Translating Between XML and Relational Databases

Miqdad Asaria 2001/2002

Abstract

This project aims to implement a framework for inter-model transformations of data, built on the Hypergraph Data Model (HDM). The work takes two data models as examples, XML and Relational Databases, and shows how they can be implemented within this framework.

Online Resources

The software, the source code, the documentation, this report, the user guide and any other project related information can be found on the project website located at:
http://www.miqdad.webstar.co.uk/individual_project_2002/

Acknowledgement

I would like to take this opportunity to thank my supervisor Dr Chris Hogger and my second marker Dr Frank Kriwaczek for their guidance and support. I also thank Dr Peter McBrien who proposed the project and provided some initial direction.

Contents Page

1. Introduction

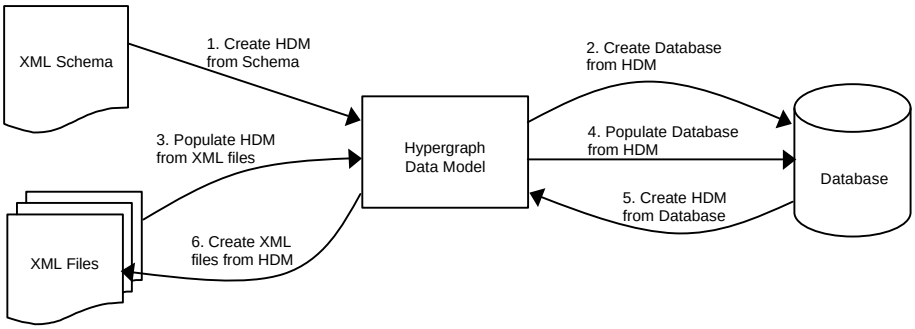
1.1 Motivation

With the invention of computers and the emergence of the Internet, the world is currently being transformed by the information revolution. Unimaginable amounts of data are now available at our fingertips from a correspondingly vast range of different sources. The problem now is to be able to make this information meaningful and usable. It is no longer possible to read everything to find what we are looking for, we need to be able to query the information to get what we want. To service this need databases have been developed to efficiently organise and query data. The other problem we face is that information resides in many incompatible formats, so even though we have all this information we are unable to fully utilise it. To service the need for data interchange we have seen the recent emergence of XML. The logical next step would be to link these two exciting technologies together to fully unleash the potential of the information available to us. This is where my project fits in.

1.2 Overview

I realised that data is modelled in different ways to make it meaningful for different purposes. XML is very useful for data interchange, databases for data storage and querying, UML for modelling object oriented concepts etc. Therefore I decided that rather than making a direct conversion tool to translate between XML and relational databases, it would be much more useful to create a framework for inter-model transformations, centred on some common data model. Then new data models could be added into this framework by providing ways to convert to and from this common data model. For reasons I discuss in the background section of the report I have chosen the Hypergraph Data Model (HDM) as this common data model.

For the use of this project I will implement the core HDM framework and add XML and relational database handling capabilities into the framework. The diagram below highlights the key features of my implementation.



XML is still a developing set of ideas [1], but with the recent publication of the XML Schema specification [2,3,4] it is fast maturing into a fundamental computing technology. The rich semantics of the XML Schema definition language, and the portable nature of XML documents have established it as the standard for information exchange. My project focuses on finding

ways of extracting and utilising the information in the XML Schema to create a powerful translation tool to translate between XML files and relational databases.

1.3 Report Structure

Chapter 2 provides a background study of the existing solutions to the translation problem, the theory behind the Hypergraph Data Model, and the technologies used in the implementation.

Chapter 3 details the actual implementation process with sub-sections dedicated to dealing with each of the transformations shown by the arrows in the diagram above.

Chapter 4 contains an evaluation of the end product, highlighting the strengths and weaknesses.

Chapter 5 draws conclusions from the project, highlighting the contribution of the project and proposes future directions for work in this area.

Chapter 6 is a bibliography of all the references used in the project. The references are numbered and referenced in the report by these numbers e.g. [1] in the text refers to the reference numbered [1] in the bibliography i.e. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, available at http://www.w3.org/TR/REC-xml, October 2000.

Chapter 7 contains the appendices, these include a user guide for the software and the complete version of the XML examples that are referred to throughout the project.

2. Background

2.1 XML

The Extensible Markup Language (XML) [1] is a subset of SGML and is a standard for defining and sharing data. XML files can be broken down into two pieces the header and the content the header gives information for parsers and applications to decide how to handle the document. A typical header looks like: `<?xml version="1.0" encoding="UTF-8" ?>`

```
<?xml version="1.0" encoding="UTF-8" ?>
<book>
      <title>The title of the book</title>
      <chapter number="1">
            <heading>Chapter 1</heading>
            <para>….</para>
            <para>….</para>
      </chapter>
      <chapter number="2">
            <heading>Chapter 2</heading>
            <para>….</para>
            <para>….</para>
      </chapter>
</book>
```

book.xml

The content part of the XML file consists of elements attributes and data. Each XML file has exactly one root element. This is the highest-level element with the first opening tag and the last closing tag in the content. All other content is contained within the root element. The root element in the above example is book.

Elements are XML tags which obey certain rules: names must start with a lowercase letter or an underscore and must not contain space; for every opening tag there must be a corresponding closing tag; closing tags should appear at the same level of the nesting hierarchy as opening tags. Elements can contain other elements (as show by the book element in the example) they may contain attributes (as shown by chapter element with attribute number) and may contain character data (as shown by the title element with character data "The title of the book").

Attributes are name value pairs declared in the opening tag of the associated element. Values must be enclosed in quotation marks and each instance of an element can only have a single value for any given attribute. The element chapter has attribute number in the above example.

XML files can also contain comments, processing instructions and namespace information, for a full description of XML the reader is encouraged to refer to [1].

Because XML is very flexible allowing users to make up custom elements and attributes, the XML documents have no meaning by themselves. We need a way of defining constraints on the documents to provide meaning to the various components, there are two established

standards that serve this purpose: DTDs and XML Schemas. I will briefly describe these in the following sections of the report.

2.1.1 XML DTD

The Document Type Definition (DTD) can be used to define, in a very limited non-XML syntax, the constraints on the XML file. The example below shows a possible DTD for book.xml example described above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT book (title, chapter+)>
<!ELEMENT chapter (heading, para+)>
<!ATTLIST chapter
    number CDATA #REQUIRED
>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT para (#PCDATA)>
<!ELEMENT title (#PCDATA)>
```

Essentially the DTD defines which elements and attributes can occur in the XML file and occurrence constraints on these elements and attributes. The above DTD defines that a book contains title and chapter elements. If an element name is followed by a + this means that the specified element must occur 1 or more times, ? means 0 or 1, * means 0 or more and if no occurrence constraint is specified it means the element should occur exactly once. #PCDATA means the element contains character data.

The attributes an element has are defined in the <!ATTLIST> tag. In the example we can see that the chapter element has a number attribute of type CDATA[1]. Occurrence constraints for attributes are specified in terms of the #REQUIRED key word.

The DTD does not support any data typing, value ranges, pattern matching, inheritance and extension of types. To overcome these limitations the XML Schema was produced as an alternative constraining language.

2.1.2 XML Schema

XML Schema [2,3,4] is an XML constraining language actually written in XML. It has a much larger vocabulary than DTD, so can produce a much richer set of semantic integrity constraints. Below we can see an example schema for book.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xsd:element name="book">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="title"/>
                <xsd:element ref="chapter" minOccurs="1" maxOccurs="10"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="chapter">
```

[1] CDATA represents character data in attributes. It can be thought of as equivalent to PCDATA for elements

```
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="heading"/>
                <xsd:element ref="para" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="number" use="required">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:integer">
                        <xsd:minInclusive value="1"/>
                        <xsd:maxInclusive value="10"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="heading" type="xsd:string"/>
    <xsd:element name="para" type="xsd:string"/>
    <xsd:element name="title" type="xsd:string"/>
</xsd:schema>
```

As we can see the schema is much more verbose than the DTD, and allows us to fully specify minimum and maximum occurrences values e.g. minimum 1 and maximum 10 chapters. It also allows us to specify minimum and maximum values e.g. 1 and 10 for chapter numbers. Schemas have many more features and whole projects could be written describing them. I will deal with the various features as we come across them in the rest of the report.

2.1.3 XML APIs

To access XML in computer programs we require API's[2]. Two main API's have emerged to access XML, SAX and DOM [6].

The Simple API for XML Parsing (SAX) sequentially parses the XML data using call back functions triggered on processing each element, attribute or processing instruction. This has the advantage of being very fast and memory efficient but results in complicated code.

SAX can be used to create a Document Object Model (DOM) this is tree representation of the whole XML file in memory. This can require a lot of memory and can be slow, but has the advantage that it provides the programmer with a familiar tree structure to write powerful recursive functions over.

In addition to these two a new standard has just emerged called JDOM, this is similar to DOM in that it works on a tree structure in memory but is more programmer oriented with compound functions built in for commonly used operations.

XML gives us a self-describing data format, this means the format is platform independent giving us portable data. The XML API's all have Java implementations. Programs written in Java have the property that they will run on any platform for which a Java Virtual Machine has been written i.e. give us portable code. The combination of Java and XML gives us portable code and portable data, therefore being ideal for the heterogeneous-network based applications that are currently dominating computing.

[2] API – Application Programming Interface, allows programs to access and modify data-structures e.g. XML using predefined functions.

2.2 Databases

2.2.1 Native XML Databases

A number of approaches exist for storing XML in databases. The most ambitious of these is the emergence of a new class of databases created specifically to handle XML data. These are called native XML databases. There are a number of different methods of implementing these, and there does not seem to be any standard way of storing or querying the data. Additionally since these databases are very new they have yet to be fully optimised.

This option is not really appropriate solution to the translation problem at the present time. The lack of standards require that programmers have to learn new query languages and write different code to access different databases. Non-XML data cannot be added to the databases to augment the XML data, and the systems are still very immature so have not been highly optimised to achieve the levels of performance and reliability of RDBMS.

Some interesting research prototypes in this area include Rufus, Lore [17], Strudel, Natix, some have even gone into commercial production like Tamino and eXcelon. This may become a much more viable option in the future as the technology matures.

2.2.2 Object Oriented Databases

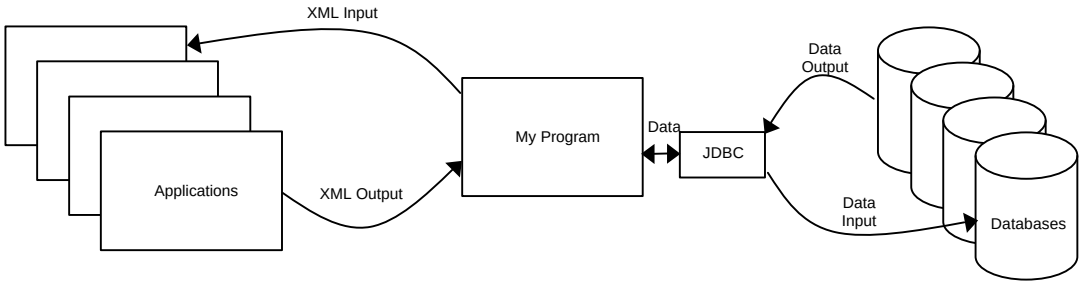
A more attractive option would be to use Object Oriented (OO) databases. These would naturally accommodate the storage of the DOM representation of the XML files. OO databases are more established than XML databases, have a standard query language in OQL, and have been around for some time now. However they are not widely used and so there has not been the same level of investment in optimising and developing them. Currently most data is stored in relational databases, ideally we would like to identify a solution to the translation problem that automatically utilises our existing data. For this reason I am more inclined towards a solution that focuses on relational databases.

2.2.3 Relational Databases

Relational databases are the most widely used, proven and optimised databases. While they may not be the ideal structure to store XML data, it would seem that for the practical reasons described above they are the only real option. Realising this all the major database vendors have extended their RDBMS products to have some sort of XML support. Some examples are the extended SQL syntax of SQL Server 2000, XML Data-types in Oracle 9i, IBM's DB2 XML Extender with its fuzzy query language. Unfortunately each vendor has taken a different path to implementing XML support with no common standard methodology for handling XML. Also it seems that at present the quality of XML support is very limited.

Ideally we would like to have a solution that is database independent, allowing the user to have the same XML support independent of which database is being used. To meet this requirement I intend to develop an intermediate application to handle all the XML functionality with all communication with the database implemented in a database independent manner through Java Database Connectivity (JDBC) drivers. If my general solution is successful the database vendors could then provide optimised database specific implementations of the XML handling

standards propagated by my implementation. The diagram below shows how my program utilises JDBC to achieve database independence.



2.3 Intermediate Modelling Languages

2.3.1 Hypergraph Data Model

A suitable common data model for the purpose of inter-model transformations is the Hypergraph Data Model (HDM). I will describe it briefly here[3] a full description can be found in [13,16]. A schema in a HDM is a triple < Nodes, Edges, Constraints > . A query over the schema is an expression whose variables are members of Nodes ∪ Edges. Nodes and Edges define a labelled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes and other edges. Constraints is a set of Boolean valued queries over the schema. Nodes are uniquely identified by their names. Edges and constraints have an optional name associated with them.

An instance I of a schema S = < Nodes, Edges, Constraints > is a set of sets satisfying the following:

i. Each construct c ∈ Nodes ∪ Edges has an extent denoted by $Ext_{S,I}(c)$, that can be derived from I.
ii. Conversely, each set in I can be derived from the set of extents $\{Ext_{S,I}(c) \mid c \in$ Nodes ∪ Edges$\}$.
iii. For each e ∈ Edge, $Ext_{S,I}(e)$ contains only the values that appear within the extents of the constructs linked by e.
iv. The value of every constraint c ∈ Constraints is true, the value of a query q being given by $q[c_1 / Ext_{S,I}(c_1), ... , c_n / Ext_{S,I}(c_n)]$ where $c_1, ..., c_n$ are the constructs in Nodes ∪ Edges.

There are eight primitive transforms that we can apply to HDM models:

1. renameNode < fromName, toName > renames a node provided toName is not already the name of a node in the model.

2. renameEdge < < fromName, $c_1, ..., c_m$ > , toName > renames an edge provided toName is not already the name of an edge in the model.

[3] This description is taken mainly from [11]

3. addConstraint c adds a new constraint c provided that c holds true.

4. delConstraint c deletes a constraint provided that c exists.

5. addNode < name, q > adds a node named name whose extent is given by the value of the query q, provided that a node of the given name does not already exist.

6. delNode < name, q > deletes a node. The query q states how the extent of the deleted node could be recovered from the extents of the remaining schema constructs. Deleting is successful provided the node exists and does not participate in any edges.

7. addEdge < < name, c_1, ..., c_m > , q > adds a new edge between a list of existing schema constructs c_1, ..., c_m. The extent of the edge is given by the value of the query q. Adding succeeds provided that the edge does not already exist, all the constructs involved in the edge do exists and the query q satisfies the appropriate domain constraints.

8. delEdge < < name, c_1, ..., c_m > , q > deletes an edge. The query q states how the extent of the deleted edge can be recovered from the extents of the remaining schema constructs. Deleting is successful provided that the edge exists and does not participate in any other edges.

We now need to define how we can use these primitive transformations to represent higher level modelling languages. All modelling languages can be broken down into a combination of extensional and constraint constructs. We need to map operations on these constructs to operations on the underlying HDM.

We can further break down the concept of extensional constructs into [15]:

a.) nodal constructs – these can be present in a model independently of any other constructs and map to HDM nodes.
b.) linking constructs – these can only exist in a model when certain other nodal constructs exist. The extent of a linking construct is a subset of the Cartesian product of the extents of these nodal constructs. Linking constructs map to edges in the HDM.
c.) nodal-linking constructs – these are nodal constructs that can exist only when certain other nodal constructs exist, and are linked to these constructs. Nodal-linking constructs map to a combination of a node and an edge in the HDM.

The two data models focussed on in this project are XML and relational databases. The mapping between XML and HDM [11] is:

1. An XML element may exist by itself and is not dependent on the existence of any other element. We can see it as a nodal construct, mapping directly to the concept of a node in HDM.
2. An XML attribute a of an XML element e may only exist in the context of e, and hence a is a nodal-linking construct. It is represented by a node < < e_a > > in the HDM, together with an associated unlabelled edge < < _, e, e_a > > connecting the HDM node representing the attribute a to the HDM node representing the element e.

A constraint states each instance of the attribute is related to at least one instance of the element.

3. Element nesting in XML can be represented by a set of edges. The nesting of elements e_1,....., e_n within an element e, can be represented by the set of edges $< <\_, e, e_1 > > \ldots < <\_, e, e_n > >$. Each such edge is an individual linking construct.
4. Additionally in XML order may or may not be of semantic importance. If we want to represent order information i.e. list based semantics we add an additional edge to an instance of an order node.

The mapping between the relational model and HDM follows a similar fashion in which relations, attributes, primary and foreign keys are represented as combinations of extensional and constraint constructs. We will go into more detail in the implementation section of the report.

2.3.2 Other Modelling Languages

Many other intermediate/common data modelling languages exist. Some have been specifically designed for XML, others are very informal. None of the other languages I came across were able to fully represent both extensional and constraint construct for multiple different data models. For more on other languages see [17,18,19,20,25].

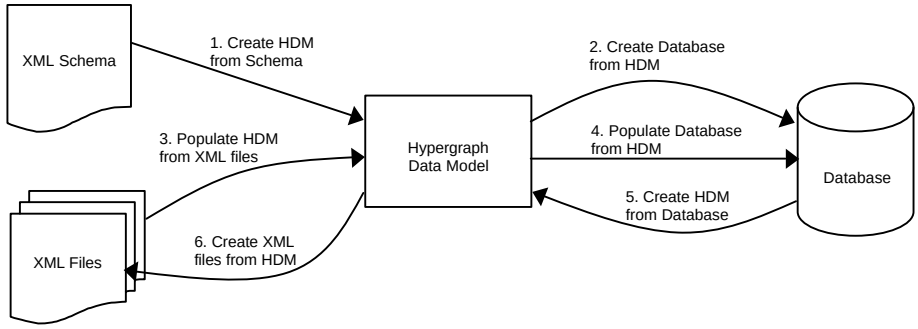
The features that convinced me to choose HDM above these other languages were:

1. HDM is not specifically designed to handle only certain modelling languages it is low level enough to be able to model any other modelling language.
2. A lot of research has already gone into HDM and how it can be used to model many different higher level modelling languages see [11,12,13,14,15,16,22] this means a lot of the theoretical work of how to use the model has been done already.
3. All the transformations in HDM are uniquely reversible, this coupled with the schema transformation repository see [22,23] allows for some very powerful applications of the model, some of which I will be discussing in more detail later in this report.
4. HDM is a directed graph based mathematical formalism, so it is structured enough to be able to prove certain properties over, and being based on a directed graph (a common mathematical structure) many of these properties have already been proven.

3. Implementation of the Software

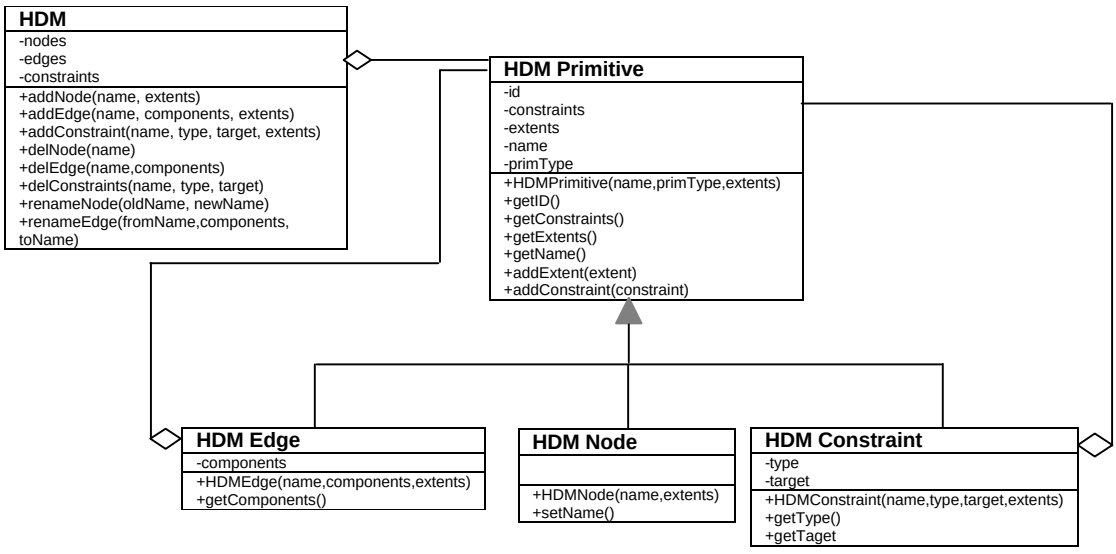
3.1 Overview of the Software

The software can be broken into four main modules, the implementation of the core HDM model, the translation of XML Schemas into this model, the translation of relational databases to and from this model, and the translation of XML files to and from this model. These main components are shown in the diagram below and I will be describing them in more detail in the following sections.



3.2 Implementation of the Hypergraph Data Model

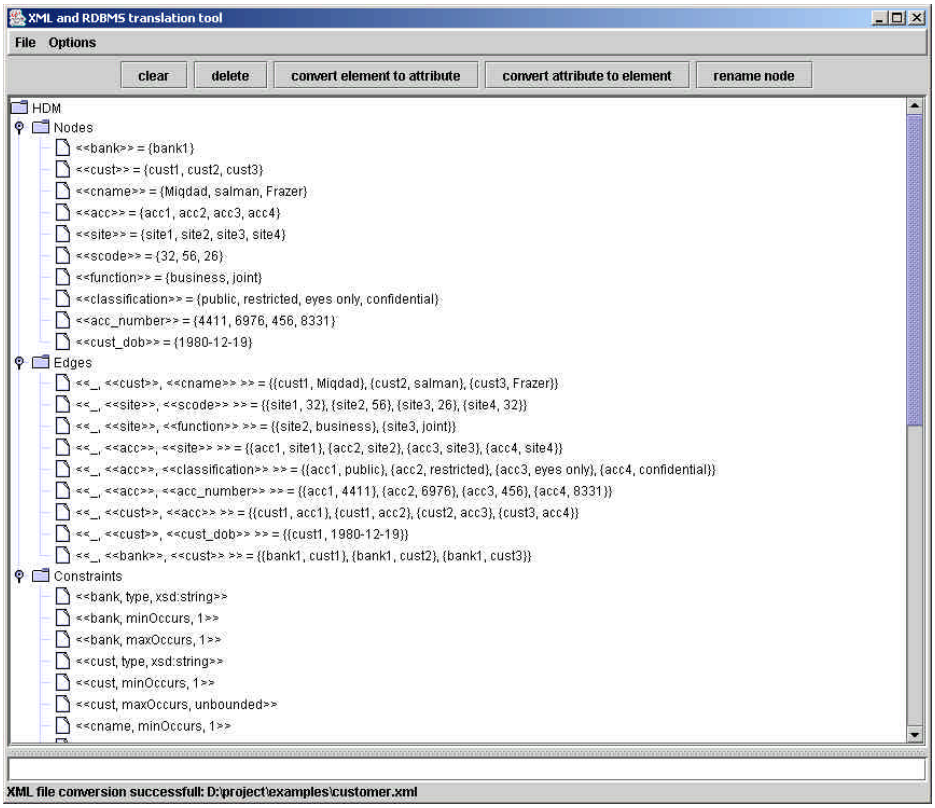
The central component of the software is the implementation of the Hypergraph Data Model. I decided that I should do the implementation of the project in Java to fully compliment the portable data of XML with the portable code. In my implementation I have attempted to stay as close to the theoretical model as possible.



The above diagram shows the classes used in my implementation of HDM. The HDM class contains collections of nodes, edges and constraints that make up the model. It also provides the eight primitive transforms over the HDM (described in the background section of the report).

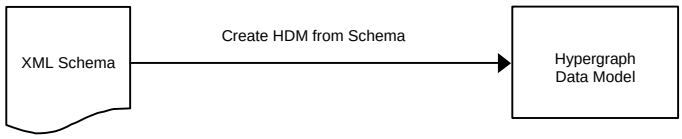
Nodes, edges and constraints are all HDM primitive components and share many common features. To optimise for this redundancy I have created an HDM Primitive super-class containing the common information and sub-classed this to create the HDM Nodes, HDM Edges and HDM Constraints. Since the model does not require edges and constraints to have unique names I decided that for ease of implementation I would give each HDM Primitive a unique ID to enable the other parts of the program to identify it.

After implementing this core component I implemented a basic graphical user interface (GUI) to display the HDM. As the HDM is the central component through which all data must pass, it seems that this is what must be graphically represented. I decided that a simple tree structure would suffice with branches for the nodes, edges and constraints.



I considered constructing a much more elaborate visualisation of the directed graph structure, but decided against this as I felt that benefits gained would be mostly cosmetic and would not really contribute to the academic content of the project. I decided to leave this as an optimisation to be implemented at the end of the project. With this in mind I ensured the that the GUI code was completely independent of the model so that it could be effectively unplugged and a new GUI plugged in, in its place.

3.3 Translating From XML Schema to HDM



The first part of the project I am going to tackle is the transformation of the XML Schema into HDM. As was described in the background section the XML Schema is actually an XML document itself. Therefore for this part of the project the schema file needed to be parsed using a Java API for parsing XML. In this instance I decided to use JDOM as this API gives the most intuitive access to the XML file. This choice inevitably involves a trade-off between ease of use on the one hand and speed and memory efficiency on the other (e.g. using SAX would be much faster and more memory efficient but would be harder to code with). I decided that the focus of the project should be to make theoretical progress not to produce a highly optimised application, and to this end the easier but less efficient API would be more appropriate. When complete the application can always be optimised to use a more efficient API.

The XML Schema consists of elements, attributes and type declarations, from these the elements are nodal constructs and attributes are link nodal. Type declarations embody the relationship between the elements and the attributes (linking constructs), contain element and attribute declarations and define constraints over these elements, attributes and relationships (constraint constructs). For the definitions of the different types of constructs please refer to the background subsection on HDM.

I will first give a quick overview of the schema parsing algorithm here and then describe some interesting aspects of it in more detail in the following subsections. The algorithm works by first parsing the schema and making lists of all the explicitly declared complex and simple types and globally declared elements. It then goes through each of the globally declared elements and for each of these it adds the element as a node in the HDM. It finds the corresponding type declaration and adds the relevant constraints from this. Through the type declaration it finds any child elements and attributes of the element, and adds edges from the parent to each of the children. It then recurses over the children. The recursion bottoms out at children that are leaf elements or attributes, these obviously have no children. Nodes and edges are added to the model with empty extents, extents of the model will be populated by XML file instances of the schema.

Throughout this section I will be referring to the bank example XML Schema and instance files reproduced in full in Appendix A – Bank Example.

3.3.1 Representing XML Nodal Constructs in HDM

Nodal constructs map directly to HDM nodes. In the XML Schema elements play the role of nodal constructs. In the schema, elements can be declared in a number of ways:

1. As top-level element declarations nested directly under the root element e.g. the bank element in the example:

```
<xsd:element name="bank" type="bankType"/>
```

2. As nested elements declared in a top level complex type declaration e.g. element cust declared in the complex type declaration of bankType:

```
<xsd:complexType name="bankType">
        <xsd:sequence>
            <xsd:element name="cust" type="customer" maxOccurs="unbounded"/>
        </xsd:sequence>
</xsd:complexType>
```

3. As nested elements declared in nested anonymous complex type declarations e.g. element scode in the anonymous complex type declaration of site:

```
<xsd:element name="site">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="scode" type="xsd:integer"/>
            <xsd:element ref="function" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

4. As references to elements declared at the top level e.g. the element in the above example with the reference to the top level element function shown below:

```
<xsd:element name="function">
<xsd:simpleType>
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="business"/>
        <xsd:enumeration value="personal"/>
        <xsd:enumeration value="joint"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:element>
```

When the algorithm is searching for the type of the element it may find that the element is actually just a reference to another element. In this case the reference should be replaced by the referenced element. Once references have been replaced we can deal with the type definition of the element. The type definition may be anonymous, in which case the element has no type attribute as in examples 3 and 4, where the type definition is included as part of the content of the element declaration (there can be several levels of nesting in this way). Alternatively the type definition may be explicitly defined, in this case either the type attribute points to a

named global type declaration such as in example 1 or the type is a base type of the xml schema definition (XSD) language such as in the declaration of the scode element in example 3. Base XSD types can be identified as those prefixed with the namespace prefix assign to the XML Schema constructs xsd in this case.

At this point the algorithm splits elements according to whether they are of simple types[4] or of complex types. Simple typed elements and attributes are added as nodes to the HDM and the relevant constraints are added to the model as will be discussed in section 3.3.4.

3.3.2 Representing XML Linking Constructs in HDM

Linking constructs map directly to HDM edges. In the XML Schema the nesting of elements play the role of the linking relationship. This nesting is represented in complex type definitions which as discussed above can be explicitly named or anonymous.

When dealing with elements of complex type, first we add the node to the HDM model with its corresponding constraints. Then we look to the complex type definition to find the children of this complex element. For example for the element cust shown in example 2 above the corresponding complex type definition is:

```
<xsd:complexType name="customer">
    <xsd:sequence>
        <xsd:element name="cname" type="xsd:string"/>
        <xsd:element name="acc" type="account" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="dob" type="xsd:date" use="optional" default="1976-11-05"/>
</xsd:complexType>
```

We can see that the cust element has three children the elements cname and acc and the attribute dob. The algorithm adds an edge between cust and each of its children (three edges in total). The algorithm then recurses on each of these children. The recursive nature of the algorithm will correctly deal with any complex children.

3.3.3 Representing XML Link Nodal Constructs in HDM

Link nodal constructs map to a node and an edge in HDM. In the XML Schema attributes play the role of link nodal constructs. As the algorithm has already added the necessary edge to between the parent element and the attribute (see 3.3.2) it now must add the attribute node. These need to be handled separately from other simple type nodes for two reasons, firstly the way constraints are defined over attributes differs to the way they are defined over elements, and secondly we will need to follow some naming convention to identify attribute in the HDM model to enable us to reverse the conversion process. The naming convention I have followed is to prefix the attribute name with its parent element name, separating the two by an underscore, e.g. in the example above (see 3.3.2) the dob attribute of the cust element would be represented by a node named cust_dob.

[4] simple type here refers to XSD base types or declared or anonymous XSD simple types

3.3.4 Representing XML Constraint Constructs in HDM

Constraints are represented in the XML Schema in a number of ways:

1. As attributes in element definitions, these can both directly state constraints and point to a location in the schema where the constraints are defined:

```
<xsd:element name="acc" type="account" minOccurs="0" maxOccurs="unbounded"/>
```

   The example above shows some of these constraints on the element, the maxOccurs and minOccurs attributes directly state values while the type attribute directs the reader to the user-defined account type to get further constraints. Where the type refers to a user-defined simple type, or the element contains an anonymous simple type definition or the type attribute value is an XSD base type, the element can have additional constraining attributes such as attributes giving default values, final values, fixed values.

2. As attributes in attribute declarations, these take a similar form to the constraints defined in element declarations, the one significant difference being the way occurrence constraints are defined. Attributes declarations have an attribute use which can take values required/optional/prohibited:

```
<xsd:attribute name="dob" type="xsd:date" use="optional" default="1976-11-05"/>
```

3. As attributes of elements or attributes that reference other elements, in such cases the constraints on referencing element or attribute are added to the constraints on the referenced attribute:

```
<xsd:element ref="function" minOccurs="0"/>
```

   In the above example the declared element has all the constraints of the function element and additionally has the minOccurs=”0” constraint.

4. As part of simple type declarations. Simple type declarations build on a base XSD type or another simple type, adding restrictions to define more refined types. When the definitions build on other simple type definitions then we need to add any constraints defined in the base type to the constraints extracted directly from the type declaration. This nesting of definitions within definitions can occur through an arbitrary number of levels:

```
<xsd:element name="function">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="business"/>
            <xsd:enumeration value="personal"/>
            <xsd:enumeration value="joint"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

The example above shows an anonymous simple type definition for the function element. The simple type is built by applying restrictions to the XSD base type string. The restrictions applied enumerate the values that the element is allowed to take i.e. business, personal or joint. Many other restricting facets can be applied in simple type definitions, the facets that can be applied depend on the XSD base type being extended. For example types built on the string base type can be constrained to conform to patterns defined by regular expressions, have minimum, maximum or specified lengths. XSD integer types can take value constraints:

```
<xsd:attribute name="number" use="required">
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
            <xsd:minInclusive value="10"/>
            <xsd:maxInclusive value="3000000"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:attribute>
```

The example above shows that the value of the number attribute must be a positive integer between 10 and 3000000. Similarly many other restricting facets exist for these and other XSD base type. Simple types as well as being derived by restriction can also be derived by union and list constructs.

5. As part of complex types. Complex types can extend other complex or simple types by restriction and by extension i.e. adding more child elements and attributes. In addition to this complex type definitions can contain further constraints in the form of the content model that they follow. The possible content models include “sequence”, “choice” or “all”. The “sequence” content model implies that the declared child elements appear in instance documents in the order that they are declared in schema’s complex type declaration. The “all” content model implies that the order is not important. The “choice” content model implies that only one of the declared children elements can occur under an instance of the parent element:

```
<xsd:complexType name="customer">
    <xsd:sequence>
        <xsd:element name="cname" type="xsd:string"/>
        <xsd:element name="acc" type="account" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="dob" type="xsd:date" use="optional" default="1976-11-05"/>
</xsd:complexType>
```

The example above shows the complex type definition for the customer type, the definition indicates that an element of type customer optionally has a dob attribute of type date with a default value of 1976-11-05. It also states that elements of customer type must contain exactly one cname child element followed by any number (including zero) of acc elements.

The algorithm adds constraints immediately after it adds the corresponding nodes. The list of constraints to add for the node are built up by following the relevant different ways of finding and collecting constraints described above. These constraints are then added to the HDM and to the node that they apply to. Constraints are represented in the program by a constraint type, a target node or element to which they apply and the extents of the constraint which provide the details of the possible values or value restrictions:

```
<xsd:element name="classification" default="business">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="public"/>
            <xsd:enumeration value="confidential"/>
            <xsd:enumeration value="eyes only"/>
            <xsd:enumeration value="restricted"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

In the above example the algorithm would add the default value constraint, the type constraint and the restricted values enumeration in the following way.

Constraint 1: type = “default”, target = “classification”, extent = {“business”}
Constraint 2: type = “type”, target = “classification”, extent = {“xsd:string”}
Constraint 3: type = “enumeration”, target = “classification”, extent = {“public”, “confidential”, “eyes only”, “restricted”}

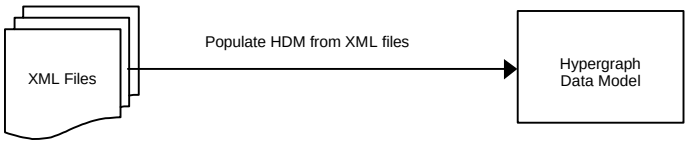
These constraints will be added to the HDM model and references to them will be added to the classification node (the node which represents the XML element classification).

In a similar way the algorithm handles the addition of many of the other constraints, some times the absence of explicitly declared constraints implies that they should take some default values e.g. minOccurs and maxOccurs both take the value “1” if not explicitly defined.

Due to the large number of different constraints I will not describe how the program deals with each different constraint here, but refer the reader to the extensively commented source code file *“XMLSchema2HDM.Java”* lines 417 to 769 where it can be seen how the program deals with each of the different constraints.

3.4 Translating Between XML and HDM

3.4.1 From XML to HDM



Once we have built a representation of the XML constructs in HDM by parsing the XML Schema, we can then populate the extents of the nodes and edges using instances of the schema. It could be argued that the XML constructs can also be built up from the XML file and there is no need for the schema at all. This is the approach taken by most of the current conversion methodologies, however I recognised that this was not a very complete approach for two reasons:
Firstly XML instances of a schema do not necessarily have to contain all the element and attributes declared in the schema e.g. those attributes declared as optional in the schema may be absent from an instance. This lack of information in the model will render it incompatible with certain valid instances of the schema that contain the additional optional information not built into the model.
Secondly the XML instance file does not contain the information about the constraints on the data, this can only be obtained from the XML Schema or DTD. Without this information data added to the model from other sources for example through the database, will not be forced to conform to the constraint in the XML Schema. This augmented model may contain data violating the constraints in the schema and so we will not be able to convert from the model back to a valid XML file instance of the schema.
For these reasons I decided the better option would be to build the model from the XML Schema and populate the extents from the XML file instances of the schema.

Again in this section I will be referring to the bank example XML Schema and instance files reproduced in full in Appendix A – Bank Example.

The algorithm for parsing the XML file uses the lists of elements, attributes, complex types and simple types built up from parsing the XML Schema. For each of the attributes, the algorithm adds the instance values from the XML file as extents to the node representing the attribute in the HDM.

From our bank example we can see that there will be an attribute named acc_number. After processing the XML Schema this will be added as a node to the HDM, the algorithm will look for all instances of the acc_number attribute, i.e. the number attribute of the element acc. In the file, it finds that these take values 4411, 6976 and 8331. These are added to the extent of the HDM node acc_number giving it the extent {“4411”, “6976”, “8331”}.

The algorithm then traverses the list of elements. The extents of the leaf elements[5] are populated in a similar fashion to attributes. Elements of complex types contain child elements and attributes. When the algorithm comes across these complex elements it must add instance values to the extent of the node representing the element and add instance values to the extents of the edges linking this element to its child elements or attributes.

Instance values for the complex elements are generated sequentially in order of their occurrence in the instance file. The convention used for these values is that they consist of a number representing the occurrence number, prefixed by the name of the element they are used to identify. From our bank.xml example the first cust element is given an instance value cust1 and the second cust2. The algorithm also recognises that the user may have already loaded an XML instance file into the model or have compatible instance data in the database, to accommodate for this before generating instance values for an element, it first checks for instance values already present for the element in the model and the related database. If there are any existing instance values the value with the highest occurrence number is set as the base value and the new instances are assigned values following on from this base value.

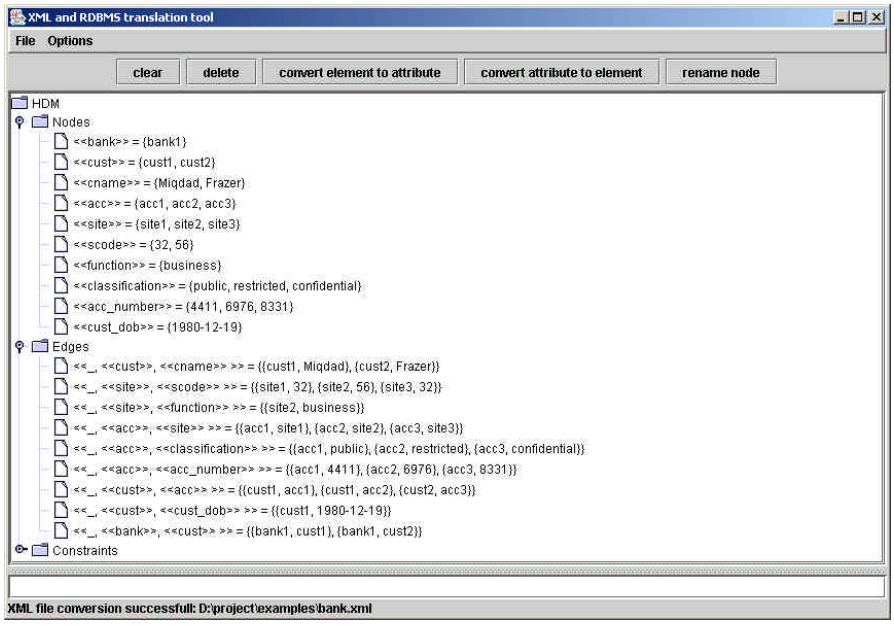
Once generated the relevant instance value is stored in the extent of the corresponding HDM node. The algorithm then looks to the complex type definition of the element to find the child elements and attributes. These children are located in XML instance files and HDM edge extents are populated with the pairs of values representing the parent element instance and the child element instance. From our bank.xml example the edge between cust and cname will be populated with the values: {(“cust1”, “Miqdad”), (“cust2”, “Frazer”)}.

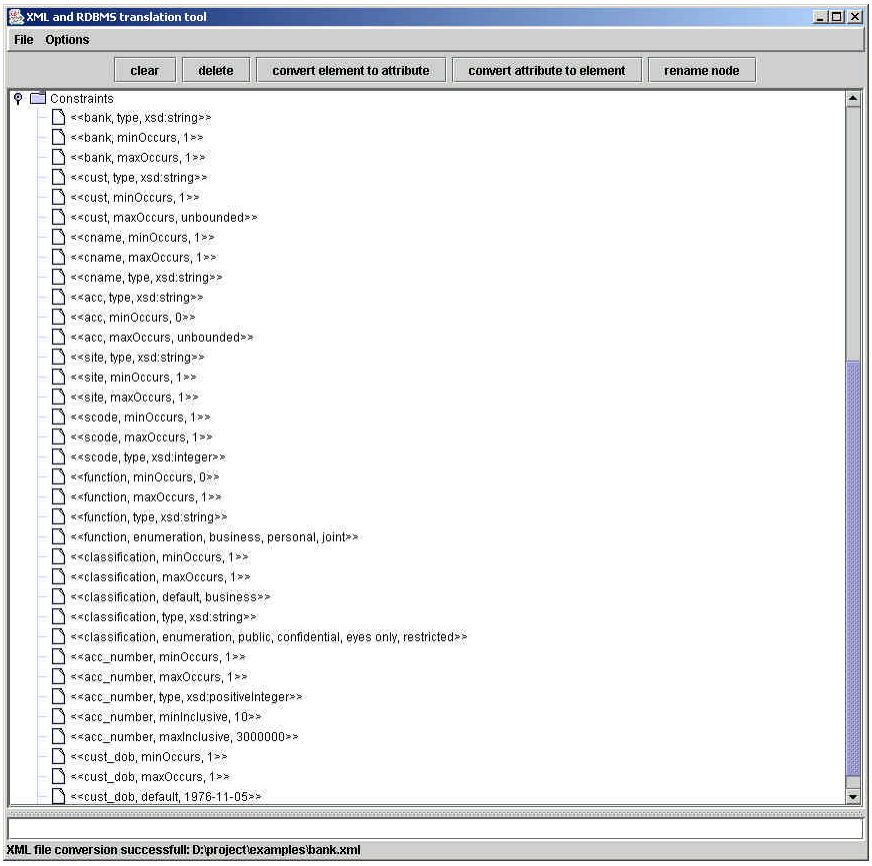
In some cases the child element will also be a complex element in which case we will also need to generate instance values for these before being able to populate edge extents e.g. the edge between bank and cust in our bank example. The extent of this edge will take values: {(“bank1”, “cust1”), (“bank1”, “cust2”)}.

There is an additional complication in that when generating instance values it would be semantically incorrect to generate more than one instance value for the root element in the XML file even if the root element already has a value in the model. This is because there can only ever be one root element in an XML file and therefore only one root element in a XML compatible HDM model. Looking again to the bank example, it would be wrong to generate an instance value “bank2” if the model already contains “bank1” generated from the processing of another XML instance file. The algorithm checks for this special case of the root element and ensures multiple root instances are not generated.

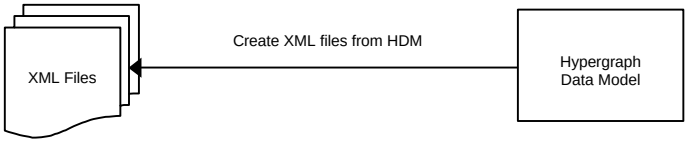
[5] Leaf elements are those elements that are of simple types and have no child elements or attributes

The full results of the parsing of the schema and XML instance for the bank example are shown in the following screen shots.





3.4.2 From HDM to XML



The conversion from HDM to XML is much simpler. The algorithm works by building up a JDOM document[6] in memory. It begins by identifying the root node in the HDM model. This node is located by searching for the node that has no parent. An XML element to represent the root node is added to the JDOM document. Next we find all the child nodes of the root node by searching for edges originating at the root element, the destination of these edges are the children of the root node. From these the nodes representing attributes[7] are added to the root element in the JDOM document along with their corresponding attribute values. To find the attribute values the edge between the root node and the node representing the attribute are identified, and the value representing the attribute in the edge extent is added as the value of the attribute in the JDOM file. Child leaf nodes are added in a similar fashion.

For child nodes that are not leaf nodes the algorithm looks to the extents of the edges linking them to their parent node. For each extent pair where the parent instance matches the current parent instance value, a new node in is created in the JDOM document to represent the child instance, any attributes that the child instance has are added by searching the extent of the edges from the node to its child attributes to find edges that originate at this particular child instance. If such a value is found in the edge extent the corresponding attribute with its instance value is added to the JDOM document. Leaf node children of the instance are added in a similar fashion. For the child nodes that are not leaf node we apply this sub-procedure recursively (i.e. the sequence of operations described in this paragraph is applied recursively to them). Once the complete JDOM document has been constructed, a standard JDOM procedure is used to write this document out as an XML file.

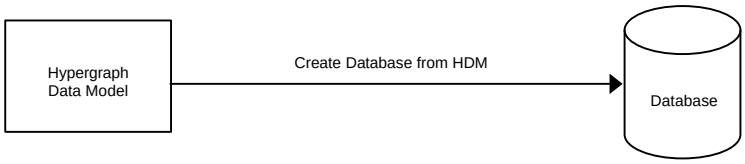
To clarify how the algorithm works we can consider our running bank example. We can identify the root node as bank and so add a bank element to our JDOM document. There are no attribute node children or leaf node children for the bank node. However there is a edge to the complex node cust. The instance value of bank is “bank1”, we look at the extent[8] of the edge linking bank and cust: {(“bank1”, “cust1”), (“bank1”, “cust2”)}. For each of “cust1” and “cust2” we add a child cust element under the bank element. Dealing first with “cust1” we look for all edges to attributes of cust with “cust1” in the source value of the edge extent. We find an edge to the attribute dob with extent: {(“cust1”, “1980-19-12”)} so we add an attribute to the cust element with name: “dob” and value: “1980-19-12”. Next we look for leaf node children, we find the node cname with edge extent containing (“cust1”, “Miqdad”) and so we add the

[6] A JDOM document is a Java object that represents an XML file as a tree structure.
[7] Attribute nodes can be identified by the naming convention employed i.e. parent element name followed by underscore followed by attribute name e.g. cust_dob
[8] The extents of the nodes are shown in the screen shot on the previous page

child element: “cname” with value: “Miqdad”. We then look for complex node children we find acc with its edge extent containing values: {(“cust1”,”acc1”), (“cust1”,”acc2”), (“cust2”,”acc3”)}, so we add two child acc elements representing “acc1” and “acc2”, we do not add and acc element for “acc3” as this relates to “cust2” not “cust1”. In this way we follow the algorithm until a complete JDOM document is built up equivalent to book.xml in Appendix A.

3.5 Translating Between Relational Databases and HDM

3.5.1 Creating a Database from HDM



To translate between the HDM model and a relational database it seems that the most natural mapping is to map complex nodes[9] to relations and leaf nodes and attributes as fields in these relations. The relationship between complex nodes and their complex children also need to be represented in the relational model. The best way to do this is by using foreign keys. Because a parent node can have many different children nodes but a child node can only have one parent node it makes sense to put this foreign key field in the relation representing the child. Following this methodology we can ensure that the database is at-least in second normal form.

Throughout the project software, all communication with the database is done through its JDBC[10] driver, this gives the program database independence i.e. it should work correctly with any database that has a JDBC driver written for it. The algorithm for creating the database generates the SQL statements to perform the creation and executes them through JDBC. When producing the SQL statements an additional complication arises in that we cannot create tables with foreign keys until the tables which the foreign keys reference are first created. The algorithm to create the database must therefore start by creating the table representing the root node, as this will be the only table that does not contain a foreign key (because foreign keys are kept in child node relations and the root is the only node that is not a child node). The primary key field and fields for all leaf and attribute child nodes are added to the created table. Child nodes are found by inspecting the model to find the destination of edges originating at the parent node.

The algorithm then finds all the complex child nodes and for each of these it creates a table representing the relation with a primary key field, a foreign key field referencing the parent's primary key field and fields representing the leaf and attribute child nodes. This process is recursively repeated on all complex child nodes.

We also need some way to represent the HDM constraints in the database. This can be achieved through the use of SQL column types, default values and column constraints:

1. SQL data types are much more general that the XSD base data types, therefore while there is a direct mapping between some XSD types and their equivalent SQL types, for others we have to augment the most similar of the SQL data type with value constraints to get an accurate mapping. The table below shows how the algorithm

[9] Complex nodes are those with child element or attribute nodes.
[10] JDBC – Java Database Connectivity

handles the conversion of XSD types[11], any obscure types (e.g. xsd:NMTOKENS) that are not specifically handled by the program are given a default value of text.

XSD Type	SQL Type	Additional Constraints
string	text	-
integer	int4	-
boolean	boolean	-
float	real	-
double	double	-
short	int2	-
long	int8	-
date	date	-
int	int4	minInclusive = "-1"
positiveInteger	int4	minInclusive = "1"
nonNegativeInteger	int4	minInclusive = "0"
negativeInteger	int4	maxInclusive = "-1"
nonPositiveInteger	int4	maxInclusive = "0"

2. The other constraints can be applied to the database as column constraints, the table below summarises how the different constraints are represented in SQL when applied on a column named test.

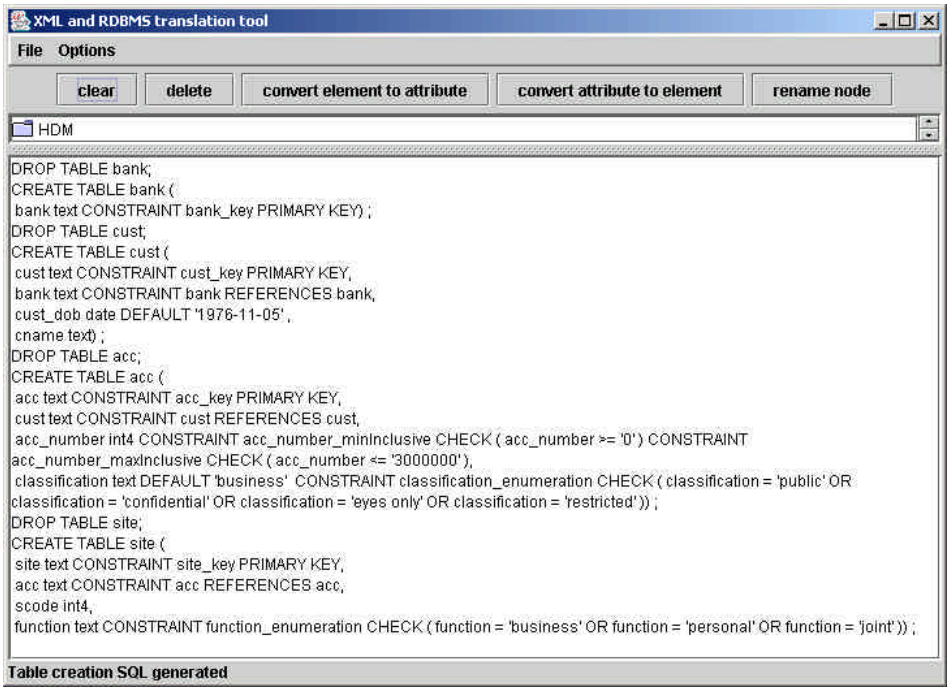
Constraint	Example Extent	SQL Column Constraint
default	{"Miqdad"}	DEFAULT 'Miqdad'
maxExclusive	{"100"}	CONSTRAINT test_maxExclusive CHECK (test < '100')
maxInclusive	{"100"}	CONSTRAINT test_manInclusive CHECK (test <= '100')
minExclusive	{"10"}	CONSTRAINT test_minExclusive CHECK (test > '10')
minInclusive	{"10"}	CONSTRAINT test_minInclusive CHECK (test >= '10')
fixed	{"true"}	CONSTRAINT test_fixed CHECK (test = 'true')
length	{"20"}	CONSTRAINT test_length CHECK (LENGTH(test) = '20')
maxLength	{"20"}	CONSTRAINT test_maxLength CHECK (LENGTH(test) <= '20')
minLength	{"5"}	CONSTRAINT test_minLength CHECK (LENGTH(test) >= '5')
totalDigits	{"4"}	CONSTRAINT test_totalDigits CHECK (LENGTH(test) <= '4')
prohibited	-	CONSTRAINT test_prohibited CHECK (test = null)
enumeration	{"valueA", "valueB", "valueC", "valueD"}	CONSTRAINT test_enumeration CHECK (test = 'valueA' OR test = 'valueB' OR test = 'valueC' OR test = 'valueD')

Many other constraints are captured by the HDM model but have not been dealt with in the conversion to relational databases, for discussion on these omissions please refer to the evaluation section of this report.

[11] I have decided to use the XSD base types as the base types for representing type constraints in the HDM model

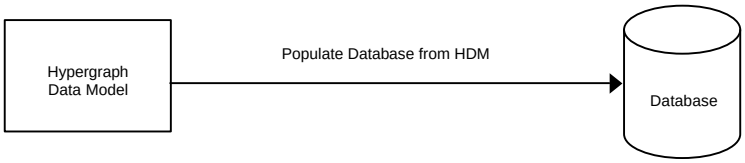
When adding a node as a field[12] in a table the algorithm gets all the constraints associated with the node, these are taken from the node's reference to constraints that have been added over it, for each of these constraints the algorithm adds any SQL equivalent to the column creation SQL statement.

The bank example of Appendix A produces the table creation SQL statements shown in the screenshot below:



[12] I use the words field and column interchangeably throughout this section

3.5.2 Populating a Database from HDM

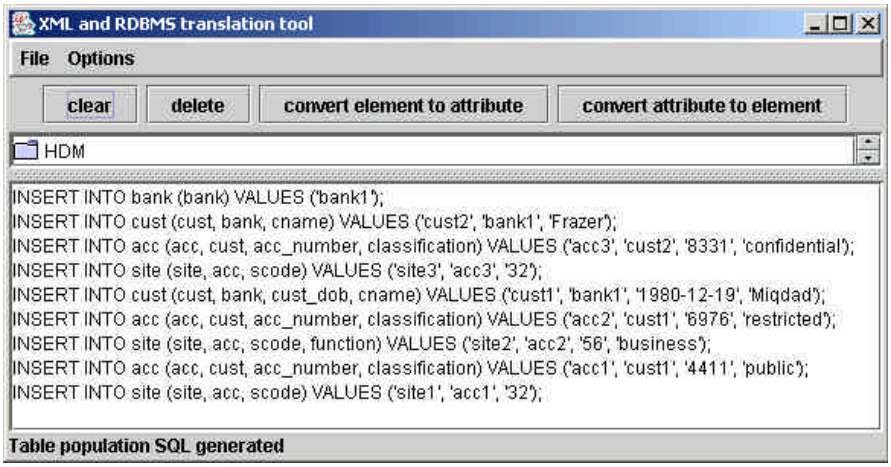


Once we have the basic HDM structure and constraints the data base schema is more or less equivalent to the XML Schema used to create the HDM. We can now populate the database with the data held in the extents of the HDM nodes and edges. To do this we must generate database insert statements to populate the tables that we have created. These SQL statements can be executed through the JDBC database driver to apply them to the database in a database independent manner. Because we are using foreign key constraints the order in which we execute the SQL statements matters. For the same reasons as explained in the table creation section we must start with inserts on the root table.

The algorithm populates the root table primary key value with the extent of the root node. For reasons already discussed this can only ever have one row. The edges from the root node are inspected and any non-null child attribute and leaf node values are added from the extents of the relevant edges.

For each complex child node, primary key values are inserted into the table representing the node from the extent of the node in the HDM. Foreign key values are added with the value of the parent node instance to which they relate. We then look to all child attribute and leaf nodes, we inspect the edge extents in which these nodes are described, if we find the primary key value in the source[13] of the edge in the extent we add the destination value of the edge in the extent to the database as the value of the field representing the node. We repeat this procedure recursively over the complex child nodes.

The bank example of Appendix A produces the table insertion SQL statements shown in the screenshot below:



[13] The source of the edge is the first value in the edge extent, while the destination is the second value.

3.5.3 Creating a HDM from a Database

As well as being able to store XML data in the database we also want to be able to do the reverse transformation, to store data from a database in an XML file. To show how we can do this there is one component of the software that still needs to be described: the conversion from the database to HDM. This conversion can obviously only be performed on databases that are compatible with a possible HDM structure, e.g. there must be some way to link all the tables into a graph structure.

The conversion algorithm first builds the basic HDM structure using the database schema information, and then populates the extents of the HDM using the data in the database. This can be thought of as being analogous to the XML to HDM translation process where first the XML Schema is used to build the basic HDM Structure and then the extents are populated using data in XML instance files.

To build the basic HDM structure we need to be able to access the database schema information. Limited access to this schema information is possible in a database independent way through JDBC. This gives us access to table names, column names and column data types. Using this information we can identify the table that should be represented by the root HDM node as being the table that does not contain any foreign key columns. This table name is added to the model as the root node, the columns of this table are added to the HDM as child element and attribute nodes and edges are added to the model linking the root node to its children. Any tables that contain a column that is a foreign key reference to the root node are added as complex child nodes and the edges between the root node and these complex children[14] are also added to the model. The procedure is then repeated recursively on the complex child nodes. Finally type constraints are added to the HDM to represent the column data types.

Once we have the basic HDM structure we populate the extents of the nodes and edges using the data in the database. To do this we deal with each table separately. For each table we select one row of data at a time, for each column of the selected row we add the value in that column of the selection to the extent of the HDM node representing that column. We then add edge extents by taking the primary key value with each column value in turn, and adding it as an edge extent. This extent is added to the edge linking the node representing the table to the node representing the column. In this way we fully populate the extents of HDM model, completing our HDM representation of the database.

[14] This relationship is represented by the foreign keys in the database.

3.6 Operations on the HDM

Semantically equivalent XML files can be represented in many syntactically different ways, for example there may two files representing exactly the same data, one in an XML file in which all data is held in attribute values while another in which all data is held as the content of leaf elements. Another such semantically insignificant syntactic difference is in the attribute and element names chosen. For example in one file we may have the element cust, while in another file we have an equivalent element customer. There may be some information in one of the XML files that is not important to store and is not present in another otherwise equivalent XML file. To combine the HDM model of these two we need to delete the constructs representing the unimportant information, e.g. one file may contain comments.

Providing some basic operations on the HDM model allows us to combine these syntactically different but semantically equivalent XML files into a single data model. These operations can also be used to convert the HDM model into a form compatible with the database schema when translating from XML to the database, and conversely convert the HDM model into a form compatible with the XML Schema when translating from the database to XML.

3.6.1 Delete

The delete operation can be applied to nodes, edges and constraints. To delete a node we must first delete any edges that the node participates in. Deleting a node or edge automatically deletes any constraints on the node or edge.

3.6.2 Rename

In my implementation edges and constraints are not named, therefore the rename operation applies only to nodes. In HDM two nodes cannot have the same name so the renaming operation is only successful if the proposed new name is not already used as the name of a node in the model. In my implementation I have followed the convention that attribute nodes are named by prefixing the attribute name by the parent element name and separating these two parts of the node name with an underscore. Renaming therefore does not allow the introduction of an underscore into a node name, as this would imply a conversion of an element into an attribute. Also for attribute nodes renaming does not allow the user to modify the part of the name representing the parent element, as this would imply that the attribute has been moved from one element to another.

Renaming a node will update all the name references to the node in the model i.e. all the edges and constraints that it is involved in.

3.6.3 Convert Attribute to Element

Attributes nodes in HDM are equivalent to leaf element nodes. To use this equivalence I have made this operation to convert attributes to leaf elements. The conversion operation must check that the operation is being applied on an attribute node and that the name of the resulting element node is not already in use. If the name of the resulting element node is already in use, the attribute should be renamed first and then converted.

3.6.4 Convert Element to Attribute

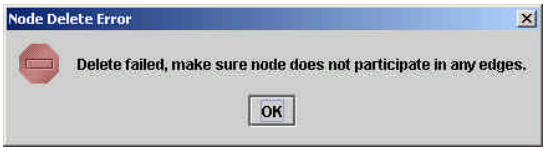
This operation is just the reverse of the attribute to element conversion. The operation must ensure that it is being applied to a leaf element node, as it does not make sense to have nodes with children as attributes. The operation must also ensure that the node is compatible with a maxOccurs constraint of 1, since attributes can only occur once in an element. If successful the operation adds the maxOccurs = "1" constraint to the resulting attribute node.

3.7 Usability Issues

I have added a number of features to my program to aid usability:

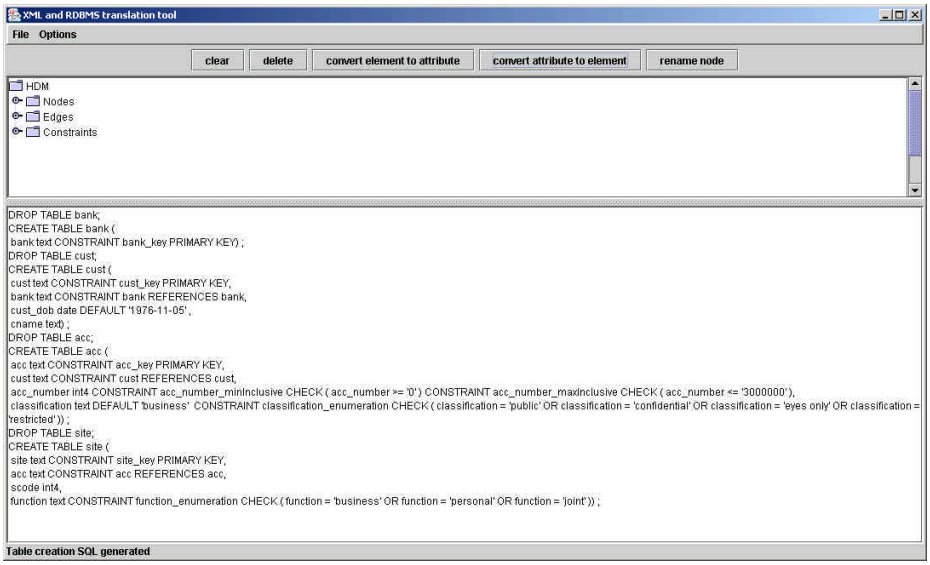
1. The database options are set in an XML configuration file, I have built a graphical interface into the program to view and set these options.

2. Meaningful error and information dialogues are produced for the user when they try and perform illegal operations e.g. illegal renaming, element attribute conversion, loading invalid XML files, SQL errors, using a non-existent database and many more.



3. The program makes extensive use of the status bar, constantly informing the user of what is happening.

4. Any SQL generated by the program is shown in the bottom pane of the GUI to keep the user informed of what the program is doing. This is particularly useful when using the program's operations to make a model conform to a database schema.



5. In addition to these features that make the software usable for the end user I have also tried to make the source code as organised and usable for anyone wishing to study it or develop the project further. All coding is done in Java and is generously commented. Commenting conforms to the javadoc conventions and so the standard Java documentation tool can be employed to produce documentation in a form familiar to all Java programmers. Standard software engineering design patterns are employed in the code wherever it is appropriate to do so. The software has been split into highly cohesive and loosely coupled modules, allowing the seamless replacement / addition of modules. Execution, compilation and documentation scripts have been implemented[15] to automate these potentially tedious functions.

3.8 Testing

As the application was developed test harnesses were used to continuously perform white box and black box testing. At the completion of each module extensive use case testing was carried out to test the module and then to test the whole software after the module was integrated into it.

[15] The source code and documentation are available from the project website: http://www.miqdad.webstar.co.uk/individual_project_2002/

4. Evaluation

4.1 Strengths and Weaknesses

The objective of the project was to produce a framework for inter-model transformations and to build in support for XML and relational databases into this framework. For evaluation purposes I have broken the project down into the nine different components that one would expect the project to consist of, and will discuss what has been achieved and left out for each of these components in the following subsections.

4.1.1 Implementation of the HDM Framework

The implementation of the core framework stayed very close to the theory [16] and was largely successful. The weakness in this part of the project lies in the graphical representation of the HDM. The tree structured employed is very basic and functional, and for users unfamiliar with HDM or graph terminology is not very intuitive. I had the option of focussing much more effort on developing an intuitive graph structure, but after discussion with my supervisor we came to the conclusion that it would be more productive to focus my limited time and effort on the theoretical progress of the project rather than attend to cosmetic details. This prioritisation of effort is where the distinction between a commercial implementation and a theoretical prototype comes in.

4.1.2 Translation from XML Schema to HDM

XML Schema is a vast language with a large range of features that can be combined in a number of different ways to very powerful effect. In my project implementation I first prioritised the different features and combinations in terms of those commonly used and generally useful. These I ensured were handled by my project correctly. With these in place my project will work with the vast number of schemas that will ever be written. I then went on to implement some of the more complicated and less often used features.

One major weakness with this schema conversion algorithm is that it is unable to handle mixed content models. Mixed content models are models of XML files that are allowed to contain child elements and textual data. Further more the textual data can be interspersed in between the child elements in every possible way. An example of a mixed content XML file is a XHTML file. In such files we can have elements such as paragraphs that contain mixed content, e.g. text may be interspersed with bold, italic, font and other child element tags. It is very difficult to pre-determine where and how the textual data will appear. One very inefficient solution that I explored was to treat textual data as special child elements and assume that there was textual data in every possible place that it could occur. This however proved very inefficient and did not work for instances where the occurrences of fellow child elements was not fixed.

My project does not fully cover the XML Schema specification, therefore when encountering schemas that it cannot fully process I had two options: either abort the parsing of the schema and tell the user the schema is incompatible with the program, or to ignore the un-handled construct during the parse and do the best possible job of understanding the rest of the schema

reporting to the user that the schema contains some features that the program does not handle so results may not be as expected. I decided to go for the second of these options, as the program could still be useful even without fully understanding the schema file.

4.1.3 Translation from HDM to XML Schema

This component is the only expected feature that is completely omitted from the project. Two reasons for the omission are: firstly it was felt that this component was not really an essential part of the conversion process. If the HDM data structure is formed from an XML Schema then all data in the model must conform to the constraints in the schema as embodied by the model. Therefore there is no need to reproduce the schema to check the validity of the data as the data must conform to the schema to be in the model. The second reason for the omission is that there are many different ways to represent the same set of constraints in an XML Schema, therefore there is no unique inverse transform from the HDM to an XML Schema. We would be able to reproduce a semantically equivalent schema but it would not necessarily be the same original schema. It can be argued that we only need to be aiming for this semantic equivalence. While this may be sufficient for human users, perhaps it would not be enough for external programs using this application to understand two schemas are the same.

In retrospect I think this functionality is more important than I had assumed when doing the implementation. When considering this choice as shown in the reasoning above, it was incorrectly assumed the conversion process would always begin with XML. When looking at the software more closely we notice that it is just as correct to start the process with a database. In such a case we will be able to create XML output, but without an XML Schema will not be able to define to the user what it is that constitutes valid XML input into the model.

4.1.4 Conversion from XML to HDM

This conversion works perfectly for valid XML instances of the XML Schema associated with the HDM. However the program has a weakness here in that it does not check the validity of the instance file against the schema. This can result in data that is incompatible with the HDM constraints being inserted into the model, leaving the HDM in an inconsistent state. One way to add the required validation functionality is to employ a third party validation tool. This tool can be invoked transparently by the program to perform validation. One such tool suitable for the job could be the IBM Schema Quality Checker API. This was not implemented in the project as I ran out of time to learn and incorporate this software.

4.1.5 Conversion from HDM to XML

This was also implemented successfully, but may also have benefited from the validation against a schema functionality described above.

4.1.6 Conversion from HDM to Database Schema

The program successfully converts the HDM structure into a relational database schema that accurately represents all the nodes, edges and most of the constraints present in the model. Some constraints however do not have a valid equivalent in a relational database, for example the minOccurs constraint cannot be represented when its value is more that zero as this constraint would be violated by an empty database table and so would never allow the table to

be created. Other constraints cannot be implemented in a database independent way, for example the pattern constraint employs regular expressions which are represented differently in different databases.

The program uses the information in the model to decompose the database to second normal form. A weakness is that it does not allow the user to provide additional input to enable it to decompose the database to higher normal forms.

4.1.7 Conversion from Database Schema to HDM

All the appropriate nodes and edges can be recreated in the HDM from a database. Constraints however are more problematic. The problem relies of the restrictions imposed by relying on JDBC for all database access. The only constraints accessible through JDBC are type constraints and therefore these are the only constraints the program can create in the HDM translation.

A possible solution to this problem is to have a separate table to hold a string representation of the constraints, used only for conversion purposes. After discussion with my supervisor it was decided that this was more of a hack around the problem, a path that may be pursued in a commercial implementation but not appropriate as a meaningful extension to the conversion process theory.

A more elegant solution could be to extent the JDBC driver to give more complete access to database metadata, but this was thought to be beyond the scope of the project.

4.1.8 Conversion from HDM to Database

This functionality enables the transfer of data in the extents of the HDM nodes and edges to be stored in database table. This functionality seems to work faultlessly.

4.1.9 Conversion from Database to HDM

This functionality enables the transfer of data from database tables to the extents of the HDM nodes and edges. This functionality seems to work faultlessly.

4.1.10 Miscellaneous Issues

Validity checking by the program is very strict, if an XML file contains an error in it the whole file is rejected by the program. It can be argued that when using very large XML files, this may not be desirable behaviour, the file may contain a small error at the end and so abort the conversion wasting all the parsing effort. More appropriate behaviour in such circumstances would be to read all the correct data into the model and report any errors to the user so they can deal with the errors. This being more of a research project, showing that the implementation of a theoretical model can recognise errors is what is important. In a commercial implementation the extra error handling functionality would obviously be desirable but this does not necessary add to the fulfilment of this prototype project's objectives. Additionally it is unlikely that this prototype will ever be used with very large XML files. Other issues concerning very large files include optimising memory usage by using

SAX to access the XML rather than JDOM, and optimising the algorithms for speed, perhaps at the expense of an elegant coding style.

Another issue that came up in the project was the trade-off involved in using JDBC for database access, on the positive side it allows the program to be database independent and work with any JDBC compatible database. But on the negative side it places restrictions on the amount of database metadata that the program can access and denies the program the opportunity to use database specific functionality that may result in more efficient and elegant code. In a commercial implementation I would expect that the program would be optimised with database specific extensions for the commonly used databases, leaving JDBC to handle the other databases.

On the whole many of the limitations of the program highlight the difference between what is expected form a theoretically based project and a commercial application. I have designed my software in a modular way so that extensions can be easily added to certain aspects of the project without requiring changes to files dealing with unrelated functionality. The project is split into modules called: Database, XML, HDM, Config and GUI. So if we want to modify / optimise database access we need only apply the necessary modifications to the Database module in which all database access functionality is implemented.

4.2 Comparison with other Solutions

Most existing solutions to the conversion problem are provided by database vendors as part of their database product. These are obviously database specific solutions and a different level and style of XML support is provided in each implementation. Most solutions are limited to converting from the database to XML and are not able to perform the reverse transformation. These solutions give very little control over the structure and naming in the XML output.

In my research I have come across one other implementation of the HDM approach to translating between XML and databases [26]. This implementation does not deal with HDM constraints and their corresponding XML Schema / DTD and database constraints. Also the approach only handles the transformation in one direction: from XML to databases.

In contrast my project is database independent to the extent that it allows the user to simultaneously combine data from multiple different databases into the same model. The user has a significant amount of control over the structure of the XML file produced, being able to convert between elements and attributes, and able to delete unwanted information form the intermediate model before conversion. It also allows the user to rename elements and attributes. The conversion process works in both directions and when transferring from XML to the database, XML Schema constraints are extensively utilised. It also has the advantage of not being limited to translate between just XML and databases, but can be logically extended to integrate data from multiple data sources and translate between one to one, one to many, many to one, and many to many, different data models.

5. Conclusions and Future Work

5.1 Contribution of the Work

This work implemented a conversion methodology using the Hypergraph Data Model. Much has been written about how such a conversion should take place. The contribution of my work to the theory around HDM is in extending the research on how different data models can be represented in the HDM by proposing a method for the conversion of XML Schema into the model. This also results in providing an improved theory of how XML should be converted to and from the HDM by utilising the schema information.

On the practical side this project provides the only fully functional implementation of the core HDM framework that I am aware of. It adds modules to this framework to handle the conversion between XML and HDM, and to handle the conversion between relational databases and HDM. It also provides the functionality for the user to alter the structure of the data while in the intermediate HDM state.

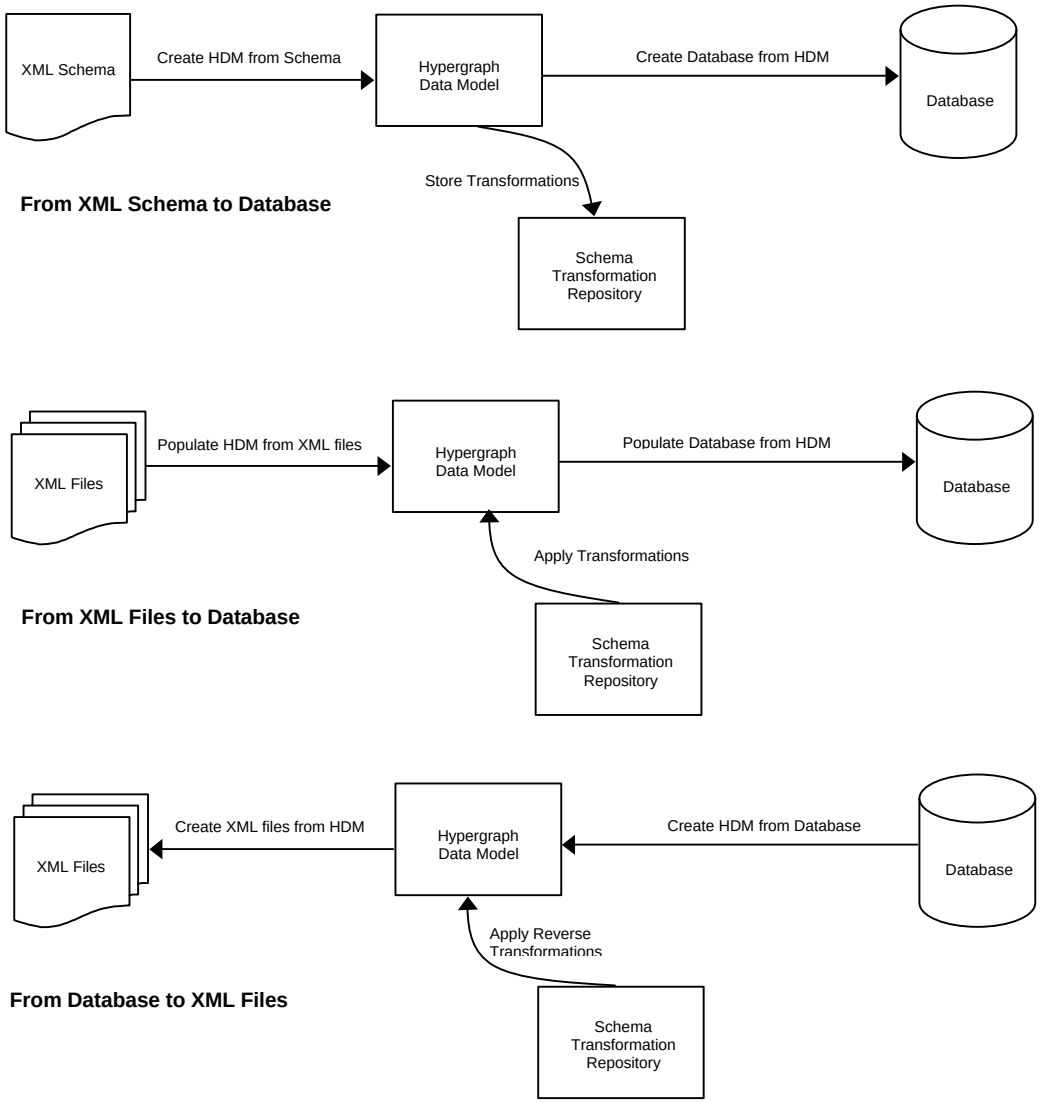
5.2 Lessons Learnt

The main thing I would do differently if re-implementing the project would be to start by writing a simple Java API for XML Schema parsing, based of the existing XML API's. The XML Schema has many components that are continually accessed in the code and such an API would simplify the access to these commonly required components.

5.3 Future Work

The software currently just translates data from one form to another, this could be made much more useful if some sort of query functionality was added. The logical way to achieve this would be to define a query language over the HDM. Queries could either be performed on the data once in HDM form, or for more efficient performance when large amounts of data are involved we could translate queries [18,19,20] over HDM to queries over the data models that are linked through it e.g. SQL queries over relational databases.

Another interesting feature to add to the software is a schema transformation repository. When we bring an XML Schema into the HDM we may want to perform certain transformation operations on it to make it conform to our database. These transformations can be stored in some transformations repository. When we bring an XML instance file into the model these transformations can be retrieved from the repository and automatically applied to make the instance file also structurally conform to the database. HDM has a very useful property that all transformations composed of basic operations are fully and uniquely reversible e.g. the reverse of an addNode transformation is a delNode transformation with the same parameters. Therefore when we take data out of the database we can automatically apply the reverse transformations from the schema repository to convert the data back to the original XML format. This can be taken further and used to convert between many different but equivalent XML formats and the database, or even just between different XML formats.



The above diagram illustrates how the schema transformation repository would fit into the project. A repository that fulfils this functionality and is specifically designed to work with HDM is currently being developed by a group at Imperial College under the project name Automed see [22,34]

Other useful extensions would be to add support for other data modelling languages [16], to improve the graphical user interface, to implement the remaining XML Schema features that have not yet been implemented, to find a way to extract constraint metadata from the database, to extend the project to deal with Object Oriented and Native XML databases, and to add an XSL module to enable the program to use appropriate style sheets to convert XML output to more specialised formats e.g. to pdf, postcript, WAP, Microsoft Word.

6. Bibliography

[1] Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, available at http://www.w3.org/TR/REC-xml, October 2000.

[2] XML Schema part 0: Primer W3C Proposed Recommendation, available at http://www.w3.org/TR/xmlschema-0/, March 2001.

[3] XML Schema part 1: Structures W3C Proposed Recommendation, available at http://www.w3.org/TR/xmlschema-1/, March 2001.

[4] XML Schema part 2: Datatypes W3C Proposed Recommendation, available at http://www.w3.org/TR/xmlschema-2/, March 2001.

[5] Eric T. Ray, Learning XML, O'Reilly, January 2001.

[6] Brett McLaughlin, Java & XML (Second Edition), O'Reilly, August 2001.

[7] Patrick Niemeyer and Jonathan Knudsen, Learning Java, O'Reilly, May 2000.

[8] Kevin Williams et al, Professional XML & Databases, Wrox, 2000.

[10] J. Worsely and J. Drake, Practical PostgreSQL, O'Reilly, January 2002.

[11] P.J. McBrien and A. Poulovassilis, A Semantic Approach to Integrating XML and Structured Data Sources – Technical Report 30/11/2000, Birkbeck College and Imperial College, London, November 2001.

[12] P.J. McBrien and A. Poulovassilis, A Formal Framework for ER Schema Transformations, Kings College London, In proceedings of ER'97, Volume 1331 of LNCS, pages 408-421, 1997.

[13] P.J. McBrien and A. Poulovassilis, Automatic Migration and Wrapping of Database Applications – A Schema Transformation Approach, Kings College London, In proceedings of ER'99, Volume 1728 of LNCS, pages 96-113, Springer-Verlag, 1999.

[14] P.J. McBrien and A. Poulovassilis, A Formalisation of Semantic Schema Integration, Information Systems, 23(5):307-344, 1998.

[15] P.J. McBrien and A. Poulovassilis, A Uniform Approach to Inter-Model Transformations, In Advanced Information Systems Engineering, 11th International Conference CaiSE'99, Volume 1624 of LNCS, pages 333-348, Springer-Verlag, 1999.

[16] A. Poulovassilis and P.J. McBrien, A General Formal Framework for Schema Transformations, Kings College London, May 1998.

[17] R. Goldman et al, From Semi-Structured Data to XML: Migrating the Lore Data Model and Query Language, In proceedings of WebDB, 1999.

[18] D. Florescu and D. Kossmann, Storing and Quering XML Data Using an RDBMS, Bulletin of the Technical Committee on Data Engineering, 22(3):27-34, September 1999.

[19] S. Abiteboul et al, The Lorel Query Language for Semi-Structured Data, Journal on Digital Libraries, 1(1), 1997.

[20] J. Shanmugasundaram et al, Relational Databases for Querying XML Documents: Limitations and Objectives, In proceedings of the 25th VLDB Conference, pages 302-314, 1999.

[21] Brett McLaughlin, Validation with Java and XML Schema, available at http://www.javaworld.com/javaworld/jw-10-2000/jw-1013-validation2_p.html, October 2000.

[22] M. Boyd and N. Tong, Automed – The Automed Repositories and API, Imperial College London, August 2001.

[23] Nikolaos Rizopoulos, Database Schema Integration Tool, MSc Thesis – Imperial College, September 2001.

[24] Nikolaos Giannadakis, Transforming XML Schemas into Relational Models, MSc Thesis – Imperial College, September 2001.

[25] Manargias Dimitris, Storage and Retrieval of XML Data, MSc Thesis – Imperial College, September 2001.

[26] Syafeeq Alias, XML to relational database converter, BEng Project – Imperial College, June 2001.

[27] Niamul Choudhury, Relational database to XML converter, BEng Project – Imperial College, June 2001.

[28] XML at Sun, available at http://wwws.sun.com/software/xml/.

[29] Apache XML Project, available at http://xml.apache.org.

[30] O'Reilly XML.COM, available at http://www.xml.com.

[31] Brett Spell, Enhancing Database Code with Metadata, available at http://www.devx.com/upload/free/features/javapro/1999/06jun99/bs0699/bs0699.asp.

[32] The World Wide Web Consortium, available at http://www.w3.org.

[33] JDOM, available at http://www.jdom.org.

[34] Automatic Generation of Mediator Tools for Heterogeneous Database Integration (AutoMed), available at http://www.doc.ic.ac.uk/automed

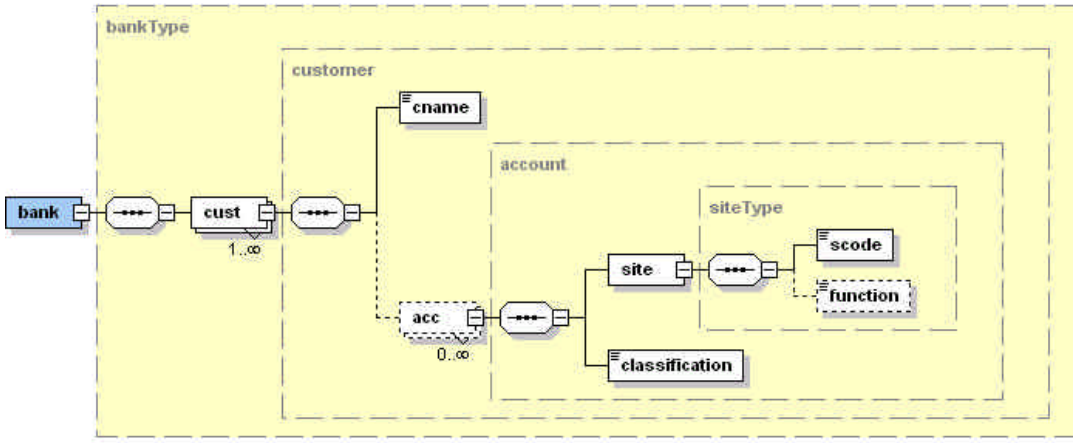
7. Appendices

7.1 Appendix A – Bank Example

bank schema – bank.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
    <xsd:annotation>
        <xsd:documentation>
            Project XML Schema File for bank accounts
         </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="bank" type="bankType"/>
    <xsd:complexType name="bankType">
        <xsd:sequence>
            <xsd:element name="cust" type="customer" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="customer">
        <xsd:sequence>
            <xsd:element name="cname" type="xsd:string"/>
            <xsd:element name="acc" type="account" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="dob" type="xsd:date" use="optional" default="1976-11-05"/>
    </xsd:complexType>
    <xsd:complexType name="account">
        <xsd:sequence>
            <xsd:element name="site">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="scode" type="xsd:integer"/>
                        <xsd:element ref="function" minOccurs="0"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="classification" default="business">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="public"/>
                        <xsd:enumeration value="confidential"/>
                        <xsd:enumeration value="eyes only"/>
                        <xsd:enumeration value="restricted"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="number" use="required">
            <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                    <xsd:minInclusive value="10"/>
                    <xsd:maxInclusive value="3000000"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:attribute>
    </xsd:complexType>
    <xsd:element name="function">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="business"/>
            <xsd:enumeration value="personal"/>
            <xsd:enumeration value="joint"/>
        </xsd:restriction>
    </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

graphical representation of bank schema – bank.xsd

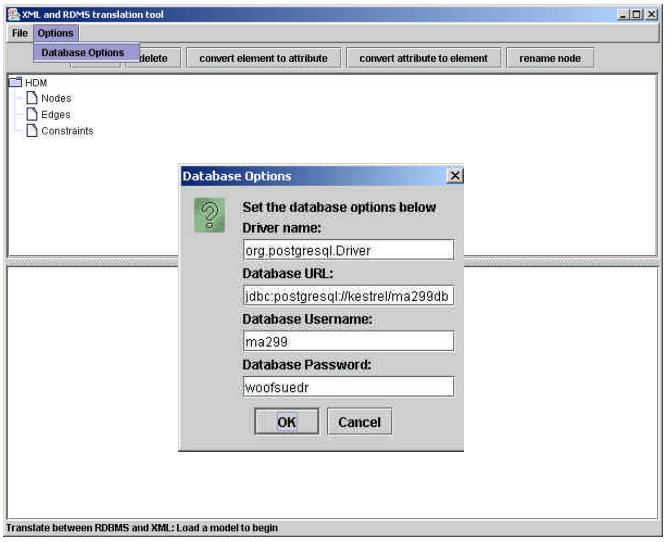


bank instance file – bank.xml

```
<?xml version="1.0"?>
<bank xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="bank.xsd">
    <cust dob="1980-12-19">
        <cname>Miqdad</cname>
        <acc number="4411">
            <site>
                <scode>32</scode>
            </site>
            <classification>public</classification>
        </acc>
        <acc number="6976">
            <site>
                <scode>56</scode>
                <function>business</function>
            </site>
            <classification>restricted</classification>
        </acc>
    </cust>
    <cust>
        <cname>Frazer</cname>
        <acc number="8331">
            <site>
                <scode>32</scode>
            </site>
            <classification>confidential</classification>
        </acc>
    </cust>
</bank>
```

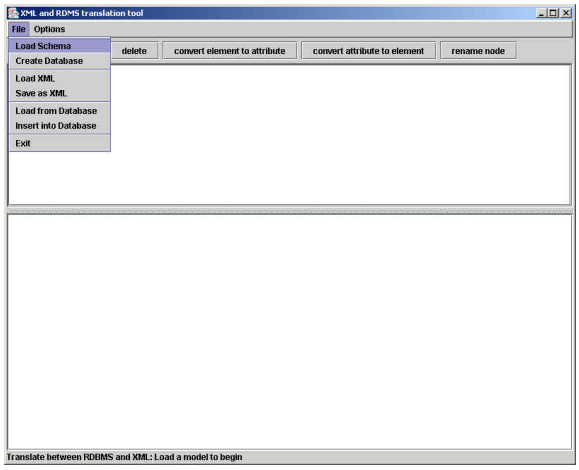
7.2 Appendix B - User Guide

To begin using the program you must first set your database options. Select database options form the options menu to get to the dialogue box shown below.



The dialogue box picture shows typical options for use of the program with postgreSQL in the Imperial College – Department of Computing Laboratories.

After having set the database options we must load an XML Schema by selecting the load schema option form the file menu.

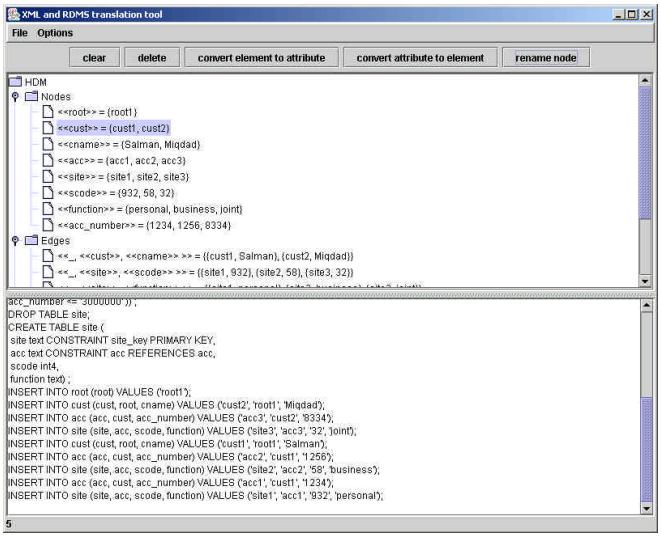


This will result in the standard file selection dialogue box appearing, with a selection mask to show .xsd files. (.xsd is the standard extension for xml schema files)

Once we have successfully loaded the schema file we then load an instance of the schema. An example schema file bank.xsd and an example instance file bank.xml can be found in the project examples directory.

Once we have loaded these files successfully we can create and populate the database by selecting the create database and insert into database options from the file menu.

At this stage the program will show the HDM representation of the data in the top pane and the SQL statements that were executed in the bottom pane. The output you should see when using the bank example files is shown in the screen shot below.



The model can be modified by using the buttons on the toolbar. Below we can see an example of the dialog box produced when we try to rename the cust node.

The clear button on the toolbar is used to clear the HDM model from memory. The model can now be reloaded the database by choosing the load from database option in the file menu.

The model can be saved as XML by choosing the save as XML option from the file menu. The standard file save dialog appears with which you specify where to save the XML file.

The data has now travelled through the complete transformations pathway. Now that you are familiar with the basics feel free to experiment further.