabulary is defined for two kinds of metadata keys, those based on ordered enumerations and those based on set theory.

Finally, an implementation prototype is created and described. This prototype is an integrated authoring environment for learning objects, and includes a context sensitive syntax highlighting editing control for both primitive and sequence learning objects, as well as a repository visualization tool for semantic versioning. This prototype provides examples of two versioning functions; rollbacks and historical visualizations.

5.2 Research Contributions

The primary contribution of this work is the description of a version model for learning objects. The majority of the research and development in the area of learning object technologies has been centered on repositories for containing and describing learning objects. These repositories have one primary goal – facilitating the reuse of learning material. With the discovery of learning objects comes the need to modify the object to fit a local context. Supporting the management of the many derivations of a learning object is a natural next step.

Secondary contributions of this work include the development of versioning vocabularies, in particular the change set vocabulary for XML documents. By directly mapping interactions with the DOM to a serializable state, the management of XML document versioning is consistent as well as straightforward to implement. As there are DOM based XML parsers for virtually every language and platform, this technique would be easy to replicate in many different environments. This change set vocabulary could be easily reused to facilitate general XML versioning.

Lastly, the production of an environment for the development and visualization of learning objects can serve as an interesting test bed for aiding in the authoring and discovery process. Most repositories and authoring tools limit an instructional designer to string matching on search queries to find learning objects. This ignores the many interconnec-

tions that exist between learning objects, both structurally and semantically. It is hoped that the techniques described here will encourage repository authors to consider a broader range of visualization techniques, regardless of whether version control is implemented.

5.3 Future Directions

5.3.1 Empirical Validation

One of the struggles with any work in the field of learning technologies is the long incubation time required for real-world user studies. Especially when modifying an already existing process, users need to feel suitably comfortable with the technology and the new workflows before they can provide good feedback. This requires that highly refined user interfaces be developed, in order to stop the new workflow (as opposed to the underlying method by which the workflow is carried out) from being rejected.

This work introduces one such workflow. Testing was not done with actual learning object authors as the principal goal of this work is to explore computational methods for supporting versioning in general. There is a need, however, for the testing of a versioning platform aimed specifically at authors to determine additional needs they may have. A number of important questions include:

- Does capturing semantic change sets increase the workload of an author to an unreasonable level?
- Does the visualization of versioning information change how often authors updates their courses?
- Does coupling the version history with the learning object itself result in scalability problems?

5.3.2 Delta Compression

The sameness principle for the version model indicates that it is the result of applying changes that makes a learning object unique, and not the specific changes that occur. This is interesting in that it provides the potential for change sets to be compressed without changing the meaning of the version. For instance, the design of the editor in the

prototype actually mapped each keystroke when an attribute value was selected to the insertion of another character in the DOM model. Thus a five character attribute value was serialized as five separate XML elements, each with their own XPath identifier. This kind of delta could be compressed by converting the five statements into a single multi-character manipulation on the DOM, without compromising the integrity of the revision history.

5.3.3 Branch Optimization

Capturing the interactions with a learning object can be done at both fine and coarse grain levels. The prototype described in Chapter 4 used extremely fine grain versioning, where individual keystrokes were invoked directly on the DOM model and stored as changes in a change set. While this allows for some comprehensive rollback features (the document could be regressed character by character), it also increases the size of the history file. If forward deltas are used, the amount of processing required to attain the newest version requires applying all elements in the history file.

The model could be expanded to include the notion of lossy tag points. A tag is typically used in version control systems to denote a particular version of interest. Lossy tag points instead indicate the collapse of the version history previous to that point into a new artifact. This significantly reduces the size of the version history, because it effectively removes all notion of versioning from before the tag point. This has the effect of speeding up regressions if forward deltas are being used, as well as minimizing the size of the history stored.

5.3.4 Visualizations

It is the firm belief of the author that visualizations of content interconnections, both within a learning object repository as well as within a course delivery tool, will greatly aid in the usability and quality of learning content produced. Nonetheless, the production of these visualizations has been largely ignored by the educational technology

community.[4] With versioning information comes even more visualization data, and integrating this in a non-overwhelming fashion is important.

The implementation prototype, for instance, provides a simple method for reducing the depth of a revision tree by collapsing nodes. Is this appropriate for the kinds of revision graphs that will exist? Does the visualization need to be more interactive? Will it scale appropriately? Is it showing the information that course authors want to see? The next step in answering these questions is to test the prototype with a group of instructional designers.

5.3.5 Intelligent Merging Facilities

A principle function of any software configuration management system is the ability to merge versions of artifacts into a new artifact. This process has always been error prone, and tends to require interaction with the content expert. The advent of more strictly structured data (e.g. XML with schemas) may provide for more precise merging routines. These routines, while non-trivial to develop, can interrogate schemas of information to determine how to greater merge content into a consistent state. The more semantics that can expressed in machine form, the more likely it is that the author need not be involved in the merging process. While there is only limited support for capturing the semantics of an XML document (e.g. XML Schemas), one could imagine schema aware repositories that could provide even deeper levels of reasoning when merging versions.

5.4 Conclusions

Reusability is cited heavily as a motivating factor for packaging educational content in the form of learning objects. While the tools and techniques for creating reusable learning objects are in their infancy, they are maturing quickly. As instructional designers begin to rely more heavily on learning object repositories for original content, the nature management of derivative works becomes an issue. This research has begun the explo-

[4] In the particular area of learning object repositories, the author knows only of some initial work that is being done on 3-D virtual environments for repositories of multimedia objects through the DISCOVER laboratory at the University of Ottawa. See http://www.discover.uottawa.ca/research/LORNET.html for more information.

ration of how learning object versioning can be supported at both a structural level, and a semantic level.

References

Barritt, C. and Lewis, D. (2002). Reusable Learning Object Strategy; http://www.cisco.com/warp/public/10/wwtraining/elearning/implement/rlo_strategy_v3-1.pdf.

Bendix, L., Dattolo, A., and Vitali, F. (2001). *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing, pp. 523-548.

Berliner, B. (1990). CVS II: Parallelizing Software Development. *USENIX Winter 1990 Technical Conference*, USENIX Association, 341-352.

Cederqvist, P. Version Management with CVS; http://ftp.cvshome.org/cvs-1.11.3/cederqvist-1.11.3.pdf.

Clark, J. & DeRose, S. (1999). XML Path Language (XPATH), Version 1.0, World Wide Web Consortium (W3C) recommendation, http://www.w3.org/TR/xpath

Collins-Sussman, B., Fitzpatrick, B.W., and Pilato, C.M. subversion.tigris.org; http://subversion.tigris.org/.

Conradi, R. and Westfechtel, B. (1998). Version Models for Software Configuration Management. *ACM Computing Surveys*, vol. 30, no. 2, 232-282.

Cooke, J. et al. Computer Science Tutorial Page; http://www.cs.usask.ca/resources/tutorials/csconcepts/index.html.

DCMI Usage Board DCMI Elements and Element Refinements - a current list; http://dublincore.org/usage/terms/dc/current-elements/.

Dennis, S., Uijtdehaage, S., and Candler, C. Introducing the Health Education Assets Library: a National Multimedia Repository; http://www.healcentral.org/index.jsp.

Department of Computer Science, University of Saskatchewan I-Help Learning Content Management System; http://www.cs.usask.ca/online.

Downes, S. Smart Learning Objects; http://education.qld.gov.au/staff/learning/courses/sdownesapril.html.

Downes, S. (2002). The Learning Object Economy; www.downes.ca/files/Learning_Object_Economy.doc.

Dublin Core Metadata Initiative Dublin Core Metadata Initiative (DCMI); http://dublincore.org/.

Duval, E. et al. (2002). Metadata Principles and Practicalities. *D-Lib Magazine*, vol. 8, no. 4

EOE Foundation Educational Object Economy; http://www.eoe.org/.

Ferraiolo, J., Fujisawa, J., and Jackson, D. (2003). Appendix B: SVG Document Object Model (DOM), Scalable Vector Graphics (SVG) 1.1 Specification, World Wide Web Consortium (W3C) recommendation, http://www.w3.org/TR/SVG/svgdom.html

CanCore Initiative (2002). CanCore Learning Object Metadata: Metadata Guidelines, Version 1.1

Friesen, N. (2001). What are Educational Objects? *Interactive Learning Environments*, vol. 9, no. 3.

Friesen, N., Mason, J., and Ward, N. (2002). Building Educational Metadata Application Profiles. *International Conference on Dublin Core and Metadata for e-Communities 2002* (DC-2002), Firenze University Press, 63-69.

Gateway to Educational Materials Consortium GEM 2.0 Elements and Semantics; http://www.geminfo.org/Workbench/GEM2_elements.html.

Gerard, R. (1969). *Computer-assisted instruction: a book of readings*. Academic Press, Inc., pp. 15-41.

Gibbons, A. S., Nelson, J., and Richards, R. (2001). The Nature and Origin of Instructional Objects. In D. A. Wiley *The Instructional Use of Learning Objects*. Association for Educational Communications and Technology

Government of Ireland The Irish Public Service Metadata Standard; http://www.gov.ie/webstandards/metastandards/manual.pdf.

Grant, J., Beckett, D., and McBride, B. (2002). RDF Test Cases: N-Triples, World Wide Web Consortium (W3C) working draf, http://www.w3.org/TR/rdf-testcases/#ntriples

Hatala, M. and Richards, G. (2003). Making a SPLASH: A Heteregeneous Peer-to-Peer Learning Object Repository . *The Twelfth International World Wide Web Conference* (WWW 2003), Association of Computing Machinery (ACM).

Hatala, M. et al. (2004). The Interoperability of Learning Object Repositories and Services: Standards, Implementations and Lessons Learned. *The Thirteenth International World Wide Web Conference* (WWW2004), 154-161.

Hodgins, W. Learning Architecture, Learning Objects: Learning API Task Force; http://www.cedma.org/guestlalo.asp.

International Organization for Standardization (ISO) (1986). *ISO 8879:1986,* Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML)

IMS Global Learning Consortium Inc. (2003). *Version 1.0 Final Specification,* IMS Simple Sequencing Information and Behavior Model, Version 1.2

IMS Global Learning Consortium Inc. (2003). IMS Learning Resource Meta-data Information Model, Version 1.2

IMS Global Learning Consortium Inc. (2003). IMS Learning Resource Meta-data Best Practices and Implementation Guide, Version 1.1

IMS Global Learning Consortium Inc. (2003). IMS Digital Repositories Specification

ISO Bulletin on JTC1/SC36 (2002). Information technology: Learning by IT

Ip, A., Morrison, I., and Currie, M. (2001). What is a learning object, technically? *WebNet 2001*, Association for the Advancement of Computing in Education (AACE).

Klein, M. and Fensel, D. (2001). Ontology versioning on the Semantic Web. *International Semantic Web Working Symposium* (SWWS), Stanford University.

Le Hors, A. et al. (2004). Document Object Model (DOM) Level 3 Core Specification, World Wide Web Consortium (W3C) recommendation, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/Overview.html

IEEE, Inc. (2002). *IEEE P1484.12.1-2002,* Draft Standard for Learning Object Metadata

Learning Object Repositories Network LORNET; http://www.lornet.org/.

Magee, M. et al. (2002). Building Digital Books with Dublin Core and IMS Content Packaging. *International Conference on Dublin Core and Metadata for e-Communities 2002* (DC-2002), Firenze University Press, Pages 91-96.

Mason, J. and Sutton, S. Education Working Group: Draft Proposal; http://dublincore.org/documents/2000/04/30/education-namespace/.

Massachusetts Institute of Technology MIT OpenCourseWare. (2003); http://ocw.mit.edu/index.html.

McGreal, R. et al. (2002). eduSource: A pan-Canadian learning object repository. *E-Learn*.

Merriam-Webster Inc. Merriam-Webster OnLine; http://www.m-w.com/.

Merrill, M.D. (March/April). Knowledge Objects., vol. 1, no. 11

Nejdl, W. et al. (2002). EDUTELLA: A P2P Networking Infrastructure Based on RDF. *The Eleventh International World Wide Web Conference* (WWW 2002), Association of Computing Machinery (ACM).

Noy, N.F. & Klein, M. (2002). Ontology Evolution: Not the Same as Schema Evolution, tech. report SMI-2002-0926, Stanford Medical Informatics, Stanford, CA

Office of the e-Envoy e-Government Metadata Initiative; http://www.govtalk.gov.uk/documents/e-Government_Metadata_Standard_v1.pdf.

OASIS DocBook Technical Committee (2002). *Committee Specification 4.2,* The DocBook Document Type

Paquette, G. and Rosca, I. (2002). Organic Aggregation of Knowledge Object in Educational Systems. *Canadian Journal of Learning and Technology*, vol. 28, no. 3.

Richards, G. and Hatala, M. (2002). POOL, POND and SPLASH - A Peer to Peer Architecture for Learning Object Repositories. *Internet 2 Conference, Workshop on Collaborative Computing in Higher Education: Peer-to-Peer and Beyond*.

Robson, R. Object-oriented Instructional Design and Web-based Authoring; http://www.eduworks.com/robby/papers/objectoriented.pdf.

Roddick, J.F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, vol. 37, no. 7, 383-393.

IMS Global Learning Consortium Inc. (2004). IMS Content Packaging Information Model, Version 1.1.4

Sosteric, M. and Hesemeier, S. (2002). When is a Learning Object not an Object: A first step towards a theory of learning objects. *International Review of Research in Open and Distance Learning*, vol. 3, no. 2.

Pemberton, S. et al. (2000). XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition), World Wide Web Consortium (W3C) recommendation, http://www.w3.org/TR/xhtml1/

Sutton, S.A. and Mason, J. (2001). The Dublin Core and Metadata for Educational Resource. *International Conference on Dublin Core and Metadata Applications 2001*, National Institute of Informatics.

Tichy, W.F. (1985). RCS: A system for version control. *Software - Practice and Experience*, vol. 15, no. 7, 637-654.

Treasury Board of Canada Government On- Line Metadata Standard; http://www.cio-dpi.gc.ca/its-nit/standards/tbits39/crit391_e.asp.

Wiley, D., Gibbons, A., and Recker, M. (2000). A reformulation of the issue of learning object granularity and its implications for the design of learning objects; http://www.reusability.org/granularity.pdf.

Wiley, D.A. The Post-LEGO Learning Object; http://wiley.ed.usu.edu/docs/post-lego.pdf.

Wiley, D. A. (2001). Connecting learning objects to instructional design theory: A definition a metaphor, and a taxonomy. In D. A. Wiley *The Instructional Use of Learning Objects*. Association for Educational Communications and Technology

Wood, L. et al. (2000). Document Object Model (DOM) Level 1 Specification (Second Edition), World Wide Web Consortium (W3C) working draft, http://www.w3c.org/TR/2000/WD-DOM-Level-1-20000929/

Wu, H. (2002). *Designing a Reusable and Adaptive E-Learning System*, master's thesis, Saskatoon, Saskatchewan, Canada, University of Saskatchewan

Appendix A

This appendix presents a normative binding of the version model presented in section 3.1 using XML Schema. All elements that are a part of the version model exist within the namespace *http://cs.usask.ca/~cab938/love/xml/11.01.2004*.

The schema defines a number of types, in particular:

- changeType: Encapsulates the structural and semantic changes which make up a given version, as well as the human readable log statement that describes that version. Each changeType is associated with a version identifier (VID).
- history: Orders the change types for a learning object, primitive or sequenced, such that the first change type corresponds to the first version of the learning object, and the last corresponds to the most recent version of the learning object. In this way applications can iterate over the change types in the history to build a revision timeline for a learning object. Each history is associate with an object identifier (OID).
- semanticType: Provides a metadata key (perspective), a new value for that key (value) and some description from a change set vocabulary identifying how that key has changed (transformation).
- structuralType: Provides a reference to a position in the learning object (path), and a function that modifies that learning object (transformation). In the case of a primitive learning object, these are XPath and DOM functions respectively (see Appendix B). In the case of a sequenced learning object these paths through a tree and tree transformations respectively (see Code Listing 3-4).

Code Listing A-1 provides an XML Schema for the version model.

Code Listing A-1: Normative XML Schema for the version model

```
[01] <?xml version="1.0" encoding="UTF-8"?>
[02] <xs:schema elementFormDefault="qualified"
[03]            xmlns:love="http://cs.usask.ca/~cab938/love/xml/11.01.2004"
[04]            xmlns:xs="http://www.w3.org/2001/XMLSchema"
[05]
     targetNamespace="http://cs.usask.ca/~cab938/love/xml/11.01.2004">
[06]     <xs:complexType name="changeType">
[07]         <xs:sequence>
[08]             <xs:element name="structural"
[09]                         type="love:structuralType"
[10]                         minOccurs="0"
[11]                         maxOccurs="unbounded"/>
[12]             <xs:element name="semantic"
[13]                         type="love:semanticType"
[14]                         minOccurs="0"
[15]                         maxOccurs="unbounded"/>
[16]             <xs:element ref="love:log"
[17]                         minOccurs="0"/>
[18]         </xs:sequence>
[19]         <xs:attribute name="vid"
[20]                       type="xs:anyURI"
[21]                       use="required"/>
[22]     </xs:complexType>
[23]     <xs:element name="history">
[24]         <xs:complexType>
[25]             <xs:sequence>
[26]                 <xs:element name="change"
[27]                             type="love:changeType"
[28]                             minOccurs="0"
[29]                             maxOccurs="unbounded"/>
[30]             </xs:sequence>
[31]             <xs:attribute name="identifier"
[32]                           type="xs:anyURI"
[33]                           use="required"/>
[34]         </xs:complexType>
[35]     </xs:element>
[36]     <xs:element name="log"
[37]                 type="xs:string"/>
[38]     <xs:complexType name="semanticType">
[39]         <xs:attribute name="value"
[40]                       type="xs:string"
[41]                       use="required"/>
[42]         <xs:attribute name="transformation"
[43]                       type="xs:string"
[44]                       use="required"/>
[45]         <xs:attribute name="perspective"
[46]                       type="xs:anyURI"
[47]                       use="required"/>
[48]     </xs:complexType>
[49]     <xs:complexType name="structuralType">
[50]         <xs:attribute name="path"
[51]                       type="xs:string"
[52]                       use="required"/>
[53]         <xs:attribute name="transformation"
[54]                       type="xs:string"
[55]                       use="required"/>
[56]     </xs:complexType>
[57] </xs:schema>
```

Appendix B

This appendix presents a binding of the Document Object Model (DOM) to a character syntax such that DOM invocations can be serialized into the XML change set form given in Appendix A.

Code Listing B-1: EBNF bindings for the DOM

```
[01] <separator> ::= ";"
[02] <createDocumentType> ::= "createDocumentType" <separator>
[03]                          <STRING> <separator> <STRING> <separator>
[04]                          <STRING>
[05] <createDocument> ::= "createDocument" <separator> <STRING>
[06]                      <separator> <STRING>
[07] <insertBefore> ::= "insertBefore" <separator> (<createElement> |
[08]                    <createTextNode> | <createComment> |
[09]                    <createCDATASection> | <createProcessingInstruction> |
[10]                    <createAttribute> | <createEntityReference> )
[11] <replaceChild> ::= "replaceChild" <separator> (<createElement> |
[12]                    <createTextNode> | <createComment> |
[13]                    <createCDATASection> | <createProcessingInstruction> |
[14]                    <createAttribute> | <createEntityReference> )
[15] <appendChild> ::= "appendChild" <separator> (<createElement> |
[16]                   <createTextNode> | <createComment> |
[17]                   <createCDATASection> | <createProcessingInstruction> |
[18]                   <createAttribute> | <createEntityReference> )
[19] <normalize> ::= "normalize"
[20] <setNamedItem> ::= <createAttribute>
[21] <removeNamedItem> ::= <removeAttribute>
[22] <setNamedItemNS> ::= <createAttribute>
[23] <removeNamedItemNS ::= <deleteNamespace>
[24] <deleteNamespace> ::= "deleteNamespace"
[25] <appendData> ::= "appendData" <separator> <STRING>
[26] <insertData> ::= "insertData" <separator> <LONG> <separator> <STRING>
[27] <deleteData> ::= "deleteData" <separator> <LONG> <separator> <LONG>
[28] <replaceData> ::= "replaceData" <separator> <LONG> <separator> <LONG>
[29]                   <separator> <STRING>
[30] <setAttribute> ::= "setAttribute" <separator> <STRING> <separator>
[31]                    <STRING>
[32] <removeAttribute> ::= "removeAttribute"
[33] <setAttributeNode> ::= <setAttribute>
[34] <removeAttributeNode> ::= "removeAttribute"
[35] <setAttributeNS> ::= <createAttributeNS>
[36] <removeAttributeNS> ::= "removeAttributeNS"
[37] <setAttributeNodeNS> ::= <createAttributeNS>
[38] <createElement> ::= "createElement" <separator> <STRING>
[39] <createDocumentFragment> ::= "createDocumentFragment" <separator>
     <STRING>
[40] <createTextNode> ::= "createTextNode" <separator> <STRING>
[41] <createComment> ::= "createComment" <separator> <STRING>
[42] <createCDATASection> ::= "createCDATASection" <separator> <STRING>
[43] <createProcessingInstruction> ::= "createProcessingInstruction"
[44]                                   <separator> <STRING>
[45] <createAttribute> ::= "createAttribute" <separator> <STRING>
[46] <createElementNS> ::= "createElementNS" <separator> <STRING> <separator>
[47]                       <STRING>
[48] <createAttributeNS> ::= "createAttributeNS" <separator> <STRING>
[49]                         <separator> <STRING>
[50] <importNode> ::= "importNode" <separator> (<createElement> |
```

```
[51]                    <createTextNode> | <createComment> |
     <createCDATASection>
[52]                    | <createProcessingInstruction> | <createAttribute> |
[53]                    <createEntityReference> )
```

The expressions from the change vocabulary above are mapped to transformation elements. Where more than one parameter is specified, the EBNF elements are separated by the token ";", and parameters are kept in order.

This binding corresponds to Level 2 of the DOM. The DOM is a general purpose API, and each level represents a new evolution of this API. The approach taken in this work is to couple an XPath expression (which points to a node in the DOM hierarchy) with a transformation to be applied to that node. Thus a number of the DOM methods are functionally equivalent (e.g. *the removeAttribute()* and *removeAttributeNode()* are equivalent when working with an attribute node in the DOM). In addition, the definition of a primitive learning object leads to using only a single XML document. Thus some of the modification functions of the DOM that are intended for multi-document environments (such as *importNode()*) are unlikely to be used. Wherever possible, these functions have been expressed in terms of single document creation functions.

Appendix C

The implementation prototype implements the version model by serializing change sets into XML and saving them within the IMS Content Packaging manifest. While the prototype was developed, changes were made to the XML Schema binding for the model. Substantial differences between the schema presented in Appendix A and the one used by the prototype include:

1. OID Identifiers: The prototype uses UNC filenames instead of the more general URIs suggested by section 3.1.2. While not technically incorrect, it is a poor programming practice as UNC names do not guarantee global uniqueness.

2. VID Identifiers: The prototype uses single branches for each learning object, and thus an OID with a version offset is enough to uniquely identify a particular version of a learning object. In a distributed situation VIDs should be generated according to the procedure outlined in section 3.1.2.

In addition, the schema used by the prototype does not correspond to the one provided in Appendix A. Instead the schema shown in Code Listing C-1 is used. While semantically similar, this schema uses slightly different element names for the structural types.

Code Listing C-1: Prototype XML Schema for the version model

```
[01] <?xml version="1.0" encoding="UTF-8"?>
[02] <xs:schema elementFormDefault="qualified"
[03]            xmlns:love="http://cs.usask.ca/~cab938/love/xml/11.01.2004"
[04]            xmlns:xs="http://www.w3.org/2001/XMLSchema"
[05]
     targetNamespace="http://cs.usask.ca/~cab938/love/xml/11.01.2004">
[06]     <xs:complexType name="changeType">
[07]         <xs:sequence>
[08]             <xs:element name="syntactic"
[09]                         type="love:structuralType"
[10]                         minOccurs="0"
[11]                         maxOccurs="unbounded"/>
[12]             <xs:element name="semantic"
[13]                         type="love:semanticType"
[14]                         minOccurs="0"
[15]                         maxOccurs="unbounded"/>
[16]             <xs:element ref="love:log"
[17]                         minOccurs="0"/>
[18]         </xs:sequence>
[19]     </xs:complexType>
[20]     <xs:element name="history">
[21]         <xs:complexType>
[22]             <xs:sequence>
[23]                 <xs:element name="change"
[24]                             type="love:changeType"
```

```
[25]                              minOccurs="0"
[26]                              maxOccurs="unbounded"/>
[27]             </xs:sequence>
[28]             <xs:attribute name="identifier"
[29]                           type="xs:anyURI"
[30]                           use="required"/>
[31]         </xs:complexType>
[32]     </xs:element>
[33]     <xs:element name="log"
[34]                 type="xs:string"/>
[35]     <xs:complexType name="semanticType">
[36]         <xs:attribute name="value"
[37]                       type="xs:string"
[38]                       use="required"/>
[39]         <xs:attribute name="transformation"
[40]                       type="xs:string"
[41]                       use="required"/>
[42]         <xs:attribute name="perspective"
[43]                       type="xs:anyURI"
[44]                       use="required"/>
[45]     </xs:complexType>
[46]     <xs:complexType name="structuralType">
[47]         <xs:attribute name="path"
[48]                       type="xs:string"
[49]                       use="required"/>
[50]         <xs:attribute name="transformation"
[51]                       type="xs:string"
[52]                       use="required"/>
[53]     </xs:complexType>
[54] </xs:schema>
```

Lastly, the grammar for mapping changes in primitive learning objects to the DOM is done at a slightly higher level than that presented in Appendix B. Instead of mapping directly to the DOM API, the grammar allows only for the creation and deletion of elements and attributes, and the insertion of text strings as children of these elements or values of these attributes. The creation of the DOM document as a whole is implicitly done when the author creates a learning object. Further, the more advanced features of the DOM (e.g. the insertion of a node before or after another node) are not implemented, but the same effect can be achieved by crafting precise XPath statements. These functions are useful for a full featured implementation, as they provide a comprehensive way to wrap a given XML parser. Nonetheless, a full demonstration of the approach can be by relying on special XPath expressions instead. Code Listing C-2 describes the grammar for the prototype DOM bindings.

Code Listing C-2: Prototype EBNF bindings for the DOM

```
[01] <separator> ::= ";"
[02] <name> :: = <STRING> //The name of the node (element or attribute)
[03] <location> ::= <LONG> //The position of the node in the child list
[04] <text_location> ::= <LONG> //A zero based character offset
[05] <InsertAttribute> ::= "InsertAttribute" <separator> <name>
[06] <InsertAttributeText> ::= "InsertAttributeText" <separator>
[07]                           <text_location> <separator> <STRING>
[08] <InsertElement> ::= "InsertElement" <separator> <location>
[09]                     <separator> <name>
[10] <InsertElementText> ::= "InsertElementText" <separator>
[11]                         <text_location> <separator> <STRING>
[12]
```

Code Listing C-3 provides an example of the output generated by the prototype for the learning object presented in Code Listing 3-1.

Code Listing C-3: Example prototype IMS Manifest

```
[01] <?xml version="1.0"?>
[02] <manifest xmlns="http://www.imsproject.org/xsd/imscp_rootv1p1p2">
[03]   <metadata />
[04]   <organizations />
[05]   <resources>
[06]     <resource d3p1:identifier="C:\_lor\brown_fox\index.xhtml"
[07]               d3p1:type="webcontent"
[08]               d3p1:href="C:\_lor\brown_fox\index.xhtml"
[09]
     xmlns:d3p1="http://www.imsproject.org/xsd/imscp_rootv1p1p2">
[10]       <love:history
[11]             xmlns:love="http://cs.usask.ca/~cab938/love/xml/11.01.2004">
[12]         <love:change>
[13]           <love:syntactic love:path="/"
[14]                           love:transformation="InsertElement:1:html" />
[15]           <love:log>
[16]           </love:log>
[17]         </love:change>
[18]         <love:change>
[19]           <love:syntactic love:path="/*[1]"
[20]                           love:transformation="InsertElement:1:body" />
[21]           <love:syntactic love:path="/*[1]/*[1]"
[22]                           love:transformation="InsertElement:1:p" />
[23]           <love:syntactic love:path="/*[1]/*[1]/*[1]"
[24]                           love:transformation="InsertTextNode:1:T" />
[25]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[26]                           love:transformation="InsertElementText:1:h" />
[27]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[28]                           love:transformation="InsertElementText:2:e" />
[29]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[30]                           love:transformation="InsertElementText:3: " />
[31]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[32]                           love:transformation="InsertElementText:4:q" />
[33]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[34]                           love:transformation="InsertElementText:5:u" />
[35]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[36]                           love:transformation="InsertElementText:6:i" />
[37]           <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
```

```
[38]                            love:transformation="InsertElementText:7:c" />
[39]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[40]                            love:transformation="InsertElementText:8:k" />
[41]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[42]                            love:transformation="InsertElementText:9: " />
[43]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[44]                            love:transformation="InsertElementText:10:b" />
[45]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[46]                            love:transformation="InsertElementText:11:r" />
[47]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[48]                            love:transformation="InsertElementText:12:o" />
[49]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[50]                            love:transformation="InsertElementText:13:w" />
[51]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[52]                            love:transformation="InsertElementText:14:n" />
[53]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[54]                            love:transformation="InsertElementText:15: " />
[55]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[56]                            love:transformation="InsertElementText:16:f" />
[57]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[58]                            love:transformation="InsertElementText:17:o" />
[59]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[60]                            love:transformation="InsertElementText:18:x" />
[61]          <love:syntactic love:path="/*[1]/*[1]/*[1]/text()[1]"
[62]                            love:transformation="InsertElementText:19:." />
[63]          <love:log>
[64]          </love:log>
[65]        </love:change>
[66]      </love:history>
[67]    </resource>
[68]  </resources>
[69] </manifest>
```