ADELAIDE UNIVERSITY AUSTRALIA

Improved Searching Mechanisms for Distributed Geospatial Image Archives

Anna Forrest

October 2001

Submitted in partial fulfillment of the requirements for the Honours degree of Bachelor of Science in the faculty of Mathematical and Computer Sciences.

Table of Contents

Table of Figures

Acknowledgements

The completion of this research project would not have been possible without help from a number of people. Firstly, I am grateful to Dr. Paul Coddington who willingly accepted the task of supervising me as an Honours research student. Paul has provided many valuable suggestions that have helped me to solve problems and improve on work already completed. His patience when explaining difficult concepts has been a pleasant virtue and much appreciated.

I would also like to thank the other lecturers and tutors I have encountered during my degrees. In particular, I must make mention of David Knight and Kevin LewKewLin who have been instrumental in helping me understand the fundamentals of computer science. David and Kevin would willingly spend their time explaining new material to me and managed to do this in a clear, concise and encouraging way.

Thanks also to my parents who have helped support me through university. Your love and encouragement has meant so much and getting home to see you on “The Farm” has always given me necessary rest and relaxation.

Finally I would like to thank all my friends for their support and understanding throughout the year.

Abstract

Many organisations are developing geospatial image archives (libraries) containing data from earth observation systems. However, a lack of standardised interfaces for querying, accessing and processing data from distributed archives makes it difficult to develop applications that can utilise multiple libraries.

The U.S. National Imagery and Mapping Agency (NIMA) has developed the Geospatial and Imagery Access Services (GIAS) specification to address this problem. Similar efforts are being made to develop interface standards by the OpenGIS Consortium (OGC).

Searching multiple archives is problematic as each may employ different image storage formats or metadata schemas (data models). This research shows how XML and XSLT, the emerging standards for data description and exchange, can be used to solve the problem of converting between different data models in a distributed geospatial image archive. This is demonstrated by the design and implementation of a new metadata conversion architecture for OLGAS, a Java program that conforms to a subset of the GIAS specification. The data models supported in OLGAS were defined using XML Schemas and the translations between different data models were achieved using XSLT. The metadata conversion architecture discussed in this paper enables federated querying across multiple image libraries that employ different data models. The architecture may also be used in libraries that support the Open GIS Consortium interface standards.

Chapter 1

Introduction

One of the major developments in Information Technology in the past decade has been the advent of distributed computing technologies such as the World Wide Web, Java [45] and the Common Object Request Broker Architecture, CORBA [48]. These developments have been the driving force behind the way society disseminates and accesses information. Many people are now departing from traditional information mediums (such as paper maps, atlases, and television reports) to gain up-to-the-minute information from on-line digital data archives [9].

There are many organizations developing online libraries of geospatial data, that is, information that relates to a geographic location at or near the surface of the Earth. Some example archives include Microsoft's Terraserver; the Australian Centre for Remote Sensing (ACRES) Digital Catalogue; the U.S. Bureau of the Census TIGER Map Server; and MapQuest [3,9]. Information in these libraries is collected from various earth observation systems (EOS) and encompasses things like maps, aerial photography, charts and satellite imagery. For example, Microsoft's Terraserver has digital imagery from Russian SPIN-2 satellites as well as digital orthorectified photographs from the U.S. Geological Survey [9]. The amount of geospatial data available is rapidly increasing with current EOS capable of producing terabytes of data per day [15]. This presents a real need to develop online data archives that provide efficient storage, access and retrieval mechanisms for large data sets.

The services of distributed geospatial libraries are important for many application areas, e.g. emergency planning, natural resource management and defence. Such applications require the timely provision of information about a place in order to minimize the loss of lives or avoid the costs associated with making bad decisions [9]. However, developing applications of this nature is severely impeded by the ability to integrate geospatial information from multiple archives. These difficulties arise because archives generally employ different data formats and standards, access mechanisms and organizational structures.

The formidable problem of integration has prompted many organizations to develop interoperability standards for geospatial data, including the Open GIS Consortium (OCG), the U.S. National Imagery and Mapping Agency (NIMA), the ISO technical committee on geographic information (ISO TC211), the U.S. Federal Geographic Data Committee (FGDC), etc. Much of their efforts have centered on standardizing file formats and metadata models. It has only been of late that work has been done to provide standard interfaces for querying, accessing and processing data from online geospatial archives.

The U.S. National Imagery and Mapping Agency (NIMA) have driven the development of standards and specifications for interfacing to distributed geospatial image archives.

NIMA have defined the U.S. Imagery and Geospatial Information System (USIGS) architecture that is targeted at defence applications. A large component of USIGS is the Geospatial and Imagery Access Server (GIAS) [51] specification, which defines standard CORBA interfaces for interfacing to a geospatial image library [4]. The Open GIS Consortium (OGC), whose ultimate goal is to enable interoperability of distributed Geographic Information System (GIS) applications, is making similar efforts. The OGC have introduced the Open GIS Catalog Interface Specification that includes standard interfaces for a variety of distributed computing platforms including CORBA, OLEDB and WWW [32].

The research presented in this thesis investigates ways to improve the searching mechanisms of distributed geospatial image archives. The motivation for this work was to develop a standards-based architecture to handle multiple image products that have different data models, e.g. GMS-5 and LANDSAT satellite images, orthorectified aerial photos, National Imagery Transfer Format (NITF) etc; support querying by clients that understand different data models; and allow federated querying over multiple servers with different data models.

In this project, emerging web standards XML [54] and XSLT [62] were used to support multiple data models within OLGAS [30], an On-Line Geospatial Imagery Access Server that conforms to a subset of the GIAS specification. XML and XSLT are languages defined by the World Wide Web Consortium (W3C) specifically for data description and transformation. There are two mechanisms in XML for defining data models, namely DTDs and XML Schemas, and transformations between them are possible using XSLT. The major motivation for using XML and XSLT was to replace the ad-hoc approach to metadata conversion in OLGAS with a standards-based architecture.

The material in this thesis adheres to the following structure. Chapter 2 presents more information about distributed geospatial image libraries. An overview of the OLGAS system, and the original state of its data model conversion architecture, forms much of this chapter. Other topics of discussion include middleware, the GIAS and Catalog Interface specifications and federated querying.

Chapter 3 describes the technologies used. The chapter begins by explaining the importance of XML and some simple examples of the language syntax are given. The discussion then turns to DTDs and XML Schemas; two mechanisms that are used to specify the structure of XML documents. The concepts of XSLT, a language for transforming XML documents into other forms, are also explained.

The discussion of Chapter 4 details how the geospatial data models were described using XML Schema. Example data model conversions are also given.

The modifications made to the OLGAS program are noted in Chapter 5. Also found here are the reasons for each modification and any problems that occurred during the implementation phase.

Finally, a summary and suggestions for future work is given in Chapter 6.

Chapter 2

Distributed Geospatial Image Archives

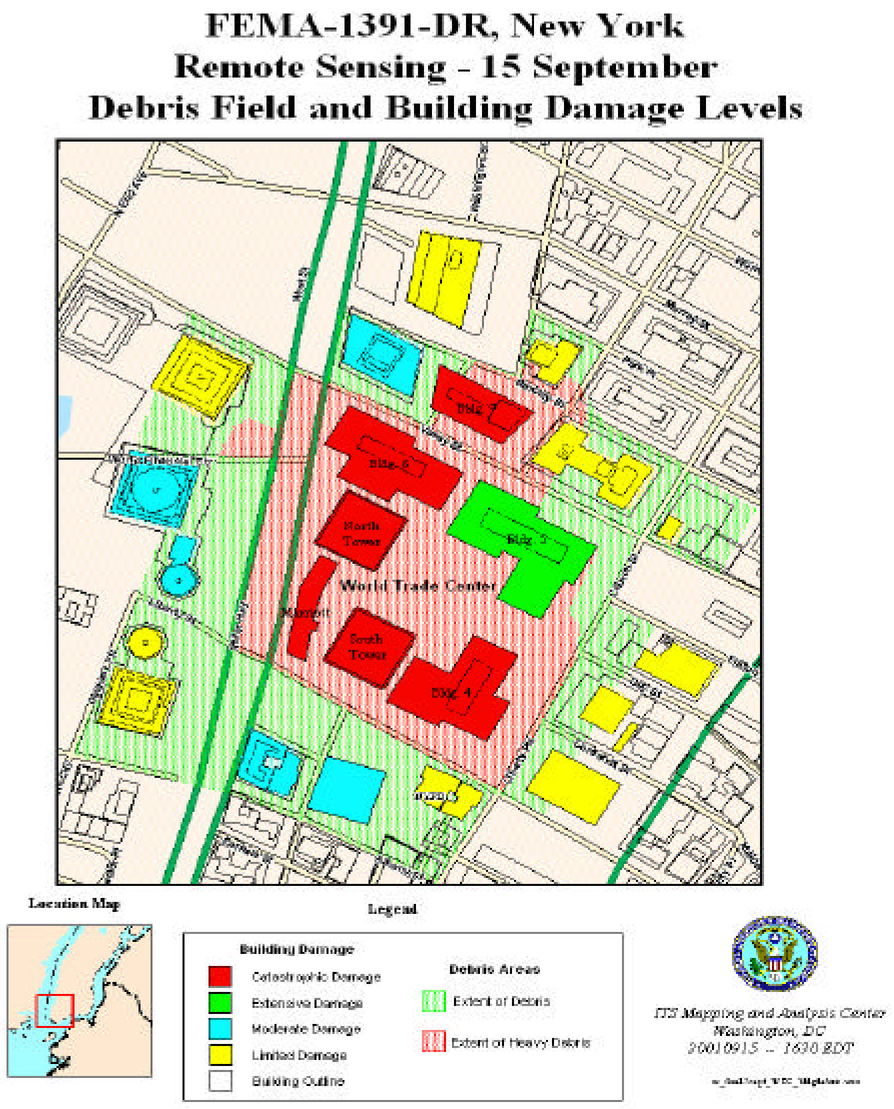
2.1 Definitions

A geospatial image archive is distributed if its users, services, and image metadata are integrated among many distinct locations [9]. The development of distributed archive systems has been necessary for a number of reasons. Sophisticated earth observation systems, such as satellites, are producing large volumes of geospatial data per day (in the order of terabytes) [15]. It is simply not practical for users to have local copies of all this data. In most cases these data sets will be stored on multiple servers and managed through a communications network. Another reason is that geospatial data exists at different physical localities. This is because its collection is decentralized – performed by many different groups. Furthermore, the ability to perform rapid processing of large amounts of geospatial data may require the use of a high performance machine that is not local to the data archive. In fact multiple high performance machines may be needed to obtain enough performance to be useful for computation-intensive and/or time-critical end-user applications.

Many application areas are now relying on geographic information systems (GIS) that access and process geospatial data from distributed data archives. An example of this is the on-demand access and analysis of satellite and photoreconnaissance images for defence Command, Control, Communications and Intelligence (C3I) systems [18]. A distributed image library system called IMAD (Image Management and Dissemination) has been developed to provide this capability [4,19]. More detail about the IMAD system is given in Section 2.5.

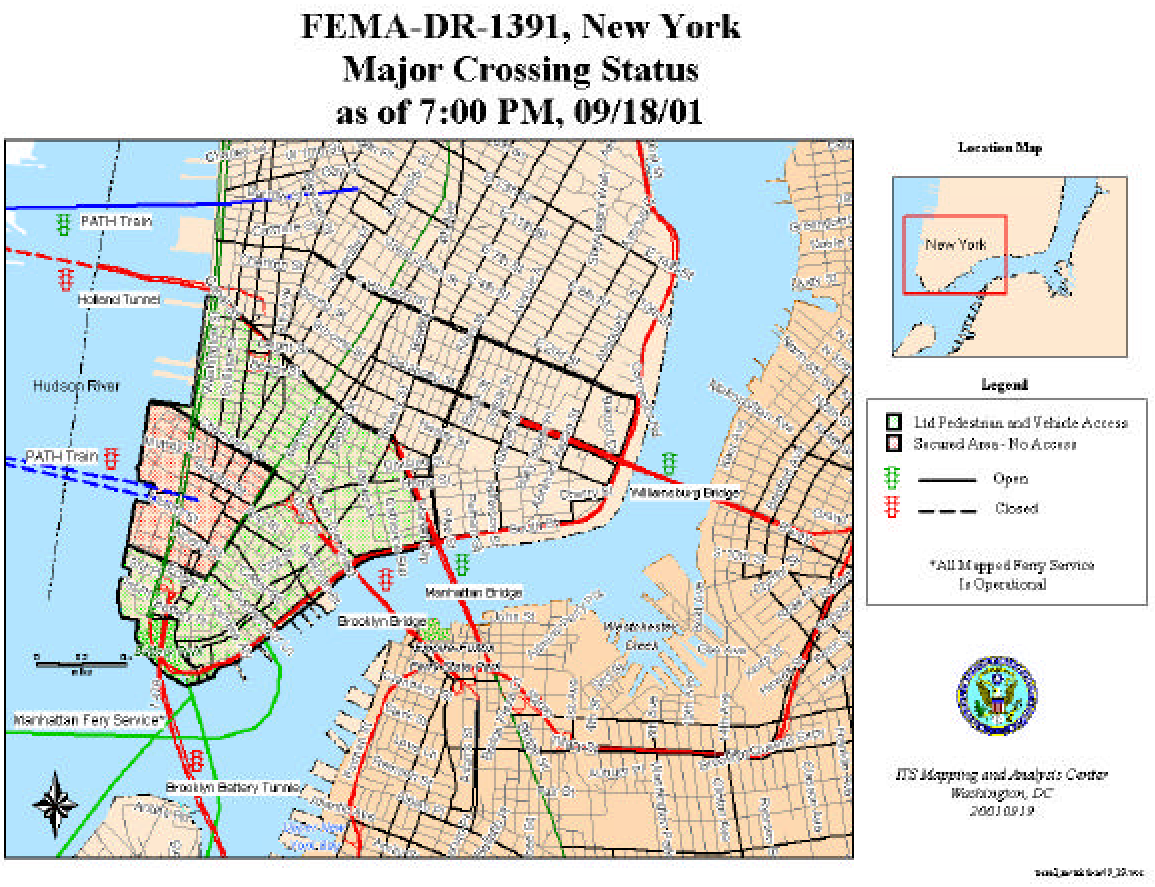
The services of distributed geospatial libraries are extremely important for emergency services planning and response. The *New York Times* recently addressed the vital role that GIS was playing in the World Trade Center recovery efforts of the September 11 terrorist attacks [16,24]. The Federal Emergency Management Agency (FEMA) Mapping and Analysis Center [10] have been developing a GIS database to provide task force leaders with data concerning potential hazards. This information includes maps of unstable building foundations, debris density, and aerial photography measuring thermal radiation. Coordinators of the recovery operation have found the development of easy to use organized digital archives essential in putting together a plan that protects the rescue workers. FEMA have posted a New York GIS archive on their website containing maps and images that are accessible to the public. Figure 1, Figure 2 and Figure 3 were obtained from this site and show the transportation routes affected, building damage levels and a post-attack aerial image [12,11,13].

Figure 1: FEMA map of WTC Debris Areas and Building Damage



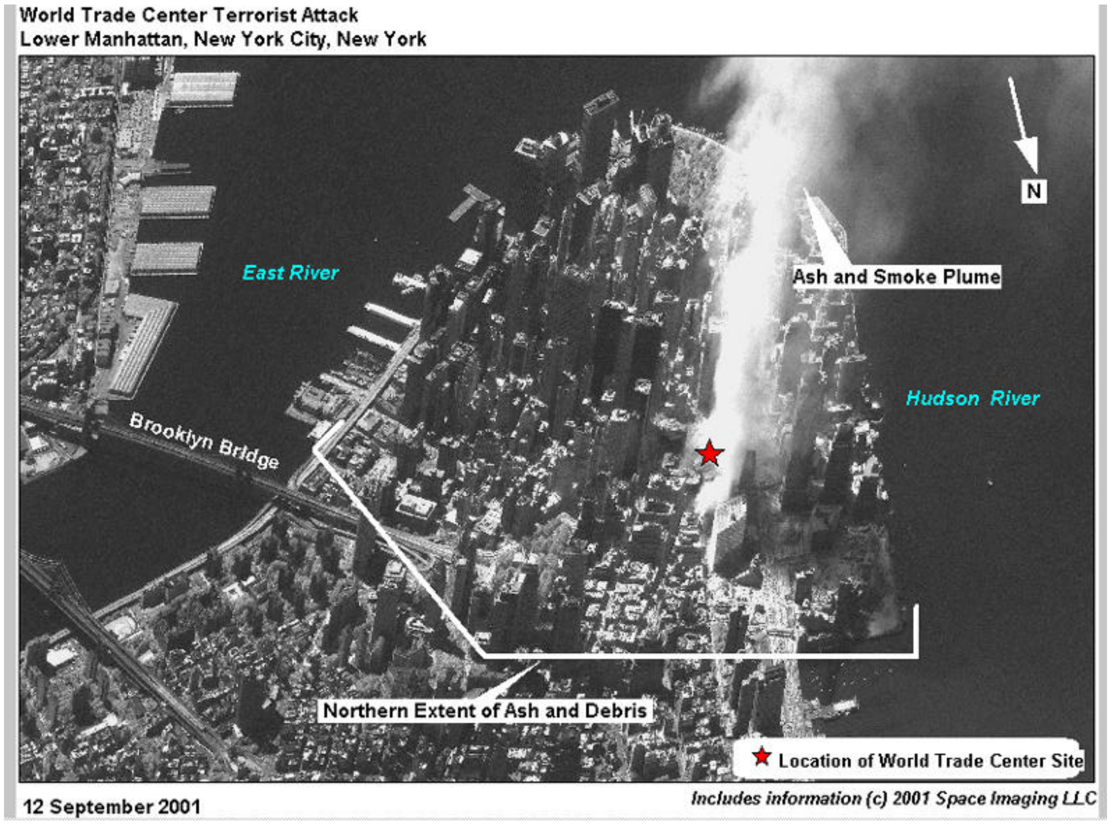
Similar data is being collected by other agencies including Intergraph Mapping and GIS Solutions who are assisting the Transit department of NYC to make decisions on how to best restore the damaged routes [37].

Figure 2: FEMA map of NY Major Transportation Route Status



Information that is useful in emergency planning and C3I systems will typically come from different distributed sources, each with different data formats, metadata schemas and interfaces for accessing and querying the data. In the past, these systems were not designed to interoperate or share data so a lot of time and effort was expended to integrate the data from different sources. Now, organizations like the Open GIS Consortium are working on defining standards for data formats, metadata and interfaces that will allow these systems to interoperate.

Figure 3: Post-attack image of New York (Courtesy of Space Imaging)


World Trade Center Terrorist Attack
Lower Manhattan, New York City, New York
East River
Ash and Smoke Plume
N
Hudson River
Brooklyn Bridge
Northern Extent of Ash and Debris
Location of World Trade Center Site
12 September 2001
Includes information (c) 2001 Space Imaging LLC

2.2 Barriers to Geospatial Data Sharing

Discovering, processing and exploiting geospatial data from multiple archives is made difficult due to one or more of the following barriers:

- There is no common interface to the data. A number of different databases are used to store data and these may have different query and access mechanisms.
- A large number of data formats exist for storing geospatial data. Some of these formats require special tools to access the data. For example, a set of HDF (Hierarchical Data Format) utilities from the NSCA website are required to extract metadata and image data from HDF data sets [29].
- Different metadata schemas (i.e. database schemas) are used to store data. Thus metadata converters are needed to convert between schemas.
- Differing data and interpretation. Methods used to collect and interpret data vary between organizations [26].

These barriers clearly demonstrate the need for metadata standards and common interfaces to geospatial image libraries. Until recently most effort from the standards community has focused on developing standard metadata fields to allow client queries to be understood by data archives that conform to the standards. A problem has arisen with several metadata standards being produced for use by different user groups. General metadata standards for searching the web are being developed, however most of the standards being produced are designed to reflect the data content in particular fields of interest [4]. For example, there are standards for geospatial, imagery, and geospatial imagery data.

Another problem can occur if metadata standards are revised. If a server is upgraded to conform to a revised data model then client applications using the previous model may not function. Handling data model revisions in a distributed environment requires interfaces for querying services that can support multiple data models or data views.

2.3 Standard Interfaces to Geospatial Image Archives

New technologies such as the World Wide Web, Java, and the Common Object Request Broker Architecture (CORBA) are making it easier to develop standard interfaces for supporting distributed client/server applications. CORBA, developed by the Object Management Group (OMG), is a standard middleware architecture that allows a set of distributed objects to communicate [22]. Object communication is achieved through a ‘request of service’ paradigm. The services that an object provides are described in an interface using OMG’s Interface Definition Language (IDL). IDL’s are language independent descriptions and enable distributed objects to be implemented in different programming languages (such as Java, C, and C++). Using IDL’s allows system components to be developed independently and integrated efficiently through an object request broker infrastructure.

The development of standards for interfacing to distributed geospatial archives was initially driven by the defence sector and the U.S. National Imagery and Mapping Agency (NIMA). NIMA have been devising specifications for a standard U.S. Imagery and Geospatial Information Service (USIGS) architecture. A major element of USIGS is the Geospatial Imagery and Access Server (GIAS) specification which provides a standard interface to a geospatial image library. The USIGS specifications adopt an object-oriented approach and are described using the Object Management Group (OMG) Interface Description Language (IDL).

More recently, the Open GIS Consortium (OGC) has been developing interface standards to enable interoperability of distributed GIS applications.

2.3.1 Geospatial and Imagery Access Service (GIAS) Specification

The GIAS defines standard CORBA interfaces for a geospatial image library. The GIAS specification is structured so that a different manager controls each library function. Each manager is specified by a class with a well-defined interface describing data, methods and error conditions of that class. The following provides a description of the essential classes:

Library – The Library class serves as the starting point for any interaction with the rest of the library. That is, it is the central point of access to all manager services offered by GIAS for the image library. It has a method `get_manager_types()` to allow a client to discover the managers supported by a particular GIAS library implementation. Another method `get_access_criteria()` allows clients to discover of the access criteria for a particular type of manager (such as username, password etc). There is also an operation to request access to a manager, `get_manager()`. A client can also discover other accessible libraries that are known to this library via `get_other_libraries()`.

Data Model Manager – This manager allows a client to discover and access the data model being used by the Library. The operations of this manager are partitioned into two groups: ancillary data access and data model access. Ancillary operations include finding out the last date the Library's data model was updated. Data model specific methods provide information about the data views that the Library supports, and the queryable and non-queryable attributes of a requested data view.

Catalog Access Manager – The operations defined in the Catalog Access Manager, `submit_query()` and `hit_count()`, allow a client to submit queries to search the holdings of a GIAS Library. The client indicates the query data view they wish to use by passing it in as a parameter to these methods. The client discovers the data views accepted by the library from interrogating the Data Model Manager. The queries must be submitted in Boolean Query Syntax (BQS), a grammar that was created specifically for use by GIAS compliant libraries. BQS is also defined in the GIAS specification.

Creation Manager – The creation manager provides methods to request the archiving and cataloging of a new product to a library. A client can nominate to include only metadata via `create_metadata()` or both the metadata and image data set using `create()`.

Order Manager – This manager provides functions to enable clients to order image products from the library. There is a method that allows clients to check if their order is valid before submission. The client may request delivery of the product via FTP, email, or hardcopy.

Array Access Manager - Allows clients to specify a geospatial region of an image to be loaded directly into an application as a data object.

Product Access Manager – This manager allows clients to determine characteristics about a specific product. The client may also request thumbnail images for a set of products.

Typically clients will access data from a GIAS library as follows. They will connect to the library through the Library Manager which will identify the other available managers. Clients will then determine an appropriate data view to use through interrogating the Data Model Manager. A BQS query is constructed and submitted to the Catalog Access Manager which handles the query and returns the results. More specifically, the server is provided with the name of a data view and a query in that data view. The server then converts this query into a data model that its internal database understands. The query is then executed and the results are converted back into the query data view and returned to the user. The client can view thumbnail images of the hits using the Product Access Manager. Once an image is selected, the client can chose to download it using either the Order Manager or Array Access Manager.

The GIAS specification does not define interfaces for locating libraries with specific characteristics, as this is a function of the CORBA Trading service. Also, the specification does not provide interfaces for processing information that is not directly related to search or delivery, because this is a function of exploitation systems (i.e. a system that provides geospatial image processing services) [51]. Standards for image processing services (or exploitation services) are also under investigation. NIMA have released the Geospatial and Imagery eXploitation Service (GIXS) specification to enable image processing [4]. The design of GIXS is based on the Java Advanced Imaging API [40] and is still largely a work in progress.

2.3.2 Open GIS Catalog Interface Specification

The Open GIS Consortium (OGC) has developed the Open GIS Catalog Interface specification to enable interoperability of distributed applications. The Catalog Interface specification is partly based on the GIAS standard which was submitted to the OGC as a candidate.

The Open GIS specification caters for a variety of distributed computing technologies, including CORBA (Common Object Request Broker Architecture), Microsoft's DCOM and the World Wide Web [32].

The specification provides support for both a GIAS-based approach and a general digital library catalog search approach. The later is a Web-based approach built on technologies such as XML [54] and Z39.50 [1], and uses a message-based client-server architecture.

The Open GIS specifications are far more general than those developed by NIMA. The GIAS standards were targeted at military applications and focus mainly on geospatial image data, however, the Open GIS framework supports any kind of geospatial data.

2.4 On-Line Geospatial Image Access Server (OLGAS)

The On-Line Geospatial Image Access Server (OLGAS) was developed by the Distributed High Performance Computing (DHPC) group of Adelaide University in collaboration with the Imagery Management and Dissemination (IMAD) group of the Defence Science and Technology Organisation (DSTO). OLGAS is a prototype data archive management system which implements a subset of the Geospatial Imagery and Access Server (GIAS) specification. It provides the basic functionality required of a geospatial image server including storage, management, searching, accessing and basic processing of image data files and metadata.

2.4.1 System Overview

The OLGAS system is portable. The code is written completely in Java and can run unchanged on any computing platform that supports a Java Virtual Machine [5]. The system also uses Java Database Connectivity (JDBC) [46] for interfacing to the image database. This enables the OLGAS program to interface to any database system that supports JDBC. This includes commercial database systems such as Informix and Oracle, as well as free public-domain databases like mySQL and PostgreSQL. The server may be accessed remotely via the Internet through standard interfaces using CORBA. The OLGAS system can be used to create a new library or as a wrapper to provide a standard interface to an existing library [5].

The current implementation of OLGAS is based on version 3.0 of the GIAS specification. However, some interfaces from version 3.3 are also supported. As mentioned in Section 2.3.1, different managers control the main features of the library, such as adding, querying and accessing data. At the time of taking on this project, the OLGAS implementation supported the following managers: a Library Manager, Catalog Access Manager (for client queries), Creation Manager (to add new image products), Array Access Manager (to download a region of an image directly into an application), and a Geo Data Set Manager (that handles image dissemination). Note that the Geo Data Set Manager interface was defined in version 3.0 of the GIAS specification and was replaced in later versions by the Order Manager which provided similar functionality.

2.4.2 Supporting Multiple Image Products and Data Models

The primary goal of this project was to provide an efficient way of handling multiple image product formats and data models in a GIAS compliant image server. This section aims to provide an account of the conversion architecture needed to support different data models whilst explaining the previous image and metadata conversion architecture of OLGAS.

2.4.2.1 Previous OLGAS Image and Metadata Conversion Architecture

The conversion architecture requirements of the system are clearly shown in Figure 4. In order to support multiple image products and formats, the server needs to know how to extract metadata and image data from image files. The Image Converter layer in OLGAS handles this by using the Java Advanced Imaging (JAI) [40] API for reading and writing image data and associated metadata for a specific image product or format.

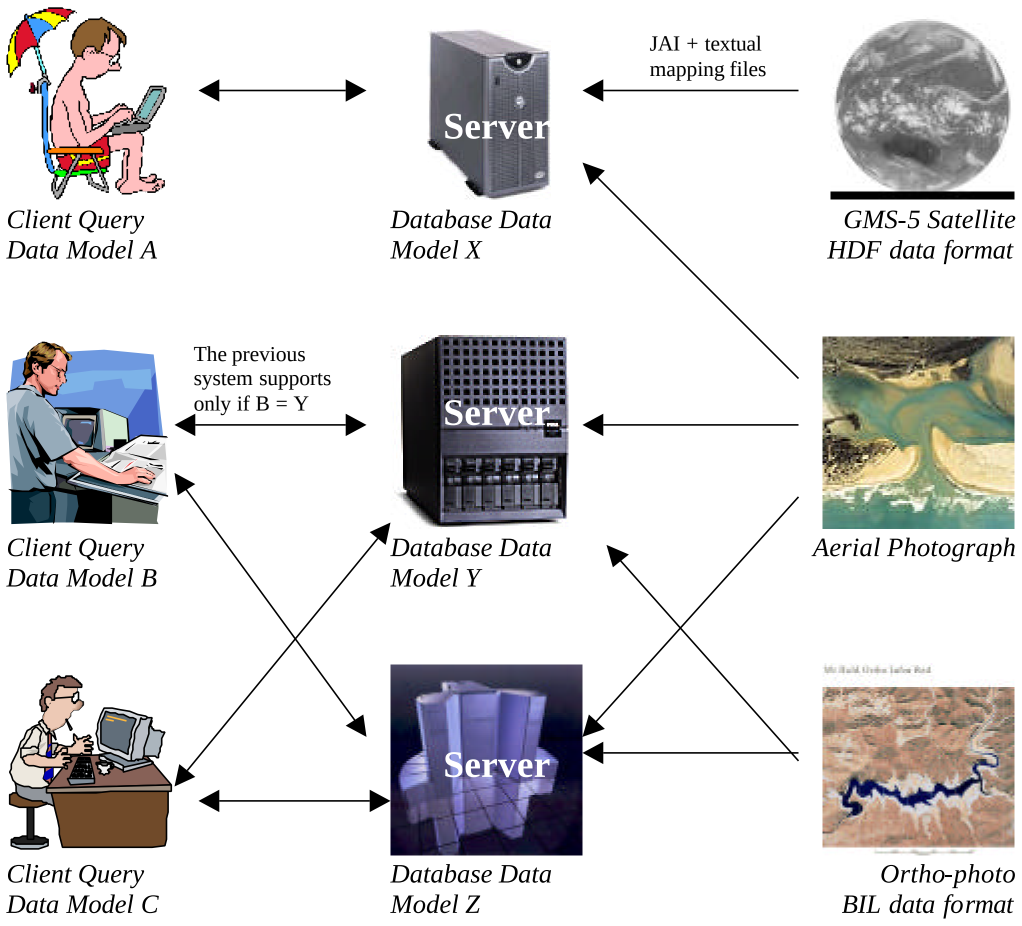
The OLGAS architecture supports the following image formats:

- National Imagery Transmission Format NITF: used mainly for military applications
- Hierarchical Data Format HDF: standard for scientific data, including GMS-5 and other satellite data
- Binary Interleaved by Line BIL: used for georectified images (orthophotos)

Note that NITF and HDF have support for metadata built into the format, whereas BIL does not and requires separate metadata and image files. Another thing to note is that many different types of image products employ the same data format. For example, the HDF format is used in GMS-5 satellite products, LANDSAT-7 products, and image products from NOAA MSPPS (Microwave Surface and Precipitation Product System). The same JAI HDF reader/writer can be used to extract the metadata for any HDF file. However, HDF products that use different data models will require different metadata converters. Other image formats can be supported in OLGAS with a JAI reader/writer and an appropriate metadata converter.

A metadata converter is required to transform the image product data model (e.g. ORTHO, GMS-5) to the standard data model used for storing metadata in the database. The previous OLGAS system supports the conversion of metadata from the abovementioned image products to the OLDA database data model. A description of OLDA fields is provided in Appendix A and will be a valuable reference throughout this section and Chapter 5.

Figure 4: Conversion Architecture Requirements


JAI + textual mapping files
Server
Server
Server
The previous system supports only if B = Y
Client Query Data Model A
Database Data Model X
GMS-5 Satellite HDF data format
Client Query Data Model B
Database Data Model Y
Aerial Photograph
Client Query Data Model C
Database Data Model Z
Ortho-photo BIL data format

The database data model is often referred to as the primary data view. The other data models used in the system (for image files and client queries) are called secondary data views.

The OLGAS MetaConverter Architecture previously used data model configuration files to describe each data view to be recognised by the OLGAS server. These configuration files specify a list of metadata fields, their types, and whether they are queryable. The following is an extract from the OLDA data view configuration file:

```
IMAGEID     char(80)    Q
NBANDS      short
YEAR        long        Q
```

The primary data view is identified during library initialisation. Previously, mappings from a secondary data view to the primary data view were specified by a couple of mapping files (textual and Java class mapping files). The OLGAS primary data view contains some mandatory attributes that must be present when adding image product metadata to the server repository. For example, the primary data view of OLGAS contains the mandatory fields:

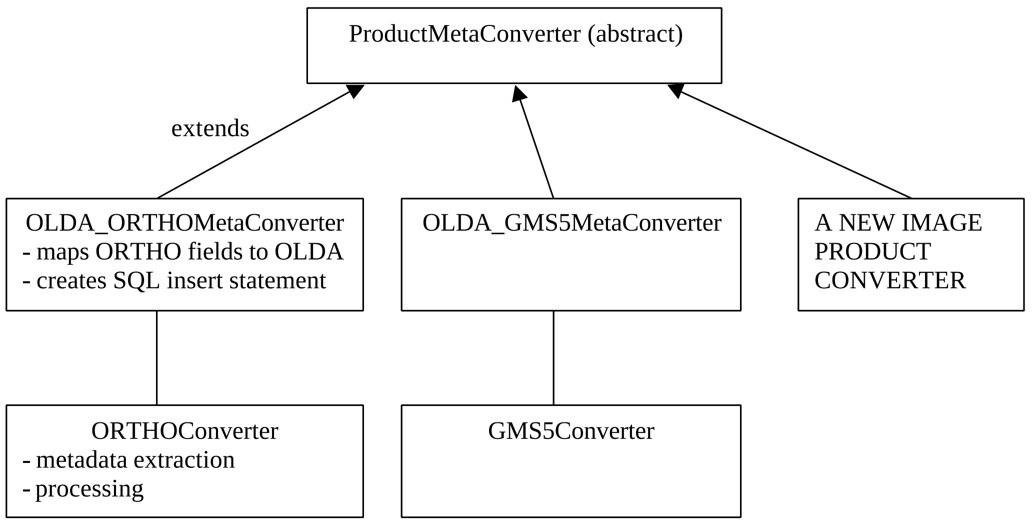
▪ ACCESSID	- which is the database access ID for an image
▪ FILENAME	- the name of the image data file
▪ URL	- the URL for the image if it is accessible from a Web Server
▪ THUMBNAIL	- the name of a file containing a thumbnail image of the data

If the product data view mapping does not assign values to these fields then the server must create them to allow the image product to be queried and utilized [25]. The textual mapping file provides a mapping from the image product data view to the primary data view. These files specify the values that each primary field should be assigned. A primary data model field may map to:

- A literal value (constant mapping)
- An equivalent secondary data model field (referred to as a 1-1 mapping)
- A value that needs to be generated by some processing or format conversion of a secondary data model field or fields.

In version 0.09 of OLGAS, a Perl script (a metadata extraction program) was required for each image product to perform the mappings in the text file. More recent versions of OLGAS achieved this using Java class files that implemented a ProductMetaConverter superclass. Please refer to Figure 5 for an illustration of the metadata conversion architecture previously used in OLGAS. This diagram shows two image product converters for the ORTHO and GMS-5 data models. The OLDA_ORTHOMetaConverter and OLDA_GMS5MetaConverter classes perform a mapping to the primary data view (OLDA) with the assistance from the ORTHOConverter and GMS5Converter classes that extract and process the image metadata. The MetaConverter mapping routine generates an SQL insert statement which is required for registering the product metadata in the database. To support any new image products another class extending the ProductMetaConverter superclass must be provided.

Figure 5: Class Diagram of Previous OLGAS Meta-Conversion Architecture


ProductMetaConverter (abstract)
extends
OLDA_ORTHOMetaConverter
- maps ORTHO fields to OLDA
- creates SQL insert statement
OLDA_GMS5MetaConverter
A NEW IMAGE PRODUCT CONVERTER
ORTHOConverter
- metadata extraction
- processing
GMS5Converter

A similar metadata conversion is needed when handling client queries that are presented in different data views. This concept is portrayed in Figure 4. When the gentleman on the beach submits a query from his wireless laptop to the image server, the attributes within his query (i.e. from data view A) must be mapped to those attributes understood by the server (i.e. data view X). Also, when the server returns the query results they must be mapped back into the query data view. As mentioned in Figure 4, the previous OLGAS system only handled queries that were in the same data model as the internal database schema.

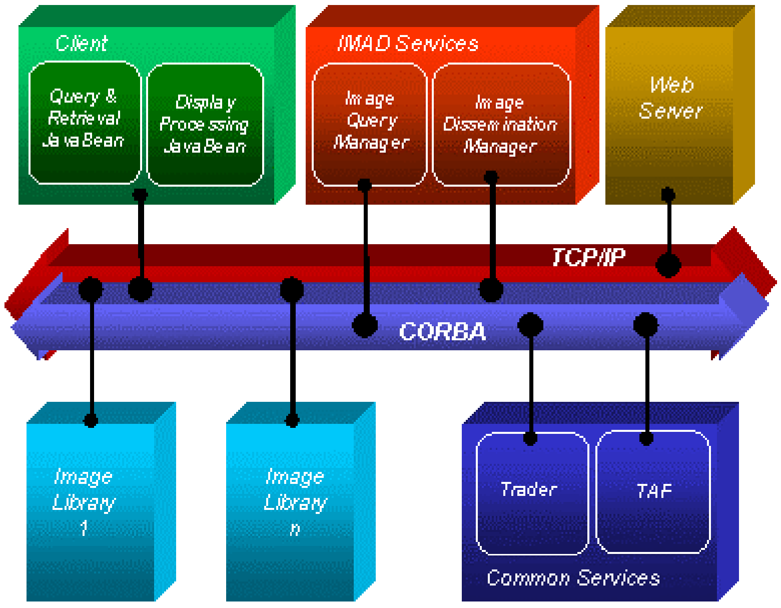
As you can see, the previous OLGAS system had a rather non-standard way of converting between the secondary and primary data views. This was largely because there were no standard mechanisms available for data model conversion at the time of developing OLGAS. However, now there are standards like XML and XSLT for describing and transforming data models and this project shows how they can be incorporated into OLGAS.

2.5 Federated Querying

Currently, the GIAS specification only defines an interface for a single image library. It does not define a middleware architecture that would enable multiple archives to be combined into a federated system [4]. The Imagery Management and Dissemination (IMAD) group of DSTO have developed a federated image data service that can interface to any number of GIAS-compliant image libraries [19].

The IMAD system provides client access to a federation of distributed archives via specialized middleware components (IMAD and CORBA services). Components of the system are displayed in Figure 6 [4].

Figure 6: Components of the IMAD system



The IMAD middleware architecture allows a client to submit queries and retrieve images from a federation of distributed libraries. The two IMAD-specific services providing this extra functionality are the Image Query Service (IQS) and the Image Dissemination Service. Before discussing these services it is worth explaining the CORBA services used by this system.

The IMAD system relies on common CORBA middleware services such as the Trader and the Transmission Availability Forecast (TAF). The Trader is a standard CORBA service that is part of the CORBA specification. Its role is to find objects (such as image library servers) that offer services requested by the client. The client may narrow this search by specifying the criteria that an object must obey. For example, “find image servers with 300 or more images”. However, for a library to be located by a trader it must first register its services with the trader. This is analogous to a Business directory (e.g. the Yellow Pages). If a business does not register an advert in the directory then clients will not discover them. To increase the service domain of large distributed systems, multiple traders may be linked together to form a federated trading system. When a local trader receives a client request for a particular service, the request is circulated to the other traders in the federation. Any hits from remote traders are collated by the local trader and returned to the client [4]. The Transmission Availability Forecast (TAF) Service is a specialized service defined and implemented by the DSTO IMAD group, which is used to determine the available bandwidth between a client and server object.

As mentioned, IQS is a middleware service that allows clients to submit a query to all libraries in the federated system. The following provides a brief overview of how client queries are handled by this service. When a client query string is submitted, the IQS contacts a local trader and obtains references to the available libraries. Query Manager threads are then created to connect to each library and perform the query.

While the Query Manager threads are busy, the IQS calls upon the TAF service to determine the available network bandwidth between each image library and the client. This information enables the IQS to collate results from the different libraries more efficiently. When the IQS receives all results back from the Query Manager threads, they are consolidated into a big hit-list. The image metadata is then collected for each result in the list and requires the IQS to detect any duplicates. The query results are packaged into a distributed query request object and are returned to the client.

Transferring large quantities of image data via current CORBA mechanisms can involve significant overheads and can lead to poor performance. The IMAD system handles image retrieval through an Image Dissemination Manager. This manager uses a factory pattern to generate a retrieval object that will deliver imagery to the client in the most appropriate form [19].

The current IMAD system assumes that all GIAS servers in the federation use the same data model. However this is an unrealistic assumption due to reasons outlined in Section 2.2 (e.g. revisions to metadata standards). Hence, the successful implementation of federated querying across multiple GIAS libraries requires that each server be able to support multiple data models, which is the main goal of this project.

Chapter 3

Emerging Language Standards for Data Description and Transformation

3.1 XML: eXtensible Markup Language

Inevitable changes in computing applications over the years have resulted in a substantial growth of proprietary data formats. This has made exchanging data between programs difficult. In the past this task was relatively simple as most programs stored their data as text. Now, conversion modules are needed to allow applications to successfully transfer and comprehend data [20]. This chapter discusses a new technology that is set to revolutionize data handling and exchange.

Extensible Markup Language, abbreviated XML, is a language defined by the World Wide Web Consortium (W3C), the body responsible for developing and endorsing interoperable standards for the Web [53].

Markup languages are useful in many computer applications as they help to determine the way a document's content should be interpreted. The most familiar example is Hypertext Markup Language (HTML), a non-proprietary format based upon Standard Generalized Markup Language (SGML). HTML documents contain a set of predefined tags, or markup, describing how they should be displayed in a Web browser [55].

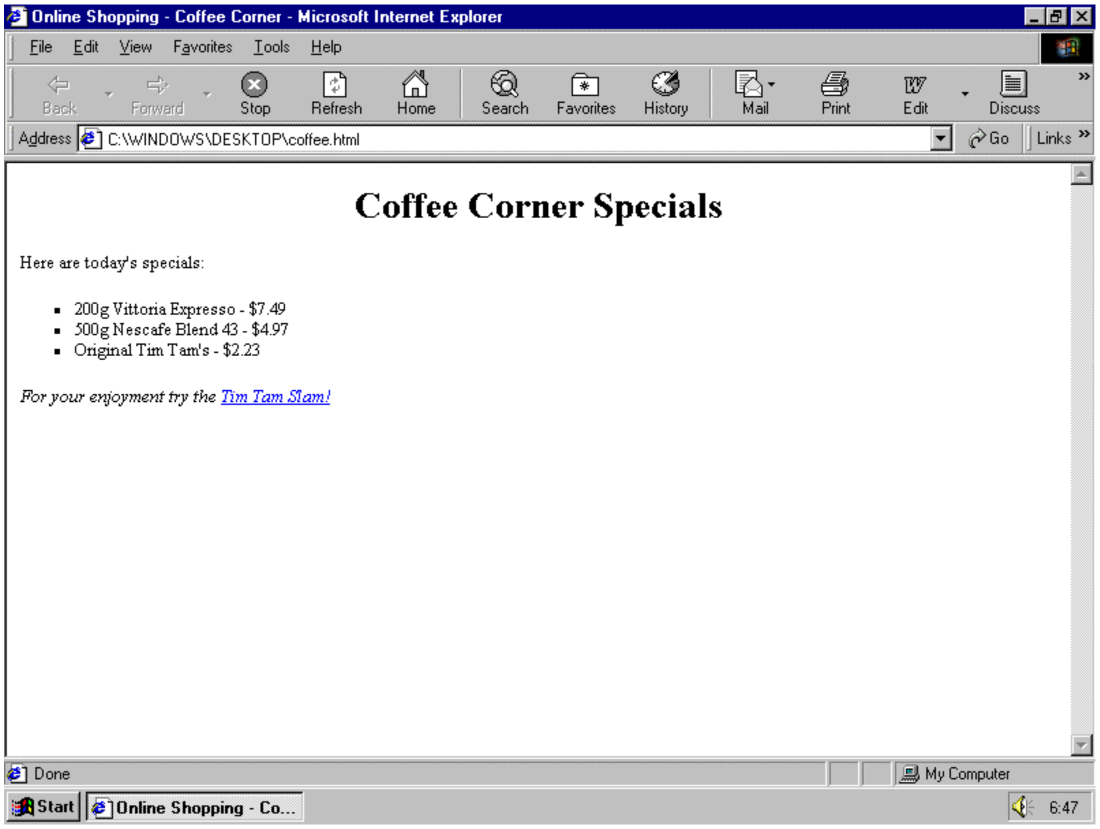
A sample HTML document is shown in Figure 7.

Figure 7: Simple HTML source code

```
<html>
   <head>
      <title>Online Shopping - Coffee Corner </title>
   </head>
   <body>
      <center>
         <h1>Coffee Corner Specials </h1>
      </center>
      <p> Here are today's specials:
      <ul>
         <li>200g Vittoria Expresso  - $7.49
         <li>500g Nescafe Blend 43   - $4.97
         <li>Original Tim Tam's      - $2.23
      </ul>
      <p> <i>For your enjoyment try the
          <a href="http://www.timtamslam.com"> Tim Tam Slam! </a>
          </i>
   </body>
</html>
```

The results of loading this page in a web browser are captured in the screenshot below:

Figure 8: HTML page viewed in Microsoft Internet Explorer 5.5


Online Shopping - Coffee Corner - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address C:\WINDOWS\DESKTOP\coffee.html
Go Links
Done
My Computer
Start Online Shopping - Co...
6:47

Coffee Corner Specials

Here are today's specials:

- 200g Vittoria Expresso - $7.49
- 500g Nescafe Blend 43 - $4.97
- Original Tim Tam's - $2.23

For your enjoyment try the Tim Tam Slam!

You will notice that the text “Coffee Corner Specials” is horizontally centered in the

Web page and this is because it is enclosed within the <center> tag. Markup is in more places than you think, not just on the Web. You may not realise it but even word processing documents are embedded with markup codes.

XML, as the name suggests, is another markup language and, like HTML, is an easier-to-use subset of SGML.
XML is extensible, meaning that you can define your own set of markup tags. This property makes XML a feasible candidate for describing data models whereas HTML is limited to data display. Let's consider a simple example to illustrate this and get a feel for the syntax and capabilities of XML.

Figure 9: XML example – video.xml

```
<?xml version="1.0"?>
<VideoStore>

  <TradingName> VideoCrazy </TradingName>

  <Description> Video Sales - not Hire </Description>

  <ContactDetails>
      <Address>
          <StreetNumber> 109 </StreetNumber>
          <StreetName> Shopping Crescent </StreetName>
          <Suburb> Shopsville </Suburb>
          <State> Australian Shopping Capital </State>
          <Postcode> 007 </Postcode>
      </Address>
      <Telephone/>
      <Email> staff@videocrazy.com.au </Email>
  </ContactDetails>

  <Videos>

      <Video serialNo="00000001">
          <Title> The Fugitive </Title>
          <Genre> Action </Genre>
          <Starring>
              <Actor firstName="Harrison" lastName="Ford"/>
              <Actor firstName="Tommy Lee" lastName="Jones"/>
          </Starring>
          <RunningTime>120</RunningTime>
          <Classification> M </Classification>
          <Rating> ***** </Rating>
          <Price> 24.95 </Price>
      </Video>

      <Video serialNo="0000002">
          <Title> Titanic </Title>
          <Genre> Drama </Genre>
          <Starring>
              <Actor firstName="Leonardo" lastName="DiCaprio"/>
              <Actor firstName="Kate" lastName="Winslet"/>
```

```
            </Starring>
            <RunningTime> 185 </RunningTime>
            <Classification> PG </Classification>
            <Rating> ** </Rating>
            <Price> 20.95 </Price>
        </Video>
        …
    </Videos>
</VideoStore>
```

This document describes a typical video store, in particularly, one concerned with video sales and not hire. An important detail to note is that this document is stored as text and thus is easily accessible. XML documents can be created using any text-editor like emacs, xedit, vi or Windows WordPad. However, specially designed XML editors are available to make this process a lot easier. Examples include XML Writer, XML Spy, XML Pro and Microsoft's free XML Notepad [20].

As we shall discover, XML documents are subject to many rules and constraints. The W3C Recommendation Specification for XML [54] provides a full account of this syntax, however a brief overview will be given here.

Two constraints are documented in the XML specification: well-formedness and validity.

The former is more important as it must be satisfied before an object can be considered as an XML document. For a discussion on the latter, please refer to Section 3.2 and 3.3. These sections introduce Document Type Definitions and XML Schemas respectively, which provide a way to specify the legal building blocks of XML documents.

An XML Parser, often referred to as an XML Processor, is responsible for reading documents and reporting violations of the constraints defined in the W3C specification. There are two forms of XML processors available, non-validating and validating. Non-validating processors will only report errors if the document is not well-formed. Validating processors perform an additional check to determine whether the document is valid (i.e. conforms to rules given in a Document Type Definition or XML Schema).

The W3C specification states that a document is well-formed if it satisfies the properties:

> *1. Taken as a whole, it matches the production labeled document*
> *2. It meets all the well-formedness constraints given in this specification*
> *3. Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.*

The first stipulation says that the object must match the "document" production (i.e. rule) defined as follows:

> *document ::= prolog element Misc**

This means the document is composed of three parts. Firstly a prolog, followed by a root element and then any number of miscellaneous productions.

It is possible for a document to be well-formed even if it has an empty prolog.

However, if a programmer wishes to include an XML declaration or a document type declaration they must do so in the prolog. Comments and other processing instructions are also permitted here. The main purpose of such declarations and processing instructions is to provide information to the XML processor. For example, the XML declaration indicates which XML version the document should conform to, which language encoding is used, and whether or not it is a standalone (self-contained) document. Figure 9 has a non-empty prolog part, due to the XML declaration on line 1.

XML documents are composed of one or more elements delimited by matching start and end tags. The only exception to this rule is an empty element. An element's content is any character data enclosed within matching tags and can include other markup. For example, the document in Figure 9 has an element called "Address" with content:

```
<StreetNumber> 109 </StreetNumber>
<StreetName> Shopping Crescent </StreetName>
<Suburb> Shopsville </Suburb>
<State> Australian Shopping Capital </State>
<Postcode> 007 </Postcode>
```

It also has an empty "Telephone" element, represented by the empty-tag <Telephone/>. Well-formed XML documents must have exactly one root element. All other elements must be enclosed in the root element and must nest properly within each other. <VideoStore> is the root element in Figure 9 and all elements in this example are properly nested. The following fragment illustrates an incorrect nesting of tags:

```
<root>
  <naughty> This example is WRONG as the tags do not nest properly
    <nesting> The closing tag for naughty should be after the closing tag for nesting.
  </naughty>
    </nesting>
</root>
```

Before progressing any further it will be beneficial to explain some terminology. In XML, elements are related as parents and children. In Figure 9, the root element VideoStore is a parent element of TradingName, Description, ContactDetails and Videos. Conversely, TradingName, Description, ContactDetails and Videos are child elements of VideoStore. TradingName and Description are sibling elements as they have the same parent.

An XML element may contain a set of attributes. These are name-value pairs and help provide extra information about an element. An attribute specification can occur in a start-tag or in an empty-tag. Each <Video> start-tag in Figure 9 has an attribute called serialNo whose value is surrounded by double quotes. The document also contains empty <Actor> elements with two attributes called firstName and lastName.

The third part of any well-formed XML document is called a miscellaneous section. This can include any number of XML comments, processing instructions or whitespace.

For an object to be an XML document the second property stipulates that all well-formedness constraints in the W3C specification be met. Essentially these are syntax rules that the document must adhere to. An exhaustive listing is clearly set out in the documentation provide by W3C, however it is worthwhile mentioning a select few here.

We have already discussed one constraint already: End tags must match start tags. The case sensitive nature of XML prevents tags such as </ThIs> and <THIS> from matching.

The next constraint requires that attribute specifications are unique: No attribute name should appear more than once in the same start-tag or empty-tag. A document containing the element <VEHICLE make="Ford" model="Laser" model="Elle McPherson"/> is not well-formed as it contains a duplicate attribute named "model".

Another rule expects that attribute values be enclosed in quotes. Double quotes are commonly used, however, single quotes are required if the attribute value contains double quotes. These examples are legal:

<student ID="9999999Z">
<student ID='9999999Z'>
<student name='Anna "The Great" Forrest'>

Yet, errors would be reported when processing the following:
<student name="Anna "The Great" Forrest">.
<student ID=9999999Z>

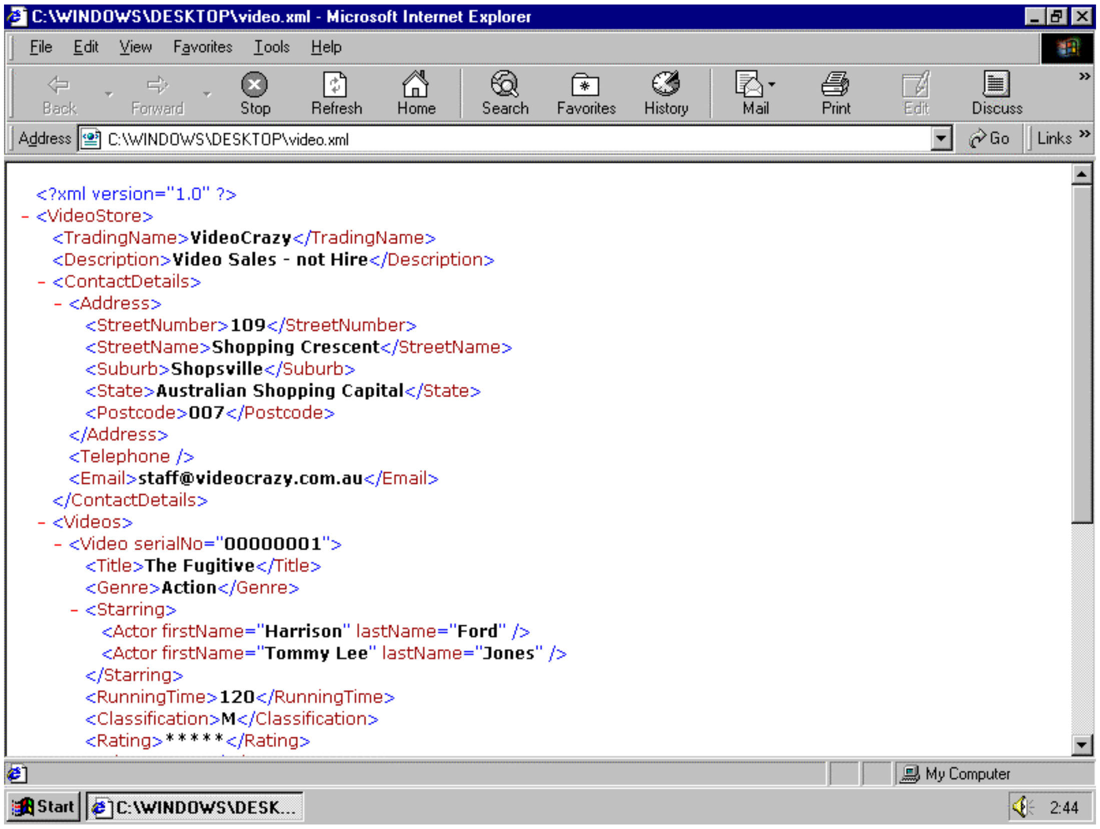
Finally, a document may be well-formed only if the parsed entities it references are also well-formed. Entities are XML's way of referring to a data storage unit. Unparsed entities may be text or binary data and are not required to satisfy any constraints. In comparison, a parsed entity contains replacement text and must be well-formed. Each entity reference in a document is replaced by the entity itself. That is, an entity reference causes the entity's data to be included in the referencing document. If the data included is not well-formed then it logically follows that the resulting document is not well-formed.

XML documents alone are not designed to do anything. Additional software is required to perform tasks such as data transmission, processing and display.

At the time of writing this paper, a complete general XML browser did not exist. Nevertheless, sufficient support is available for XML in Internet Explorer 5.5 and Netscape Navigator 6. Both browsers are able to display XML documents directly.

The next figure shows the outcome of loading video.xml from Figure 9 in Internet Explorer 5.5.

Figure 10: XML document viewed in Microsoft Internet Explorer 5.5


C:\WINDOWS\DESKTOP\video.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss
Address C:\WINDOWS\DESKTOP\video.xml
Go Links
My Computer
Start C:\WINDOWS\DESK...
2:44

```
  <?xml version="1.0" ?>
- <VideoStore>
    <TradingName>VideoCrazy</TradingName>
    <Description>Video Sales - not Hire</Description>
  - <ContactDetails>
    - <Address>
        <StreetNumber>109</StreetNumber>
        <StreetName>Shopping Crescent</StreetName>
        <Suburb>Shopsville</Suburb>
        <State>Australian Shopping Capital</State>
        <Postcode>007</Postcode>
      </Address>
      <Telephone />
      <Email>staff@videocrazy.com.au</Email>
    </ContactDetails>
  - <Videos>
    - <Video serialNo="00000001">
        <Title>The Fugitive</Title>
        <Genre>Action</Genre>
      - <Starring>
          <Actor firstName="Harrison" lastName="Ford" />
          <Actor firstName="Tommy Lee" lastName="Jones" />
        </Starring>
        <RunningTime>120</RunningTime>
        <Classification>M</Classification>
        <Rating>*****</Rating>
```

You may notice that even the tags are displayed in the web page in Figure 10. This is because we have not told the browser how to treat the content of the user-defined elements. So how can we sensibly display the information from a XML document on the web?

A common solution is to use style sheet languages such as Cascading Style Sheets (CSS) or eXtensible Stylesheet Language (XSL). These languages allow you to dictate how each element in the XML document should be formatted.

XSL also contains a transformation language, called XSLT, which is capable of converting XML documents into other document types. Transforming XML to HTML for display purposes is probably one of the most common uses of XSLT today. Another popular use, fundamental to this project work, is XSLT's ability to transform one data model into another data model. Michael Kay has published a comprehensive book, XSLT Programmer's Reference, which contains many real-world applications of XSLT [23]. Online support and discussion groups for the book are available at http://www.wrox.com. More information about XSLT, as well as concrete example transformations, is given in Section 3.4.

Internet Explorer 5.5 and Netscape Navigator 6 have good support for style sheets, with partial support for XSLT. This means that XML may need to be transformed to HTML outside of the browser using a fully conformant XSLT 1.0 Processor.

Another solution is to use a scripting language like JavaScript to load, access and display XML data from within the HTML program code. Internet Explorer 5.5 supports the scripting languages JavaScript and VBScript. As this topic is not directly applicable to this project, any interested readers are directed to the discussion in Inside XML [20].

3.2 DTDs: Document Type Definitions

Because XML allows you to create your own set of tags, it is also up to you to state the syntax they must follow. For example, can a <RunningTime> element be the child of a <Video> element? Is it necessary for a <Video> element to contain a <Title> element? Or, is the serialNo attribute of a <Video> element mandatory or optional? There are two mechanisms available for specifying the legal structure and syntax of XML documents - a Document Type Definition (DTD) or an XML Schema.

Initially developed for SGML, DTDs have been around for years. In fact, formal rules regarding their use are built into the XML 1.0 Recommendation [38]. XML Schema, on the other hand, is new to the scene - officially released as a W3C Recommendation in May 2001 [57,58,59].

To appreciate their similarities and differences, the next two sections are devoted to a detailed discussion of DTDs and XML Schemas.

The simplest way to explain DTDs, as with most concepts in Computer Science, is by examples. Recall Figure 9 from Section 3.1 - the XML document describing a video store. Figure 11 below depicts a suitable DTD for validating such a class of XML documents.

Figure 11: DTD example – video.dtd

```
<!DOCTYPE VideoStore [

    <!ELEMENT VideoStore (TradingName, Description?, ContactDetails, Videos)>
    <!ELEMENT TradingName (#PCDATA)>
    <!ELEMENT Description (#PCDATA)>
    <!ELEMENT ContactDetails (Address, Telephone, Facsimile?, Email?)>
    <!ELEMENT Address (StreetNumber, StreetName, Suburb, State, Postcode)>
    <!ELEMENT Telephone (#PCDATA)>
    <!ELEMENT Facsimile (#PCDATA)>
    <!ELEMENT Email (#PCDATA)>
    <!ELEMENT StreetNumber (#PCDATA)>
    <!ELEMENT StreetName (#PCDATA)>
    <!ELEMENT Suburb (#PCDATA)>
    <!ELEMENT State (#PCDATA)>
    <!ELEMENT Postcode (#PCDATA)>
    <!ELEMENT Videos (Video*)>
    <!ELEMENT Video (Title, Genre?, Starring, RunningTime, Classification?,
                        Rating?, Price)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Genre (#PCDATA)>
    <!ELEMENT Starring (Actor+)>
    <!ELEMENT Actor (#PCDATA)>
    <!ELEMENT RunningTime (#PCDATA)>
    <!ELEMENT Classification (#PCDATA)>
    <!ELEMENT Rating (#PCDATA)>
    <!ELEMENT Price (#PCDATA)>

    <!ATTLIST Video serialNo ID #REQUIRED>
    <!ATTLIST Actor firstName CDATA #REQUIRED>
    <!ATTLIST Actor lastName CDATA #REQUIRED>

]>
```

A DTD is made up of the following markup declarations

- Element Type Declarations
- Attribute-List Declarations
- Entity Declarations
- Notation Declarations

Figure 11 demonstrates the use of both element type and attribute-list declarations. Element type declarations define the legal elements of a XML document and specify the content they are permitted. These declarations are of the form:

    <!ELEMENT element-name content>

An element's content may be declared as:

- EMPTY.
  E.g. A declaration such as <!ELEMENT Pen EMPTY> is equivalent to <Pen />
- #PCDATA.
  The term PCDATA means parsed character data. Elements that are to store character data only should be declared in the DTD as #PCDATA. Note that markup is not permitted within these elements.
  Figure 11 restricts <TradingName> to character data using the declaration:

      <!ELEMENT TradingName (#PCDATA)>.
- ANY.
  Element’s declared with content ANY can contain character data, markup or both.
  E.g. <!ELEMENT Flexible ANY>.
- Or a Child Element List.
  Relationships between elements can be established using this content model.
  You can stipulate which child elements are allowed for a parent element by giving their names in parenthesis as follows:

      <!ELEMENT parent (child)> or
      <!ELEMENT parent (child1, child2, child3, …)

  For example, to specify that a <Milk> element be a child of a <Cow> element you declare:

      <!ELEMENT Cow (Milk)>
      <!ELEMENT Milk (#PCDATA)>

  It may be desirable for the <Cow> element to be composed of a sequence of children such as <Milk> and <Beef>.
  This is achieved by the following declarations:

      <!ELEMENT Cow (Milk, Beef)>
      <!ELEMENT Milk (#PCDATA)>

<!ELEMENT Beef (#PCDATA)>

Producing valid XML documents requires that the children elements appear in exactly the same order as they are defined in the element type definition. Therefore, <Cow> <Milk> Shake </Milk> <Beef> Burger </Beef> </Cow> is valid and <Cow> <Beef> Burger </Beef> <Milk> Shake </Milk> </Cow> is not. The declaration for <Cow> above requires that exactly one <Milk> element is present followed by exactly one <Beef> element. DTDs provide a mechanism, similar to regular expressions, for denoting the number of occurrences of child elements.

child+	means one or more occurrences of <child>
child*	means zero or more occurrences of <child>
child?	means zero or one occurrences of <child>
child	means exactly one occurrence of <child>

From Figure 11, we can deduce a general organization of the root element <VideoStore>. There must be exactly one <TradingName> element, which “may” be followed by a <Description>. After this there should be one <ContactDetails> element, then one <Videos> element. The <Videos> element can contain zero or more occurrences of <Video>.

In a DTD, you can declare attributes with an attribute-list declaration: <!ATTLIST element-name attribute-name data-type default-value>. Three attributes are defined in video.dtd

```
<!ATTLIST Video serialNo ID #REQUIRED>
<!ATTLIST Actor firstName CDATA #REQUIRED>
<!ATTLIST Actor lastName CDATA #REQUIRED>
```

The <Video> element has one attribute called serialNo. The attributes firstName and lastName both belong to <Actor>.

The data-type and default-value constructs help the parser determine whether the attribute value given is legal. Attribute data types fall into three categories: string types (i.e. CDATA), tokenized types (i.e. ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS), or enumerated types.
CDATA, meaning character data, is the most common attribute type. The value of CDATA attributes can be set to any string of text not including markup. The abovementioned attributes of <Actor> expect values of this type.
Another very important attribute type is ID. Applications typically use ID to make elements unique. Each video in the video store example is identified by a serialNo attribute. This may be useful to distinguish multiple copies of the same video. A validating parser will report a violation if two or more of these attributes have the same value.
Enumerated types provide a list of values that an attribute can have. Any value provided that is not in the list will cause an error.

The default-value allows developers to specify what should occur if no attribute value is present in the XML document. There are four choices available:

- #REQUIRED

- #IMPLIED
- #FIXED value
- default

#REQUIRED signifies that an attribute value must be presented by the user. The #IMPLIED keyword indicates that no default value is supplied and the author is free to provide one. An attribute's value can be permanently fixed if declared using the #FIXED keyword. Lastly, you can specify a default value to be used if the attribute is omitted.

It is also possible to state entity and notation declarations in a DTD. The concept of an entity was discussed previously in Section 3.1. Entities are declared in a DTD using <!ENTITY name definition> and can be referenced in a document using an entity reference, &name;.
A DTD that describes financial statements would find it useful to include an entity declaration for the end of financial year.
<!ENTITY yrend "30th June, 2001">
Any reference to &yrend; in the financials would be replaced by the text 30th June, 2001 by the XML parser.
The purpose of a notation declaration is to indicate the format of any non-XML data (e.g. image formats like GIF and JPG).

To use a DTD, an XML document must have a <!DOCTYPE> declaration in its prolog. The DTD can be included in the XML source file, which is an ideal way to verify your own data. Alternatively, a DTD can be external, which is an effective solution for a group of people wanting to exchange data.
The basic syntax for an internal declaration is:
<!DOCTYPE rootname [DTD]>
The result of incorporating the DTD from Figure 11 into the document video.xml can be seen in Appendix B.

3.3 XML Schemas

A recent alternative to the classic DTD is XML Schemas. At the expense of complexity, Schemas are far more precise and powerful than DTDs. The sheer volume of documentation can often be a reliable measure of complexity. The W3C describes XML Schemas in three documents:

- http://www.w3.org/TR/xmlschema-0/ Part 0 - XML Schema primer
- http://www.w3.org/TR/xmlschema-1/ Part 1 - XML Schema structures
- http://www.w3.org/TR/xmlschema-2/ Part 2 - XML Schema data types

The primer provides an easily approachable introduction to the Schema definition language. A more formal definition is contained in parts 1 and 2 of the XML Schema specification.

An important difference between XML Schemas and DTDs is that Schemas are XML documents themselves. Therefore, they can be managed by XML authoring tools, e.g. XML Pro, and even styled or transformed using XSL [6]. The XML syntax of Schemas makes their descriptions far more extensible than the DTD syntax [38].

The vastly improved typing system of XML Schemas is perhaps the most significant change. In terms of validation, this allows Schemas more control over the *content* of documents, not just their structure, making them more suitable for data intensive applications. There are many built-in data types supported by Schema including:

- Document-oriented types of DTDs
- Data-oriented types (integer, float, string, date, time, short, boolean etc).

As we shall see, the programmer can create additional data types in Schema.

Let’s revisit the video store example again. A full Schema definition for video store documents can be found in Figure 12.

Figure 12: XML Schema example – video.xsd

```
<xsd:schema xmlns:xsd=”http://www.w3.org/1999/XMLSchema”>

    <xsd:annotation>
        <xsd:documentation>
            Schema for Video Store Documents
            Created by: Programmer X etc
        </xsd:documentation>
    </xsd:annotation>

    <xsd:element name=”VideoStore” type=”VideoStoreType”/>

    <xsd:complexType name=”VideoStoreType”>
        <xsd:element name=”TradingName” type=”xsd:string”/>
        <xsd:element name=”Description” type=”xsd:string” minOccurs=”0”/>
        <xsd:element name=”ContactDetails” type=”ContactDetailsType”/>
        <xsd:element name=”Videos” type=”VideosType”/>
    </xsd:complexType>

    <xsd:complexType name=”ContactDetailsType”>
        <xsd:element name=”Address” type=”AddressType”/>
        <xsd:element name=”Telephone” type=”xsd:long”/>
        <xsd:element name=”Facsimile” type=”xsd:long” minOccurs=”0”/>
        <xsd:element name=”Email” type=”xsd:string” minOccurs=”0”/>
    </xsd:complexType>

    <xsd:complexType name=”AddressType”>
        <xsd:element name=”StreetNumber” type=”xsd:integer”/>
        <xsd:element name=”StreetName” type=”xsd:string”/>
        <xsd:element name=”Suburb” type=”xsd:string”/>
        <xsd:element name=”State” type=”xsd:string”/>
        <xsd:element name=”Postcode”>
            <xsd:simpleType base=”xsd:integer”>
                <xsd:pattern value=”\d{4}”/>
            </xsd:simpleType>
        </xsd:element>
    </xsd:complexType>

    <xsd:complexType name=”VideosType”>
        <xsd:element name=”Video” minOccurs=”0” maxOccurs=”unbounded”/>
            <xsd:complexType>
                <xsd:element name=”Title” type=”xsd:string”/>
                <xsd:element name=”Genre” type=”xsd:string” minOccurs=”0”/>
                <xsd:element name=”Starring” type=”StarringType”/>
                <xsd:element name=”RunningTime” type=”xsd:short”/>
                <xsd:element name=”Classification”>
                    <xsd:simpleType base=”xsd:string”>
                        <xsd:enumeration value=”G”/>
                        <xsd:enumeration value=”PG”/>
                        <xsd:enumeration value=”M”/>
                        <xsd:enumeration value=”MA”/>
                        <xsd:enumeration value=”R”/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name=”Rating” minOccurs=”0”>
                    <xsd:simpleType base=”xsd:string”>
                        <xsd:enumeration value=”*”/>
                        <xsd:enumeration value=”**”/>
                        <xsd:enumeration value=”***”/>
                        <xsd:enumeration value=”****”/>
                        <xsd:enumeration value=”*****”/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:element name=”Price” type=”xsd:float”/>
                <xsd:attribute name=”serialNo” type=”serialNoType”
                        use=”required”/>
            </xsd:complexType>
        </xsd:element>
    <xsd:complexType>

    <xsd:complexType name=”StarringType”>
        <xsd:element name=”Actor” minOccurs=”1” maxOccurs=”unbounded”>
            <xsd:complexType>
                <xsd:attribute name=”firstName” type=”xsd:string”
                        use=”required”/>
                <xsd:attribute name=”lastName” type=”xsd:string”
                        use=”required”/>
            </xsd:complexType>
        </xsd:element>
    </xsd:complexType>

    <xsd:simpleType name=”serialNoType” base=”xsd:ID”>
        <xsd:pattern value=”\d{8}”/>
    </xsd:simpleType>
</xsd:schema>
```

This example appears daunting, however it is not as bad as it seems. The first observation to make is that it is a well-formed XML document. The root element is <xsd:schema> and contains an attribute declaring a target namespace for the schema tags – “http://www.w3.org/1999/XMLSchema”. By convention, Schema tags are usually prefixed by *xsd*.

In Schemas, you declare elements and attributes using the <xsd:element> and <xsd:attribute> tags respectively. You can specify their type, whether built-in or user-defined, when you declare them. In the video store schema, video.xsd, line 10 declares a <VideoStore> tag of type “VideoStoreType” – a user-defined type. Line 27 defines the element <StreetNumber> to be an integer – one of Schema’s built-in data types.

Types in Schema are either simple or complex. The types that come built into the Schema specification are simple, however, users can create simple or complex types by using the <xsd:simpleType> and <xsd:complexType> declarations respectively.

Complex types are used to define elements that enclose subelements or have attributes, whereas the purpose of simple types is to hold data like numbers, strings, or dates/times.
Note that an attribute must not be structured and hence cannot be represented by a complex type. In a nutshell, complex types have replaced the child element list declarations of a DTD and simple types have replaced #PCDATA.

<VideoStore> is the root element of a video store document and, as previously noted, it can have four subelements. Please refer to Line 12 in Figure 12 to examine how this relationship is expressed using a complex type declaration named “VideoStoreType”. The subelements <TradingName> and <Description> are denoted by xsd:string – one of Schema’s predefined simple types. The remaining subelements contain markup and, similarly to their parent, are declared using complex types.

There are three attribute declarations in Figure 12 on lines 64, 72 and 73. The first of these is an attribute for the <Video> element named “serialNo”. The remaining attributes “firstName” and “lastName” occur in the complex type declaration for an <Actor>.

Schema developers can indicate if an attribute is mandatory or provide defaults, by using the ‘use’ and ‘value’ attributes of <xsd:attribute>. Possible values for the ‘use’ attribute are:

- required
- optional
- fixed
- default
- prohibited

The first four are analogous to the <!ATTLIST> default-values of a DTD, in order these are #REQUIRED, #IMPLIED, #FIXED value, and default. If an attribute’s use is fixed or default, then its value must be set using the ‘value’ attribute of <xsd:attribute>. If prohibited use is indicated, the attribute must not appear in the document at all. The presence of all attributes defined in Figure 12 is mandatory for a document to be considered valid.

Default values can also be specified for elements by using the ‘fixed’ or default’ attributes of <xsd:element>. The ‘fixed’ attribute is useful in cases where you need to describe data that can’t be altered, e.g. the date for the end of financial year <xsd:element name=”YearEnding2001” type=”xsd:date” fixed=”2001-06-30”/>. The ‘default’ attribute is used to provide a value for an element if it is not assigned a value by the user. For example, in most cases tax is charged at the rate of 10%, however, it can differ in some circumstances. It would be beneficial to declare this as a default rate so that documents need only specify a value for abnormal rates. <xsd:element name=”TaxRate” type=”xsd:float” default=”10.00”/>. Another example may be to specify a default language such as English.

In a DTD you could restrict the number of times an element could occur using operators such as ?, +, and *. Schemas provide similar functionality through the <xsd:element> attributes ‘minOccurs’ and ‘maxOccurs’. These attributes have a default value of 1 meaning that an element must appear exactly once. The following declaration ensures that an element, E, occurs at least X times and at most Y times, where X and Y are integers and X<=Y: <xsd:element name=”E” minOccurs=”X” maxOccurs=”Y” type=”…” />

Schemas provide what are called facets, to restrict the data that a <xsd:simpleType> can hold. For example, say you want to define an element called studentMark that can hold values between 0 and 100 inclusive. There are facets defined called xsd:minInclusive and xsd:maxInclusive that allow you to do this:

```
<xsd:element name=”StudentMark”>
    <xsd:simpleType base=”xsd:integer”>
        <xsd:minInclusive value=”0”/>
        <xsd:maxInclusive value=”100”/>
    </xsd:simpleType>
</xsd:element>
```

Other popular facets include <xsd:Length>, <xsd:minLength>, <xsd:maxLength>, <xsd:Pattern>, and <xsd:Enumeration>. Figure 12 uses the facet <xsd:Pattern> to indicate that a Postcode is to contain 4 digits. Therefore, the document in Figure 9 would be invalid as its Postcode has only 3 digits, “007”. The video schema supplies a list of possible values for the <Classification> and <Rating> elements using <xsd:enumeration>. For example, the only classifications allowed are G, PG, M, MA, and R.

Other things that make Schemas attractive include:

- The ability to inherit syntax from other schemas
- Good support for documentation
- Ability to create choices/ sequences/ attribute groups/ all groups

With choices, you declare a number of elements with the intention that only one of them will be chosen. Element order may be stipulated by declaring an element sequence, similarly to DTDs. The attribute group is used to group together all attribute definitions belonging to an element. An all group is used to specify a group of elements that may appear 0 or 1 times and in any order [20].

The DTD is by no means obsolete technology. They are simpler to learn than Schemas and can be much easier for humans to digest. Also, there is much more software support for DTDs as they have been around the longest. Having said this, XML Schemas are definitely the technology of choice if you need to validate data items.

3.4 XSLT: eXtensible Stylesheet Language Transformations

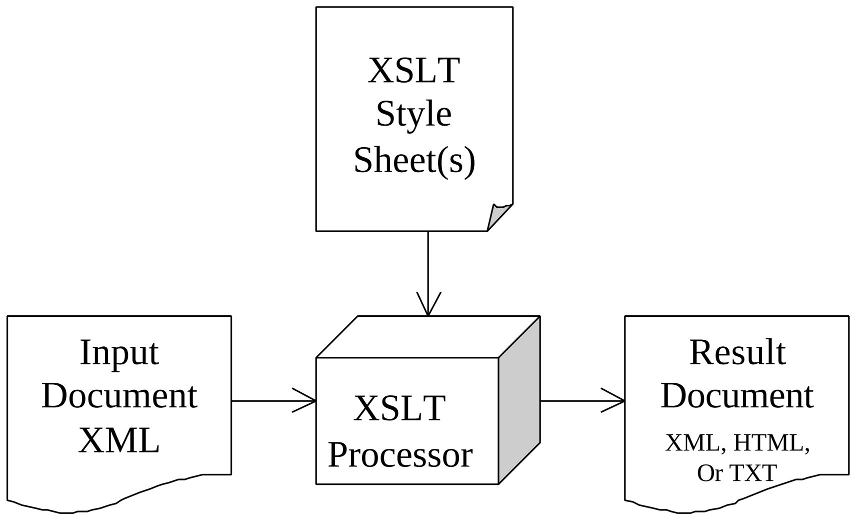
XSLT, which stands for Extensible Stylesheet Language: Transformations, is a language for transforming the structure of XML documents [23]. It was published as a W3C Recommendation on 16 November 1999, and fits into the large XML family of standards. The XSLT specification can be found at http://www.w3.org/TR/xslt.

There are two main reasons why XML transformations may be necessary:

- To communicate with human readers. Humans can "read" raw XML files, however, this is seldom likely to be the intended method of communication. Most of the time, data needs to be transformed into a suitable presentation format such as HTML, PDF, or even audible sound. These types of transformations are performed by XSLT.
- To transfer data between different applications using different data models. For example, loading data into an application might require the document to be in the form of an SQL script. XLST can be used to transform the data to the required SQL format. The rationale behind using XSLT for this project was to handle the transformation between multiple image data models.

Having established the importance of transformations, let's consider how XSLT goes about processing the XML documents into the required output.
In XSLT, required transformations are expressed in a stylesheet as a set of rules. These rules describe what output should be generated when particular patterns are encountered in the source XML document. Note that it is the role of an XSLT processor to apply the stylesheet to a source XML document and produce a result document, as shown in Figure 13.

Figure 13: Transformation Process Overview



There are several XSLT processors available. The Oracle XML Developer's Kit

(XDK) consists of XML parsers for Java and C++ that include XSLT processors. This project made use of Oracle's XDK Java XSLT Processor, which can be downloaded for free from the Oracle web site at http://technet.oracle.com/tech/xml. Saxon is another Java implementation of XSLT developed by Michael Kay, the author of XSLT Programmer's Reference. The source code can be downloaded at http://users/iclway.co.uk/mhkay/saxon/. Other popular processors include Apache's Xalan (http://xml.apache.org), Sun's Transformation API for XML (http://java.sun.com/xml/download.html) and Microsoft's MSXML3 product (http://msdn.microsoft.com/xml).

We are now ready to examine an example of using XSLT to transform XML. Again, consider the XML document from Figure 9. Let's assume the "VideoCrazy" store wants to display a table of their products in a browser. This requires an XML to HTML conversion and one possible XSLT stylesheet to do this is shown in Figure 14.

Figure 14: XSLT stylesheet – video.xsl

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

<xsl:template match="/">
    <html>
    <head>
        <title> Video Catalog for <xsl:value-of select="VideoStore/TradingName"/> </title>
    </head>
    <body>
        <center>
            <h1> <xsl:value-of select="//TradingName"/>'s Catalog Summary </h1>
        </center>
        <table border="2">
            <tr>
                <td>TITLE</td>
                <td>GENRE</td>
                <td>RUNNING TIME</td>
                <td>CLASSIFICATION</td>
                <td>SELLING PRICE</td>
            </tr>
            <xsl:for-each select="VideoStore/Videos/Video">
                <tr>
                    <td><xsl:value-of select="Title"/></td>
                    <td><xsl:value-of select="Genre"/></td>
                    <td><xsl:value-of select="RunningTime"/></td>
                    <td><xsl:value-of select="Classification"/></td>
                    <td><xsl:value-of select="Price"/></td>
                </tr>
            </xsl:for-each>
        </table>
    </body>
    </html>
</xsl:template>
</xsl:stylesheet>
```

You can link a XSLT stylesheet to a source document by including a<?xml-stylesheet?> processing instruction in the document's prolog. For example, you can specify that the stylesheet video.xsl should be used to transform the video.xml document by inserting the following instruction into the prolog of video.xml:

<?xml-stylesheet type="text/xsl" href="video.xsl"?>

However, this processing instruction is not mandatory for a transformation to be carried out by a XSLT processor. Processors that operate through a command line interface can determine the stylesheet(s) to apply from command line arguments. The <?xml-stylesheet?> instruction is mainly used to perform transformations within browsers without requiring extra scripting [23].

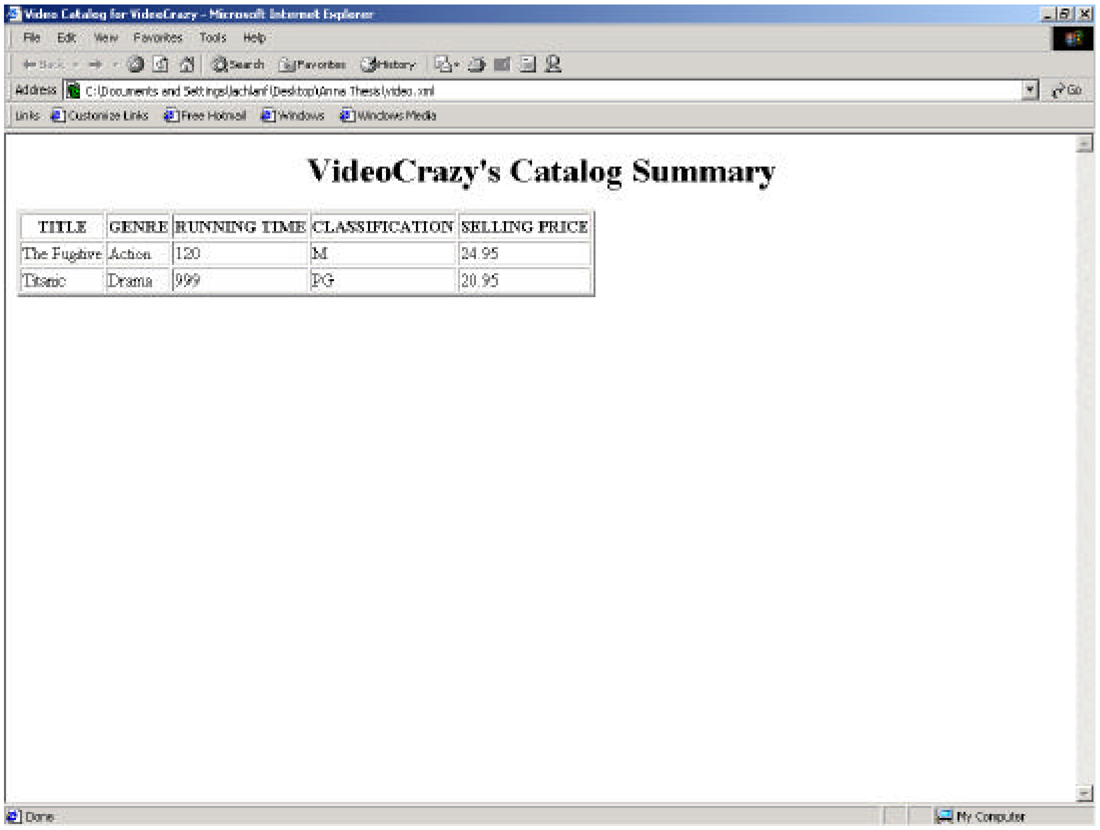
Microsoft provides an implementation for XSL in Internet Explorer 5 and 5.5 known as MSXSL. But this does not support the XSLT 1.0 standard because it was developed when XSL was only a W3C Working Draft. The changes made to XSLT before it became a recommendation have resulted in a dialect of MSXSL that bears little resemblance to the standard.

To successfully load and transform XML documents in IE5 and 5.5 you must either

- Use the WD-xsl dialect of MSXSL to create your stylesheets. The namespace URI of the <xsl:stylesheet> element should be http://www.w3.org/TR/WD-xsl
- Download and install Microsoft's MSXML3, which provides a full implementation for both dialects (WD-xsl and XSLT 1.0).

Alternatively you could transform the document into HTML before loading it in IE using any compliant XSLT 1.0 processor like Saxon.
Note that a XSLT 1.0 processor must process the stylesheet in Figure 14 because it declares the namespace URI http://www.w3.org/1999/XSL/Transform.
Whatever approach you take will result in the following web page:

Figure 15: Results of XML to HTML Transformation


Video Catalog for VideoCrazy - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address
Links Customize Links Free Hotmail Windows Windows Media
Go
Done
My Computer

C:\Documents and Settings\lachlanf\Desktop\Anna Thesis\video.xml

VideoCrazy's Catalog Summary

TITLE	GENRE	RUNNING TIME	CLASSIFICATION	SELLING PRICE
The Fugitive	Action	120	M	24.95
Titanic	Drama	999	PG	20.95

A stylesheet is represented by the <xsl:stylesheet> element of the XSLT namespace. The <xsl:stylesheet> must be the outermost element of every stylesheet module. It is mandatory to state the version of XSLT being used and to declare the XSLT namespace. There is an alias element, <xsl:transform>, that serves the same purpose. The immediate child elements of an <xsl:stylesheet> element are referred to as top-level elements and may fall into three categories: XSLT-defined, vendor-defined or user-defined. This paper only describes a selection of the XSLT-defined elements, i.e. those that form part of the XSLT namespace. A complete list of top-level elements may be obtained from the W3C XSLT Specification, http://www.w3.org/TR/xslt.

The XSLT specification has two top-level elements <xsl:import> and <xsl:include> that allow a modular development of a stylesheet program. For example, the behaviour of stylesheet A may be inherited in stylesheet B if B has defined a top-level include or import element that links A to itself. If A was imported by B, the definitions of B will have higher precedence that those in A. However, if B includes A then all definitions have equal precedence. These elements make stylesheet development easier as an object-oriented approach may be adopted and a library of reusable components can be created.

The most important top-level element in a stylesheet is <xsl:template>. It is within this element that rules are specified to transform a source tree into a result tree. A template will be invoked either by pattern matching, or explicitly by name. Commonly a template element has a match attribute that determines which nodes are eligible to be

processed by this template rule. For example, <xsl:template match=”CHAPTER[@title=’History’]>
  <xsl:text> Hello World </xsl:text>
</xsl:template>
would match the CHAPTER elements in a source tree that have a title attribute equal to “History” and create the text node with content “Hello World” in the result tree. If the match attribute is absent, <xsl:template> must contain a name attribute. Two low-level elements are used to control invocation of templates and these are <xsl:apply-templates> (for pattern matching) and <xsl:call-templates> (for explicitly named templates). A number of low-level elements can be used within a template body to assist in the formation of a result tree. These include:

- Elements that generate output nodes such as <xsl:value-of>, <xsl:element>, <xsl:attribute>, <xsl:comment> and <xsl:text>. Figure 14 demonstrates the use of <xsl:value-of>.
- Elements that copy nodes from a source document to a result document, namely <xsl:copy> and <xsl:copy-of>.
- Elements for conditional processing and iteration such as <xsl:if>, <xsl:choose>, <xsl:when>, <xsl:otherwise> and <xsl:for-each>. The element <xsl:for-each> was used in Figure 14 to iterate over each Video in the Video Store.

Variables and parameters are defined in a stylesheet using the <xsl:param> and <xsl:variable> elements. The <xsl:param> element may appear at the top-level of a stylesheet to define global parameters or as a lower-level element to define local parameters to be passed to a specific template. A name attribute must be present for the parameter and a default value may be specified using the select attribute. Parameters may be used to pass in extra information to assist the processor e.g. which part of the source document to process [23]. Parameter passing may be achieved via command-line arguments or the <xsl:with-param> element. Similarly, variable declarations can be global (top-element) or local (i.e. defined as a child of a template element). Once variables in XSLT are declared they cannot be changed. They are useful to define values that will be used multiple times in a stylesheet and avoid the need to recalculate results.

There is also a top-level element called <xsl:output> to control the serialization process, i.e. the stage where a result tree is written to a serial output file. This element has a method attribute to specify the format of the result document (such as xml, html, or text). Serialization is not mandatory for an XSLT processor. Some processors make the results available via a DOM (Document Object Model) API where users must traverse the nodes in the result tree using method calls. However, most XSLT processors support both the DOM and serialization.

Another top-level element worthy of note is the <xsl:script> element, which allows an author to write extension functions in Java or JavaScript. Some reasons why you might want to do this are summarized below:

- Access system services not directly available via XSLT. These may include determining the current date, writing to a log file or accessing trigonometric functions.

- Perform complex processing that is cumbersome to express in XSLT. For example, date manipulations or a conversion between different geospatial coordinate systems.

The main purpose for the <xsl:script> element is to tell the processor where to find the extension function implementations referenced in the stylesheet and what language the implementation uses. The <xsl:script> element may contain a src attribute, in which case the extension function implementation is external and specified by a URI. In this case the <xsl:script> element must be empty. However, if no src attribute is present, the element content of <xsl:script> contains the implementation. An example of its use is in Figure 31.

Many standard functions are provided in the XSLT specification for use in expressions. Some of the predefined functions are characterized below:

- Data type casting (boolean, number, string)
- Arithmetic number manipulation (ceiling, floor, round)
- String manipulation (concat, contains, starts-with, substring, translate, etc)
- Summation operators (sum, count)
- Boolean operators (false, true, not)
- Retrieving node names and identifiers (generate-id, local-name, name, namespace-uri)
- Node finding (document, key, id)
- Context information (current, last, position)

3.5 Oracle XML Developer Kit (XDK) for Java

The Oracle XML Developer Kit for Java consists of a number of components to assist in the development of XML applications. The components can be evaluated for free and are available on Oracle technet website, http://technet.oracle.com/tech/xml. No source code is provided, only a binary distribution.

The components utilized for this work included the XML Schema Processor (which is based on the Oracle XML Parser v2 component of this distribution), the XSLT Processor and the XML SQL Utility (XSU). As mentioned in Section 3.3, XML Schemas are a relatively new addition to the XML toolbox and until recently had little software support [47]. Oracle has introduced one of the first implementations for XML Schemas [23]. The XML Schema Processor supports the parsing and validation of XML files against an XML Schema definition file. The XSLT Processor supports the XSLT language for transforming XML documents. Lastly, the XML SQL Utility provides a simple API for loading data from an XML document into a database and generating an XML document as the result of an SQL query. There are two classes forming the XSU API: OracleXMLSave and OracleXMLQuery.

OracleXMLSave is the class enabling data from an XML document to be mapped to object-relational tables or views. The user must first create an instance of this class by passing in a database connection and table name to the constructor method. After that, insert, update and delete operations may be performed on the table.

OracleXMLQuery performs the generation of XML given an SQL query. In the generated XML, the rows returned by the SQL query are enclosed in a <ROWSET> element; a <ROW> tag is wrapped around each individual hit.

Consider the “student” table in Figure 16. Assume we have an XML document called students.xml containing data about students. The OracleXMLSave class can insert the entries from students.xml into the student table via the code in Figure 17. An example of generated XML from the query “select * from student” is shown in Figure 18.

Figure 16: Database schema for student

```
CREATE TABLE student (
        ID int,
        NAME char(30),
        COURSE char(30)
);
```

Figure 17: OracleXMLSave – Insert Processing

```
import oracle.xml.sql.dml.OracleXMLSave;
import java.sql.Connection;
…
  …
  Connection my_conn = new Connection(…..);
  OracleXMLSave my_save = new
                              OracleXMLSave(my_conn, “student”);
  my_save.insertXML(“students.xml”);
  my_save.close();
  …
…
```

Figure 18: Generation of XML from table using OracleXMLQuery

```
<?xml version=”1.0”?>
<ROWSET>
        <ROW num=”1”>
                <ID>001</ID>
                <NAME>Bloggs</NAME>
                <COURSE>Arts</COURSE>
        </ROW>
        <ROW num=”2”>
                <ID>398</ID>
                <NAME>Smith</NAME>
                <COURSE>Computing</COURSE>
        </ROW>
        <!-- additional rows -->
</ROWSET>
```

Chapter 4

Data Model Definitions and Conversions

The intent of Chapter 3 was to explain theoretical concepts for a subset of the XML family of standards. This chapter aims to show how these standards can be utilized to define and translate between different data models for real world examples.

4.1 Using XML for Data Format and Exchange

There are a number of application domains that already use XML-related technologies to describe data models. One example is music publishing on the Internet. Recordare is a company that is developing MusicXML technology to represent musical scores and sheet music [36]. The MusicXML language consists of two Document Type Definitions, partwise.dtd and timewise.dtd, which can be downloaded from the Recordare website, http://www.recordare.com. The Partwise DTD represents a music score by part/instrument whereas the Timewise DTD models the score by time/measure. Recordare have also provided XSLT stylesheets to convert the Partwise format into the Timewise format and vice versa. Let's assume we want to store a number of different music scores in a database. By convention, we elect to setup the database using the Partwise model. Suppose we want to add the following new items to the database: Score A (in Partwise format), Score B (in Timewise format) and Score C (in MIDI format). Before each score can be inserted it must be converted to the underlying database schema (Partwise) and an appropriate SQL insertion command must be generated (assuming the database uses SQL). The XSLT stylesheet from Recordare can convert Score B to the Partwise format, but a MIDI to Partwise stylesheet must be created to convert Score C. Once the scores are documents that conform to the Partwise model, Oracle's XML SQL Utility can be used to insert them into the database directly. Retrieving the scores in different formats requires stylesheets that can convert from the Partwise model into the Timewise model and MIDI model respectively.

Another example pertains to e-commerce applications. XML DTDs and Schemas have been used to create a number of formats for exchanging business data. These include Extensible Business Reporting Language (XBRL), the Market Data Definition Language (MDDL) and the Interactive Financial Exchange (IFX) [6].

There is an excellent registry of public XML Schemas on the Internet at http://www.xml.org/xml/registry.jsp where you will observe a vast range of data models created for a wide range of purposes.

The remaining sections of this chapter focus on the design of geospatial data models and their conversion. Two models are described, namely OLDA (the database schema used in the OLGAS system) and ORTHO. The decision to work with the ORTHO model was made based on the complexity of the other models like GMS-5.

4.2 Design of the Primary Data Model (OLDA)

This section details the process of designing an XML-based data model for OLDA. OLDA is the primary data model used in the OLGAS database. The OLDA model is a very basic data schema for describing geospatial image data. However, it contains the main fields that would be used by most geospatial image data models. The OLDA metadata fields and types are listed in Appendix A. The letter Q indicates fields that are queryable by clients.

The first step that needed to be addressed was whether to use a DTD or an XML Schema to define the data model. In the initial stages of this work, XML Schemas were a working draft and did not have the same degree of software support as DTDs (i.e. tools to validate XML documents against a Schema were scarce). However, XML Schemas have a superior data typing system that allows more restrictions to be placed on the data content of documents. For example, consider the IMAGEID field in Appendix A. The type of IMAGEID is a string with maximum length of 80. Figure 19 shows how this field could be declared using a DTD and an XML Schema. A DTD element declaration for IMAGEID could not handle such restrictions because the acceptable element content is any parsed character data (#PCDATA). However, types are easily constrained in Schemas using facets as we discovered in Section 3.3. The maxLength facet is used in this example to constrain the length of the string to 80 characters.

Figure 19: IMAGEID definition

DTD declaration for IMAGEID:

```
<!ELEMENT IMAGEID (#PCDATA)>
```

XML Schema declaration for IMAGEID:

```
<xsd:element name="IMAGEID">
        <xsd:simpleType base="xsd:string">
                <xsd:maxLength value="80"/>
        </xsd:simpleType>
</xsd:element>
```

In addition, DTDs cannot validate numeric data fields. Consider the HOUR field in Appendix A which is defined as a long. It also makes sense for this field to accept only numbers in the range 0 to 23. Using a DTD we cannot even guarantee that the content of the HOUR field is a number! However, by using XML Schema we can ensure that the HOUR field is a long and obeys the constraints (i.e. the range 0 .. 23). The possible declarations for HOUR can be found in Figure 20.

Figure 20: HOUR definition

DTD declaration for HOUR:

```
<!ELEMENT HOUR (#PCDATA)>
```

XML Schema declaration for HOUR:

```
<xsd:element name=”HOUR”>
        <xsd:simpleType base=”xsd:long”>
                <xsd:minInclusive value=”0”/>
                <xsd:maxInclusive value=”23”/>
        </xsd:simpleType>
</xsd:element>
```

The main disadvantage of using Schemas was their complexity and this will become clear when the overall structure of the OLDA Schema is discussed.

A number of possible representations could be developed for the OLDA data model. Common differences that can arise in development concern the representation of data as elements or attributes. One of the initial designs for the OLDA model is shown in Figure 21. This figure gives a Schema definition and a brief example of a conforming document. This design was centered on an attribute-based description of the data. The XML Schema was very compact and simple to write, however, this design presented some major flaws. For instance, it does not ensure the presence of particular OLDA fields. The only guarantee is that documents will contain at least one field. This is because the minOccurs attribute in the field element declaration is set to 1. Also, the design does not prevent duplicate fields (i.e. a document could contain two field elements with a fname attribute = IMAGEID). The major weakness of this model, however, was that conforming XML documents were specifying the type of fields and this is clearly incorrect. This design was promptly discarded and another solution was formed to overcome these flaws.

Figure 21: Initial design of OLDA Schema (Incorrect)

OLDA Schema:
```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

    <xsd:element name=”fields” type=”fieldsType”/>

    <xsd:complexType name=”fieldsType”>
        <xsd:element name=”field” minOccures=”1” maxOccurs=”unbounded”>
            <xsd:complexType>
                <xsd:attribute name=”fname” type=”xsd:string” use=”required”/>
                <xsd:attribute name=”ftype” type=”xsd:string” use=”required”/>
                <xsd:attribute name=”queryable” type=”xsd:boolean
                            use=”default” value=”False”/>
                <xsd:element name=”value” type=”xsd:string”/>
            </xsd:complexType>
        </xsd:element>
    <xsd:complexType>

</xsd:schema>
```

Conforming XML document:
```
<?xml version=”1.0”?>
<fields>
        <field fname=”IMAGEID” ftype=”char(80)” queryable=”True”>
                <value> 987-456-123 </value>
        </field>
        <field fname=”NBANDS” ftype=”short”>
                <value> 3 </value>
        </field>
        <!-- additional field elements -->
</fields>
```

The final OLDA Schema was fairly large and therefore is not provided in full. However a code extract is given in Figure 22 which captures the essential features of the design.

Figure 22: OLDA.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

    <xsd:element name="FIELDS" type="fieldsType"/>

    <xsd:complexType name="fieldsType">
        <xsd:all>

        <xsd:element name="IMAGEID">
            <xsd:complexType>
                <xsd:element name="value">
                    <xsd:simpleType base="xsd:string">
                        <xsd:maxLength value="80"/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:attribute name="primary" type="xsd:boolean"
                        use="fixed" value="False"/>
                <xsd:attribute name="queryable" type="xsd:boolean"
                        use="fixed" value="True"/>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name="HOUR">
            <xsd:complexType>
                <xsd:element name="value">
                    <xsd:simpleType base="xsd:long">
                        <xsd:minInclusive value="0"/>
                        <xsd:maxInclusive value=”23”/>
                    </xsd:simpleType>
                </xsd:element>
                <xsd:attribute name="primary" type="xsd:boolean"
                        use="fixed" value="False"/>
                <xsd:attribute name="queryable" type="xsd:boolean"
                        use="fixed" value="True"/>
            </xsd:complexType>
        </xsd:element>

        <xsd:element name=”NBANDS”>
            <xsd:complexType>
                <xsd:element name=”value” type=”xsd:positiveInteger”/>
                <xsd:attribute name=”primary” type=”xsd:boolean”
                        use=”fixed” value=”False”/>
                <xsd:attribute name="queryable" type="xsd:boolean"
                        use="fixed" value="False"/>
            </xsd:complexType>
        </xsd:element>

        <!-- other OLDA field definitions -->

        </xsd:all>
    </xsd:complexType>
</xsd:schema>
```

The root element of the OLDA Schema is <FIELDS> and is a complex structure containing element definitions for each field listed in Appendix A. The extract in Figure 22 shows element definitions for the IMAGEID, HOUR and NBANDS fields. The IMAGEID field is an identification value assigned by the producer, HOUR is part of the time that the image was taken and NBANDS is the number of spectral bands in the image (e.g. 4 for GMS-5, 3 for RBG photo). In the Schema, each field is declared as a complex structure which allows them to contain attributes. There are three components defined for each field:

- A “value” element
- A “primary” attribute
- A “queryable” attribute

The primary attribute is a boolean that indicates whether the field is a primary key of the database. The queryable attribute is a boolean that specifies if a client can query the field. Note that these attributes take on a fixed value and cannot be changed in a conforming document. The rationale for the two attribute definitions was to enable the creation of a stylesheet to transform the OLDA Schema into a series of SQL database initialization commands. More information about this is provided in Section 4.3.

Clearly, the value element was needed to store the data for a particular field (i.e. the parent element). Any constraints imposed on the data were specified in the <value> element via simpleType facets. The fragment of code in Figure 22 demonstrates how the constrictions of IMAGEID and HOUR were achieved in their respective value elements. The value element for NBANDS uses the positiveInteger type of Schema with no other restrictions.

The OLDA Schema makes use of an all-group. Each element within an all-group is allowed to occur 0 or 1 times and in any order. This decision prevented documents from containing duplicate fields and allowed documents to list the fields as they saw fit.

A conforming document for the OLDA data model is shown in Figure 23, and an invalid example is given in Figure 24.

Figure 23: Valid OLDA XML document

```
<?xml version=”1.0”?>
<FIELDS>
    <NBANDS>
        <value> 3 </value>
    </NBANDS>
    <IMAGEID>
        <value> 981279Z </value>
    </IMAGEID>
</FIELDS>
```

Figure 24: Invalid OLDA XML document

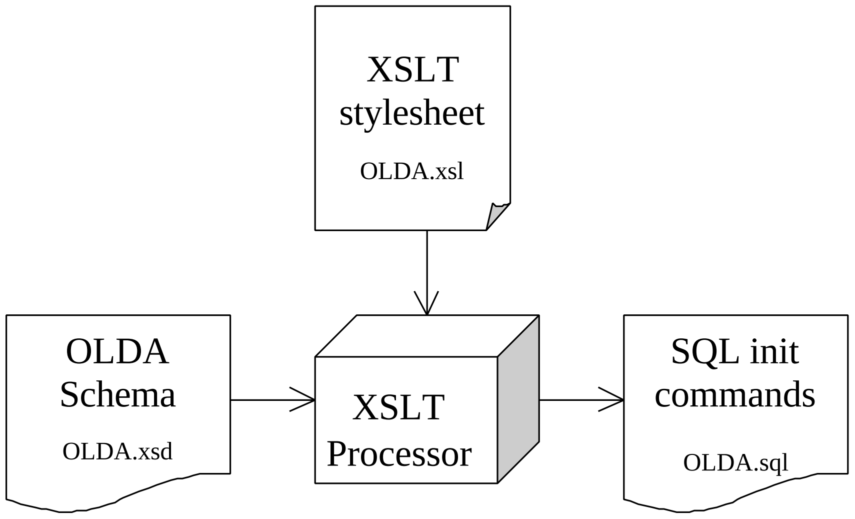
```
<?xml version="1.0"?>
<FIELDS>
    <NBANDS>
        <value> 3 </value>
    </NBANDS>
    <HOUR>
        <value> 1999 </value>
    </HOUR>
    <NBANDS>
        <value> 1 </value>
    </NBANDS>
</FIELDS>
```

4.3 Database Initialisation

To handle metadata storage and queries, OLGAS requires a database program with a JDBC driver to be installed and running. It is also necessary to set up a data model (or database schema) to specify the form of metadata to be stored. In this study, the OLDA data model was used to setup the database.

Initialising the database with a schema is achieved via the SQL `create table` command. Another SQL command, called `create view`, is used to specify which fields of a table are queryable. Previous implementations of OLGAS used a Perl script, called create_datamodel, to take the text file in Appendix A and generate the appropriate SQL initialization commands. The current system now performs this task using an XSLT stylesheet and processor as portrayed in Figure 25.

Figure 25: Generation of SQL setup commands from OLDA Schema



The stylesheet for this transformation is attached in Appendix C. It outputs three SQL commands: `drop table`, `create table`, and `create view` for a table named

OLDA_metadata. The main part of the stylesheet is in a template rule that matches the xsd:all element (in the OLDA.xsd file xsd:all contains all the field elements such as IMAGEID and HOUR etc.). Within this template three variables are created:

- all_fields - contains all the xsd:element nodes in OLDA.xsd that have a name attribute corresponding to an OLDA metadata field.
- primary_field - contains all OLDA metadata nodes that have a primary attribute set to “True” (normally a node list of length 1).
- query_field - contains all OLDA metadata nodes that have a queryable attribute set to “True”.

To form the `create table` statement, the name of each node in the all_fields variable is printed and another template is called to determine and output the type of the field. Then the primary key for the table is output by printing the name of the node in the primary_field variable. Finally, the name of the nodes in the query_field variable are output in an SQL `create view` statement.

If you wish to specify a different data model, X, to set up the database then this can be achieved in one of two ways.

1. Use an existing Schema definition for X. This method is favourable if X is a large and complex model, however, a new stylesheet must be developed to generate the SQL initialization commands.
2. Create a new Schema definition for X that has a similar form to the OLDA Schema, but uses the metadata field names from model X. This would be an appropriate method if no schemas exist for X. Another advantage of this approach is that the stylesheet in Appendix C could be applied to X to generate the SQL initialization commands. A script is provided in the new implementation, called create_datamodel.java, to assist the creation of a new XML Schema. create_datamodel.java accepts a file describing the metadata fields (similar to the file in Appendix A) and generates an XML Schema similar to the one in Figure 22.

4.4 Design of a Secondary Data Model (ORTHO)

Georectified aerial photographs (orthophotos), such as the image of Mt Bold in Figure 4, can often be described using a simple data model we denote by ORTHO. This is a fairly minimalist model containing fewer metadata fields than models such as GMS-5 and NITF. Hence, it was the model chosen for experimenting with the design and implementation of an XML-based conversion architecture.

The methodology used to define the ORTHO data model in XML Schema was similar to that outlined in Section 4.2: Design of the Primary Data Model (OLDA). Figure 26 shows a fragment of the ORTHO Schema defined. Again, the metadata fields are enclosed in the root element named <FIELDS>. Each field may occur 0 or 1 times and in any order as they are defined in an all-group.

Figure 26: ORTHO.xsd code fragment

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <xsd:element name="FIELDS" type="fieldsType"/>

  <xsd:complexType name="fieldsType">
    <xsd:all>

    <xsd:element name="Version" type="xsd:string"/>
    <xsd:element name="DataFile" type="xsd:string"/>
    <xsd:element name="DataSetType" type="xsd:string"/>
    <xsd:element name="DataType" type="xsd:string" default="Raster"/>
    <xsd:element name="ByteOrder" type="xsd:string" default="MSBFirst"/>
    <xsd:element name="Datum" type="xsd:string"/>
    <xsd:element name="Projection" type="xsd:string" default="RAW"/>
    <xsd:element name="CellType" type="xsd:string"/>
    <xsd:element name="Xdimension" type="xsd:positiveInteger" default="1"/>
    <xsd:element name="Ydimension" type="xsd:positiveInteger" default="1"/>
    <xsd:element name="NrOfLines" type="xsd:nonNegativeInteger"/>
    <xsd:element name="NrOfCellsPerLine" type="xsd:nonNegativeInteger"/>
    <xsd:element name="NrOfBands" type="xsd:nonNegativeInteger"/>
    <xsd:element name="Eastings" type="xsd:float" minOccurs=”1”/>
    <xsd:element name="Northings" type="xsd:float" minOccurs=”1”/>
    <xsd:element name="SenseDate" type="xsd:string"/>
    <xsd:element name="LastUpdated" type="xsd:string"/>
    <xsd:element name="Value" type="xsd:string"/>

    <!-- other element definitions here -->

    </xsd:all>
  </xsd:complexType>

</xsd:schema>
```

The design of the ORTHO schema was a lot simpler than the OLDA design for the following reasons. The OLDA schema needed to contain extra information to make it easier to initialize the database. The ORTHO data model was not used by the database and therefore did not need to define the “queryable” and “primary” attributes. This meant that each field could be defined as a simple type instead of a complex type. Also, the ORTHO schema in Figure 26 did not use any facets to restrict field content. Only the predefined types of XML Schema were used. These were adequate and allowed the necessary restrictions e.g. the NrOfLines field cannot be negative, the Xdimension must be a positive integer, etc. A more precise ORTHO Schema definition could be realized using facets, e.g. <xsd:enumeration> could be used to enumerate the possible values of fields like CellType (which could be Unsigned8BitInteger or Signed8BitInteger) and Projection (which could be RAW or METER) etc.

An example of a valid ORTHO document is given in Figure 27.

Figure 27: Valid ORTHO XML document

```
<?xml version="1.0"?>
<FIELDS>
    <Version> "5.5" </Version>
    <Xdimension> 25 </Xdimension>
    <Ydimension> 25 </Ydimension>
    <Eastings> 218159.2660504 </Easting>
    <Northings> 600056.12 </Northings>
    <NrOfLines> 4715 </NrOfLines>
    <NrOfCellsPerLine> 6008 </NrOfCellsPerLine>
</FIELDS>
```

4.5 Primary to Secondary Data Model Conversions

This section discusses the development of XSLT stylesheets to perform translations from a primary data model to a secondary data model and vice versa. This process is explained by considering the two-way mapping of the OLDA (primary) and ORTHO (secondary) data models. Three different stylesheets were created:

- ORTHO_OLDA_database.xsl – for loading ORTHO image metadata into the database
- ORTHO_OLDA.xsl – to convert queries in the ORTHO model to queries in the OLDA model
- OLDA_ORTHO.xsl – to convert queries in the OLDA model to queries in the ORTHO model

The stylesheet ORTHO_OLDA_database.xsl is given in Appendix D. Two different stylesheets were needed to map the secondary data view to the primary data view. One was responsible for the conversion of product data models and was used when adding a new image product to the database. The other was required to convert client data models for handling queries. Different stylesheets were needed because each of these transformations has slightly different functionality. The database loading stylesheet is required to generate metadata fields that are mandatory for inserting a product into the repository and clearly we do not want these fields to be generated for every client query. Now an overview of the different types of conversions will be given.

A common mapping that is often required when converting between different data models is a one-to-one mapping. This is where a field from one model maps directly to a field in another model. The fields may have the same name or a different name. There were a number of fields in the ORTHO model that mapped directly onto OLDA fields. An example of this is the NrOfBands field from the ORTHO model which corresponds to the NBANDS field in the OLDA model. Figure 28 illustrates how this mapping was expressed in the various stylesheets developed. There are very subtle differences. The first stylesheet maps <NrOfBands> to <NBANDS><value> so that the resultant OLDA document conforms to the OLDA Schema. The second stylesheet contains the line of code:

<xsl:apply-templates select="comment()[1]"/>
which calls a template to process the first comment node in the <NrOfBands> element. This is because the BQS query operators needed to be preserved in XML comments

during the transformation, please refer to Section 5.2.

Figure 28: XSLT – One to one mapping

ORTHO_OLDA_database.xsl

```
<xsl:template match=”//NrOfBands”>
    <xsl:element name=”NBANDS”>
        <xsl:element name=”value”>
            <xsl:value-of select=”.”/>
        </xsl:element>
    </xsl:element>
</xsl:template>
```

ORTHO_OLDA.xsl

```
<xsl:template match=”//NrOfBands”>
    <xsl:element name=”NBANDS”>
        <xsl:apply-templates select=”comment()[1]”/>
        <xsl:value-of select=”.”/>
    </xsl:element>
</xsl:template>
```

OLDA_ORTHO.xsl

```
<xsl:template match=”//NBANDS”>
    <xsl:element name=”NrOfBands”>
        <xsl:value-of select=”.”/>
    </xsl:element>
</xsl:template>
```

Data model conversions are seldom always one-to-one mappings. Sometimes it may be necessary to scale a field (e.g. m to km conversion) or to calculate a field value based on multiple fields from the input document. Figure 29 shows the conversion of fieldX (in kilometers) to fieldY (in meters) and how the SOUTH field of the OLDA model was calculated from multiple ORTHO fields. In the calculation of SOUTH the Ydimension represents the physical distance per pixel in the Y direction and NrOfLines is the number of pixels in the Y direction.

Figure 29: XSLT – Simple arithmetic conversions

Kilometers to meters:

```
<xsl:template match=”//fieldX”>
    <xsl:element name=”fieldY”>
        <xsl:value-of select=”1000*(//fieldX)”/>
    </xsl:element>
</xsl:template>
```

Calculation for OLDA SOUTH field:

```
<xsl:element name=”SOUTH”>
    <xsl:value-of select=”(//Northings) – (//Ydimension)*(//NrOfLines)”/>
</xsl:element>
```

Also, one-to-many mappings are frequently encountered when converting between data models. This is when one field in the source document maps to multiple fields in the target document. For example, the ORTHO SenseDate field maps to the OLDA fields YEAR, MONTH, DAY, HOUR, MIN, and SEC. Data extraction can be handled by XSLT functions such as `substring()` as shown in Figure 30. The substring function has three parameters: the first is the containing string; the second is the position in the containing string of the first character to be included in the result string; the third is optional and specifies the length of the result string [25]. Suppose the element <SenseDate> had the value “Thu Feb 11 10:57:32 GMT 1999” then the resulting OLDA document would have a YEAR and HOUR element as follows:

```
<YEAR><value>1999</value></YEAR>
<HOUR><value>10</value></HOUR>
```

Figure 30: XSLT – One to many mapping

ORTHO_OLDA_database.xsl

```
<xsl:template match=”//SenseDate”>
    <xsl:element name=”YEAR”>
        <xsl:element name=”value”>
            <xsl:value-of select=”substring(//SenseDate,25,4)”/>
        </xsl:element>
    </xsl:element>
    <xsl:element name=”HOUR”>
        <xsl:element name=”value”>
            <xsl:value-of select=”substring(//SenseDate,12,2)”/>
        </xsl:element>
    </xsl:element>

    <!-- and so on -->
</xsl:template>
```

In some situations, complex field transformations may need to be performed that cannot be expressed, or are cumbersome to express, using XSLT. A common and rather challenging problem is to convert Latitude-Longitude fields (in degrees, minutes, seconds) to UTM (in meters). As explained in Section 3.4, XSLT extension functions provide a way to invoke a Java or JavaScript program from the stylesheet. The XSLT Processor detects extension functions through the <xsl:script> element which informs the processor where to locate the implementation. Extension functions are also extremely useful if the processing you require already exists in a Java library. Suppose there was a Java class, *java.metautil.Fields*, consisting of the function *latlon_2_utm(args)* to perform the transformation described above. Figure 31 shows how to call this function from within a stylesheet. Note that the class *java.metautil.Fields* must be included in the classpath.

Figure 31: XSLT – Data model conversion via extension functions

```
<?xml version=”1.0”?>
<xsl:stylesheet …
        xmlns:fields=”java:java.metautil.Fields”>

    <xsl:script language=”java” implements-prefix=”fields”
                src=”java:java.metautil.Fields”/>
    …
    <xsl:template match=”//LATFIELD”>
        <xsl:element name=”UTMFIELD”>
            <xsl:value-of select=”fields:latlon_2_utm(//LATFIELD)”/>
        </xsl:element>
    </xsl:template>
</xsl:stylesheet>
```

Chapter 5

A New Metadata Conversion Architecture

The purpose of this chapter is to describe the modifications that were made to the Image and Metadata Conversion architecture of OLGAS to enable the data model conversions outlined in Chapter 4.

Section 2.4.2.1 described the metadata conversions required in the OLGAS system. Three different types of conversions needed to be addressed by the Metadata Conversion architecture and they are listed below:

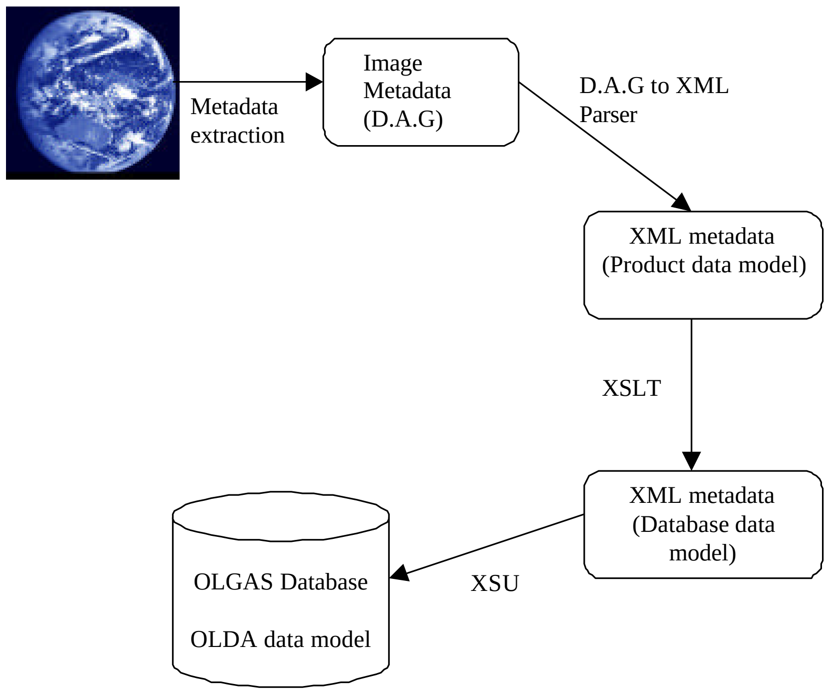
- Before image metadata is stored in a repository it needs to be transformed from the data model used in the image product to the data model used in the system's internal database.
- Query submissions that are in a different data model to the one employed in the database need to be transformed to the database data model before the query can be executed.
- The results from a query are returned in the database data model and may need to be transformed back into a model that the client understands.

Development of a new Metadata Conversion architecture for OLGAS involved modifying the implementation of two managers defined by the GIAS specification. The Creation Manager, which handles the submission of image products to a library, was altered to cater for the first conversion listed above. The Catalog Access Manager also required changes in order to support multiple query data views from clients.

5.1 Creation Manager

As mentioned in Section 2.3.1, the Creation Manager code handles the addition of image metadata and products to a library. It consists of two operations: `create()` and `create_metadata()`. The `create()` method allows a client to nominate an image product and associated metadata to the library. The `create_metadata()` method makes it possible for clients to add metadata only and this is useful if the image is not currently available in digital form. If an image product is nominated then the system must be able to recognize the image format, extract its metadata and convert it to the data model used by the library catalog. The diagram in Figure 32 provides an overview of the architecture that was designed to utilize the XSLT data model conversions described in the previous chapter.

Figure 32: Conversion architecture – Support for multiple product data views



When an end-user requests to add a new image product, the system checks to see if the product can be read/written and whether the image metadata can be extracted. Next the system searches for the availability of a stylesheet for transforming the product data model into the database data model. The system expects to find the stylesheet files in a specific directory and for them to obey the following naming convention, <ProductDataModel>_<DatabaseDataModel>_database.xsl.
If no stylesheet is available then submission of this metadata will fail and the `create()` method will return an exception.

When image metadata fields and values are extracted from the image product they are placed into a GIAS data structure called a Directed Acyclic Graph (D.A.G). A DAG to XML parser was developed to generate an XML document representation of the metadata in the DAG. We needed to have the image metadata in XML syntax before it could be transformed using XSLT to the database schema. The DAG to XML parser accepts a DAG structure containing the metadata and returns a File containing the XML created. The XML document structure created by this parser can be seen in Figure 33. This figure shows that documents are generated with a root element called <FIELDS>. This was to conform to the schema definitions of the secondary data views (such as the ORTHO model in Figure 26). Each metadata field in the DAG is output as a child element of <FIELDS>. More specifically, an opening field tag is output followed by the value and then a closing field tag.

Figure 33: D.A.G. to XML parser output

```
<?xml version=”1.0”?>
<FIELDS>
    <field1> value1 </field1>
    <field2> value2 </field2>
    <field3> value3 </field3>
    …
</FIELDS>
```

This DAG to XML parser, along with other helper classes that needed to be developed were placed in a package called DHPC_GIAS.xml_utils.

An XSLT processor was plugged-in to the code to carry out the metadata conversion. The XML file produced from the DAG to XML parser was passed in as a parameter to the processor along with the appropriate stylesheet that contains rules for transforming that product into the database data model. In the class diagram of the existing OLGAS meta-conversion architecture shown in Figure 5, we can see that the ProductMetaConverter class is abstract and is subclassed by each new image product with the ability to convert metadata. In the new implementation, ProductMetaConverter is no longer abstract. The tasks performed by this class are as follows:

1. An Image Converter class is loaded for a product. (Image Converters are classes that contain methods for reading and writing image files, extracting metadata and additional image processing).
2. A `get_metadata()` method from Image Converter class is called to retrieve product metadata in a DAG structure
3. A DAG to XML parser is called upon to generate an XML document in the product data model
4. The XSLT stylesheet (e.g. ORTHO_OLDA_database.xsl) for transforming the product data model to the database data model is located. Also, a mandatory.xsl stylesheet is setup (this is specific to OLGAS and deals with the mandatory fields that are required by the OLGAS database).
5. XSLT processor is instantiated
6. XSLT processor takes the XML document from step 3 and the stylesheet from step 4 and produces an XML document conforming to the database data model.
7. The document produced from step 6 is then returned to the caller, namely the FileAccessFunctions class which controls the check-in process for the metadata.

To support the addition of a new image product, the only thing that must be supplied is an appropriate XSLT stylesheet e.g. ORTHO_OLDA_database.xsl. Of course this assumes that there is an Image Converter available, a JAI reader/writer interface, for extracting the metadata from the new product. Also note that the need to supply a subclass for each new product has been eliminated.

A problem encountered with this conversion was how to handle the mandatory metadata fields that were needed to enable a product to be stored in the database. In OLGAS, the database required the following metadata fields to be present:

ACCESSID, FILENAME, URL and THUMBNAIL. An explanation of these fields was given in Section 2.4.2.1. The conversion architecture uses a separate stylesheet called mandatory.xsl to deal with fields that are mandatory to the database. The behaviour of mandatory.xsl is inherited in the

<ProductDataModel>_<DatabaseDataModel>_database.xsl

modules via an <xsl:include> element. Please refer to Appendix D. Note that the same mandatory.xsl stylesheet is used for each new image product because its content is specific to the database data model.

In mandatory.xsl, a global parameter was defined for each mandatory field:

```
<xsl:param name=”accessed” select=”null”/>
<xsl:param name=”filename” select=”null”/>
<xsl:param name=”url” select=”null”/>
<xsl:param name=”thumbnail” select=”null”/>
```

Step 4 in the ProductMetaConverter algorithm above states that a mandatory XSLT Stylesheet object is setup. The setup phase involves passing the values for all mandatory database fields to the mandatory XSLT Stylesheet object and this must be done before instantiating an XSLT processor. The XSLT Stylesheet class has operations to set and remove the value of top-level parameters (i.e. global parameters). For example, to set the ACCESSID parameter the following method call is needed:

```
<MandatoryStylesheetObject>.setParam(“accessed”,objID);
```

where objID is a String representing the value of the ACCESSID field. The segment of code in Figure 34 shows how mandatory field elements are created for the resulting XML document.

Figure 34: mandatory.xsl code fragment

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

    <xsl:param name=”accessid” select=”null”/>

    <!-- other global parameter definitions -->

    <xsl:template match=”/”>

        <xsl:element name=”ACCESSID”>
            <xsl:if test=”(count(//ACCESSID) = 0)”>
                <xsl:value-of select=”($accessid)”/>
            </xsl:if>
            <xsl:if test=”(count(//ACCESSID) > 0)”>
                <xsl:value-of select=”(//ACCESSID)”/>
            </xsl:if>
        </xsl:element>

        <!-- definition of the other mandatory fields -->

    </xsl:template>
</xsl:stylesheet>
```

Let’s consider the element definition of ACCESSID in more detail. The value of the global parameter “accessid” is assigned to the output ACCESSID element only if there are no ACCESSID elements specified in the source XML document (i.e. if the product metadata or user does not supply a value for ACCESSID). The second if-test handles the case when the input document has specified the value for ACCESSID. The other elements FILENAME, URL and THUMBNAIL are defined in a similar manner.

Step 7 above states that a FileAccessFunctions class controls the check-in of the XML document. To check-in metadata, this class makes a call to the `store_metadata()` routine of the DBLibrary class (an implementation for the database library). Previously, the `store_metadata()` routine accepted an SQL insert statement. This method has been altered to accept an XML document which is then submitted to a DataBaseMgr class for direct insertion into the database using Oracle’s XML SQL Utility. The insertion of XML documents into a database is achieved through the class OracleXMLSave and an example was given in Figure 17.

5.2 Catalog Access Manager

The role of the Catalog Access Manager is to submit client queries to search the holding of a GIAS Library. Version 3.3 of the GIAS specification now allows a client to specify the query data view they wish to use. This addition requires the server implementation to be able to convert from the client query view to the data view used in the library and vice versa. Please refer again to Figure 4: Conversion Architecture Requirements.

As mentioned, queries must be submitted in Boolean Query Syntax, a grammar that is specific to the GIAS Specification. This allows a client to interact with a catalog in a uniform manner regardless of the database’s implementation, native query language (SQL), and physical schema (data model) [51]. It is based on the notion of field-operator-value triplets called factors. Each factor represents a condition of interest. For example, a client query to recover images from the year 2000 will contain the factor: YEAR = 2000 assuming there is a YEAR field in the data view. A complete query is assembled by relating the factors with Boolean operators “and” and “or”. So a client wishing to retrieve images from the year 2000 with a size no smaller than 256 by 512 pixels would form the query:

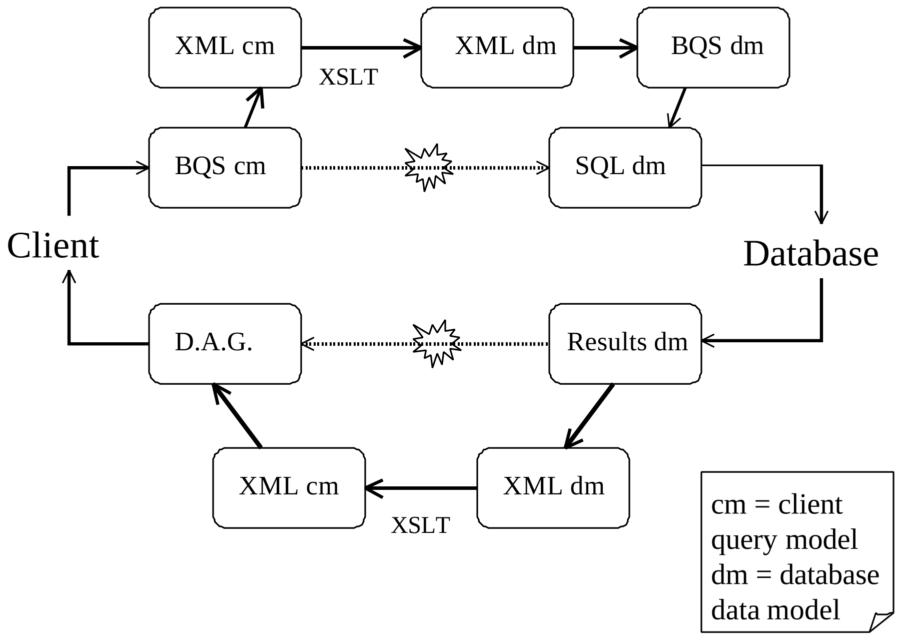
Figure 35: Example BQS query

NROWS >= 256 and NCOLS >= 512 and YEAR = 2000.

A complete definition for BQS is set out in the GIAS specification in Backus-Naur Form.

In the previous version of OLGAS, BQS queries were parsed directly into an SQL statement that was sent to the database. This was possible because the system required the query data view to be the same as the database data view. Now, the system implementation needs to handle multiple query data views. Thus, we must ensure that the SQL statement created only includes fields that are in the data view understood by the database. Figure 36 includes an outline of the architecture designed to enable conversions from the client query data view to the database data view. The bold arrows in the diagram reveal the modifications that were made to OLGAS to support multiple query models.

Figure 36: Conversion architecture – Support for multiple query data views



Firstly, a parser was developed to convert a BQS query into a well-formed XML document. Again, the rationale for doing this was to be able to use a stylesheet to transform the query metadata fields into fields that would be recognized in the database. As mentioned above, a BQS query is composed of <field-operator-value> triplets. The BQS_XML parser was developed to generate XML obeying the form shown in Figure 33, yet with the ability to remember the operators that made up and separated each factor. This was essential to reconstruct a sensible BQS query, after the application of a stylesheet, from the XML (now in the database data model). The decision was made to preserve operators by placing them in an XML comment. This is explained more clearly in Figure 37. It was first hoped that the stylesheet used for loading metadata into a database could be applied here to convert the client query data view to the database data view. However, the goals of each conversion are slightly different and hence each stylesheet required different functionality. For example, the database loading stylesheet generated metadata fields that were mandatory for setting up the database. We would not want these fields to be generated for every query. The stylesheet for converting queries needed to have a template to match the comment nodes and copy them to the result document.

A function was created, called regenerate_BQS, to reconstruct a sensible BQS query from an XML document with the form shown in Figure 37. The following basic algorithm was used to accomplish this task:

```
1. read xml_declaration and root element, discard
2. While not end of file, read line
     o If comment => extract operator
     o If open tag => extract metadata field name
     o If closing tag => discard
     o Else, must have value => extract value
```

We have now generated a BQS query that contains metadata fields understood by the database. The query can now be transformed into an SQL statement by using the BQS_SQL parser from the old system.

Figure 37: BQS_XMLParser output

```
An example BQS query looks like:

       not <field-operator-value> and
           <field1-operator1-value1> or
           <field2-operator2-value2> and
           <field3-operator3-value3>

BQS_XMLParser generates XML document with form:

       <?xml version=”1.0”?>
       <FIELDS>
          <!-- not -->
          <field>
             <!-- operator -->
             value
          </field>
          <!-- and -->
          <field1>
             <!-- operator1 -->
             value1
          </field1>
          …
       </FIELDS>
```

Note that queries presented in the database data model are parsed directly into an SQL statement as before.

The results returned from a query are in the database model and may need to be converted to the query data view before being transported back to the client. The previous version of OLGAS put the results straight into a DAG structure and returned this to the client. The metadata conversion architecture developed here needed to intercept the results and transform them to the correct model using XSLT before storing them in a DAG for the client.

The results DAG is formed in the `query_results()` method of the DBLibrary class, a class containing routines to interact with the database. This method was modified in the following way:

- Check whether the query data model is the same as the database data model
- If yes => skip the XSLT conversion code and convert results to DAG as before
- Otherwise
    - Instantiate a ResultsHandler class (a class defining two important methods: one to generate XML from the database results – similar to how OracleXMLQuery class operates; and another to generate a DAG given an XML document)
    - Check for presence of XSLT converter to transform database data view into query data view (e.g. ORTHO_OLDA.xsl)
    - If converter unavailable => exception (scream madly etc)
    - Otherwise
        - Invoke ResultsHandler.generate_XML to get an XML document in the database data view
        - Run the XSLT processor the convert the XML in the database data view to an XML document in the query data view
        - Extract name value pairs from the XML query data view and store in a DAG

5.3 Data Model Manager

The Data Model Manager is a recent addition to the GIAS specification and provides operations to allow a client to discover the data views supported by a library. The Data Model Manager was not complete at the time of writing this thesis, however a design for how to implement it based on the new data model architecture using XML is given below.

The manager provides some basic functions to access ancillary data like `get_data_model_date()` which returns the last date the library's data model was updated. A fixed attribute, called 'last_updated', could be added to the root element of the database schema (in this case the OLDA Schema) to record the date of the last update to the model. Then the Data Model Manager implementation could easily extract the date information using a very simple stylesheet (in fact only one template rule is required):

```
<xsl:template match=”/”>
        <xsl:value-of select=”@last_updated”/>
</xsl:template>
```

Other ancillary functions could be incorporated into the same stylesheet, and a global parameter could be used to indicate to the XSLT processor which section of the stylesheet to process to generate the function result.

The manager also provides functions that allow a client to discover information about the data views supported by a library. An essential operation is `get_data_views()`. It could be assumed that a data view is supported if it has a XML Schema and XSLT stylesheets to convert to and from the internal database data model. As the server expects a particular naming convention for the stylesheets and schemas it could be implemented by searching for particular files. For example, if ORTHO.xsd, ORTHO_OLDA.xsl and OLDA_ORTHO.xsl were discovered then the server could add the ORTHO model to the list of data views it supports.

Another Data Model Manager function is `get_attributes()`. This accepts a data view and returns a list of metadata fields that describe that data view. This type of information could be extracted easily from an XML Schema defining a data view using XSLT.

Chapter 6

Summary and Future Work

This project has shown that it is possible to use XML and XSLT, the emerging web standards, to handle geospatial metadata and data model conversions within the constraints of the GIAS software architecture. This was demonstrated by the design and implementation of such an architecture for the OLGAS program, a geospatial image server that implements a subset of the GIAS interface standards.

Data models were defined using XML Schema and rules to specify how to transform one data model into another were expressed in XSLT stylesheets. An architecture was designed to handle the addition of image products that employ different data models to a repository; and support client queries which may be in a different data view to the internal database. This functionality is essential since it allows data to be integrated from different distributed sources. It also allows more clients access to the data as querying can be performed in a number of different data views.

The architecture described within this paper could be applied in any other geospatial image archive that implements the GIAS standard interfaces. Minor changes would be required if the server database schema was different to the schema in the OLGAS database. That is, the primary data model would need to be defined in XML Schema along with some XSLT stylesheets to handle the mandatory metadata fields and the transformations to and from the secondary data models. However the MetaConverter Java class files would not require changing.

The system developed could be used in geospatial image servers that use different interface standards such as the Open GIS Catalog Interface specification. However, the system will need to supply a specific parser to convert the query language used in the Open GIS specifications (i.e. the OGC_Common Catalog Query Language) to XML. In the system implemented this is equivalent to the BQS to XML parser.

The same problem of handling metadata and data model conversions is a general one for any kind of data archive, and is particularly important for enabling federated querying across multiple archives (e.g. federated query requires the system to handle multiple servers and clients that use different data models).

A conversion architecture to handle multiple data models is essential, even in application areas that have a well-defined standard data model in use by all products, clients and servers. The reason is that most standard data models will undergo periodic revisions, leading to the inevitability of clients and servers using a different version of the data model at any one time.

There is much more that could be done to extend on the work completed here. The first task would be to provide a complete implementation for the Data Model Manager. The implementation of the Data Model Manager is essential because it is the mechanism by which clients can discover the data models that are supported by a server. The client can be confident in using any of the data models returned by this manager to query the server's database. Some ideas on how to implement the Data Model Manager were suggested in Section 5.3. These suggestions involved using simple XSLT stylesheets to extract information about the data models, e.g. the date of the last modification, from their XML Schema definitions. Discovery of the secondary data models may involve a simple directory search for secondary data model definitions (.xsd files) and the corresponding stylesheet converters (.xsl files) that are required to translate the secondary data model to and from the primary data model.

Once the Data Model Manager is complete, support for a wider range of data models could be provided. The addition of a new data model would require an XML Schema to define the legal metadata fields and types and three XSLT stylesheets to support all the conversions that are required to and from the database data model. After adding a series of new data models they system should be thoroughly tested to check if all GIAS managers are exhibiting the correct behaviour.

Other future work may include extending the OLGAS system to support both the GIAS and Open GIS Catalog Interface specifications. This would involve building a higher level software architecture to abstract over both interface specifications.

Appendices

A: OLDA data model fields

IMAGEID	char(80)	Q
ICAT	char(16)	Q
PLATID	char(64)	Q
NBANDS	short	
ABPP	short	
NCOLS	long	
NROWS	long	
CLASS	char(1)	Q
RELEASE	char(256)	Q
PRODFMT	char(64)	Q
PRODSNME	char(16)	Q
PRODUCERCD	char(64)	
MEANGSD	float	
DESCRIP	char(512)	Q
CLOUDCVR	float	Q
TIMECOLL	char(16)	
YEAR	long	Q
MONTH	long	Q
DAY	long	Q
HOUR	long	Q
MIN	long	Q
SEC	long	Q
NORTH	float	Q
SOUTH	float	Q
EAST	float	Q
WEST	float	Q
NWLAT	float	
NWLON	float	
NELAT	float	
NELON	float	
SWLAT	float	
SWLON	float	
SELAT	float	
SELON	float	
DROPOUTS	long	Q

B: XML DTD Example

```
<?xml version="1.0"?>
<!DOCTYPE VideoStore [

    <!ELEMENT VideoStore (TradingName, Description?, ContactDetails, Videos)>
    <!ELEMENT TradingName (#PCDATA)>
    <!ELEMENT Description (#PCDATA)>
    <!ELEMENT ContactDetails (Address, Telephone, Facsimile?, Email?)>
    <!ELEMENT Address (StreetNumber, StreetName, Suburb, State, Postcode)>
    <!ELEMENT Telephone (#PCDATA)>
    <!ELEMENT Facsimile (#PCDATA)>
    <!ELEMENT Email (#PCDATA)>
    <!ELEMENT StreetNumber (#PCDATA)>
    <!ELEMENT StreetName (#PCDATA)>
    <!ELEMENT Suburb (#PCDATA)>
    <!ELEMENT State (#PCDATA)>
    <!ELEMENT Postcode (#PCDATA)>
    <!ELEMENT Videos (Video*)>
    <!ELEMENT Video (Title, Genre?, Starring, RunningTime, Classification?,
                                   Rating?, Price)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Genre (#PCDATA)>
    <!ELEMENT Starring (Actor+)>
    <!ELEMENT Actor (#PCDATA)>
    <!ELEMENT RunningTime (#PCDATA)>
    <!ELEMENT Classification (#PCDATA)>
    <!ELEMENT Rating (#PCDATA)>
    <!ELEMENT Price (#PCDATA)>

    <!ATTLIST Video serialNo ID #REQUIRED>
    <!ATTLIST Actor firstName CDATA #REQUIRED>
    <!ATTLIST Actor lastName CDATA #REQUIRED>

]>
<VideoStore>

   <TradingName> VideoCrazy </TradingName>

   <Description> Video Sales - not Hire </Description>

   <ContactDetails>
      <Address>
         <StreetNumber> 109 </StreetNumber>
         <StreetName> Shopping Crescent </StreetName>
         <Suburb> Shopsville </Suburb>
         <State> Australian Shopping Capital </State>
         <Postcode> 007 </Postcode>
      </Address>
      <Telephone/>
      <Email> staff@videocrazy.com.au </Email>
   </ContactDetails>

   <Videos>

      <Video serialNo="00000001">
         <Title> The Fugitive </Title>
         <Genre> Action </Genre>
         <Starring>
            <Actor firstName="Harrison" lastName="Ford"/>
            <Actor firstName="Tommy Lee" lastName="Jones"/>
         </Starring>
         <RunningTime>120</RunningTime>
         <Classification> M </Classification>
         <Rating> ***** </Rating>
         <Price> 24.95 </Price>
      </Video>

      <Video serialNo="0000002">
         <Title> Titanic </Title>
         <Genre> Drama </Genre>
         <Starring>
            <Actor firstName="Leonardo" lastName="DiCaprio"/>
            <Actor firstName="Kate" lastName="Winslet"/>
         </Starring>
         <RunningTime> 185 </RunningTime>
         <Classification> PG </Classification>
         <Rating> ** </Rating>
         <Price> 20.95 </Price>
      </Video>
      ...
   </Videos>
</VideoStore>
```

C: OLDA.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.1"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xsd="http://www.w3.org/1999/XMLSchema"
                exclude-result-prefixes="xsd"
                >

<xsl:output method="text"/>

<xsl:template match="/">

   <xsl:text>
       &#xA;drop table OLDA_metadata;
       &#xA;create table OLDA_metadata(
   </xsl:text>

   <xsl:apply-templates select="/xsd:schema/xsd:complexType/xsd:all"/>
</xsl:template>


<xsl:template match="/xsd:schema/xsd:complexType/xsd:all">

   <xsl:variable name="all_fields"
    select="xsd:element[not(@name='FIELDS' or @name='value')]"/>

   <xsl:variable name="primary_field"
    select="xsd:element[xsd:complexType/xsd:attribute[@name='primary' and
                                                  @value='True']]"/>

   <xsl:variable name="query_field"
    select="xsd:element[xsd:complexType/xsd:attribute[@name='queryable' and
                                                  @value='True']]"/>

      <xsl:text>&#xA;</xsl:text>
      <xsl:for-each select="$all_fields">
         <!-- print the name of the field -->
         <xsl:value-of select="@name"/>

         <xsl:apply-templates select="xsd:complexType/xsd:element[@name='value']"/>
         <xsl:if test="position()!=last()"> <xsl:text>,&#xA;</xsl:text> </xsl:if>
         <xsl:if test="position()=last()">,
            primary key (<xsl:value-of select="$primary_field/@name"/>);
         </xsl:if>
      </xsl:for-each>

      <!-- Create a SQL view to display the queryable fields -->
      <xsl:text> &#xA;create view OLDA_query as &#xA;select </xsl:text>

      <xsl:for-each select="$query_field">
         <xsl:value-of select="@name"/>
         <xsl:if test="position()!=last()">, </xsl:if>
      </xsl:for-each>

      <xsl:text> &#xA;from OLDA_metadata;</xsl:text>

</xsl:template>
```

```
<xsl:template match="xsd:complexType/xsd:element[@name='value']">
   <xsl:if test="@type">
      <!-- here we have a built-in type -->
      <xsl:choose>
         <xsl:when test="@type='xsd:float'"> real </xsl:when>
         <xsl:when test="@type='xsd:short'"> smallint </xsl:when>
         <xsl:otherwise> <xsl:value-of select="@type"/> </xsl:otherwise>
      </xsl:choose>
   </xsl:if>
   <xsl:if test="not(@type)">
      <xsl:apply-templates select="xsd:simpleType"/>
   </xsl:if>
</xsl:template>


<xsl:template match="xsd:simpleType">
   <xsl:choose>
      <xsl:when test="@base='xsd:string'">
         <xsl:variable name="string_length" select="xsd:maxLength/@value"/>
         <xsl:if test="$string_length='1'"> char </xsl:if>
         <xsl:if test="$string_length!='1'"> char(
                       <xsl:value-of select="$string_length"/>) </xsl:if>
      </xsl:when>
      <xsl:when test="@base='xsd:long'">
         <xsl:text> int </xsl:text>
      </xsl:when>
      <xsl:otherwise> <xsl:value-of select="@base"/> </xsl:otherwise>
   </xsl:choose>
</xsl:template>

</xsl:stylesheet>
```

D: ORTHO_OLDA_database.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.1"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <xsl:include href="mandatory_fields.xsl"/>

   <xsl:output method="xml" version="1.0" indent="yes"/>

   <xsl:variable name="sense_date">
      <xsl:value-of select="normalize-space(//SenseDate)"/>
   </xsl:variable>

   <xsl:variable name="cell_type">
      <xsl:value-of select="normalize-space(//CellType)"/>
   </xsl:variable>

<xsl:template match="/">

   <!-- the top element of the OLDA XML document -->
   <xsl:element name="FIELDS">

   <!-- Literal Mappings e.g. ICAT = "VIS" -->
   <xsl:element name="ICAT">
      <xsl:element name="value">
         <xsl:text>VIS</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="PLATID">
      <xsl:element name="value">
         <xsl:text>LANDSAT</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="CLASS">
      <xsl:element name="value">
         <xsl:text>U</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="RELEASE">
      <xsl:element name="value">
         <xsl:text>Registered users</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="PRODUCERCD">
      <xsl:element name="value">
         <xsl:text>South Australian Department of Environment and Heritage</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="MEANGSD">
      <xsl:element name="value">
         <xsl:text>30</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="DESCRIP">
      <xsl:element name="value">
         <xsl:text>Landsat image of Adelaide region</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="CLOUDCVR">
      <xsl:element name="value">
         <xsl:text>999</xsl:text>
      </xsl:element>
   </xsl:element>

   <xsl:element name="DROPOUTS">
      <xsl:element name="value">
         <xsl:text>0</xsl:text>
      </xsl:element>
   </xsl:element>


   <!-- One-to-One mappings e.g.IMAGEID == DataFile -->
   <xsl:element name="IMAGEID">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//DataFile)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="NBANDS">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//NrOfBands)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="NCOLS">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//NrOfCellsPerLine)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="NROWS">
```

```
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//NrOfLines)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="PRODFMT">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//DataSetType)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="PRODSNME">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//DataType)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="NORTH">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//Northings)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="EAST">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//Eastings)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="ICORDS">
      <xsl:element name="value">
         <xsl:value-of select="normalize-space(//Datum)"/>
      </xsl:element>
   </xsl:element>


   <!-- Algebraic manipulations -->
   <xsl:element name="SOUTH">
      <xsl:element name="value">
        <xsl:value-of select="//Northings - (//Ydimension * //NrOfLines)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="WEST">
      <xsl:element name="value">
         <xsl:value-of select="//Eastings - (//Xdimension * //NrOfCellsPerLine)"/>
      </xsl:element>
   </xsl:element>

   <!-- this shows how it is possible to extract data from SenseDate
        NOTE: if extraction is complex may want to use an extension function. -->
   <xsl:element name="YEAR">
      <xsl:element name="value">
         <!-- get the year from SenseDate string -->
         <xsl:value-of select="substring($sense_date,25,4)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="MONTH">
      <xsl:element name="value">
         <!-- get the month from SenseDate string -->
         <xsl:variable name="month">
            <xsl:value-of select="substring($sense_date,5,3)"/>
         </xsl:variable>
         <xsl:choose>
            <xsl:when test="$month='Jan'">01</xsl:when>
            <xsl:when test="$month='Feb'">02</xsl:when>
            <xsl:when test="$month='Mar'">03</xsl:when>
            <xsl:when test="$month='Apr'">04</xsl:when>
            <xsl:when test="$month='May'">05</xsl:when>
            <xsl:when test="$month='Jun'">06</xsl:when>
            <xsl:when test="$month='Jul'">07</xsl:when>
            <xsl:when test="$month='Aug'">08</xsl:when>
            <xsl:when test="$month='Sep'">09</xsl:when>
            <xsl:when test="$month='Oct'">10</xsl:when>
            <xsl:when test="$month='Nov'">11</xsl:when>
            <xsl:when test="$month='Dec'">12</xsl:when>
            <xsl:otherwise>Invalid month</xsl:otherwise>
         </xsl:choose>
      </xsl:element>
   </xsl:element>

   <xsl:element name="DAY">
      <xsl:element name="value">
         <!-- get the day from SenseDate string -->
         <xsl:value-of select="substring($sense_date,9,2)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="HOUR">
      <xsl:element name="value">
         <!-- get the hour from SenseDate string -->
         <xsl:value-of select="substring($sense_date,12,2)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="MIN">
      <xsl:element name="value">
         <!-- get the minute from SenseDate string -->
         <xsl:value-of select="substring($sense_date,15,2)"/>
      </xsl:element>
   </xsl:element>

   <xsl:element name="SEC">
      <xsl:element name="value">
         <!-- get the minute from SenseDate string -->
         <xsl:value-of select="substring($sense_date,18,2)"/>
      </xsl:element>
   </xsl:element>

   <!-- need to convert SenseDate into Zulu format
        Zulu format is DDHHMMSSZMONYY-->

   <xsl:element name="TIMECOLL">
      <xsl:element name="value">
         <xsl:value-of select="substring($sense_date,9,2)"/>
         <xsl:value-of select="substring($sense_date,12,2)"/>
         <xsl:value-of select="substring($sense_date,15,2)"/>
         <xsl:value-of select="substring($sense_date,18,2)"/>
         <xsl:text>Z</xsl:text>
         <xsl:value-of select="substring($sense_date,5,3)"/>
         <xsl:value-of select="substring($sense_date,27,2)"/>
      </xsl:element>
   </xsl:element>

   </xsl:element>

</xsl:template>

</xsl:stylesheet>
```

Bibliography

[1] ANSI/NISO (1992), ANSI/NISO Z39.50-1992: *Information Retrieval Service Definition and Protocol Specification for Open Systems Interconnection,* NISO Press.

[2] Chang K (Sep 23, 2001*). From 5,000 Feet Up, Mapping Terrain for Ground Zero Workers*, <http://www.earthdata.com/news_nytimes_wtc_9-23-2001.htm>.

[3] Coddington P et al (1998). *Implementation of a Geospatial Imagery Digital Library using Java and CORBA*, Proc. of Technologies of Object-Oriented Languages and Systems Asia '98 (TOOLS 27), Beijing, Sept 1998, J. Chen et al., (IEEE, 1998). <http://www.dhpc.adelaide.edu.au/reports/047/abs-047.html>.

[4] Coddington P et al (1999). *Interfacing to On-line Geospatial Imagery Archives*, Proc. of Australasian Urban and Regional Information Systems Assoc. Conf. (AURISA'99), Leura, NSW, November 1999. <http://www.dhpc.adelaide.edu.au/reports/071/abs-071.html>.

[5] Coddington P (1999). *User Guide for the OLGAS Image Server*. <http://www.dhpc.adelaide.edu.au/reports/056/abs-056.html>.

[6] Cover R (ed). *The XML Cover Pages*, <http://www.oasis-open.org/cover/sgml-xml.html>.

[7] Crane Softwrights Ltd., *Practical Transformation Using XSLT and XPath*, <http://www.cranesoftwrights.com/training/#ptux-dl>.

[8] Department for Environment and Heritage, *Land & Maps*, <http://www.environment.sa.gov.au/mapland/image.html>.

[9] *Distributed Geolibraries* Spatial Information Resources 1999, National Academy Press, Washington DC, <http://www.nap.edu/html/geolibraries/>.

[10] Federal Emergency Management Agency (FEMA), *GIS Geographic Information Systems*, <http://www.gismaps.fema.gov/index.htm>

[11] FEMA, *New York Major Crossing Status*, <http://www.gismaps.fema.gov/2001graphics/dr1391/rm_dm15sept_WTC_bldg debris.jpg>.

[12] FEMA, *New York Remote Sensing – 15 September Debris Field and Building Damage Levels*, <http://www.gismaps.fema.gov/2001graphics/dr1391/travel_restrictions09_19.jpg>.

[13] FEMA, *Post-attack Image of New York,* <http://www.gismaps.fema.gov/2001graphics/dr1391/nyc_pan_12sep.jpg>.

[14] GenaWarehouse International. *Imagis FAQs – ERMapper ERS Files*, <http://www.genaware.com/html/support/faqs/imagis/imagis13.htm>.

[15] Green K, *Managing Terabytes from Terra*, HPCwire, [Online, accessed 30 March, 2001]

[16] Greenman C (Oct 4, 2001). *Mapping the Hazards to Keep Rescuers Safe*, The New York Times, <http://www.nytimes.com/2001/10/04/technology/circuits/04MAPS.html>.

[17] Harold E (2001). *Chapter 17 of the XML Bible, Second Edition: XSL Transformations*, <http://www.ibiblio.org/xml/books/bible/updates/14.html>.

[18] Harwick K, Coddington P. *Interfacing to Distributed Active Data Archives*, Future Generation Computer Systems 16, 73 (1999). <http://www.dhpc.adelaide.edu.au/reports/050/abs-050.html>.

[19] Hildebrandt J., Grigg M., Bytheway S., Hollamby R., & James S. *Dynamic C2 application of Imagery and GIS information using backend repositories*. DSTO technical report.

[20] Holzner S (2001). *Inside XML*, New Riders Publishing, U.S.A.

[21] HPCwire, *XML gets nod from net standards group* [Online, accessed 6 May 2001].

[22] Java Developer Connection, *Introduction to CORBA: Short Course*, <http://developer.java.sun.com/developer/onlineTraining/corba/corba.html>.

[23] Kay M (2001). *XSLT Programmer's Reference 2nd Edition*, Wrox Press Ltd., Birmingham.

[24] Letham G (2001). *GIS Database Developed for New York WTC Recovery Efforts*, <http://spatialnews.geocomm.com/features/wtc>.

[25] Mason K (1999). *Implementation of Product and Metadata Conversion in the OLGAS Library*, unpublished.

[26] Mason K (1999). *The Architecture of the OLGAS Library for Image and Meta-Data Conversion*, unpublished.

[27] McGrath R (2001). *An Experimental Comparison of HDF4, HDF5, and XML Representations of the Same Dataset*, <http://hdf.ncsa.uiuc.edu/HDF5/XML/EOSData/h4-h5-xml.htm>.

[28] *Metadata in GIS*, <http://www.ci.anchorage.ak.us/gis/gisinternet/htmls/>.

[29] National Center for Supercomputing Applications, *Information, Support, and Software from the Hierarchical Data Format (HDF) Group of NCSA*, <http://hdf.ncsa.uiuc.edu>.

[30] *OLGAS home page*, <http://www.dhpc.adelaide.edu.au/projects/olgas/index.html>.

[31] Open GIS Consortium, <http://www.opengis.org/>.

[32] Open GIS Consortium, *Specifications*, <http://www.opengis.org/techno/specs.htm>.

[33] PCI Geomatics, *ER Mapper Rasters (ERS)*, <http://www.pcigeomatics.com/cgi-bin/pcihlp/GDB>.

[34] Race P., Rice D., & Vera R (2001). *JavaTM Image I/O API Guide*, Sun Microsystems, U.S.A.

[35] Radiya A., & Dixit V (2000). *The basics of using XML Schema to define elements*, <http://www-106.ibm.com/developerworks/xml/library/xml-schema/>.

[36] Recordare, *MusicXML Definition*, <http://www.recordare.com/xml.html>.

[37] SpatialNews, *GIS, Data & Mapping Related News*, <http://spatialnews.geocomm.com/dailynews/2001/sep/11/index.html>.

[38] St. Laurent S (1999). *Describing Your Data: DTDs and XML Schemas*, <http://www.xml.com/pub/a/1999/12/dtd>.

[39] Sun Microsystems, *Introduction to CORBA*, <http://developer.java.sun.com/developer/onlineTraining/corba/corba.html>.

[40] Sun Microsystems, *Java Advanced Imaging API*, <http://java.sun.com/products/java-media/jai/>.

[41] Sun Microsystems, *Java API for XML Parsing Release: 1.0*, <http://java.sun.com/xml/jaxp-docs-1.0.1/docs/api/>.

[42] Sun Microsystems, *Java API for XML Parsing Release: 1.1*, <http://java.sun.com/xml/jaxp-docs-1.1/docs/api/index.html>.

[43] Sun Microsystems, *Java Remote Method Invocation*, <http://java.sun.com/products/jdk/rmi/>.

[44] Sun Microsystems, *Java Technology and XML*, <http://www.java.sun.com/xml>.

[45] Sun Microsystems, *The Java Platform*, <http://java.sun.com/aboutJava/>.

[46] Sun Microsystems, *The JDBC Database Access API*, <http://java.sun.com/products/jdbc/>.

[47] The Apache Software Foundation, *The Apache XML Project*, <http://xml.apache.org/>.

[48] The Object Management Group, *Common Object Request Broker Architecture (CORBA)*, <http://www.omg.org/corba/>.

[49] U.S National Imagery and Mapping Association, <http://www.nima.mil/>.

[50] U.S National Imagery and Mapping Association, *U.S. Imagery and Geospatial Information System (USIGS) specifications*, <http://www.nima.mil/sandi/arch/products2.html>.

[51] U.S National Imagery and Mapping Association, *Geospatial and Imagery Services (GIAS) specifications*, <http://www.nima.mil/sandi/arch/addinfo.html>.

[52] Wason J. *Converting from XML Schema data types to SQL data types*, <http://www.ecs.soton.ac.uk/~jlw98r/xmlSchema2SQLdatatypes.htm>.

[53] World Wide Web Consortium, <http://www.w3.org>.

[54] World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 Specification*, <http://www.w3.org/TR/REC-xml>.

[55] World Wide Web Consortium, *HyperText Markup Language Home Page*, <http://www.w3.org/MarkUp>.

[56] World Wide Web Consortium, *Schema Tutorial*, <http://www.w3schools.com/schema>.

[57] World Wide Web Consortium, *XML Schema Part 0: Primer*, <http://www.w3.org/TR/xmlschema-0>.

[58] World Wide Web Consortium, *XML Schema Part 1: Structures*, <http://www.w3.org/TR/xmlschema-1>.

[59] World Wide Web Consortium, *XML Schema Part 2: Datatypes*, <http://www.w3.org/TR/xmlschema-2>.

[60] World Wide Web Consortium, *XML Tutorial*, <http://www.w3schools.com/XML>.

[61] World Wide Web Consortium, *XSL Tutorial*, <http://www.w3schools.com/XSL>.

[62] World Wide Web Consortium, *XSL Transformations (XSLT) 1.0 Specification*, <http://www.w3.org/TR/xslt>.

[63] Wrox, *Wrox.com Programmer to Programmer*, <http://www.wrox.com>.

[64] XML Global, *A resource site for XSLT, XSL & XML technologies*, <http://www.xslt.com>.

[65] *XMLSoftware*, <http://www.xmlsoftware.com>.