Imperial College of Science, Technology and Medicine
(University of London)
Department of Computing

Translating between XML and Relational Databases using XML Schema and Automed

Andrew Charles Smith
acs203

Submitted in partial fulfillment of the requirements for the MSc Degree in Advanced Computing of the University of London and for the Diploma of
Imperial College of Science, Technology and Medicine.

September 2004

Acknowledgements

I would like to thank my supervisor, Peter McBrien, for his support during this project and his help with the Automed API. I would also like to thank Lucas Zamboulis for his help with the XML queries and Nicolas Debarnot for his help with LaTeX.

Contents

List of Figures

List of Tables

Abstract

XML and relational databases are two of the most important mechanisms for storing and transferring data. A reliable and flexible way of moving data between them is very desirable goal. The way data is stored in each method is very different which makes the translation process difficult. To try and abstract some of the differences away a low-level common data model can be used. To successfully move data from one model to the other a way of describing the schema is needed. Until recently there was no widely accepted way of doing this for XML. Recently, however, XML Schema has taken on this role. This project takes XML conforming to XML Schema definitions and transforms in into relational databases via the low-level modeling language HDM. In the other direction a relational database is transformed into an XML Schema document and an XML instance document containing the data from the database. The transformations are done within the Automed framework providing a sound theoretical basis for the work. A visual tool that represents the XML Schema in a tree structure and allows some manipulation of the schema is also described.

1 Introduction

Most business data is stored and maintained in relational DBMSs and looks likely to remain that way for the foreseeable future. These systems provide efficient and reliable access to data for users within an organisation. However, the recent huge growth of eBusiness and the need to transfer data to and from customers and other organisations over the Internet has meant some data needs to escape from this restrictive model. XML [19] has emerged as the dominant standard for representing and exchanging this data, so mechanisms for moving data stored in RDBMSs to XML and visa versa has become a very important area of study.

Approaches to extracting data from one modeling language and representing it in another can be broadly split in two. The first chooses one of the languages as the common data model and transforms the data from the other in it. This suffers from the complication that the two languages rarely have a simple correspondence between their modeling constructs. The second uses a common, lower-level language to represents the semantics of both modeling languages. Transformations within the lower-level language allow a mapping to be created. This has the advantage that specific complexities in both the initial modeling languages are abstracted away. A further advantage is that once a given modeling language has been specified in the low-level language it can be converted into any other language that has also been specified in the low-level language. The Hypergraph data model (HDM) [28] is one such language and will be described in Section 5.

HDM has the further advantage of being graphical in nature. Humans are far better at interpreting graphical data than they are at interpreting textual data. Using a graphical representation of the two different models makes it much easier to see the similarities and differences between them. A number of graphical representations for both XML and relational data exist, Section 2.4 describes some of them. They suffer from the disadvantage that their representations are intended only for the one model. HDM allows data from many different models to be represented using the same simple constructs. In Section 6.1 it is shown how data from XML Schema and the relational model can be represented by the same HDM graphs. This ability to represent data from different models with the same graphical structures makes it much easier to see the similarities and differences between schemas. Having data from a number of different models represented with the same HDM structures also makes the process of translation between any of those models easier.

The success of any tool that maps between XML and RDBMSs, be it directly or via a lower-level language, is heavily dependent on the quality of the schema defining the structure of the data to be transferred. An ambiguous or incomplete schema can lead to an imprecise mapping or loss of data. Mechanisms for inferring relational schemas from non-relational ones are well documented but methods for inferring an XML schema are less well understood.

A number of tools already exist that can move data between XML and RDBMSs [21, 14, 8, 22], (Section 3.4 will look at some of these in more detail), but many suffer from the drawback that the XML schema must be specified in advance by a human expert. For a complex data set this can

```
<xsd:complexType name="shoes">
  <xsd:sequence>
    <xsd:element name="price" type="xsd:integer" />
    <xsd:element name="colour" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="shoeid" type="xsd:integer" />
</xsd:complexType>

<xsd:element name="shoe" type="shoes">
  </xsd:key name="unique_shoeid">
    <xsd:selector xpath="./" />
    <xsd:field xpath="@shoeid" />
  </xsd:key>
</xsd:element>
```

Figure 1: A simple XML Schema fragment

be a lengthy and difficult task. A tool that automatically infers a precise XML schema directly from the relational schema is obviously very desirable. This field is still quite new and there are only a few tools that do this [5, 3].

Choosing a format to describe the XML schema in has, until recently, been difficult. DTDs have historically been used. They, however, have a number of disadvantages. The specification of a DTD is written in non-XML syntax and offers only limited data typing. XML Schema [20] has emerged recently as the *de facto* standard for defining XML schemas. It is a powerful and comprehensive standard that supports rich built-in types and allows the creation of further complex types based on the built-in ones as well as many other useful features.

This project will present a method for moving data from an XML format into an RDBMS and visa versa based on XML Schema using HDM as an intermediary.

Figure 1 shows an XML Schema fragment containing a complex type an element of that type and an identity constraint. Figure 3 shows a low-level graphical model of the complex type. As can be seen this is very general but maintains the essential aspects of the schema, i.e. a parent node linked to a number of child nodes representing the complex type with its sub-elements.

Figure 2 shows the relational table that could result if the complex type were transformed into the relational model. Each complex type generates a relation, the elements and attributes within that complex type become attributes of the relation and the key identity constraint becomes a relational primary key. The HDM representing this relation is exactly the same as the one representing the complex type. In this case the parent node represents the relation and the child nodes are the table's attributes.

From this simple example we can see the power of representing both models in a general graphical format. Firstly it is very easy to see the relationships between the various elements of each structure and secondly the general nature of the HDM allows us to represent both schemas with the same graph highlighting the similarity of the schemas something that not be immediately obvious from comparing the XML Schema file to the SQL table definition unless one knew both definition languages.

The rest of this report expands on this idea showing how much more complex XML Schemas and the XML instance documents they constrain can be translated into relational databases and conversely how relational databases can be represented as XML Schemas and XML instance documents, using the HDM as an intermediary.

The rest of this report is structured as follows: Section 2 provides an example of why it might be necessary to move data between XML and an RDBMS. It goes on to describe both models briefly along with methods for querying data stored in each model. Ways of representing the schema of each model are given and a description of the aspects of XML Schema most relevant to

```
CREATE TABLE shoes
(shoeid INTEGER NOT NULL,
 price INTEGER NOT NULL,
 colour VARCHAR(20) NOT NULL,
 CONSTRAINT shoes_pk PRIMARY KEY (shoeid),
);
```

Figure 2: SQL definition of relational table

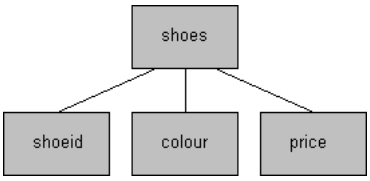

Figure 3: HDM representation of simple XML Schema fragment and relational table

data storage is given.

Section 3 discusses some of the difficulties in translating data from one model to the other and highlights the main points that need to be addressed for a translation to be successful. It goes on to describes a number of approaches that have already been taken to solving this problem.

Section 4 presents my method of solving the main problems associated with the translation process. It introduces the idea of using a low-level abstract data model to help with the process.

Section 5 describes one such model, the HDM, that has been used with some success to represent various high level languages. The HDM representations of XML and the relational model that have already been defined in the literature are presented. A new extension to the relational model to allow it to represent type information is given.

Section 6 presents an HDM representation of XML Schema. This is a new model and represents one of the main contributions of this report. The most significant XML Schema constructs are described in terms of the HDM along with some examples of how these can be related to the previously described relational model. All the primitive transformations such as adding and deleting constructs to and from the new model are described. An example of a composite transformation is also given.

Section 7 describes formally how the new XML Schema constructs can be transformed into constructs in the relational HDM model. This is the second major contribution of this work. An example of a transformation is given.

Section 8 provides a number of case studies describing how some of the more complicated XML Schema constructs can be transformed into relational constructs.

Section 9 describes the implementation of a visual tool that allows some manipulation of the XML Schema and some transformations to be done on it.

Section 10 offers some conclusions and suggestions for further work.

2 XML and RDBMS

XML, the *Extensible Markup Language*, has become a widely accepted standard for data exchange over the Internet and Relational Database Management Systems (RDBMS) are where most of the data in the world is kept at present. Being able to exchange data between the two is an important objective. This section will describe aspects of these two data models with the help of an example scenario in which data exchange needs to take place.

2.1 Motivating Example

A shoe company sells shoes to many different shops. They store the shoe maker, price etc. in their RDBMS. The seller may want to send this information to the shops electronically in a format that will be useful to them, i.e. not on paper or in a flat file. The seller decides to send the information in XML. The shops get the data. It is of interest to them, and they decide to store it in their RDBMSs. These will almost certainly differ from shop to shop. Ideally the seller and all the shops would agree on a common format, i.e. a specific XML schema, that the XML data would adhere to, however, this may not always be possible. A more general approach would allow the seller the generate an XML schema document describing the data he has sent to the shops based on his relational schema. The shops would then use this schema to transform the XML into a format they could easily store in their RDBMSs.
In summary:

- The seller will create an XML schema of the subset of the relational schema that the shops need.
- They will query the RDBMS and generate the result in XML conforming to the schema generated in step one and send the XML data and the schema to the shops.
- The shops will transform the XML schema and data into a form that matches their relational schema.
- They will put the XML data into their relational tables using the transformed schema.

The following sections will describe the technologies both parties are using and give examples of the documents and schemas that may be created.

2.2 RDBMS

The vast majority of data stored in the world today is stored in RDBMSs as is the case for the shoe seller and the shoe shops in our example. RDBMSs are popular because they provide very efficient and robust implementations of a well understood and researched model. The relational model will be dealt with very briefly here as the concepts are generally very well understood. The interested reader can look at [6, 7] for more information.

A relational model is characterised by the following features [7, 6]:

- It has simple structures. Relations are expressed using two-dimensional tables who elements are data items, independent of the physical data representation.
- The model provides a solid foundation for data consistency through normalization and integrity rules.
- Candidate keys provide a way of uniquely identifying tuples in a relation. These can be made up of one or more columns with non-null values. One of these candidate keys can be chosen by the database designer to be a *primary* key.
- *Foreign* keys allow relations to be joined together. A foreign key is a set of one or more columns in any relation which may hold the value(s) found in the primary key column(s) of the relation we wish to join to.

- The model allows the set oriented manipulation of relations, which has led to the development of powerful nonprocedural languages like relational algebra (set theory) and relational calculus (logic). This set oriented approach means there is no concept of order in the relational model.

A relational database is described by a schema. The schema can be described in terms of the commands used to create them, usually in SQL Data Definition Language (DDL). Figure 4 shows four of the tables that the shoe seller may use to store their shoe data and the SQL DDL commands used to create them.

2.3 XML

XML [19] falls into the mark-up language family. Markup languages use sets of tags that have meaning to a parser of that language to define document elements. HTML is an example of a mark-up language. Unlike HTML, which has a strict set of allowable tags and rules about how they can be used. These rules define the HTML *grammar*. XML places no restriction on the tag set and imposes only very limited rules on how they may be used. Anyone can come up with a set of tags and define how they are to be used. This new set of XML tags and the accompanying grammar define a new language. In this way XML is a meta-language, a language that can be used to define other languages. It is also completely extensible, a given set of tags can always have new tags added, hence the name. The only restriction on an XML document is that it should be *well-formed* i.e. each opening tag must have a closing tag and no tags should be nested out of order. For example the following is not well-formed as the tags are not nested in order:

```
<shoe>
  <make>
  </shoe>
</make>
```

Figure 5 is an example of a simple well-formed XML document. The first two lines are *processing instructions* to help a parser deal with what follows. The line that starts with `<!--` is a comment. The rest of the file describes the two shoes from the relational tables in Figure 4. The order that the elements are defined in is significant unlike in the relational model. An XML document can also include white space which in some cases is significant.

2.3.1 Constraining XML

The great flexibility of XML leads to some problems. Two documents containing the same data may be created with different XML tags or with the same tags but in a different order. This will obviously cause problems when the document comes to be processed. What is needed is some way of constraining the XML that firstly tells documents authors how they should lay their XML out and secondly checks after the documents has been created that no mistakes have been made. A document that describes the set of constraints on an XML document is called a schema and an XML document that conforms to these constraints is called *valid*. Until recently Document Type Definitions (DTDs) were the preferred method for specifying a set of constraints for a document. Figure 6 shows a DTD that could be used to constrain the XML in Figure 5.

2.3.2 XML Schema

Recently, DTDs have begun to fall out of favour and as such they will not be discussed in detail. They suffer from a number of drawbacks. Most crucially they do not conform to XML syntax themselves, as can be seen from the example, and have very limited support for data types. The lack of support for data types is becoming particularly significant as XML is used more to store data. To overcome this the W3C has come up with XML Schema[1] [20] as a new way of specifying

[1]Note: We differentiate between an XML schema and the XML Schema standard defined by the W3C by capitalising the S

```
+--------+--------+--------+-------+
| shoeid | colour | makeid | price |
+--------+--------+--------+-------+
|      1 | red    |      6 |    50 |
|      2 | blue   |      4 |   100 |
+--------+--------+--------+-------+
CREATE TABLE shoes
(shoeid INTEGER NOT NULL,
 price INTEGER NOT NULL,
 colour VARCHAR(20) NOT NULL,
 makeid INTEGER NOT NULL,
 CONSTRAINT shoes_pk PRIMARY KEY (shoeid),
 CONSTRAINT shoes_make_fk FOREIGN KEY (makeid) REFERENCES makes
);
+--------+----------+
| makeid | makename |
+--------+----------+
|      6 | adidas   |
|      4 | nike     |
+--------+----------+
CREATE TABLE makes
(makeid INTEGER NOT NULL,
 makename VARCHAR(20) NOT NULL,
 CONSTRAINT makes_pk PRIMARY KEY (makeid)
);
+-----------+-------------+
| countryid | countryname |
+-----------+-------------+
|         1 | china       |
|         2 | usa         |
|         3 | germany     |
+-----------+-------------+
CREATE TABLE countries
(countryid INTEGER NOT NULL,
 countryname VARCHAR(20) NOT NULL,
 CONSTRAINT countries_pk PRIMARY KEY (countryid)
);
+--------+-----------+
| shoeid | countryid |
+--------+-----------+
|      1 |         1 |
|      2 |         1 |
|      2 |         2 |
+--------+-----------+
CREATE TABLE shoe_country
(shoeid INTEGER NOT NULL,
 countryid INTEGER NOT NULL
);
```

Figure 4: Tables and SQL DDL statements for the shoes relational database

```
<?xml version="1.0" encoding="UTF-8"?>

<shoes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="file:shoes.xsd">
<!-- XML representation of two shoes -->
    <shoe shoeid="1">
      <make makeid="6">
        <makename>adidas</makename>
      </make>
      <price>50</price>
      <colour>red</colour>
      <country>china</country>
    </shoe>
    <shoe shoeid="2">
      <make makeid="4">
        <makename>nike</makename>
      </make>
      <price>100</price>
      <colour>blue</colour>
      <country>usa</country>
      <country>china</country>
    </shoe>
</shoes>
```

Figure 5: shoes.xml: XML document of shoe data

XML constraints. Figure 7 is an example of the XML Schema used to constrain the XML shoe document in Figure 5. Given that the XML in Figure 5 conforms to the constraints we can say that it is both *well-formed* and *valid*. Note that all XML Schema language components in Figure 7 are preceded by `xsd:` this tells the parser that they come from the XML Schema *namespace* defined in the processing instruction in line 2. See [19] for more information about namespaces.

The XML Schema standard is extremely comprehensive and allows very rich constraints to be created. Of particular interest to data exchange and transformation are XML Schema's support for both simple and complex data types and its support for key and unique constraints. As can be seen from the example XML Schema documents also conform to standard XML syntax which means they can be parsed by existing XML tools.

The complexity of the XML Schema standard has led a number of authors to suggest portions of the standard that can be easily understood but that are still able to offer sufficient power for specific tasks such as data storage [30, 24].

The example XML Schema includes the components deemed to be of most use when using XML Schema to define data storage and transfer schemas. Other authors[14, 33] have used simpler but non-standard schemas to constrain XML. I felt it was better to work with a portion of an accepted standard schema definition language. As time allows this work can be extended to incorporate more of the XML Schema standard.

The following is a list of the XML Schema constructs I will be discussing in the rest if this report.

- `element`: These define the name and type of elements that may be present in an XML document conforming to this schema. Elements can be of simple or complex type. Occurrence constraints determines how often an element must occur. Setting `minOccurs` to 0 will make the element optional. Setting `maxOccurs` to n means the element can be repeated n times. Setting it to `unbounded` means the element can occur an unlimited number of times.

```
<?xml version='1.0' encoding='UTF-8'?>

<!ELEMENT country (#PCDATA)*>

<!ELEMENT colour (#PCDATA)>

<!ELEMENT price (#PCDATA)>

<!ELEMENT makename (#PCDATA)>

<!ELEMENT make (makename)>
<!ATTLIST make
    makeid CDATA #IMPLIED
  >

<!ELEMENT shoe (country|colour|price|make)>
<!ATTLIST shoe
    shoeid CDATA #IMPLIED
  >

<!ELEMENT shoes (shoe)*>
<!ATTLIST shoes
    xsi:noNamespaceSchemaLocation CDATA #IMPLIED
    xmlns:xsi CDATA #IMPLIED
  >
```

Figure 6: DTD for shoe example

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name = "validCountries">
      <xsd:restriction base="xsd:string">
          <xsd:enumeration value="china" />
          <xsd:enumeration value="germany" />
          <xsd:enumeration value="usa" />
      </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="shoeType">
    <xsd:sequence>
      <xsd:element name="make">
        <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="makename" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="makeid" type="xsd:integer" use="required" />
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="price" type="xsd:integer" />
      <xsd:element name="colour" type="xsd:string" minOccurs = "0" />
      <xsd:element name="country" type="validCountries" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="shoeid" type="xsd:integer" />
  </xsd:complexType>

  <xsd:element name="shoes">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="shoe" type="shoeType" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
      <xsd:key name="uniqueShoeid">
        <xsd:selector xpath=".//shoes/shoe"/>
        <xsd:field xpath="@shoeid"/>
      </xsd:key>
      <xsd:key name="uniqueMakeid">
        <xsd:selector xpath=".//shoes/shoe/make"/>
        <xsd:field xpath="@makeid"/>
      </xsd:key>

  </xsd:element>
</xsd:schema>
```

Figure 7: shoes.xsd: XML Schema for shoe example

- `attribute`: These define the name and type of the attributes. Attributes are always of simple type and can occur once or not at all. They are optional by default. The `use` attribute in the declaration can be set to `required` to make the attribute mandatory in an instance document.

- `simpleType`: A number of simple types are defined within the XML Schema standard. Examples include `integer, float` and `string`. Other simple types can be defined by the schema author.

- `complexType`: Complex types allow elements and attributes to be combined to form new types. The `shoeType` is an example of a complex type. The way in which the elements and attributes are combined can be controlled in a number of ways. In the example the elements of the complex type form a sequence. This specifies that in the XML document, all the elements must appear in the order given in the complex type. See [17] for more details. Types can be defined outside the main schema definition. `countriesType` and `shoeType` are global types that can be referred to by name and reused. Complex types can also be extended or restricted. This is discussed in Section 8.

- `key/keyref`: Certain attributes or elements within a complex type can be defined to be unique or key elements. Key elements are not only unique but must also be present in the XML document. They are like primary keys in the relational model. In the example the `key name = "uniqueShoeid"` constraint specifies that all elements of type `//shoes/make/shoe` must have a unique `shoeid` attribute. The `selector` and `field` attribute's values are given in a simplified XPath specification discussed briefly in Section 2.4.1. The `keyref`s are used to reference `key` elements defined in other parts of the XML document. The `selector` attribute can include a number of different XPaths separated by the | character. This can allow a `keyref` to reference a 'choice' of elements and is discussed more fully in Section 8.3.

- `ID/IDREF`: The ID type is equivalent to the ID in a DTD. It defines an attribute to be unique within a document. IDs are more limited than keys in that they can only apply to attributes and ID values cannot be an arbitrary string. For instance they must start with a letter. `IDREF`s are used in the same way as `keyrefs`. `ID/IDREF` apply to the global namespace. They do not allow elements with the same name but in different namespaces to have different values as `key/keyrefs` do. These shortcomings mean that `key/keyref`s are more useful and so they will be used as the ID constraint mechanism in this project.

An XML document that conforms to a given XML Schema is called an *instance* document of that schema. Together `complexType`, `simpleType`, `element` and `attribute` determine the appearance of elements and attributes and their content in instance documents.

While XML instance document is being parsed a post-validation-info set (PSVI) [17] containing the XML Schema constructs consulted during the parsing is created. This can be accessed programatically via the XML Schema API [25].

2.4 Representing Data Graphically

As can be seen from the above, XML and the relational model are fundamentally different ways of representing data. There is no simple correspondence between their modeling constructs. It is also no immediately obvious what the relationships between elements within each model are. To try and make these relationships easier to see a number of graphical models of both XML and the relational model have been defined. Graphical models also have the advantage that humans are much better at interpreting data graphically than by looking at textual representations. They allow similarities and differences to more easily recognised. This section describes some graphical representations specifically oriented to the XML or relational model. Section 5 describes a graphical model that can be used to describe many different models making it easier to see the similarities and differences between schemas from those different models.

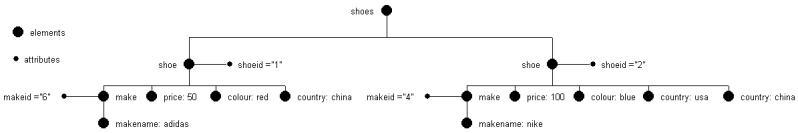

Figure 8: XPath Data Model of XML in Figure 5

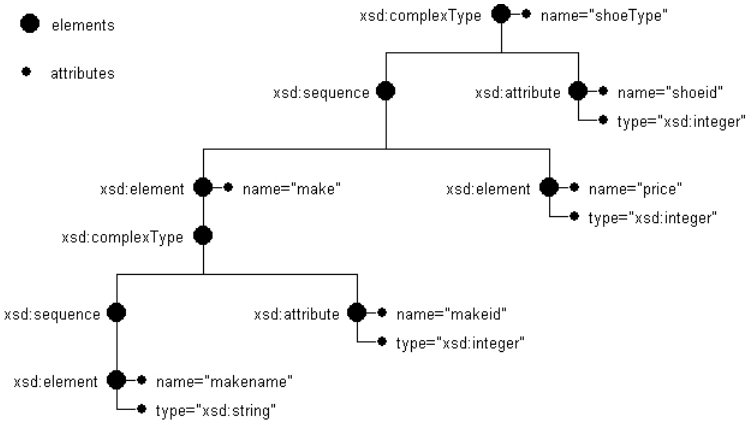

Figure 9: XPath Data Model of a portion of the XML Schema in Figure 7

2.4.1 XPath

The XPath [4] data model is often used to describe XML data. XPath is a non-XML language used to identify particular parts of XML documents. It allows XML documents do be viewed as trees. Elements, attributes, comments, processing instructions and character data are represented as nodes on the tree and the relationship between them represented by the tree's branches. There are two different types of branch: element branches that define the hierarchical relationship nodes have to each other and attribute branches that lead from an element node to any attributes it may have. Figure 8 shows an XPath representation of the XML from Figure 5.

XPath expressions identify particular elements or attributes in the tree and are formed by tracing the structure of the tree from a context node. Using the syntax '//' sets the context node to be the root node. For example the following XPath expression identifies the shoe made by nike in the XML document in Figure 5:

```
//shoes/shoe/make[makename="nike"]
```

The `//` at the beginning of the expression tells the parser to start looking from the root of the document and then follow the tree branches through the `shoe` and `make` nodes to find the value of the `makename` element. Attributes are identified by preceding the name with a `@`. For example:

```
//shoes/shoe[@shoeid = "1"]
```

Identifies the shoe with `shoeid` 1.

As mentioned before a key advantage XML Schema has over other XML schema mechanisms is that XML Schema documents conform to XML syntax themselves. For example Figure 9 shows an XPath representation of a portion of the shoeType complexType in the XML Schema from Figure 7.

We can then use XPath expressions to get information about the XML Schema elements. For example if we assume the context node is the element `xsd:complexType name = "shoeType"` then we could find the type of the price element with the following XPath expression:

```
xsd:sequence/xsd:element[@name = "price"]@type
```

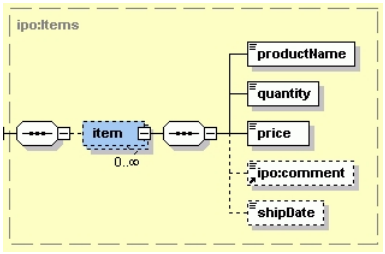

Figure 10: xmlspy graphical representation of an XML Schema fragment

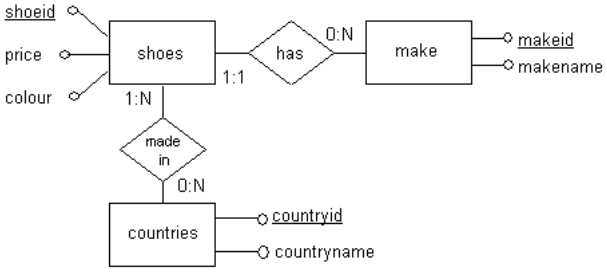

Figure 11: ER Data Model of the RDBMS Schema in Figure 4

2.4.2 XMLSPY

xmlspy® [1] is a commercially available product that provides graphical tools to help design XML Schemas. An example of their graphical representation of a schema fragment is given in Figure 10. This allows the designer to concentrate on the semantics of their design rather than getting bogged down in the complicated XML Schema syntax and is obviously a much easier way to create schemas than with a text editor.

2.4.3 Entity-Relationship Models

Figure 11 similarly represents the two relational tables from Figure 4 in the popular Entity Relationship(E-R) model. The square boxes represent the relations, the nodes represent fields (entities) in those relations and the links between the boxes represent relationships between relations. Each relation can have a primary key represented by an underlined field name. For example `shoeid` is a primary key. Each relationship has a cardinality associated with it representing a one-to-many, zero-to-many or many-to-many relationship between the two relations. In the Figure there is a one-to-many relationship from shoes to countries. Each shoe can be made in one or more countries. There is a zero-to-many relationship from countries to shoes. Each country may be a manufacturer of zero or many different shoes.

As can be seen from the figures the similarity between the two data sets is more apparent than before. The data in both models is represented by nodes and relationships between bits of data by graph edges. The job in translating from one to the other will be to flatten the XPath and combine the different relations in the E-R model. This graphical approach to representing data will be discussed in more detail in Section 5.

2.5 Querying

Data stored anywhere, be it in XML documents or RDBMSs is useless unless it can be retrieved. XML and RDBMSs both allow data retrieval via well defined query languages.

2.5.1 XQuery

XQuery [18] is an attempt by the W3C to create a powerful vendor independent and easy-to-use method for query and retrieval of XML data. It has grown out of a number of earlier standards like XQL and XML-QL. As of November 2003 XQuery 1.0 was a W3C Working Draft.

The data model that XQuery uses is based on that of XPath [4]. XPath alone is useful for simple data extraction, XQuery builds on XPath with FLWOR expressions. The name comes from the `For, Let, Where, Order by` and `Return` keywords that make up the expression. These expressions are the building blocks of XQuery. The following is an example of a FLWOR expression to retrieve the colour of the shoe made by nike in the document in Figure 5:

```
1    for $i in document("shoes.xml")//shoes/shoe
2    where $i/make/makename = "nike"
3    return
4        <result>
5            {$i/colour}
6        </result>
```

Line 1 sets up a loop to select all the XML identified by the XPath expression `shoeshoe` from the document `shoes.xml` and stores it in the variable $i. Line 2 is the XPath query the results must match. Lines 4 and 6 nest the result inside a `result` tag. Line 5 is the XPath expression of the field being queried. An XML document that looks like this:

```
<result>
  <colour>blue</colour>
</result>
```

will be produced.

2.5.2 SQL

SQL stands for Structured Query Language and is the standard language by which RDBMSs are queried. SQL is based on the relational algebra underlying RDBMSs. As with RDBMSs themselves, SQL is well understood and only an illustrative example will be provided here.

```
SELECT colour FROM shoes,make WHERE shoe.makeid = makes.makeid and make = "nike"
```

This joins the `shoes` and `makes` tables and then retrieves the colours of all the shoes made by nike. The result will be 'blue'.

3 Moving Data between RDBMSs and XML

We start this section by discussing the structural differences between XML and relational databases. There follows a brief description of the translation process and how current systems perform this process.

3.1 XML/Relational Schemas

Both XML and relational data may be constrained by schemas. In XML this schema is optional whereas in RDBMSs it is an inherent part of the database. The XML schema may be stored within the document or as a separate file. The relational DDL statements that create a database and thereby its schema must be issued before any data can be stored in it. This schema then acts as a mandatory check on the validity of tuples before they are inserted. To be of most use, an XML schema should also be created first, and any XML documents created should be validated against it. It is important to note that in XML this process is not mandatory.

3.1.1 Differences between XML and relational schemas

The data models provided by RDBMSs and XML are fundamentally different as can be seen from the two schemas in Figures 4 and 7. This makes translation between the two difficult.

An important difference between the XML and relational data schemas is XML's support for *nesting*. Element types are allowed to contain other element types as is the case in the example in Figure 7 where make is a subtype of shoe. These can be used to build arbitrarily deep part-of hierarchies. It is required that all component element types are rooted in a single element type. This is in contrast to RDBMSs, which are essentially flat.

Associated with nesting comes the notion of order. Unlike relations and tuples in RDBMSs, the element types and elements of an XML document adhere to both an explicit and implicit order. The order can be explicitly specified in the XML schema or implicitly at the XML document level where the order of data elements is defined by their position in the document. This ordering may be of semantic importance.

As XML Schema asserts itself as the dominant schema language for XML it brings specific problems to the translation process due to its semantic richness. On top of nesting and ordering XML Schema introduces the concept of complex types and other structures that are difficult to map directly to relational structures. It also greatly increases the number of ways a relational schema may be expressed in XML leading to questions of what the 'best' representation is. Addressing these differences is a very important part of the translation process.

3.2 The Translation Process

There are 3 main aspects to the translation process:

1. Creation of a relational schema from a given XML schema or an XML schema from a given relational schema.

2. Deciding whether the schema that has been created is the most appropriate one or giving a human expert the chance to influence the creation of the schema. This is most important when creating an XML schema as the possibilities are numerous.

3. Providing a way of querying the data held in one format using the query language of the other format or moving the data from one format to the other.

Problem 1 is of great importance to the process. Without a precise description of the destination the data is to go to, problem 3 becomes very difficult. Well established techniques exist to translate a given XML schema into a relational one [14] but creating an XML schema from a relational schema is a relatively new field. Until recently systems that exported relational data as XML like Silkroute [21] assumed that such a schema existed before the export process began. The automatic

creation of an XML schema from a relational one is an important task and will be investigated in depth in Section 7. Two newer approaches provide ways of generating the schemas from the RDBMS schema [5, 3].

Problem 2 stems from the differences between the relational and semi structured XML models. There is no obvious one-to-one mapping from structures in the relational model to XML and so approximations must be made. These can be done in a number of ways some more appropriate to a given task than others. Bohannon et al. describe an automatic approach in [14], discussed briefly in Section 3.5.1, while a more user directed approach is adopted by L.Popa et al.[11].

If a relational database is simply being used as a place to store XML documents and we have access to that RDBMS then the data can be queried directly using SQL. If we do not and the queries need to be done on the original XML then a way of translating them into SQL is needed. XML views of the underlying relational data are a popular way of doing this and are the approach taken in [10]. If, on the other hand, relational data is being exposed via XML two main approaches have been adopted to accessing it. The first also uses XML views. The relational data is then either exported into an XML document based on these views and queries issued on the XML document itself or the queries are issued on the views and then translated [9]. The second adopted in [3] creates an XML Schema representation of the relational schema. Queries are then issued against that. Section 3.6 will discuss these approaches in more detail.

3.3 Main problems

The differences between XML data and that stored in RDBMSs means that moving data from one format to the other poses some significant problems. The first is capturing the complexity of XML Schema in the semantically simpler relational model and the second is trying to create an efficient and appropriate representation of a relational schema in XML Schema that takes advantage of the the latter's flexibility.

The following are some of the issues that need to be addressed when moving data from XML to the relational model. The first three deal with trying to maintain the structure of the XML document. The final point must be addressed so that the created relations have data integrity:

1. How to model flat XML data.
2. How to model nested XML data.
3. How to maintain the order of the XML elements.
4. How to make sure that the resulting relations have a suitable candidate key.

XML can be used for a very wide variety of tasks. This report concentrates on the aspects of XML that are most closely related to data storage, i.e. elements, attributes and the way they are nested together. My model does not support mixed content and white space in an XML document and processing instructions that do not relate to the XML Schema used to validate the document are ignored. Ways of handling these could be introduced into the model in future.
When moving data from the relational model to XML the following must be taken into account:

1. How to model relations and attributes.
2. How to model primary keys and foreign keys.
3. How will the resulting XML data be nested.

3.4 Existing Approaches

Nearly all current systems that translate between XML and RDBMSs fall into two distinct groups:

- Systems that use relational databases as a way of storing XML data, thereby taking advantage of the power of RDBMSs.
- Systems that use XML as a way of exporting relational data to the Internet.

3.4.1 Storing XML in relational databases

Existing relational databases have many powerful tools to manipulate, store and retrieve data. Storing XML in these relational systems means consumers of the data are able to take advantage of these tools to store and query the XML. There are a number of different ways of storing the XML in an RDBMS. Each method had advantages and disadvantages.

The most straightforward approach is to store XML documents as a whole within a single database attribute (BLOBs or CLOBs). An advantage of this method is any XML document can be stored, including documents without an XML schema. The disadvantage is that the data cannot be queried and none of the structure of the XML document is preserved.

Any method for storing XML in a relational database that does not store the documents as a single entity must have at least two steps. The first is to create a relational schema to store the XML in and and the second is to *shred* the XML document to capture the data from it and store that data in the created schema. There are two main ways this has been done. They are called the edge schema and the element schema.

In the edge schema proposed by Florescu and Kossman in [22], the whole XML tree is stored in a *fixed* relational schema that records details of nodes and edges in a way that is generic to all XML data. Because the schema is fixed it is not necessary to have an XML schema file describing the data. Figure 12 shows the XML from Figure 5 represented as an edge schema. Each element and attribute is assigned a unique id, a parent id, an order, a type, a name and a value. The id uniquely identifies each node in the XML tree. The parent id identifies the parent node and the order identifies the position of the node in its branch of the tree. Name is the name of the element or attribute and value is its value. Type identifies the node as an element or attribute.

With reference to the problems mentioned in Section 3.3, flat and nested data are dealt with in the same way, `id` provides the candidate key and the order is preserved by the `ord` column.

As with the previous method, any XML document can be stored in this way, in addition data from the document can be searched up to a point. Unfortunately queries over this schema can quickly become cumbersome or even intractable if there is a high degree of nesting in the original XML document. For example the SQL to get the colour of the 'nike' shoe would be as follows:

```
SELECT e3.value FROM edge_schema e1,edge_schema e2,edge_schema e3
WHERE e1.name='makename' AND e1.value='nike' AND
e1.pid = e2.id AND
e2.pid = e3.pid AND e3.name = 'colour';
```

The complexity of the above query on a relatively simple XML document shows that this method is not a good way to store a very large, deeply nested XML document.

The element schema proposed by Shanmugasundaram et al. in [8] overcomes some of the problems of storing all the XML data in a single relational table by providing a one to one mapping between multi-valued XML elements and relational tables. This method requires an XML schema describing the XML. Each potentially multi-valued schema element is represented as a new relational table. In [8] Shanmugasundaram et al. use DTD as their schema language. Any * or + element definition becomes a table. This method can be extended if the schema is described by an XML Schema document. This is the approach I have adopted and is described in detail in Section 7.1.

In common with the edge schema each table has an `id`, a `pid` and an `ord`. These have the same meaning as in the edge schema. The other fields in the element schema are the names of child attributes or single-valued child elements. The nesting of elements is represented as foreign key associations from the id of a child element to the `pid` of its parent element. The element schema representation of the XML in Figure 5 is shown in Figure 13 as well as the SQL DDL used to create the tables.

The nesting of the unbounded element country inside shoe is represented by the `es_country_fk` foreign key constraint on the `es_country` table. As with the previous method the `id` column is added as a candidate key and the order of elements is recorded in the `ord` column.

id	pid	ord	type	name	value
1	0	0	element	shoes	NULL
2	1	1	element	shoe	NULL
3	2	1	attribute	shoeid	1
4	2	1	element	make	NULL
5	4	1	attribute	makeid	m1
6	4	1	element	makename	adidas
7	2	2	element	price	50
8	2	3	element	colour	red
9	2	4	element	country	china
10	1	2	element	shoe	NULL
11	10	1	attribute	shoeid	2
12	10	1	element	make	NULL
13	12	1	attribute	makeid	m2
14	12	1	element	makename	nike
15	10	2	element	price	100
16	10	3	element	colour	blue
17	10	4	element	country	usa
18	10	5	element	country	china

Figure 12: Edge Schema representation of the XML from Figure 5

As can be seen the use of the XML schema to help create the RDBMS schema has resulted in something must closer to the relational schema in Figure 4. Querying this schema is also far easier than the edge schema.

```
SELECT es_shoe.colour FROM es_shoe, es_make
WHERE es_make.pid = es_shoe.id AND es_make.makename = 'nike'
```

Generating this schema is more complicated than the simple edge schema but something of this nature is essential for large, deeply nested XML documents.

Each of the methods described above represents the XPath tree in Figure 7 at a different level of detail. Storing the whole document in a CLOB takes the XPath tree as a whole and stores it. The edge schema flattens the XPath into a single table. The element schema converts each non-leaf node into a relational table. This shows the power of more abstract representations. Three different ways of looking at the XPath tree have resulted in three different relational schema representations. These would not have been so obvious from simply looking at the XML Schema and certainly not from the XML itself. This suggests that a graphical representation of the XML schema may be the most useful way of representing an XML document that we wish to convert to a different data model.

3.4.2 Exporting relational data to XML

In many cases we are not interested in converting a whole database into XML, what we really want are selected bits of data, i.e. the results of queries.

Silkroute [21] was one of the first systems that allowed a general and efficient way to export relational data to XML. It defined an intermediate declarative query language called RXL to express the general transformations from the relational store to the XML view.

Data is exported from the relational system in two steps. First a virtual XML view of the relational data is created in RXL. Then the consumer application queries this view using XML-QL to extract the data in XML format.

```
+------+
| id   |
+------+
|    1 |
+------+
CREATE TABLE es_shoes
(id INTEGER NOT NULL,
 CONSTRAINT shoes_pk PRIMARY KEY (id)
)

+----+-----+-----+--------+-------+--------+--------+----------+
| id | pid | ord | shoeid | price | colour | makeid | makename |
+----+-----+-----+--------+-------+--------+--------+----------+
|  2 |   1 |   1 |      1 |    50 | red    |      1 | adidas   |
|  5 |   1 |   2 |      2 |   100 | blue   |      2 | nike     |
+----+-----+-----+--------+-------+--------+--------+----------+
CREATE TABLE es_shoe
(id INTEGER NOT NULL, pid INTEGER NOT NULL, ord INTEGER NOT NULL,
 shoeid INTEGER NOT NULL,
 price INTEGER NOT NULL,
 colour VARCHAR(20) NOT NULL,
 makeid INTEGER NOT NULL,
 makename VARCHAR(20) NOT NULL,
 CONSTRAINT es_shoe_pk PRIMARY KEY (id),
 CONSTRAINT es_shoe_fk FOREIGN KEY (pid) REFERENCES es_country
)

+----+-----+-----+-----------+-------------+
| id | pid | ord | countryid | countryname |
+----+-----+-----+-----------+-------------+
|  4 |   2 |   1 |         1 | china       |
|  7 |   5 |   1 |         2 | usa         |
|  8 |   5 |   2 |         1 | china       |
+----+-----+-----+-----------+-------------+
CREATE TABLE es_country
(id INTEGER NOT NULL, pid INTEGER NOT NULL, ord INTEGER NOT NULL,
 countryid INTEGER NOT NULL,
 countryname VARCHAR(20) NOT NULL,
 CONSTRAINT es_country_pk PRIMARYKEY(id)
 CONSTRAINT es_country_fk FOREIGN KEY (pid) REFERENCES es_shoe
)
```

Figure 13: Element Schema representation of Figure 5

```
SELECT XMLElement("COLOUR-TAG",colour) FROM shoes
WHERE price = 50;

<COLOUR-TAG>red<\COLOUR-TAG>
```

Figure 14: SQL/XML query and result

Once the virtual view of the relational data has been constructed and the XML-QL query formulated, the two queries are composed. The result of the composition is another RXL query that extracts only the necessary data from the relational database.

Recently major database vendors including IBM, Microsoft, Oracle, and Sybase have been working on a proposed new standard called SQL/XML to allow RDBMSs to provide XML results to SQL queries. Oracle 9i supports some of the proposed standard [23]. Figure 14 shows a SQL/XML query on the shoes table and the resulting XML. Within the XMLElement the first field is the name of tag to encapsulate the result and the second the name of the of the field in the table to query. A disadvantage of this approach is that it creates unconstrained XML documents. There is nothing stopping two groups creating different XML documents from the same relational data. This could lead to confusion if they were processed together.

3.4.3 Generating an XML schema from a Relational schema

The methods described above all assume that an XML schema describing the XML already exists. This section will look at two methods for inferring an XML schema from a relational schema.

Lee et al describe two algorithms Nesting-based Translation (NeT) and Constraints-based Translation (CoT) in [5]. NeT works on a single relational table deriving a nested structure from the flat relational model by repeatedly applying a nest operator to create a hierarchical XML schema. CoT extends this idea by considering possible inclusion dependencies between relational tables within the same database. These constraints are derived either from the database via the ODBC/JDBC interface or provided by a schema expert. In this way CoT attempts to merge multiple interconnected tables into a coherent and hierarchical parent-child structure.

Both the algorithms generate their output in a generic schema language defined by the authors. Lui et al. provide an algorithm that generates XML Schema directly [3]. They define a number of mapping rules that take advantage of the comprehensiveness of the XML Schema standard to accurately model relational constraints. These include things like primary and foreign keys, null/not null constraints and uniqueness constraints. Figure 15 shows the XML Schema created by this method for the relational schema in Figure 4. As can be seen from the example, primary and foreign keys and uniqueness constraints are defined in terms of ID and IDREF types and null/not null constraints in terms of minOccurs and maxOccurs XML Schema constraints. The algorithm also delivers a schema with a good degree of nesting. The use of `ID/IDREF` has the disadvantage that numeric relational keys need to have a letter prepended to them because elements of type `ID` cannot begin with a number. For example the `shoeid` becomes `s1` when the shoe table is represented in XML as in Figure 16. This is necessary because they use XQuery to query the created schemas and need the dereferencing support that the `ID/IDREF` constructs provide. The method I propose does not need dereference support for queries and so I will use the more flexible `key/keyref` constructs to model primary and foreign key constraints.

3.5 Choosing the most appropriate schema

As can be seen from the above there are a number of different relational and XML schemas that describe the same data. Two automatic techniques for overcoming this problem are described below. The first creates an efficient relational schema to store XML data and the second provides a generic search method to get XML out of any relational storage scheme. Others propose a more interactive approach. IBM have created the prototype of a system that allows some level of

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "shoe_XML">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="shoe" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="price" type="xsd:integer" />
              <xsd:element name="colour" type="xsd:string" />
              <xsd:element name="countries" minOccurs="0"
                           maxOccurs="unbounded">
                <xsd:complexType>
                  <xsd:attribute name="countryid" type="xsd:IDREF" />
                </xsd:complexType>
              </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="shoeid" type="xsd:ID" />
            <xsd:attribute name="makeid" type="xsd:IDREF" />
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="makes" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="makename" type="xsd:string" />
            </xsd:sequence>
            <xsd:attribute name="makeid" type="xsd:ID" />
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="countries" minOccurs="0"
                    maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="countryname" type="xsd:string" />
            </xsd:sequence>
            <xsd:attribute name="countryid" type="xsd:ID" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 15: XML Schema created by Lui et al. method

```
<?xml version="1.0" encoding="UTF-8"?>

<shoe_XML xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='file:sch_shoes.xsd'>
  <shoe shoeid="s1" makeid="m1">
    <price>50</price>
    <colour>red</colour>
    <countries countryid="c1" />
  </shoe>
  <shoe shoeid="s2" makeid="m2">
    <price>100</price>
    <colour>blue</colour>
    <countries countryid="c2" />
    <countries countryid="c1" />
  </shoe>

  <makes makeid="m1">
    <makename>adidas</makename>
  </makes>
  <makes makeid="m2">
    <makename>nike</makename>
  </makes>
  <countries countryid="c1">
    <countryname>china</countryname>
  </countries>
  <countries countryid="c2">
    <countryname>usa</countryname>
  </countries>
</shoe_XML>
```

Figure 16: XML conforming to the XML Schema in Figure 15

user interaction in the choice of schema mappings [11]. The Net and CoT algorithms described above also suggest the participation of a human expert to help the system create the best possible schema.

3.5.1 LegoDB

LegoDB is primarily concerned with storing XML in a relational database in the most efficient manner. It creates a number of possible relational schemas based on a given XML Schema and chooses the most *efficient* via a cost-based approach. The method relies on the fact that many different XML Schemas can describe exactly the same set of documents and that these different XML Schemas may result in relational schemas that differ widely in the performance they can offer. The authors note that the possible number of these different schemas is very large, possible even infinite, so they define an algorithm to limit the number created.

Given an XML Schema the technique transforms it into a number of other different but semantically equivalent schemas. The schemas are presented in a modified schema language *P-schema* that adds statistics about the underlying XML data for later performance analysis. They make use of XML Query Algebra that abstracts away some of the more complex feature of XML-Schema. The transforms are done at the XML Schema level rather that the relational because XML Schema is semantically far richer and can express complex constraints that would be very difficult to express in a relational context with relative ease.

3.6 Querying the data

Data can be extracted from an relational database and expressed as XML in a number of ways. If we have direct access to the relational database and know the schema we want to query we could use SQL/XML [23]. If, however, we want to generate queries on the XML there are various options. Firstly we could move all the data from the relational database into one or a number of XML documents and then query those directly. This is time-consuming but may be necessary if the original relational database is not available. For example the data needs to be sent to a customer who has no access to the sellers RDBMS. If the relational database is available XML views of the underlying relational data can be created and queries made on those views. These queries are then translated into the corresponding SQL queries that are then executed on the RDBMS. This is the approach taken by SilkRoute [21] and XPERANTO [12].

A more recent approach [3] has been to create an XML Schema of the underlying relational database and then execute queries on that schema. This has the advantage over using XML views that any integrity constraints on the original relational data can be preserved. Users creating queries can find out what the constraints are by looking at the XML Schema document. Using the XML Schema also simplifies the query translation process.

Conversely it is important that XML that has been stored in a relational can be queried efficiently. Again if we have access to the RDBMS we can simply use SQL to extract the data. The queries may be more or less complicated depending on the mechanism used to create the schema but the method will be the same. If the queries need to be done on the XML then things are a bit more complicated. Shanmugasundaram et al. [10] recognise that there are a number of different possible relational schemas that can be generated from a given XML schema and provide a way of querying any of them without having to provide a specific query processor for each.

Their query technique provides a *reconstruction XML view* over the generated relational schema that allows XML queries to be issued on relational schemas generated by any technique, removing the need for a new query processor for each different schema that may be created. The XML View (virtually) reconstructs the stored XML from the rows of the relational tables based on the method used to store the XML. Queries over the stored documents can then be treated as queries over the XML view regardless of the underlying relational schema.

4 An Approach to the Translation Using Graphical Techniques

A system that translates between XML and RDBMSs needs to have the following features:

1. A way to create a relational schema from a given XML document or XML Schema.
2. A way to create an XML schema from a given relational schema.
3. A method for deciding whether the schema that has been created is the most appropriate one or giving a human expert the chance to influence the creation of the schema.
4. A way of querying the data in either format or moving it from one format to the other.
5. A graphical way of describing each schema independently of its model would make it easier to see any correspondences that may exist as humans are much better at interpreting information graphically than textually. Section 6.1 provides a number of examples of this.

The following sections will describe a system that has all of the above features and how the various difficulties associated with them are overcome.

4.1 XML to Relational

As noted previously it is possible for an XML document to have no schema describing it. This, however, can lead to XML documents with the same information in them having different structures. Documents such as these could be stored in a relational database using Florescu and Kossman's method described in [22]. As stated above, using this method makes getting at the data again is difficult. To get an accurate and efficient transformation a schema is required. For the rest of this report I will assume that XML documents I wish to transform are constrained by an XML Schema document.

When an XML document constrained by an XML Schema is parsed, the post schema validation info set (PSVI) contains the XML Schema structures. These are used to create the relational schema. A method similar to the element schema proposed by Shanmugasundaram et al. is adopted. The use of XML Schema as the schema language rather than DTDs potentially greatly improves the quality of the relational schema produced. For example if any `key` constructs exist in the XML Schema these can be used as candidate keys in the relational schema removing the need the `id` field used in the element schema.

The following are some general rules about how XML Schema constructs will be transformed into relational constructs:

- XML Schema `complexTypes` will be transformed into relations.
- XML Schema `attributes` and `elements` that are of a simple type within a given complex type will become attributes in the corresponding relation. Any elements with a `minOccurs` of 0 will be make null-able. Elements with a `maxOccurs` of `unbounded` will generate relations.
- XML Schema `key` constraints on a given complex type will become primary keys on the corresponding relation.
- Single value XML Schema `keyref` constraints on a given complex type will become foreign keys on the corresponding relation. Transforming `keyrefs` and `keys` that contain a choice in the XPath of their `selector` attribute is discussed in Section 8.3.
- XML Schema `elements` within a complex type that are of a complex type themselves will generate a link relation between the two relations generated by the enclosing complex type and the complex type of the element.

Section 7.1 describes this process in detail and a number of special cases are discussed in Section 8.

4.2 Relational to XML Schema

The algorithm for creating an XML schema from a relational schema is based on that proposed by Lui et al [3]. Again XML Schema is used as the schema language. My query method does not use any query languages that rely on the dereferencing capability of the `ID/IDREF` constructs so I'm able to use the more flexible `key/keyref` to define primary and foreign key constraints.

The following are some general rules about how relational constructs will be transformed into XML Schema:

- Relations will become XML Schema `complexTypes`.
- Non-key attributes within a relation will become `elements` of simple type within the corresponding complex type. If they are null-able the occurrence constraint `minOccurs` will be set to 0.
- Primary key attributes will become XML Schema `attributes` within the corresponding complex type and an XML Schema `key` identity constraint will be created.
- Foreign key attributes will become XML Schema `attributes` within the corresponding complex type and an XML Schema `keyref` identity constraint will be created.
- A GUI will be provided to allow the user to influence the way the resulting XML Schema will be nested.

Section 7.1 describes this process in detail and a number of special cases are discussed in Section 8.

4.3 The Appropriateness of the Schema

Creating an appropriate XML schema for a given relational data set may depend on things that were not known when the translation algorithm was written. For this reason I have created a GUI tool that allows a schema expert to design the XML Schema they think best, from the given relational schema. Any number of different schemas may be created. The tool will check the validity of each one and will not allow illegal transformations. The tool will be discussed in detail in Section 9

4.4 Querying the Schema

All transformations of data and schemas are done within the Automed framework [2] using the HDM as a Common Data Model(CDM). The IQL [31] query language within Automed allows all the data in the relational database to be moved to XML or queries can then be run on the Automed XML Schema that will be translated to queries on the relational database itself.

4.5 An Abstract Data Model

A way of defining the semantics of each model's constructs independently of their data models would help to ease translation between them. Section 2.4 presented two graphical languages that go some way to abstracting away the differences. However, they were designed as high-level languages for a specific model and cannot be used to represent constructs in a different model. For example there is no way to represent nesting or order in the E-R model. What is needed is a unifying lower-level language in terms of which constructs, transformations and inter-model links for both modeling languages can be defined. The HDM described in the next section is one such model. The Automed editor [26] provides a way of showing the relationships between the various components of a schema.

Figure 17 shows a screen shot from the Automed editor with an HDM representation of a schema on the left, a relational one of the same schema in the middle and a XML Schema representation on the right.

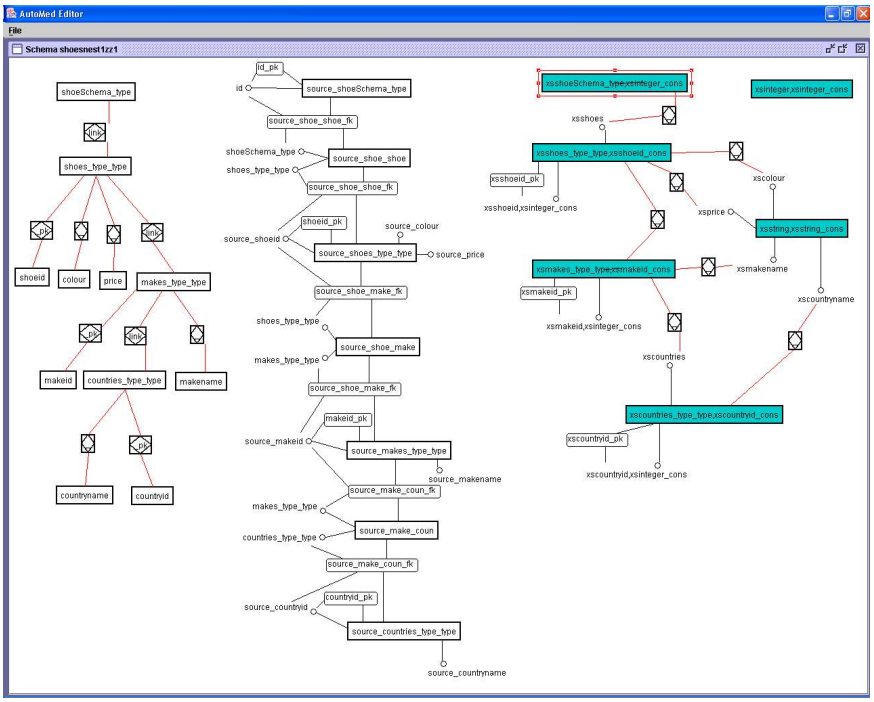

Figure 17: A screen shot of the Automed editor

5 HDM

The HDM [27] has been used to model data from a wide variety of sources including relational data [28] and XML [29]. HDM is a low-level modeling language based on a hypergraph data structure of nodes and edges. This structure and an associated set of constraints make up the language. A small set of primitive transformation operations that can be carried out on schemas represented in the HDM is also defined. Constructs and transformations in higher-level modeling languages are then defined in terms of these. This allows the automatic derivation of transformation rules in the HDM. The general nature of the language offers advantages over the model-specific techniques described in Section 2.4 when coming to describe data from differing sources such as RDBMSs and XML Schema. For example XPath cannot be used to represent relational schemas and the E-R model cannot be used to represent XML whereas HDM can represent both as shown in Section 6.1.

A schema in the HDM is a triple $S = \langle Nodes, Edges, Constraints \rangle$. The $Nodes$ and $Edges$ define a labeled, nested, directed hypergraph. It is nested in the sense that nodes can link to any number of other *nodes* and other *edges*. A query over S is an expression whose variables are members of $Nodes \cup Edges$. $Constraints$ is a set of boolean queries over S. Nodes are uniquely identified by their name. Edges and constraints may have an optional name.

An instance I of the schema S above is a set of sets that satisfies the following:

- Each construct $n \in Nodes$ has an optional type t that constrains the values that can be held in n.
- Each construct $c \in Nodes \cup Edges$ has an extent denoted by $Ext_{S,I}(c)$, that can be derived from I.
- Each set in I can be derived from the set of extents $\{Ext_{S,I}(c) | c \in Nodes \cup Edges\}$.
- For each $e \in Edges$, $Ext_{S,I}(e)$ contains only values that appear within the constructs linked by e.

- For each $c \in Constraints$, the query $c[v_1/Ext_{S,I}(v_1), ..., v_n/Ext_{S,I}(v_n)]$ evaluates to true, where $v_1, ..., v_n \in Nodes \cup Edges$ are the variables of c.

The function $Ext_{S,I}$ is called an *extension mapping*. A *model* is a triple $\langle S, I, Ext_{S,I} \rangle$. Two schemas are *equivalent* if they have the same set of instances. Two schemas S and S' can also be *conditionally equivalent* with respect to a condition f if any instance of S' satisfying f is also an instance of S.

The primitive transformations of the HDM are listed below. Each transformation is a function that when applied to a model returns a new model. Each transformation has a proviso that states when the transformation is successful. If a transformation is not successful it returns an 'undefined' model, denoted by $\emptyset$. Any transformation applied to $\emptyset$ returns $\emptyset$.

1. $renameNode\langle fromName, toName \rangle$ renames a node. Proviso: $toName$ is not the name of an existing node.

2. $renameEdge\langle\langle fromName, c_1, ..., c_m \rangle, toName \rangle$ renames an edge. Proviso: $toName$ is the name of an existing edge.

3. *addConstraint* c adds a new constraint c. Proviso: c evaluates to true.

4. *delConstraint* c removes constraint c. Proviso: c exists.

5. addNode($\langle\langle name, q \rangle\rangle$) adds a node *name* whose extent is given by the value of the query q. *name* is not the name of an existing node.

6. delNode($\langle\langle name, q \rangle\rangle$) removes a node. q is a query that states how the extent of the deleted node could be recovered from the extents of the remaining schema constructs. Proviso: the node exists and does not participate in any edges.

7. $addEdge\langle\langle name, c_1, ..., c_m \rangle, q \rangle$ adds a new edge between a sequence of existing constructs $c_1, ...c_n$. q is the extent. Proviso: *name* is not an existing edge, $c_1, ..., c_m$ exist and q satisfied the appropriate domain constraints.

8. $delEdge\langle\langle name, c_1, ..., c_m \rangle, q \rangle$ deletes the edge *name*. q states how the deleted edge could be recovered from the extents of the remaining schema constructs. Proviso: the edge exists and participates in no other edges.

A transformation t is *schema-dependent* with respect to a schema S if t does not return $\emptyset$ for *any* model of S otherwise t is *instance-dependent* with respect to S. If S can be transformed in S' they have the same set of instances and must therefore be equivalent. If a proviso f is necessary to transform S to S' then S and S' are conditionally equivalent with respect to f.

Every successful transformation t is reversible by a transformation t^{-1}. For example the transformation addNode($\langle\langle n, q \rangle\rangle$) can be reversed by delNode($\langle\langle n, q \rangle\rangle$).

5.1 Higher Level Modeling Languages in HDM

In general, any semantic modeling language consists of two types of construct: *extensional constructs* and *constraint constructs*. The *scheme* of a construct uniquely identifies it. Extensional constructs represent the set of data values in a given domain. There are three classes:

- **Nodal**: These may be present independently of any other constructs. They map onto nodes in the HDM.

- **Linking**: The extent of a linking construct is a subset of the Cartesian product of the extents of the constructs that it links. Obviously linking constructs cannot exist in isolation. They map onto edges in the HDM.

- **Nodal-Linking**: These are nodal constructs that can only exist when certain other constructs link to them. They are mapped to a node and an edge in the HDM

Relational Construct		HDM Representation	
Construct	**relation (R)**		
Class	nodal	Node	$\langle\langle \mathsf{rel}{:}r \rangle\rangle$
Scheme	$\langle\langle r \rangle\rangle$		
		Node	$\langle\langle \mathsf{rel}{:}r{:}a \rangle\rangle$
Construct	**attribute (A)**	Edge	$\langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a \rangle\rangle$
Class	nodal-linking	Links	$\langle\langle \mathsf{rel}{:}r \rangle\rangle$
	constraint	Cons	if (n=**null**)
Scheme	$\langle\langle r, a, n, t \rangle\rangle$		then makeCard($\langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a \rangle\rangle$,{0,1},{1..$N$})
			else makeCard($\langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a \rangle\rangle$,{1},{1..$N$})
		Cons	$x \in \langle\langle \mathsf{rel}{:}r{:}a \rangle\rangle \iff x \in t$
Construct	**primary key(P)**	Links	$\langle\langle \mathsf{rel}{:}r{:}a_1 \rangle\rangle, ..., \langle\langle \mathsf{rel}{:}r{:}a_n \rangle\rangle$
Class	constraint	Cons	$x \in \langle\langle \mathsf{rel}{:}r \rangle\rangle \leftrightarrow x = \langle x_1, ..x_n \rangle$
Scheme	$\langle\langle r, a_1, ..., a_n \rangle\rangle$		$\wedge \langle x, x_1 \rangle \in \langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a_1 \rangle\rangle \wedge ...$
			$\wedge \langle x, x_n \rangle \in \langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a_n \rangle\rangle$
Construct	**foreign key(F)**	Links	$\langle\langle \mathsf{rel}{:}r{:}a_1 \rangle\rangle, ..., \langle\langle \mathsf{rel}{:}r{:}a_n \rangle\rangle$
Class	constraint	Cons	$\langle x, x_1 \rangle \in \langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a_1 \rangle\rangle \wedge ...$
Scheme	$\langle\langle r, r_f, a_1, ..., a_n \rangle\rangle$		$\wedge \langle x, x_n \rangle \in \langle\langle \_, \mathsf{rel}{:}r, \mathsf{rel}{:}r{:}a_n \rangle\rangle$
			$\rightarrow \langle x_1, ..., x_n \rangle \in \langle\langle \mathsf{rel}{:}r_f \rangle\rangle$

Table 1: Definition of relational model constructs in the HDM

Constraint constructs are restrictions on the extents of the extensional constructs, they also can be used to represent the structure of the data schema in a semi-structured data model, for example XML Schema. Constraints are directly supported in the HDM. Table 1 shows how relational constructs are represented in the HDM.

Once the constructs of the new modeling language, M, have been defined in the HDM we need to define the following transformations for M within the HDM:

- For every construct in M we need an *add* transformation to add the equivalent nodes, edges or constraints to the underlying HDM schema. This transformation consists of zero or one HDM *addNode* transformations, the operand being taken from the Node field in the construct definition, followed by zero or one *addEdge* transformations taken from the Edge field. Finally a sequence of zero or more *addConstraint* transformations taken from the Cons(traint) field, are done.
- We need a *del* transformation for every construct in M, that reverses its *add* transformation. This consists of a sequence of HDM *delConstraint* transformations followed possibly by a *delEdge* and possibly a *delNode*.
- For those constructs that have a textual name a *rename* transformation is defined in terms of the HDM *renameNode* and *renameEdge* transformations.

Given a representation of M in the HDM these transformations can be derived automatically by applying one or more of the primitive HDM transformations defined above.

5.2 The Relational Model

The relational model can be taken to consist of relations, attributes, a primary key for each relation and foreign keys. Relations are represented in the HDM by nodes. The extent of a relation is its primary key. Attributes are also represented by nodes and have their own extent. They cannot exist independently of a relation, however, and so an accompanying edge in the HDM must be defined. Relational attributes can be constrained in that they can be defined as null or not null. This constraint is represented in the HDM as a choice between the attribute being optional (**null** $\rightarrow$ {0,1}) or mandatory (**null** $\rightarrow$ {1}). The primary key is a constraint that checks whether

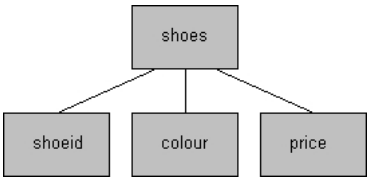

Figure 18: HDM representation of simple data

the extent of r is the same as the extents of the key attributes $a_1, .., a_n$. A foreign key is a set of attributes $a_1, ..a_n$ appearing in r that are the primary key of another relation r_f.

The model described in [28] is extended here with an additional type constraint being added to the attribute construct. First we define a type t to be the set of all possible values of that type. We define two basic types:

integer as the set of all integers

string as the set of all possible strings of any length

If we have an attribute of type t the type constraint can now be stated as follows:

$x \in \langle\langle \mathsf{rel}{:}r{:}a \rangle\rangle \iff x \in t$

A shorthand $\mathsf{setType}(\langle\langle \mathsf{rel}{:}r{:}a \rangle\rangle, \mathsf{t})$ is introduced for the above constraint.

The following subsections present ways in which different relational schemas may be represented in the HDM.

5.2.1 Unique Relational Tuples

Each tuple of data in relation **shoes** is unique: eg. (1,red,50), (2,blue,100), (3,green,75). The HDM representation is shown in figure 18

5.2.2 Relational Tuples with Repeated Elements

One of the fields in the tuple is repeated: (1,red,50,nike), (2,blue,100,nike), (3,green,75,adidas). There are two ways of representing this:

1. Include all the fields in a single table as above.
2. Create a second set of tuples with the repeated elements and an index to each value: (1,nike), (2,adidas). The original tuples then become (1,red,50,1), (2,blue,100,1), (3,green,75,2)

The HDM representation of the two options is shown in figure 19.

5.2.3 More Complex Relational Tuples

All the fields in a number of tuples may be the same except for 1: (1, red, 50, nike, china), (1, red, 50, nike, usa), (1,red,50,nike,korea), (2,blue,100,nike,china). Again there are various ways of representing this:

1. Again put all the fields in a single table. This has the distinct disadvantage that the key would have to be expanded to include all the elements in the table as every one could be repeated.
2. As in the previous example create a second set of tuples with the repeated elements and an index to each value: (1,china), (2,usa), (3,korea) and then create a link tuple: (1,1), (1,2), (1,3), (2,1). The original tuples become: (1,red,50,nike), (2,blue,100,nike).

The HDM representation of the two options are shown in figure 20.

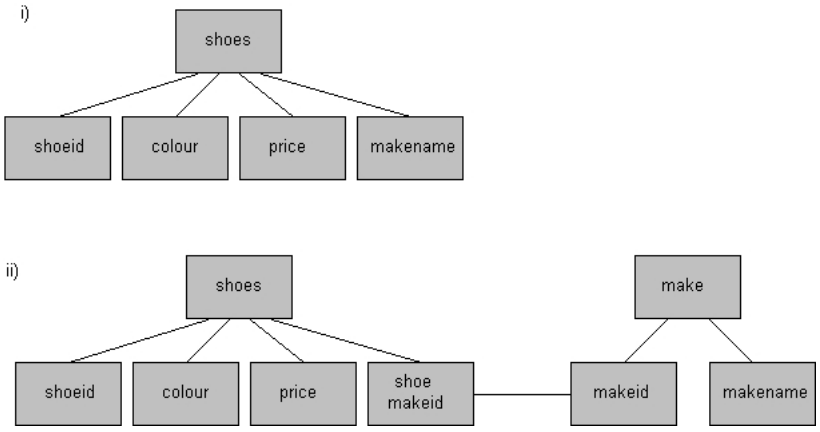

Figure 19: HDM representation of foreign key type link

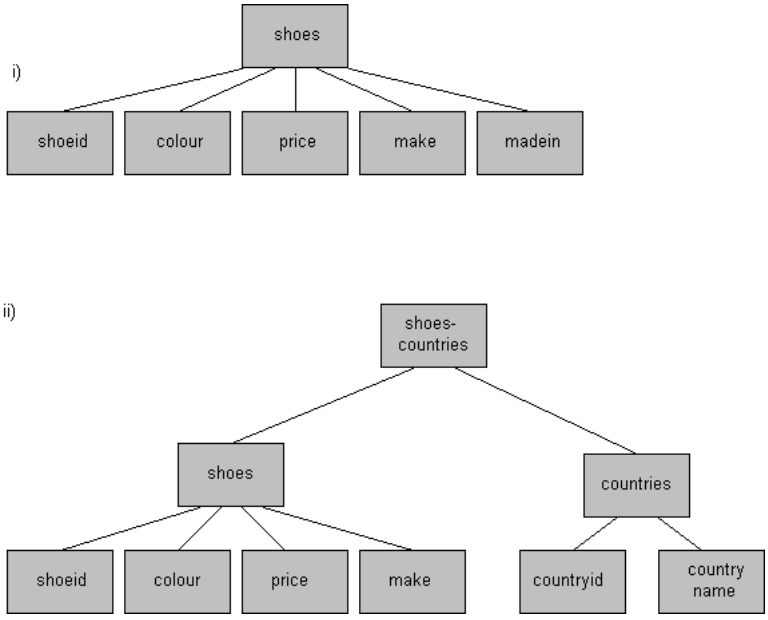

Figure 20: HDM representation of nesting type link

XML Construct		HDM Representation	
Construct	**element (Elem)**		
Class	nodal,set	Node	$\langle\langle \mathsf{xml}{:}e \rangle\rangle$
Scheme	$\langle\langle e \rangle\rangle$		
Construct	**attribute (Att)**	Node	$\langle\langle \mathsf{xml}{:}e{:}a \rangle\rangle$
Class	nodal-linking	Edge	$\langle\langle \_, \mathsf{xml}{:}e, \mathsf{xml}{:}e{:}a \rangle\rangle$
	constraint,list	Links	$\langle\langle \mathsf{xml}{:}e \rangle\rangle$
Scheme	$\langle\langle e, a \rangle\rangle$	Cons	$\mathsf{makeCard}(\langle\langle \_, \mathsf{xml}{:}e, \mathsf{xml}{:}e{:}a \rangle\rangle$,0:1,1:N)
Construct	**nest-list (List)**	Edge	$\langle\langle \_, \mathsf{xml}{:}e, \mathsf{xml}{:}e_s \rangle\rangle, \langle\langle \_, \langle\langle \_, \mathsf{xml}{:}e, \mathsf{xml}{:}e_s \rangle\rangle, \mathsf{order} \rangle\rangle$
Class	linking	Links	$\langle\langle \mathsf{xml}{:}e \rangle\rangle, \langle\langle \mathsf{xml}{:}e_s \rangle\rangle$
	constraint, list	Cons	$\mathsf{makeCard}(\langle\langle \_, \langle\langle \_, \mathsf{xml}{:}e, \mathsf{xml}{:}e_s \rangle\rangle, \mathsf{order} \rangle\rangle$,1:1,0:N)
Scheme	$\langle\langle e, e_s \rangle\rangle$		
Construct	**nest-set (Set)**	Edge	$\langle\langle \_, \mathsf{xml}{:}e, \mathsf{xml}{:}e_s \rangle\rangle$
Class	linking,set	Links	$\langle\langle \mathsf{xml}{:}e \rangle\rangle, \langle\langle \mathsf{xml}{:}e_s \rangle\rangle$
Scheme	$\langle\langle e, e_s \rangle\rangle$		

Table 2: Definition of XML constructs in the HDM

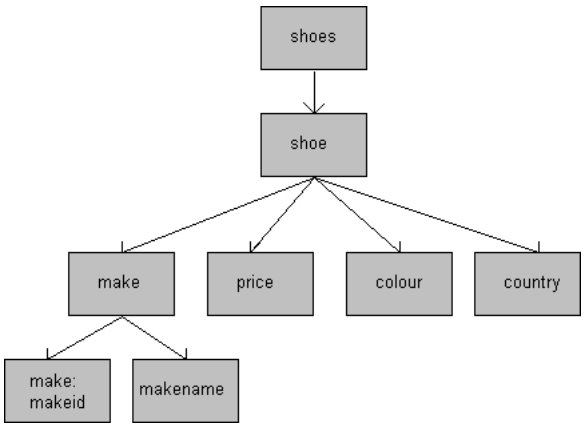

Figure 21: HDM representation of the XML in Figure 5

5.3 XML

The ordered nature of data in an XML document requires some form of list concept to be introduced into the HDM. This is done with the reserved order node. The extent of the order node is the set of natural numbers and represents the ordinality of the members of the edge set. Table 2 shows the definition of XML in the HDM.

XML elements are nodal constructs represented in HDM as a node. XML attributes are nodal-linking constructs. They are represented by a node and an unlabeled edge linking the new node to its parent element. A constraint states that each instance of an attribute is related to at least one instance of the element. XML allows any number of elements to be nested within a given element. This nesting is represented by a set of edges, each of which is an individual linking construct. These may have list or set semantics. For list semantics there is an extra unlabeled edge linking each element of the edge set to the order node.

A special HDM node called pcdata is defined to store plain text that appears between element tags. An element $\langle\langle \mathsf{e} \rangle\rangle$ can then be associated with a piece of plain text by means of an unlabeled edge $\langle\langle \_, \mathsf{e}, \mathsf{pcdata} \rangle\rangle$.

Figure 21 shows an HDM representation of the XML in Figure 5.

Elements and attributes in an XML document are uniquely identified by their position within the document. In representing an XML document in the HDM each node is assigned a unique identifier based on its position. For example the XML elements `shoe` and `make` from Figure 5

can be represented by the following instance of the HDM schemes: $\langle\langle \mathsf{shoes} \rangle\rangle$, $\langle\langle \mathsf{shoe} \rangle\rangle$, $\langle\langle \mathsf{make} \rangle\rangle$ and $\langle\langle \mathsf{makename} \rangle\rangle$, where ss, s1, s2, m1, m2, mid1, mid2, mn1, mn2 are unique identifiers:

$\langle\langle \mathsf{shoes} \rangle\rangle = \{ss\}$
$\langle\langle \mathsf{shoe} \rangle\rangle = \{s1, s2\}$
$\langle\langle \mathsf{make} \rangle\rangle = \{m1, m2\}$
$\langle\langle \mathsf{makename} \rangle\rangle = \{mn1, mn2\}$
$\langle\langle \mathsf{pcdata} \rangle\rangle = \{adidas, nike\}$

The attribute scheme for makeid is shown below:

$\langle\langle \mathsf{make : makeid} \rangle\rangle = \{\langle m1 : mid1\rangle, \langle m2 : mid2\rangle\}$
$\langle\langle \_, \mathsf{make}, \mathsf{make : makeid} \rangle\rangle = \{\langle m1, m1 : mid1\rangle, \langle m2, m2 : mid2\rangle\}$

The nesting relationships are shown below:

$\langle\langle \_, \mathsf{shoes}, \mathsf{shoe} \rangle\rangle = \{\langle ss, s1\rangle, \langle ss, s2\rangle\}$
$\langle\langle \_, \langle\langle \_, \mathsf{shoes}, \mathsf{shoe} \rangle\rangle, \mathsf{order} \rangle\rangle = \{\langle\langle ss, s1\rangle, 1\rangle, \langle\langle ss, s2\rangle, 2\rangle\}$
$\langle\langle \_, \mathsf{shoe}, \mathsf{make} \rangle\rangle = \{\langle s1, m1\rangle, \langle s2, m2\rangle\}$
$\langle\langle \_, \mathsf{hdmn}\_, \mathsf{shoe}, \mathsf{make}, \mathsf{order} \rangle\rangle = \{\langle\langle s1, m1\rangle, 1\rangle, \langle\langle s2, m2\rangle, 1\rangle\}$
$\langle\langle \_, \mathsf{make}, \mathsf{makename} \rangle\rangle = \{\langle m1, mn1\rangle, \langle m2, mn2\rangle\}$
$\langle\langle \_, \langle\langle \_, \mathsf{make}, \mathsf{makename} \rangle\rangle, \mathsf{order} \rangle\rangle = \{\langle\langle m1, mn1\rangle, 1\rangle, \langle\langle m2, mn2\rangle, 1\rangle\}$
$\langle\langle \_, \mathsf{makename}, \mathsf{pcdata} \rangle\rangle = \{\langle mn1, adidas\rangle, \langle mn2, nike\rangle\}$
$\langle\langle \_, \langle\langle \_, \mathsf{makename}, \mathsf{pcdata} \rangle\rangle, \mathsf{order} \rangle\rangle\rangle = \{\langle\langle mn1, adidas\rangle, 1\langle, \langle\langle mn2, nike\rangle, 1\rangle\}$

6 XML Schema

This section presents a definition of the XML Schema model in terms of the HDM. As mentioned in Section 2.3.2 the `complexType`, `element` and `attribute` items in an XML Schema document determine where the elements and attributes in XML instance documents conforming to the schema appear. The `type` attribute in an `element` or `attribute` definition specify what type of data can appear in that item. `key` and `keyref` serve as identity constraints on the data. In my model the Automed constructs closely model those found in an XML Schema document with the **type** construct used to represent both simple and complex types.

XML Schema is a constraint language and defines the type and structure of the data that appears in an XML *instance* document conforming to the schema. An XML Schema document can exist in isolation with no instance documents. This might happen at the beginning of the design process. More commonly there are instance documents.

It is the former that will be the focus of this section. The latter will be dealt with in more detail in Section 9. Transformations on the Automed XML Schema can be used to generate restructured XML Schema documents.

The two ways the document can be used affects the extents of the Automed constructs. If there is an instance document then the information we are moving from one schema to the other will be the data values stored in the instance document. The extents of the XML Schema constructs will then be these data values. If, however, we simply have an XML Schema document with no instance document the information we are interested in are the *types* of the various elements and attributes.

Note that complex types and the complex type nest constructs constrain the structure of the instance document whereas simple types, keys and keyrefs constrain the data in it.

As with the relation model in Section 5.2 we will assume a set of predefined simple types: $ST = \{\langle\langle \mathsf{xs}{:}integer\rangle\rangle, \langle\langle \mathsf{xs}{:}string\rangle\rangle\}$.

1. An XML Schema **type** defines the structure of an XML document conforming to this XML Schema and the data values that can appear in it. Complex types define the structure of the document. Simple types constrain the data values that can appear in it. Unnamed types take their name from their parent element with a suffix of `_typen` if this is the `n`th unnamed type in the XML Schema. Named types can exist on their own and are **nodal** constructs represented in HDM by a node $\langle\langle \mathsf{xs}{:}t\rangle\rangle$ where t is the name of the type.

2. XML Schema **elements** are always associated with a type and are thus **nodal-linking** constructs represented by an HDM node $\langle\langle \mathsf{xs}{:}e{:}t\rangle\rangle$ and an edge linking the element to it's type $\langle\langle \mathsf{xs}{:}t\rangle\rangle$ where $\langle\langle \mathsf{xs}{:}t\rangle\rangle \in ST$ or a complex type defined elsewhere in the schema document. Elements act as place holders for data. A cardinality constraint states that each element can only have one type.

3. An XML Schema **attribute** $\langle\langle \mathsf{xs}{:}pt{:}a\rangle\rangle$ is associated with a parent complex type $\langle\langle \mathsf{xs}{:}pt\rangle\rangle$ and is thus a **nodal-linking** construct. A cardinality constraint states that each instance of an attribute is related to one and only one parent type. Attributes can only be of simple type and the type is represented by a label in the scheme.

4. Nesting of elements of simple type within a complex type is represented by the **complexTypeNest** construct. This is a **linking** construct made up of a parent type $\langle\langle \mathsf{xs}{:}pt\rangle\rangle$ and a child element $\langle\langle \mathsf{xs}{:}e{:}t\rangle\rangle$. In HDM it is represented as an edge from the parent type to the child and a cardinality constraint which states that each instance of $\langle\langle \mathsf{xs}{:}pt\rangle\rangle$ is associated with 0 or more children and each instance of $\langle\langle \mathsf{xs}{:}e{:}t\rangle\rangle$ is associated with exactly one type. The order of elements within a complex type is not recorded in the model at present. If the ordering is of semantic importance an optional label could be added to this construct and the attribute construct.

5. **Keys** are **constraint** constructs on the data in an XML instance document conforming to this schema. There is a name, a parent type $\langle\langle \mathsf{xs}{:}pt\rangle\rangle$ and an attribute $\langle\langle \mathsf{xs}{:}pt{:}a\rangle\rangle$ that is the

key. Each XML attribute $\langle\langle \mathsf{xml}{:}e{:}a \rangle\rangle$ where $\langle\langle \mathsf{xml}{:}e \rangle\rangle$ is of type $\langle\langle \mathsf{xml}{:}pt \rangle\rangle$ must have a unique value and must appear. Keys are defined to always be attributes.

6. **Keyrefs** are data constraints that link an attribute in one complex type to another complex type. At present XML Schema `keys` and `keyrefs` that have multiple XPaths in their `selector` attribute, separated by a |, are not supported. A way of dealing with them is discussed in Section 8.3.

It should be noted that an element with the same name can appear within a number of different complex types. Using the XML to HDM mapping technique described in [29] this would be represented as a number of **complexTypeNest** constructs linking one element node to different type nodes. For example if a person type with key personid and a company type with key companyid both had a name element we could get the name of the person with personid = 1 with the following set based query:

$$\{name | \langle personid, name \rangle \in \langle\langle \_, \mathsf{personType}, \mathsf{name} \rangle\rangle \wedge personid = 1\}$$

The company name for company 7 could be retrieved with the similar query:

$$\{name | \langle companyid, name \rangle \in \langle\langle \_, \mathsf{companyType}, \mathsf{name} \rangle\rangle \wedge companyid = 7\}$$

It is also possible to to uniquely identify each element or attribute in an XML document by using its entire XPath as its identifier. It's extent could then be derived directed from the element or attribute.

In XML, elements can be identified by their position in the document. For instance in [29] each node is assigned a unique identifier generated from it's position in the document. If we have an XML Schema describing the document this is no longer necessary. In particular if all the complex types defined in the schema have key constraints then any element can be identified with the key of it's parent type. For example we could query the schema in Table 5 to get the makename of shoe 1 as follows:

$$\{makename \,|\langle makeid, makename \rangle \in \langle\langle \_, \mathsf{make\_type1}, \mathsf{makename} \rangle\rangle \wedge \\ \langle shoeid, makeid \rangle \in \langle\langle \_, \mathsf{shoeType}, \mathsf{make} \rangle\rangle \wedge shoeid = 1\}$$

The result is adidas.

6.1 Some Examples

This section first presents an example of how an XML Schema can be represented by HDM constructs along with their extents textually and then gives a number of graphical examples. The graphs representing the XML Schemas are the same as those representing the relational schemas presented in Section 5.2 showing how the general graphical approach used in the HDM can represent both XML Schemas and relational schemas in the same graph.

The instances of the schemes representing the XML Schema document in Figure 7 are shown in Table 4. The constraints are not included as they have no extents.

The make type is effectively nested under shoeType and shoeType is nested under the element shoe. As the complex type under the make element is unnamed it has been given the name make_type1. Derived types such as validCountries are discussed more fully in Section 8.

XML Schema **types** and **elements** add an HDM **node**, **attributes** add an HDM **node** and **edge** and **complexTypeNests** add an **edge**.

We will now look at some graphical representations in the HDM of simple XML Schemas. Section 8 will look in detail at some more complicated examples.

6.1.1 One Complex Type to represent the whole schema

All the data elements could be placed in a single complex type. There will be an HDM parent node for the type and all the other elements will be linked to it. If the complex type is to have a

XML Schema Construct		HDM Representation	
Construct	**type**		
Class	nodal	Node	$\langle\langle$xs:$t\rangle\rangle$
Scheme	$\langle\langle t\rangle\rangle$		
Construct	**element (Elem)**	Node	$\langle\langle$xs:$e\rangle\rangle$
Class	nodal-linking	Edge	$\langle\langle \_,$ xs:e:t, xs:$t\rangle\rangle$
	constraint	Links	$\langle\langle$xs:$t\rangle\rangle$
Scheme	$\langle\langle e, t\rangle\rangle$	Cons	makeCard($\langle\langle \_,$ xs:e:t, xs:$t\rangle\rangle, 0:N, 1:1$)
			$x \in \langle\langle$xs:e:$t\rangle\rangle \iff x \in \langle\langle$xs:$t\rangle\rangle$
Construct	**attribute (Att)**	Node	$\langle\langle$xs:pt:$a\rangle\rangle$
Class	nodal-linking	Edge	$\langle\langle \_,$ xs:pt, xs:pt:$a\rangle\rangle$
	constraint	Links	$\langle\langle$xs:$pt\rangle\rangle$
Scheme	$\langle\langle pt, a, t\rangle\rangle$	Cons	makeCard($\langle\langle \_,$ xs:pt, xs:pt:$a\rangle\rangle, 0:N, 1:1$)
			$t \in ST : x \in \langle\langle$xs:$pt$:$a\rangle\rangle \iff x \in \langle\langle$xs:$t\rangle\rangle$
Construct	**complexTypeNest (CTN)**	Edge	$\langle\langle \_,$ xs:pt, xs:e:$t\rangle\rangle$
Class	linking	Links	$\langle\langle$xs:$pt\rangle\rangle$,$\langle\langle$xs:e:$t\rangle\rangle$
Scheme	$\langle\langle pt, e\rangle\rangle$		
Construct	**key (K)**	Links	$\langle\langle$xs:pt:$a_1\rangle\rangle, ..., \langle\langle$xs:$pt$:$a_n\rangle\rangle$
Class	constraint	Cons	$x \in \langle\langle$xs:pt$\rangle\rangle \leftrightarrow x = \langle x_1, ..x_n\rangle$
Scheme	$\langle\langle k, pt, a_1, ..., a_n\rangle\rangle$		$\wedge \langle x, x_1\rangle \in \langle\langle \_,$ xs:pt, xs:pt:$a_1\rangle\rangle \wedge ...$
			$\wedge \langle x, x_n\rangle \in \langle\langle \_,$ xs:pt, xs:pt:$a_n\rangle\rangle$
Construct	**keyRef (KR)**	Links	$\langle\langle$xs:pt:$a_1\rangle\rangle, ..., \langle\langle$xs:$pt$:$a_n\rangle\rangle$
Class	constraint	Cons	$\langle x, x_1\rangle \in \langle\langle \_,$ xs:pt, xs:pt:$a_1\rangle\rangle \wedge ...$
Scheme	$\langle\langle kr, pt, t_f, a_1, ..., a_n\rangle\rangle$		$\wedge \langle x, x_n\rangle \in \langle\langle \_,$ xs:pt, xs:pt:$a_n\rangle\rangle$
			$\rightarrow \langle x_1, ..., x_n\rangle \in \langle\langle$xs:$t_f\rangle\rangle$

Table 3: Definition of XML Schema constructs in the HDM

Types
$\langle\langle$shoeType$\rangle\rangle = \{$integer$\}$
$\langle\langle$make_type1$\rangle\rangle = \{$integer$\}$
$\langle\langle$validCountries$\rangle\rangle = \{$china, usa, germany$\}$
Elements
$\langle\langle$price$\rangle\rangle = \{$integer$\}$
$\langle\langle$colour$\rangle\rangle = \{$string$\}$
$\langle\langle$make$\rangle\rangle = \langle\langle$make_type1$\rangle\rangle = \{$integer$\}$
$\langle\langle$country$\rangle\rangle = \langle\langle$validCountries$\rangle\rangle = \{$china, usa, germany$\}$
$\langle\langle$makename$\rangle\rangle = \{$string$\}$
Attributes
$\langle\langle$shoeType : shoeid$\rangle\rangle = \{$integer$\}$
$\langle\langle \_,$ shoeType, shoeType : shoeid$\rangle\rangle = \{\langle$integer, integer$\rangle\}$
$\langle\langle$make_type1 : makeid$\rangle\rangle = \{$integer$\}$
$\langle\langle \_,$ make_type1, make : makeid$\rangle\rangle = \{\langle$integer, integer$\rangle\}$
complexTypeNests
$\langle\langle \_,$ shoeType, price$\rangle\rangle = \{\langle$integer, integer$\rangle\}$
$\langle\langle \_,$ shoeType, colour$\rangle\rangle = \{\langle$integer, string$\rangle\}$
$\langle\langle \_,$ shoeType, make$\rangle\rangle = \{\langle$integer, make_type1$\rangle\}$
$\langle\langle \_,$ shoeType, country$\rangle\rangle = \{\langle$integer, validCountries$\rangle\}$
$\langle\langle \_,$ make_type1, makename$\rangle\rangle = \{\langle$integer, string$\rangle\}$

Table 4: HDM schema instance of the XML Schema document in Figure 7

Types
$\langle\langle \text{shoesType} \rangle\rangle = \{1, 2\}$
$\langle\langle \text{make\_type1} \rangle\rangle = \{6, 4\}$
$\langle\langle \text{validCountries} \rangle\rangle = \{\text{china}, \text{usa}, \text{germany}\}$
Elements
$\langle\langle \text{price} \rangle\rangle = \{50, 100\}$
$\langle\langle \text{colour} \rangle\rangle = \{\text{blue}, \text{red}\}$
$\langle\langle \text{make} \rangle\rangle = \langle\langle \text{make\_type1} \rangle\rangle = \{6, 4\}$
$\langle\langle \text{country} \rangle\rangle = \langle\langle \text{validCountries} \rangle\rangle = \{\text{china}, \text{usa}, \text{germany}\}$
$\langle\langle \text{makename} \rangle\rangle = \{\text{nike}, \text{adidas}\}$
Attributes
$\langle\langle \text{shoeType} : \text{shoeid} \rangle\rangle = \{1, 2\}$
$\langle\langle \_, \text{shoeType}, \text{shoeType} : \text{shoeid} \rangle\rangle = \{\langle 1, 1\rangle, \langle 2, 2\rangle\}$
$\langle\langle \text{make\_type1} : \text{makeid} \rangle\rangle = \{6, 4\}$
$\langle\langle \_, \text{make\_type1}, \text{make} : \text{makeid} \rangle\rangle = \{\langle 6, 6\rangle, \langle 4, 4\rangle\}$
complexTypeNests
$\langle\langle \_, \text{shoeType}, \text{price} \rangle\rangle = \{\langle 1, 50\rangle, \langle 2, 100\rangle\}$
$\langle\langle \_, \text{shoeType}, \text{colour} \rangle\rangle = \{\langle 1, \text{red}\rangle, \langle 2, \text{blue}\rangle\}$
$\langle\langle \_, \text{shoeType}, \text{make} \rangle\rangle = \{\langle 1, 6\rangle, \langle 2, 4\rangle\}$
$\langle\langle \_, \text{shoeType}, \text{country} \rangle\rangle = \{\langle 1, \text{china}\rangle, \langle 2, \text{china}\rangle, \langle 2, \text{usa}\rangle\}$
$\langle\langle \_, \text{make\_type1}, \text{makename} \rangle\rangle = \{\langle 6, \text{adidas}\rangle, \langle 4, \text{nike}\rangle\}$

Table 5: HDM schema instance of the XML document in Figure 5

key attribute it will be linked to the HDM parent constraint in the same way as the elements are.

```
<complexType name="shoes">
<sequence>
    <element name="colour" type="string" />
    <element name="price" type="string" />
</sequence>
<attribute name="shoeid" type="integer" />
```

The HDM representation is shown in figure 18.

6.1.2 A Complex Type for each part of the schema with no nesting

A complex type represents each group of schema elements as below:

```
<complexType name="Shoe">
<sequence>
    <element name="price" type="string" />
    <element name="colour" type="string" />
</sequence>
<attribute name="shoeid" type="integer" />

<complexType name="make">
<sequence>
    <element name="makename" type="string" />
</sequence>
<attribute name="makeid" type="integer" />
```

6.1.3 Keyrefs

A keyref from the first complex type to the second could be created. A key first needs to be created for the second complex type.

```
<key name = "unique_makeid">
    <selector xpath="./makes"/>
    <field xpath="@makeid"/>
<key>
```

and then a keyref to it created.

```
<keyref name = "fk_shoe_makeid" refer="unique_makeid">
    <selector xpath="./shoes" />
    <field xpath="@makerefid"/>
</keyref>
```

The Shoe complex type will now look like this:

```
<complexType name="Shoe">
<sequence>
    <element name="makerefid" type="integer" />
    <element name="price" />
    <element name="colour" />
</sequence>
<attribute name="shoeid" />
```

The HDM representation is shown in figure 19 part (ii). The keyref is represented by the edge linking the **shoemakeid** node to the **makeid** node.

6.1.4 Nesting

If we wished to nest the country the shoe was made in inside the shoe we could define a new complex type to store the country name and an id as follows:

```
<complexType name="countries">
<sequence>
    <element name="countryname" type="string" />
</sequence>
<attribute name="countryid" type="integer" />
```

We could then nest an element of this type inside shoes as follows. We represent the fact that a shoe can be made in more than one country by making the `maxOccurs` occurrence constraint on `shoescountries` `unbounded`:

```
<complexType name="shoe">
<sequence>
    <element name="price" type="string" />
    <element name="colour" type="string" />
    <element name="make" type="string" />
    <element name="shoescountries" type="countries" maxOccurs = "unbounded" />
</sequence>
<attribute name="shoeid" type="integer" />
```

The HDM representation is shown in figure 20 part (ii). The nesting is represented by the node named after the element and the edges linking it to its own type and its parent type.

6.2 The Primitive Transformations

We now present the primitive transformations on the Automed XML Schema model. These are automatically derivable from the definitions of the XML Schema constructs given in Table 3. Note that the provisos for `addEdge` and `addNode` apply to all the operations below.

6.2.1 Type

If a **type** construct has a key then the extent becomes the type of it's key element or if there is an instance document the extent of it's key. If the key is an attribute the extent will be the extent of the node portion of the attribute.

If there is no key element then the extent of a type is defined to be integer. If there is an instance document the extent is the set of numbers uniquely identifying the position of an element of this type in the document. For example if there were five elements of this type in the instance document the extent of the type would be {1,2,3,4,5} identifying the first to fifth elements. In this way we record the order of elements of this type within the instance document.

- $\mathsf{renameType_{xs}}(t, t') \rightarrow \mathsf{renameNode}(\langle\langle \mathsf{xs}{:}t \rangle\rangle, \langle\langle \mathsf{xs}{:}t' \rangle\rangle)$
- $\mathsf{addType_{xs}}(t, q) \rightarrow \mathsf{addNode}(\langle\langle \mathsf{xs}{:}t \rangle\rangle, q)$
- $\mathsf{delType_{xs}}(t, q) \rightarrow \mathsf{delNode}(\langle\langle \mathsf{xs}{:}t \rangle\rangle, q)$

6.2.2 Element

If an element is of simple type then the extent of an XML Schema **element** is the name of that type. If the element is of complex type the extent is the extent of the complex type. The extent of the edge is this nodal-linking construct is void. A shorthand function $\mathsf{setType}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle, \langle\langle \mathsf{xs}{:}t \rangle\rangle)$ is introduced for the constraint as stated in Table 3.

- $\mathsf{renameElem_{xs}}(e, e') \rightarrow \mathsf{renameNode}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle, \langle\langle \mathsf{xs}{:}e\text{'}{:}t \rangle\rangle)$
- $\mathsf{addElem_{xs}}(e, t, q) \rightarrow \mathsf{addNode}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle, q); \mathsf{addEdge}(\langle\langle \_, \mathsf{xs}{:}e{:}t, \mathsf{xs}{:}t \rangle\rangle, \mathsf{void});$ $\mathsf{addConstraint}(\mathsf{setType}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle, \langle\langle \mathsf{xs}{:}t \rangle\rangle))$
- $\mathsf{delElem_{xs}}(e, t, q) \rightarrow \mathsf{delConstraint}(\mathsf{setType}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle, \langle\langle \mathsf{xs}{:}t \rangle\rangle));$ $\mathsf{delEdge}(\langle\langle \_, \mathsf{xs}{:}e{:}t, \mathsf{xs}{:}t \rangle\rangle, \mathsf{void}); \mathsf{delNode}(\langle\langle \mathsf{xs}{:}e \rangle\rangle, q)$

6.2.3 Attribute

The extent of an XML Schema **attribute** is the extent of the edge linking the attribute node to it's parent, i.e. a tuple whose first element is the type of the key of the parent type and whose second element is the type of the attribute node and a single value representing the value of the attribute. The setType function is used to set the type constraint.

- $\mathsf{renameAtt_{xs}}(a, a') \rightarrow \mathsf{renameNode}(\langle\langle \mathsf{xs}{:}pt{:}a \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a\text{'} \rangle\rangle)$
- $\mathsf{addAtt_{xs}}(pt, a, t, q_{att}, q_{assoc}) \rightarrow \mathsf{addNode}(\langle\langle \mathsf{xs}{:}pt{:}a \rangle\rangle, q_{att});$ $\mathsf{addEdge}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}pt{:}a \rangle\rangle, q_{assoc}); \mathsf{addConstraint}(\mathsf{setType}(\langle\langle \mathsf{xs}{:}pt{:}a \rangle\rangle, \langle\langle \mathsf{xs}{:}t \rangle\rangle))$
- $\mathsf{delAtt_{xs}}(pt, a, t, q_{att}, q_{assoc}) \rightarrow \mathsf{delConstraint}(\mathsf{setType}(\langle\langle \mathsf{xs}{:}pt{:}a \rangle\rangle, \langle\langle \mathsf{xs}{:}t \rangle\rangle));$ $\mathsf{delEdge}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}pt{:}a \rangle\rangle, q_{assoc}); \mathsf{delNode}(\langle\langle \mathsf{xs}{:}pt{:}a \rangle\rangle, q_{att})$

6.2.4 ComplexTypeNest

The extent of a complex nest type is a tuple whose first element is the type of the key of the parent type and whose second element is the type of the child node.

- Complex type nest constructs are unnamed and so cannot be renamed.
- $\mathsf{addCTN_{xs}}(pt, e, q) \rightarrow \mathsf{addEdge}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e \rangle\rangle, q)$
- $\mathsf{delCTN_{xs}}(pt, e, q) \rightarrow \mathsf{delEdge}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e \rangle\rangle, q)$

6.2.5 Key

Keys are constraints and so have no extent. As above a shorthand function is introduced for the constraint: $\mathsf{setKey}\ (k, \langle\langle \mathsf{xs}{:}pt \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a_1 \rangle\rangle \cdots \langle\langle \mathsf{xs}{:}pt{:}a_\mathsf{n} \rangle\rangle)$

- $\mathsf{renameKey}_{\mathsf{xs}}(k, k') \rightarrow \mathsf{renameConstraint}(k, k')$
- $\mathsf{addKey}_{\mathsf{xs}}(k) \rightarrow \mathsf{addConstraint}(\mathsf{setKey}(k, \langle\langle \mathsf{xs}{:}pt \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a_1 \rangle\rangle \cdots \langle\langle \mathsf{xs}{:}pt{:}a_\mathsf{n} \rangle\rangle))$
- $\mathsf{delKey}_{\mathsf{xs}}(k) \rightarrow \mathsf{delConstraint}(\mathsf{setKey}(k, \langle\langle \mathsf{xs}{:}pt \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a_1 \rangle\rangle \cdots \langle\langle \mathsf{xs}{:}pt{:}a_\mathsf{n} \rangle\rangle))$

6.2.6 KeyRef

Keyrefs are constraints and so have no extent. The shorthand function for this constraint is: $\mathsf{setKeyRef}(kr, \langle\langle \mathsf{xs}{:}pt \rangle\rangle, \langle\langle \mathsf{xs}{:}t_f \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a_1 \rangle\rangle \cdots \langle\langle \mathsf{xs}{:}pt{:}a_\mathsf{n} \rangle\rangle)$

- $\mathsf{renameKeyRef}_{\mathsf{xs}}(kr, kr') \rightarrow \mathsf{renameConstraint}(kr, kr')$
- $\mathsf{addKeyRef}_{\mathsf{xs}}(kr) \rightarrow \mathsf{addConstraint}(\mathsf{setKeyRef}(kr, \langle\langle \mathsf{xs}{:}pt \rangle\rangle, \langle\langle \mathsf{xs}{:}t_f \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a_1 \rangle\rangle \cdots \langle\langle \mathsf{xs}{:}pt{:}a_\mathsf{n} \rangle\rangle)$
- $\mathsf{delKeyRef}_{\mathsf{xs}}(kr) \rightarrow \mathsf{delConstraint}(\mathsf{setKeyRef}((kr, \langle\langle \mathsf{xs}{:}pt \rangle\rangle, \langle\langle \mathsf{xs}{:}t_f \rangle\rangle, \langle\langle \mathsf{xs}{:}pt{:}a_1 \rangle\rangle \cdots \langle\langle \mathsf{xs}{:}pt{:}a_\mathsf{n} \rangle\rangle)$

6.3 Composite Transformations

Composite transformations can be constructed by using a number of primitive transformations together to make a new, more complex, transformation. This process can be used to create very powerful transformations. An example of a simple composite transformation on the XML Schema model is given below. Another example is given in Section 9.7.

6.3.1 Transforming an Element to an Attribute

This transformation is necessary if an element is made into a key or may be done for other reasons.

If we have an element $\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle$ with parent type $\langle\langle \mathsf{xs}{:}p\mathsf{t} \rangle\rangle$ the composite transformation will be as follows:

$\mathsf{transElement}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle)$

The operations that make up this composite transformation are as follows: (The extent of the attribute is the union of the extent of it's node q_{att} and edge components q_{assoc}.)

$\mathsf{addAtt}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}pt{:}e \rangle\rangle, \{\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle \cup \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t \rangle\rangle\})$
$\mathsf{delCTN}_{\mathsf{xs}}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t \rangle\rangle, q_{assoc})$
$\mathsf{delElement}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}e{:}t \rangle\rangle, q_{att})$

If we call the schema in Table 4 an instance I_{xs} of schema S we could create I'_{xs} by transforming the colour element within shoeType into an attribute.

$\mathsf{addAtt}_{\mathsf{xs}}(\langle\langle \mathsf{shoeType} : \mathsf{colour} \rangle\rangle, \{\{string\}, \{\langle integer, string \rangle\}\})$
$\mathsf{delCTN}_{\mathsf{xs}}(\langle\langle \_, \mathsf{shoeType}, \mathsf{colour} \rangle\rangle, \{\langle integer, string \rangle\})$
$\mathsf{delElement}_{\mathsf{xs}}(\langle\langle \mathsf{colour} \rangle\rangle, \{string\})$

Assume I_{xml} is an instance of S where S has been generated from the XML document in Figure 5. Doing the same transformation as above to give us I'_{xml} could be written as follows. Note that the only difference between them is the value of the extents.

$\mathsf{addAtt}_{\mathsf{xs}}(\langle\langle \mathsf{shoeType} : \mathsf{price} \rangle\rangle, \{\{50, 100\}, \{\langle 1, 50 \rangle, \langle 2, 100 \rangle\}\})$
$\mathsf{delCTN}_{\mathsf{xs}}(\langle\langle \_, \mathsf{shoeType}, \mathsf{price} \rangle\rangle, \{\langle 1, 50 \rangle, \langle 2, 100 \rangle\})$

$\mathsf{delElement}_{\mathsf{xs}}(\langle\langle \mathsf{price} \rangle\rangle, \{50, 100\})$

We can see that in both cases $Ext_{S,I} = Ext_{S,I'}$.

7 Inter-model Transformations in HDM

The HDM representation of higher level modeling languages in general and in particular of XML and the relational model is such that it is possible to unambiguously represent constructs of both models within the same HDM schema. This allows inter-model transformations which can be used for automatic, reversible inter-model *translation* of data and queries. This is made possible because the extents of add and delete transformations in one modeling language can be expressed in terms of the extents of constructs in some other modeling language. Additionally new *inter-model edges* which do not belong to either higher-level language can be defined. This allows associations to be built between constructs in different modeling languages and allows navigation between them. This is obviously of particular use when translating between XML and the relational model.

Template transformations can be used to specify a generic way of transforming constructs in one modeling language to another. This enables transformations on specific constructs to be automatically generated. See [28] for examples of inter-model transformations.

7.1 Transforming between XML Schema and the Relational model

Section 6.2 showed how an XML Schema can be represented in the HDM and Section 5.2 shows how the relational model can be represented in the HDM. In this section we will assume that we are transforming an XML instance document constrained by an XML Schema. The extents of these transformations will be the data items in the XML document. This will make them consistent with the extents of constructs in the relational model. If we a dealing with an XML Schema document in isolation then the XML Schema constructs do not have data items as extents but type names. These are meaningless values in the relational model so all the extents in these transformations are void.

The following section will show how each of the XML Schema constructs described in Section 6 map to constructs in the relational model. The rules and primitive operations are specified at the level of the HDM. As all transformations in Automed are reversible the transformations from the relational model to the XML Schema model are automatically derivable.

makePrimaryKey and makeForeignKey functions are introduced as a shorthand for the constraint queries for primary and foreign keys introduced in table 1 and a copyType function copies the type constraint from one construct to another. The copyKey function copies a key constraint.

1. Each HDM node representing an XML Schema **type** xs:t generates a **relation** of the same name.

   addNode($\langle\langle$rel:$t\rangle\rangle$, ($\langle\langle$xs:$t\rangle\rangle$)

   If the type does not have a key then a relational **attribute** called id is created to be the new relation's key. The extent of the attribute is that of $\langle\langle$xs:$t\rangle\rangle$.

   addNode($\langle\langle$rel:t:$id\rangle\rangle$, $\langle\langle$xs:t:$\rangle\rangle$)
   addEdge($\langle\langle$_, rel:t, rel:t:$id\rangle\rangle$, $\{\langle x, x\rangle | x \in \langle\langle$xs:$t$:$\rangle\rangle\}$)
   addConstraint(makeCard($\langle\langle$_, rel:t, rel:t:$id\rangle\rangle$, $1:1, 1:1$))
   addConstraint(setType($\langle\langle$rel:t:$id\rangle\rangle$, integer))

2. Each HDM node representing an XML Schema **attribute** $\langle\langle$xs:pt:$a\rangle\rangle$ generates a relational **attribute** of the same name:

   addNode($\langle\langle$rel:pt:$a\rangle\rangle$, $\langle\langle$xs:pt:$a\rangle\rangle$)
   addEdge($\langle\langle$_, rel:pt, rel:pt:$a\rangle\rangle$, $\langle\langle$_, xs:pt, xs:pt:$a\rangle\rangle$)
   addConstraint(copyCard($\langle\langle$_, xs:pt, xs:pt:$a\rangle\rangle$, $\langle\langle$_, rel:pt, rel:pt:$a\rangle\rangle$))
   addConstraint(copyType($\langle\langle$xs:pt:$a\rangle\rangle$, $\langle\langle$rel:pt:$a\rangle\rangle$))

3. Each HDM node representing an XML Schema **element** of simple type and the **complexTypeNest** construct linking it to it's parent, generate a relational **attribute** of the same name as the element:

   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}pt{:}e\rangle\rangle, \langle\langle \mathsf{xs}{:}e{:}t\rangle\rangle)$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel}{:}pt, \mathsf{rel}{:}pt{:}e\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle)$
   $\mathsf{addConstraint}(\mathsf{copyCard}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle, \langle\langle \_, \mathsf{rel}{:}pt, \mathsf{rel}{:}pt{:}e\rangle\rangle))$
   $\mathsf{addConstraint}(\mathsf{copyType}(\langle\langle \mathsf{xs}{:}e{:}t\rangle\rangle, \langle\langle \mathsf{rel}{:}pt{:}e\rangle\rangle))$

   Note that during the reverse operation relational key attributes will generate XML Schema **attributes** while non-key attributes will generate XML Schema **elements**.

4. Each HDM node representing an XML Schema **element** of complex type and the **complexTypeNest** construct linking it to it's parent will generate a **relation** with the same name as the element, linking the parent relation of the element to the relation representing it's type. The new relation will have two **attributes** representing the keys of the two linked relations. In this way nests are turned into link tables. The types of the two columns are the types of the keys of the complex types they are linking, $\mathsf{k}_1, \mathsf{k}_2$.

   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}e\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle))$
   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}e{:}t\rangle\rangle, \{\langle x\rangle | \langle x, y\rangle \in \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}t\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e\rangle\rangle)$
   $\mathsf{addConstraint}(\mathsf{makeCard}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}t\rangle\rangle, 1:1, 1:1))$
   $\mathsf{addConstraint}(\mathsf{copyType}(\langle\langle \mathsf{xs}{:}t{:}k_1\rangle\rangle, \langle\langle \mathsf{rel}{:}e{:}t\rangle\rangle))$
   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}e{:}pt\rangle\rangle, \{\langle y\rangle | \langle x, y\rangle \in \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}pt\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle)$
   $\mathsf{addConstraint}(\mathsf{makeCard}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}pt\rangle\rangle, 1:1, 1:1))$
   $\mathsf{addConstraint}(\mathsf{copyType}(\langle\langle \mathsf{xs}{:}pt{:}k_2\rangle\rangle, \langle\langle \mathsf{rel}{:}pt{:}e\rangle\rangle))$

   A primary key constraint consisting of both of the new columns is added.

   $\mathsf{addConstraint}(\mathsf{makePrimaryKey}(\langle\langle \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}pt, \mathsf{rel}{:}e{:}t\rangle\rangle))$

   A foreign key constraint is also added to each of the new columns.

   $\mathsf{addConstraint}(\mathsf{makeForeignKey}(\langle\langle \mathsf{rel}{:}e, \mathsf{rel}{:}t, \mathsf{rel}{:}e{:}t\rangle\rangle))$
   $\mathsf{addConstraint}(\mathsf{makeForeignKey}(\langle\langle \mathsf{rel}{:}e, \mathsf{rel}{:}pt, \mathsf{rel}{:}e{:}pt\rangle\rangle))$

5. An XML Schema **element** with an maximum occurrence constraint greater than 1 will generate a **relation** linking the parent to instances of the element. The relation will have two attributes, one to hold the key of the parent with type k and one for the element.

   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}e\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle))$
   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}e{:}pt\rangle\rangle, \{\langle x\rangle | \langle x, y\rangle \in \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}pt\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle)$
   $\mathsf{addConstraint}(\mathsf{makeCard}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}t\rangle\rangle, 1:1, 1:1))$
   $\mathsf{addConstraint}(\mathsf{copyType}(\langle\langle \mathsf{xs}{:}t{:}k\rangle\rangle, \langle\langle \mathsf{rel}{:}e{:}t\rangle\rangle))$
   $\mathsf{addNode}(\langle\langle \mathsf{rel}{:}e{:}e\rangle\rangle, \{\langle y\rangle | \langle x, y\rangle \in \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}e\rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e{:}t\rangle\rangle)$
   $\mathsf{addConstraint}(\mathsf{makeCard}(\langle\langle \_, \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}pt\rangle\rangle, 1:1, 1:1))$
   $\mathsf{addConstraint}(\mathsf{copyType}(\langle\langle \mathsf{xs}{:}e{:}t\rangle\rangle, \langle\langle \mathsf{rel}{:}pt{:}e\rangle\rangle))$

   The primary key again consists of both of the new columns.

   $\mathsf{addConstraint}(\mathsf{makePrimaryKey}(\langle\langle \mathsf{rel}{:}e, \mathsf{rel}{:}e{:}pt, \mathsf{rel}{:}e{:}t\rangle\rangle))$

A foreign key constraint is also added to the column referencing the elements parent relation.

$\mathsf{addConstraint}(\mathsf{makeForeignKey}(\langle\langle \mathsf{rel}{:}e, \mathsf{rel}{:}pt, \mathsf{rel}{:}e{:}pt \rangle\rangle))$

6. The XML Schema **key** constraint is transformed into a relational **primary key** constraint. The name of the XML Schema **key** is not included in the relational model.

   $\mathsf{addConstraint}(\mathsf{copyKey}(\langle\langle k, \mathsf{xs}{:}pt, \mathsf{xs}{:}pt{:}a_1 \cdots \mathsf{xs}{:}pt{:}a_n \rangle\rangle,$
   $\langle\langle \mathsf{rel}{:}pt, \mathsf{rel}{:}pt{:}a_1 \cdots \mathsf{rel}{:}pt{:}a_n \rangle\rangle))$

7. XML Schema **keyref** constraints are transformed into a relational **foreign key** constraint. As above the name of the XML Schema **keyref** his not included in the relational model.

   $\mathsf{addConstraint}(\mathsf{copyKey}(\langle\langle kr, \mathsf{xs}{:}pt, \mathsf{xs}{:}t_{ref}, \mathsf{xs}{:}pt{:}a_1 \cdots \mathsf{xs}{:}pt{:}a_n \rangle\rangle,$
   $\langle\langle \mathsf{rel}{:}pt, \mathsf{rel}{:}t_{ref}, \mathsf{rel}{:}pt{:}a_1 \cdots \mathsf{rel}{:}pt{:}a_n \rangle\rangle))$

   As mentioned previously my XML Schema model does not at present support XML Schema `keys` or `keyrefs` that have a choice component in the XPath for their `selector` attribute. Possible solutions are discussed in Section 8.3.

8. Once all the relational model constructs have been created the XML Schema model constructs are semantically redundant and so can be systematically removed.

7.1.1 An example

Using the above transformations we can transform the schema in table 5 into a relational schema. Not all the transformations are shown and the cardinality and type constraints are left out for brevity.

1. Types to Relations:

   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{shoeType} \rangle\rangle, \{1, 2\})$
   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{make\_type1} \rangle\rangle, \{6, 4\})$

2. Attributes to Attributes:

   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{shoeType} : \mathsf{shoeid} \rangle\rangle, \{1, 2\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{shoeType}, \mathsf{rel} : \mathsf{shoeType} : \mathsf{shoeid} \rangle\rangle, \{\langle 1, 1\rangle, \langle 2, 2\rangle\})$
   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{make\_type1} : \mathsf{makeid} \rangle\rangle, \{6, 4\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{make\_type1}, \mathsf{rel} : \mathsf{make\_type1} : \mathsf{makeid} \rangle\rangle, \{\langle 6, 6\rangle, \langle 4, 4\rangle\})$

3. Elements with simple type to Attributes

   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{shoeType} : \mathsf{price} \rangle\rangle, \{50, 100\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{shoeType}, \mathsf{rel} : \mathsf{shoeType} : \mathsf{price} \rangle\rangle, \{\langle 1, 50\rangle, \langle 2, 100\rangle\})$
   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{shoeType} : \mathsf{colour} \rangle\rangle, \{red, blue\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{shoeType}, \mathsf{rel} : \mathsf{shoeType} : \mathsf{colour} \rangle\rangle, \{\langle 1, red\rangle, \langle 2, blue\rangle\})$
   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{make\_type1} : \mathsf{makename} \rangle\rangle, \{nike, adidas\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{make\_type1}, \mathsf{rel} : \mathsf{make\_type1} : \mathsf{makename} \rangle\rangle, \{\langle 6, nike\rangle, \langle 4, adidas\rangle\})$

4. Elements with complex type i.e nested elements, to link Relations

   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{make} \rangle\rangle, \{\langle 1, 6\rangle, \langle 2, 4\rangle\})$
   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{make} : \mathsf{shoeType} \rangle\rangle, \{1, 2\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{make}, \mathsf{rel} : \mathsf{make} : \mathsf{shoeType} \rangle\rangle, \{\{\langle 1, 1\rangle, \langle 2, 2\rangle\})$
   $\mathsf{addNode}(\langle\langle \mathsf{rel} : \mathsf{make} : \mathsf{makeType} \rangle\rangle, \{6, 4\})$
   $\mathsf{addEdge}(\langle\langle \_, \mathsf{rel} : \mathsf{make}, \mathsf{rel} : \mathsf{make} : \mathsf{makeType} \rangle\rangle, \{\{\langle 1, 6\rangle, \langle 2, 4\rangle\})$

5. Key constraints to Primary Keys

   addConstraint(copyKey(⟨⟨uniqueShoeid, xs : shoeType, xs : shoeType : shoeid⟩⟩,
   ⟨⟨rel : shoeType, rel : shoeType : shoeid⟩⟩))
   addConstraint(copyKey(⟨⟨uniqueMakeid, xs : makeType, xs : makeType : makeid⟩⟩,
   ⟨⟨rel : makeType, rel : makeType : makeid⟩⟩))

6. New constraints from the link table

   addConstraint(makePrimaryKey(⟨⟨rel : make, rel : make : shoeType, rel : make : makeType⟩⟩))
   addConstraint(makeForeignKey(⟨⟨rel : make, rel : shoeType, rel : make : shoeType⟩⟩))
   addConstraint(makeForeignKey(⟨⟨rel : make, rel : makeType, rel : make : makeType⟩⟩))

The process by which the Automed relational schema can be transformed into SQL DDL statements is described in Section 9.9.

8 Case Studies

There are many more complicated schemas that can be represented in XML Schema. The following are some of the more interesting ones. These are beyond the scope of what my GUI can do and so are discussed in theory here.

1. A derived complex type that extends or restricts another one.
2. Types with the same structure but a different name. E.g an invoice address and a home address. How do we identify which is which.
3. XML Schemas with multi value keys all of which are optional but at least one must be there.
4. The X.521 model used by LDAP. This is recursive and allows self-referencing.

8.1 Derived Complex Types

The XML Schema standard allows new types based on existing ones to be created. These are called derived types. An example of an extension on a complex type is given in the following XML Schema fragment:

```
<xsd:complexType name = "address">
    <xsd:sequence>
        <xsd:element name = "number" type = "xsd:integer" />
        <xsd:element name = "street" type = "xsd:string" />
    </xsd:sequence>
    <xsd:attribute name = "addrid" type = "xsd:integer" />
</xsd:complexType>

<xsd:complexType name = "internationalAddress">
    <xsd:complexContent>
        <xsd:extension base="address">
            <xsd:sequence>
                <xsd:element name = "country" type = "xsd:string" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Extending a complex type adds a new type based on the original type but with extra elements and/or attributes. The complex type internationalAddress contains all the elements from it's base type address as well as the new country element. If we represent this graphically in HDM we notice that the two parent nodes share three common child nodes. An HDM schema fragment representing this is shown below:

$\langle\langle \mathsf{address} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{internationalAddress} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{number} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{street} \rangle\rangle = \{\mathsf{string}\}$
$\langle\langle \mathsf{addrid} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{country} \rangle\rangle = \{\mathsf{string}\}$
$\langle\langle \_, \mathsf{address}, \mathsf{number} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{integer} \rangle\}$
$\langle\langle \_, \mathsf{address}, \mathsf{street} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{string} \rangle\}$
$\langle\langle \_, \mathsf{address}, \mathsf{addrid} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{integer} \rangle\}$
$\langle\langle \_, \mathsf{internationalAddress}, \mathsf{number} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{integer} \rangle\}$
$\langle\langle \_, \mathsf{internationalAddress}, \mathsf{street} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{string} \rangle\}$
$\langle\langle \_, \mathsf{internationalAddress}, \mathsf{addrid} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{integer} \rangle\}$
$\langle\langle \_, \mathsf{internationalAddress}, \mathsf{country} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{string} \rangle\}$

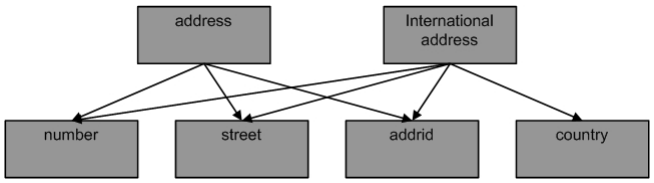

Figure 22: HDM of the address schema

Using the transformations defined in the previous section this schema could be transformed to a relational schema consisting of two distinct relations. Any other relation wishing to reference both a normal address and an international address would need two foreign keys, one for each relation.

Looking at the HDM representation in Figure 22, an obvious improvement to this would be to get rid of one of the parent nodes. This would also allow us to remove all the edges from the child nodes to that parent. This assumes that the difference between an address and and international address does not have significant semantic importance as now our ability to tell a normal address from an international one is lost. Dealing with the case where the difference is significant is dealt with in the next section.

We would then have the simpler Automed schema:

$\langle\langle \mathsf{address} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{number} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{street} \rangle\rangle = \{\mathsf{string}\}$
$\langle\langle \mathsf{addrid} \rangle\rangle = \{\mathsf{integer}\}$
$\langle\langle \mathsf{country} \rangle\rangle = \{\mathsf{string}\}$
$\langle\langle \_, \mathsf{address}, \mathsf{number} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{integer} \rangle\}$
$\langle\langle \_, \mathsf{address}, \mathsf{street} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{string} \rangle\}$
$\langle\langle \_, \mathsf{address}, \mathsf{addrid} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{integer} \rangle\}$
$\langle\langle \_, \mathsf{address}, \mathsf{country} \rangle\rangle = \{\langle \mathsf{integer}, \mathsf{string} \rangle\}$

The HDM transformations are as follows:

1. Union the extents of address and internationalAddress
2. Make the country element a child of address:
3. Delete all the edges linking the child nodes to internationalAddress.

In the relational model this would now transform into a single relation:

```
         Table "address"
  Column  |       Type        | Modifiers
----------+-------------------+-----------
 addrid   | integer           | not null
 number   | integer           |
 street   | character varying |
 country  | character varying |
Indexes:
    "addr_pk" primary key, btree (addrid)
```

8.2 Complex Types with the same structure but different names

In the following XML Schema fragment invoiceAddress and homeAddress both have the same structure.

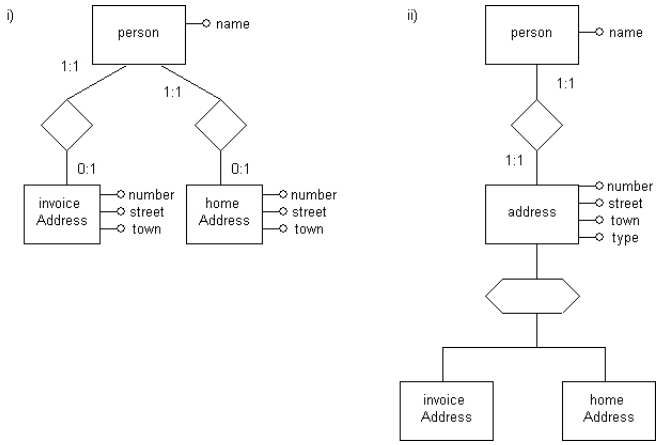

Figure 23: ER representation of address schema

```
<xsd:complexType name = "invoiceAddress">
    <xsd:sequence>
        <xsd:element name = "number" type = "xsd:integer" />
        <xsd:element name = "street" type = "xsd:string" />
        <xsd:element name = "town" type = "xsd:integer" />
    </xsd:sequence>
    <xsd:attribute name = "addressId" type = "xsd:integer" />
</xsd:complexType>

<xsd:complexType name = "homeAddress">
    <xsd:sequence>
        <xsd:element name = "number" type = "xsd:integer" />
        <xsd:element name = "street" type = "xsd:string" />
        <xsd:element name = "town" type = "xsd:integer" />
    </xsd:sequence>
    <xsd:attribute name = "addressId" type = "xsd:integer" />
</xsd:complexType>
```

The canonical representation would create two XML Schema **types** each with all the elements in it, and would result in two relations once transformed. This might be a good way of transforming the schema and would make identifying each separate address easy. However, it might be better to have a single complex type for all the data. This could make accessing the correct address harder. We need a way of identifying an address as either a home address or an invoice address. In ER the new address table becomes a generalisation shown in Figure 23 and the child tables (home address and invoice address) have a type attribute identifying one from the other. We need addressType as part of the key because merging the tables might result in duplicate address ids.

In XML Schema an address type could be defined with an element addressType to identify different types of addresses. This element could have an enumeration as it's type as shown below.

The new XML Schema fragment is shown below.

```
<xsd:simpleType name = "addressType_type">
    <xsd:restriction base = "xsd:string">
        <xsd:enumeration value = "local" />
        <xsd:enumeration value = "international" />
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name = "address">
    <xsd:sequence>
        <xsd:element name = "number" type = "xsd:integer" />
```

```
            <xsd:element name = "street" type = "xsd:string" />
            <xsd:element name = "town" type = "xsd:string" />
        </xsd:sequence>
        <xsd:attribute name = "addrId" type = "xsd:integer" />
        <xsd:attribute name = "addrType" type = "addressType_type" />
    </xsd:complexType>
```

The transformations on the XML Schema model would be as follows:

1. Add a new addrType attribute to be used as part of the key in the primary type.
2. Remove the other type and all its elements and edges.

This can now be represented as a single relational table. The new addrType element would become part of the key of the address table allowing each address to be uniquely identified.

```
          Table "address"
  Column  |       Type        | Modifiers
----------+-------------------+-----------
 addrId   | integer           | not null
 number   | integer           |
 street   | character varying |
 addrType | integer           | not null
Indexes:
    "addr_pk" primary key, btree (addrid, addrtype)
```

8.3 Keys and Keyrefs with a Choice in the XPath of the Selector

The `selector` element in both `key` and `keyref` definitions can have multiple components in the XPath definition defined in it's `xpath` attribute. In this way it is possible to define identity constraints on a number of elements in a single definition. This has the proviso that all the elements must contain the element or attribute referenced in the `xpath` attribute of the `field` element. Figure 24 provides an example of this. The key definition unique_vehicleId specifies that both cars and planes must have an id attribute and that that attribute across all planes and cars. The keyref definition registrationId that references this key specifies that the id in the vehicleType type must contain a value from EITHER a plane element OR a car element because the key it's referencing has a choice in its `selector`.

Transforming the above XML Schema into the relational model is not straightforward because plane and car are elements of different complex types. If they were of the same type a transformation to the relational model would create a single relation with a primary key on id attribute and all the constraints would be maintained, however in this case they are not of the same type.

A canonical transformation to the relational model using the method presented in this report would create a carType relation with a primary key on id, similarly a planeType relation would be generated also with a primary key on id. A registrationType relation with two foreign keys, one for planeType and one for carType would be needed. This transformation would lose the either or constraint that the choice on the XML Schema `key` enforced. An id in the planeType relation could match an id in the carType relation.

Looking at the HDM representation of the schema in Figure 25 some sort of generalisation of the id element suggests itself. This is indeed a way to maintain the correct constraints. A generalisation table called vehicleId could be created. It would have a single attribute id that would act as the primary key of the relation. The extent of the attribute would be a union of the id attributes from the other two relations. The registrationType relation would then only need one foreign key referencing the id in vehicleId. To maintain the link between the planeType and carType relations and the vehicleId relation, foreign keys referencing vehicleId could be added to the id attributes of planeType and carType.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:complexType name = "planeType">
<xsd:sequence>
<xsd:element name = "wingspan"></xsd:element>
</xsd:sequence>
<xsd:attribute name = "id" type = "xsd:integer"></xsd:attribute>
</xsd:complexType>
<xsd:complexType name = "carType">
<xsd:sequence>
<xsd:element name = "cc"></xsd:element>
</xsd:sequence>
<xsd:attribute name = "id" type = "xsd:integer"></xsd:attribute>
</xsd:complexType>
<xsd:complexType name = "registrationType">
<xsd:sequence>
<xsd:element name = "county"></xsd:element>
</xsd:sequence>
<xsd:attribute name = "id" type = "xsd:integer"></xsd:attribute>
</xsd:complexType>
<xsd:complexType name = "databaseType">
<xsd:sequence>
<xsd:element name = "plane" type = "planeType" minOccurs="0"  maxOccurs="unbounded"></xsd:element>
<xsd:element name = "car" type = "carType" minOccurs="0"  maxOccurs="unbounded"></xsd:element>
<xsd:element name = "registration" type = "registrationType" minOccurs="0"  maxOccurs="unbounded"></xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:element name = "vehicleDatabase" type = "databaseType">
    <xsd:key name = "unique_vehicleId">
    <xsd:selector xpath="./plane | ./car"/>
    <xsd:field xpath="@id"/>
    </xsd:key>

    <xsd:keyref name = "registrationId" refer = "unique_vehicleId">
    <xsd:selector xpath="./registration"/>
    <xsd:field xpath="@id"/>
    </xsd:keyref>
</xsd:element>
</xsd:schema>
```

Figure 24: XML Schema fragment showing a choice in the selector XPath

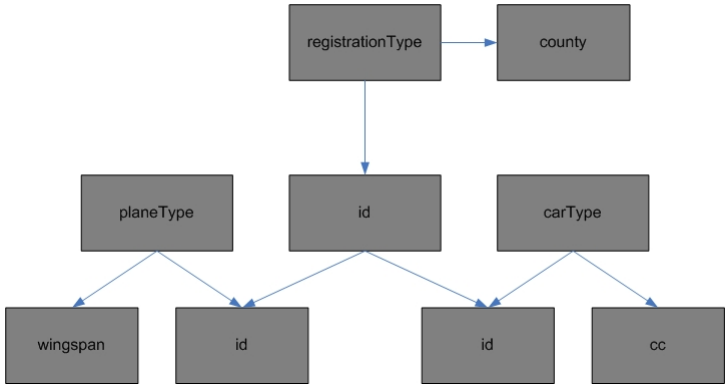

Figure 25: HDM of the vehicle schema

```
   Table "vehicleid"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
Indexes: vehicleid_pk primary key btree (id)

         Table "planetype"
  Column  |       Type        | Modifiers
----------+-------------------+-----------
 id       | integer           | not null
 wingspan | character varying |
Indexes: plane_pk primary key btree (id)
Foreign Key constraints: plane_id_fk FOREIGN KEY (id) REFERENCES vehicleid(id)

         Table "cartype"
 Column |       Type        | Modifiers
--------+-------------------+-----------
 id     | integer           | not null
 cc     | character varying |
Indexes: car_pk primary key btree (id)
Foreign Key constraints: car_id_fk FOREIGN KEY (id) REFERENCES vehicleid(id)

    Table "registrationtype"
 Column |       Type        | Modifiers
--------+-------------------+-----------
 id     | integer           | not null
 county | character varying |
Indexes: reg_pk primary key btree (id)
Foreign Key constraints: reg_id_fk FOREIGN KEY (id) REFERENCES vehicleid(id)
```

Figure 26: SQL tables representing an XML Schema with a choice in one it's key selectors

The relations resulting from this transformation are shown in Figure 8.3. It can be seen that a choice in selector XPath of a key generates a generalisation table of the two relations referenced in the key. The transformations to form a generalisation from two relations are well-defined within the HDM and integrating them with my model would not be hard. This once again shows the advantage of using an abstract model that can model many different constructs. We can take advantage of transformations that have been defined but have nothing to do with transforming from XML Schema to the relational model.

8.4 X.521 and LDAP

X.521[32] defines object classes that can be used to access X.500 based directory services. X.500 is a powerful naming standard but until recently implementations have been cumbersome and not suitable for general use. LDAP, the Lightweight Directory Access Protocol [13], based on X.521 object classes was designed at the University of Michigan as a simple way of accessing X.500 based services, specifically email addresses. It has a number of interesting features with respect to data integrity. This section is presented more as a way of showing how X.521 and LDAP can be represented in both XML Schema and the relational model. A full coverage of the transformations between the schemas is beyond the scope of this report but some possible solutions are suggested.

Figure 28 shows an XML Schema representation of a portion of the X521 standard. Figure 27 shows a diagram of the relationships between the elements. While not complete it is enough to show all the important points.

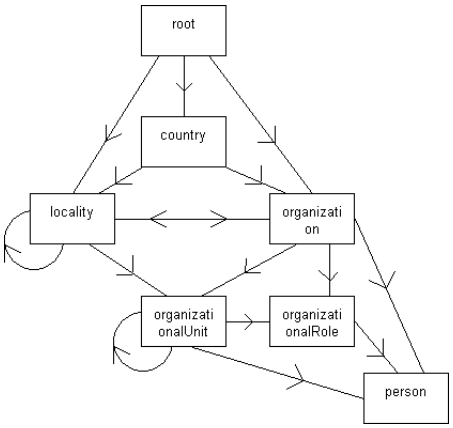

Figure 27: A portion of the X521 naming standard

8.4.1 Self referencing and Recursion

We can see from figure 27 that the locality node can contain itself. In XML Schema, relationships between elements can be created with nesting or with key/keyref combinations. In the X.521 model we can model the self-referencing nature of the locality node by nesting locality node inside itself. In the relational model this is done with self-referencing foreign keys.

As well as self referencing nodes groups of nodes together can create recursive definitions. For example an organisation can contain a locality which in turn can contain an organisation which could contain another locality etc. In XML Schema these are represented as group of complex types that can reference each other while in the relational model they become groups of relations with foreign keys that reference each other.

Using the method described in this report the X.521 XML Schema could be represented with a relation for each node of the X.521 model with relational attributes for each XML Schema attribute and an additional id attribute to act as a foreign key. The nesting of different types of node could be represented as link tables.

8.4.2 LDAP

The representation described above will work for a description of entire organisational hierarchy where the root may include a number of countries followed by some localities that may or may not have sub-elements. The structure of an LDAP address is also hierarchical is nature. Each entry in an LDAP database has a unique **Distinguished Name** or **DN**. This name is built up from the unique **Relative Distinguished Names (RDN)** at each level of the naming hierarchy above the name we are interested in. An example of an LDAP address is shown below:

```
dn:  cn = andrew, o = ic, c = uk
```

Starting from the root node in Figure 27 and reading the address backwards it must have one of a country, locality or organisation not more. Here we have a country with an RDN of uk. The next element must then be one of a country or an organization. We chose an organisation with an RDN of ic. The final element must be a locality, an organisational unit, an organisation role or a person. We chose a person. The LDAP standard includes an inetOrgPerson object that includes the cn attribute. In this way the DN is built up.

The constraint that each level of the hierarchy should be one of a number of options is not enforced by the XML Schema in Figure 28. It would allow a country to be followed by an unlimited number of organisations for example. XML Schema does, however, provide a mechanism to achieve this 'one and only one of the following' with the `choice` tag. Elements within a complex type can be declared within `choice` tags rather than `sequence` tags as we have done up till now. The

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:complexType name = "root">
    <xs:sequence>
    <xs:element name = "country" type = "country" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisation" type = "organisation" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "locality" type = "locality" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:sequence>
</xs:complexType>
<xs:complexType name = "country">
    <xs:sequence>
    <xs:element name = "organisation" type = "organisation" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "locality" type = "locality" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:sequence>
    <xs:attribute name = "c" type = "xs:string" />
    <xs:attribute name = "d" type = "xs:string" />
</xs:complexType>
<xs:complexType name = "locality">
    <xs:sequence>
    <xs:element name = "organisation" type = "organisation" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisationalUnit" type = "organisationalUnit" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "locality" type = "locality" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:sequence>
    <xs:attribute name = "l" type = "xs:string" />
</xs:complexType>
<xs:complexType name = "organization">
    <xs:sequence>
    <xs:element name = "person" type = "person" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisationalRole" type = "organisationalRole" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisationalUnit" type = "organisationalUnit" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "locality" type = "locality" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:sequence>
    <xs:attribute name = "o" type = "xs:string" />
</xs:complexType>
<xs:complexType name = "organisationalUnit">
    <xs:sequence>
    <xs:element name = "person" type = "person" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisationalRole" type = "organisationalRole" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisationalUnit" type = "organisationalUnit" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:sequence>
    <xs:attribute name = "ou" type = "xs:string" />
</xs:complexType>
<xs:complexType name = "organisationalRole">
    <xs:sequence>
    <xs:element name = "person" type = "person" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:sequence>
    <xs:attribute name = "or" type = "xs:string" />
</xs:complexType>
<xs:complexType name = "person">
    <xs:attribute name = "p" type = "xs:string" />
</xs:complexType>

<xs:element name = "root" type = "root" />

</xs:schema>
```

Figure 28: XML Schema fragment showing a portion of the X521 standard

```
<xs:complexType name = "root">
    <xs:choice>
    <xs:element name = "country" type = "country" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "organisation" type = "organisation" minOccurs = "0" maxOccurs = "unbounded" />
    <xs:element name = "locality" type = "locality" minOccurs = "0" maxOccurs = "unbounded" />
    </xs:choice>
</xs:complexType>
```

Figure 29: XML Schema fragment showing choice tags

`choice` tags mean that one and only one of the elements within the `choice` tags must appear in an element of this complex type in an instance document of this schema. Figure 29 shows a fragment of the previous schema with `choice` tags instead of `sequence` tags. This could be used to enforce the constraint that the address must start with one of country, organisation or locality.

There is no equivalent of the `choice` tag constraint in the relational model. The constraint that the root must reference one of the three elements defined within it could be constructed as a generalisation in the same way as described in the previous section. Assume that relations exist for all the nodes in Figure 27 and each relation has an id attribute and a nextChild attribute acting as a reference to the next element in the LDAP address. A rootChildren relation could be defined with an id attribute whose extent was a union of the country, organisation and locality id attributes. A foreign key could then be added to the nextElement attribute of the root relation referencing rootChildren. This would make sure that the next element after the root was one of a country, organsation or locality but would not let us know which. Solving that problem requires relational queries that are beyond the scope of this report.

Translating XML Schema's ability to define **choice** constraints presents a great challenge and is a very interesting topic for further study.

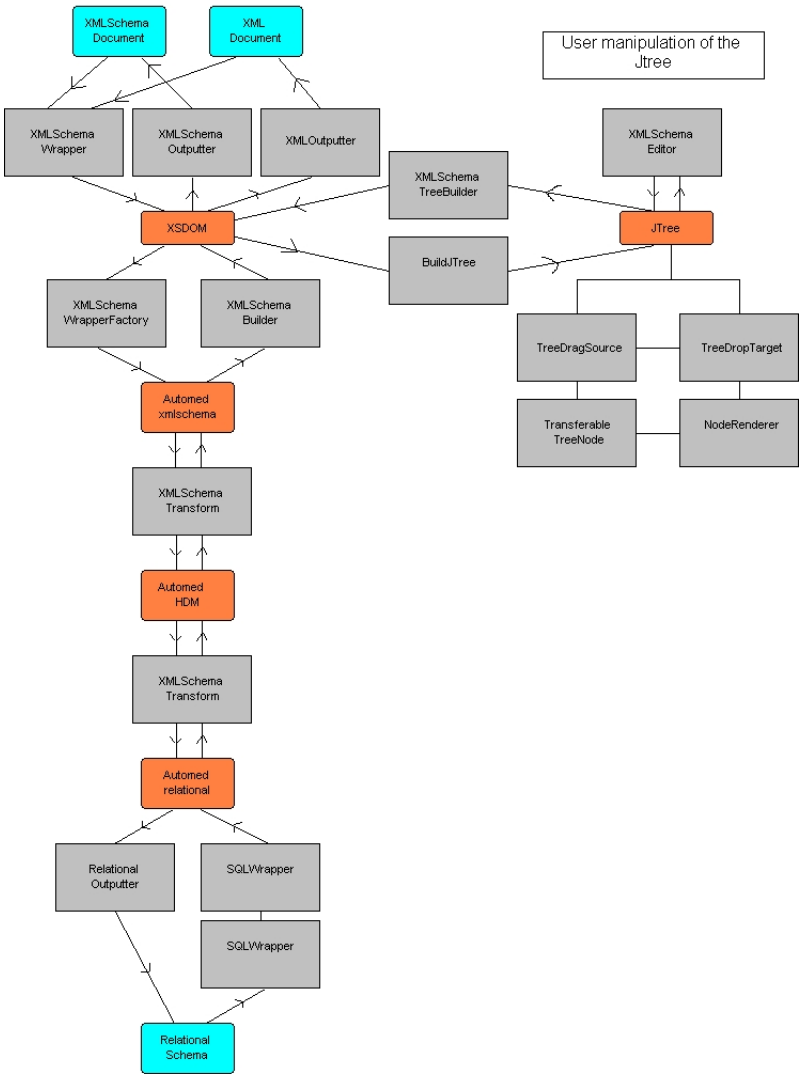

Figure 30: The class hierarchy

9 Implementation

There are two tasks that the implementation performs: the transformation of relational schemas to XML schemas and visa versa and allowing the user a degree of control over the end products of these transformations. The first of these is achieved within the Automed framework[2] and is described in section 9.1, the second via a GUI described in section 9.6. The full class hierarchy is shown in figure 30.
The different classes have the following functions:

XMLSchemaTransform Methods in this class transform constructs in the XML Schema model to HDM and back again. Other methods transform constructs from the HDM model to the relational model. Finally there are methods to transform XML Schema constructs to other XML Schema constructs.

XMLSchemaOutputter Methods in this class take XML Schema constructs and create an XML Schema document based on those constructs.

RelationalOutputter Methods in this class take SQL constructs and create a relational database based on them. They also transfer any data if the transformation was from an XML document rather than simply an XML Schema document.

BuildJTree Methods in this class convert the XSDOM representation of the schema document to DefaultMutableTreeNode objects that can be displayed as a JTree. The other classes in the **tree** branch of the API allow user manipulation of the JTree.

XMLSchemaBuilder Methods in this class convert the Automed XML Schema model into an XSDOM representation.

XMLSchemaTreeBuilder Methods in this class convert the JTree to XSDOM.

An XML document is transformed into a relational database as follows:

1. `XMLSchemaWrapper` converts the document to XSDOM
2. Optionally transforms can be done via the visual tool.

   `BuildJTree` take the XSDOM and creates a JTree representation

   The JTree can be manipulated

   `XMLSchemaTreeBuilder` converts the JTree back to XSDOM
3. `XMLSchemaWrapperFactory` creates Automed `SchemaObjects` from the XSDOM
4. `XMLSchemaTransforms` transforms the XML Schema model to HDM and then to the SQL model.
5. `RelationalOutputter` creates the SQL tables and inserts any data if we were transforming an XML document.

A relational database schema is transformed into an XML instance document as follows:

1. `SQLWrapper` and `SQLWrapperFactory` convert the database in the Automed SQL model
2. `XMLSchemaTranforms` transforms the SQL model to HDM and then XML Schema
3. Optionally transforms can be done via the visual tool.

   `BuildJTree` take the XSDOM and creates a JTree representation

   The JTree can be manipulated
4. `XMLSchemaOutputter` creates the XML Schema document from the Automed XML Schema constructs and `XMLOutputter` generates an XML document conforming to the schema containing the data from the database.

9.1 Automed

The Automed framework supports the transformation and integration of schemas that are expressed in different data modeling languages. To avoid the complications of using a chosen high level data model as the common data model Automed uses the HDM discussed in section 5. The Automed framework is implemented as a number of Java APIs [2].

A schema or data from virtually any source can be incorporated into the Automed framework by writing a *wrapper* for that data source. This is a java class that directly interrogates the data source and then creates the necessary Automed `Schema` and `SchemaObjects` to represent the data source in Automed. The design and implementation of a wrapper is discussed in detail in section9.3.

In Automed a schema is transformed by incrementally applying a set of primitive transformations to it. This set of transformations forms the transformation pathway from the source schema

that we started with to a target schema. A key advantage of the Automed approach is that the transformations and thus the transformation pathways are reversible. This is achieved by embedding the extent of the construct created or removed in each transformation. The extent defines how data associated with the new or removed construct can be derived from other existing constructs in the original schema. The reversibility of transformations enables automatic translation of queries posed on any schema on a given pathway into appropriate queries on a particular target schema also on that pathway. This means that a schema created by an Automed wrapper can be transformed any number of times and a query on any schema on that transformation pathway will retrieve data from the original wrapped data source.

9.1.1 IQL

IQL [31] is a functional intermediate query language that can be used within the Automed framework to query any schema along a given transformation pathway. Queries posed on a target schema are translated to a source schema that has been generated directly from a data source. Code within the wrapper for that data source then interrogates it directly to get a result. All extent queries in Automed are expressed in IQL.

IQL supports a `bag` collection type and supports a number of predefined operators. See [31] for a detailed description of it's capabilities.

9.2 Representing a Relational Schema in Automed

The `SQLWrapper` and `SQLWrapperFactory`[26] classes are used to create an Automed representation of a relational schema. The `SQLWrapperFactory` creates constructs that closely match those of the underlying database. For example the relational model relations and attributes are modeled as `table` and `column` constructs. I have use the SQL model as my representation of relational data in Automed. Two of the optional features I have used are the type of a column and whether or not it is null-able.

9.3 The Automed XML Schema Wrapper

This is a new wrapper that allows XML documents constrained by an XML Schema or the XML Schema document itself to be represented in Automed. A distinguishing feature of this wrapper is that the extents of the schema objects can either be data items if an XML document is represented or types if only the XML Schema document is represented. The *structure* of the XML document is constrained by the complex type and element definitions in the schema document and the *data values* are constrained by the simple types assigned to the elements or attributes. See section 2.3.2 for a more detailed discussion of XML Schema.

The XML Schema wrapper is made up of two classes:

- The `XMLSchemaWrapperFactory` class that creates the Automed schema objects. This class also contains methods to create the Automed XML Schema model constructs. See Section 6 for a description of the constructs.
- The `XMLSchemaWrapper` class that establishes a connection to the XML Schema document and creates an internal XSDOM representation of it.

See [26] for a detailed discussion about using the Automed API to create a wrapper.

An outline of the algorithm to creates a representation of the XML Schema document in Automed is given below. Details are given in the following sections:

1. Methods in the `XMLSchemaWrapper` class establish a connection to the specified XML Schema document.
2. When a connection has been established methods in `XMLSchemaWrapper` use the XML Schema API[25] to extract the different components from the schema document. These

are used to create an internal XSDOM representation. See Section 9.5 for details about XSDOM and the translation process.

3. The internal XSDOM representation is used by the `XMLSchemaWrapperFactory` to populate the Automed Schema.

9.4 The HDM transformations

All the transformations done by the `XMLSchemaTransform` class are via the HDM model. To allow the XML Schema type information to be represented the HDM model is extended to add a type label to the node construct. It was decided to adopt this approach rather than go directly from XML Schema to the SQL model to make the wrapper more general. Code has been written to convert an XML Schema document into the basic HDM components of nodes, edges and constraints. It should now be possible to easily transform these structures into the other data models that Automed supports.

The transformations needed to go from HDM to the XML Schema model are described in the following section. The transformations are done in two passes. The first transforms all the HDM parent **nodes** to XML Schema **type** constructs. In the second pass all the HDM child **nodes** are transformed into XML Schema **elements** and the HDM **edges** linking the parent and child **nodes** are transformed into XML Schema **complexTypeNest** constructs. As with the transformation from XML Schema to the relational model the transformations are reversible so they are only described in the XML Schema to HDM direction. The transformations from the SQL model to and from HDM are not described in detail as they did form a core aspect of the project.

9.4.1 Transforming between XML Schema and HDM

Two optional labels have been added to the HDM node construct. The first for the type of the node and the second for any boundedness constraints there might be.

- An HDM node is a **parent** if it appears as the second element in an HDM edge scheme. For example if there was an HDM edge scheme $\langle\langle \_, shoe, colour \rangle\rangle$ the $\langle\langle shoe \rangle\rangle$ node would be a parent node.
- An element is a **child** if it appears as the third element in an HDM edge scheme. In the above example $\langle\langle colour \rangle\rangle$ would be a child node.
- The set of **children-p** of a particular parent (p) are all those child nodes that have the same parent. For example in figure 18 the children are $\langle\langle shoeid \rangle\rangle$, $\langle\langle colour \rangle\rangle$ and $\langle\langle price \rangle\rangle$.
- An HDM node p is a **grandparent** if one of the children in **children-p** is a parent node itself.

The transformations to go from HDM to XML Schema are as follows:

1. Each HDM node representing an XML Schema **element** generates an attribute of the same name For each HDM parent node

   $\mathsf{addType_{xs}}(\langle\langle \mathsf{xs}{:}nodeName \rangle\rangle, \langle\langle \mathsf{nodeName} \rangle\rangle)$

2. For each HDM child node represented by the HDM **node** and **edge** constructs:

   $\langle\langle nodeName, nodeType \rangle\rangle$

   $\langle\langle \_, parentNode, nodeName \rangle\rangle$

   an XML Schema **element** construct and a **complexTypeNest** construct linking the new XML Schema child element to it's XML Schema parent type

   $\mathsf{addElem_{xs}}(\langle\langle \mathsf{xs}{:}nodeName{:}nodeType \rangle\rangle)$

   $\mathsf{addCTN_{xs}}(\langle\langle \_, \mathsf{xs}{:}parentNode, \mathsf{xs}{:}nodeName{:}nodeType \rangle\rangle)$

are created.

If there is no type associated with the HDM node then an XML Schema element of type anyType is created.

3. If there is a key or keyref constraint associated with an HDM edge

   $\langle\langle \_, parentNode, childNode \rangle\rangle$

   then an XML Schema **attribute** construct with scheme

   $\mathsf{addAtt}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}parentNode{:}childNode \rangle\rangle)$

   is created.

   This transformation is only valid if the HDM child node has a simple type associated with it.

4. If there is a key constraint associated with an HDM edge an XML Schema **key** construct with scheme

   $\mathsf{addKey}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}keyName{:}parentNode{:}keyField \rangle\rangle)$

   is created.

   This transformation is only valid if the XML Schema **attribute** $\langle\langle \mathsf{xs}{:}parentNode{:}keyField \rangle\rangle$ exists.

5. If an HDM edge links to two leaf nodes then an XML Schema **keyref** construct

   $\mathsf{addKeyRef}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}keyrefName{:}parentType{:}refType : keyField \rangle\rangle)$

   is created.

   This transformation is only valid if XML Schema **attribute** constructs representing both the HDM child nodes exist and there is an XML Schema **key** construct for the referenced table.

The transformations are done in 3 passes:

1. All the HDM parent nodes are transformed in to XML Schema **type** constructs.
2. The HDM child nodes are transformed to XML Schema **element** and **complexTypeNest** constructs.
3. Finally the XML Schema **key** and **keyref** constructs are added.

The transformations from XML Schema to HDM have an extra pass. In the final pass the XML Schema key constraints and nesting generate named HDM edges: _pk for a key constraint, _fk for a keyref and link for a nesting.

9.4.2 Transforming between HDM and SQL

HDM to SQL is also done in three passes similar to those described above. First the tables are added then the columns are added to the tables and finally any primary or foreign key constraints are added to the tables. When transforming from SQL to HDM the key constraints from SQL are not maintained. This is so the user can choose their own keys and nesting for the final XML Schema in the GUI. Another reason this was done is that at the moment key constraints can only be added in the GUI, not removed. The key constraints may be maintain in the future as the functionality of the GUI improves.

9.5 XSDOM

As mentioned in Section 2.3.2 the XML Schema standard is very complex. To try and get away from some of this complexity and to allow me to focus on those aspects of the XML Schema standard that are most relevant to data storage and transformation I wrote the XSDOM classes. They are based on JDOM standard [16].

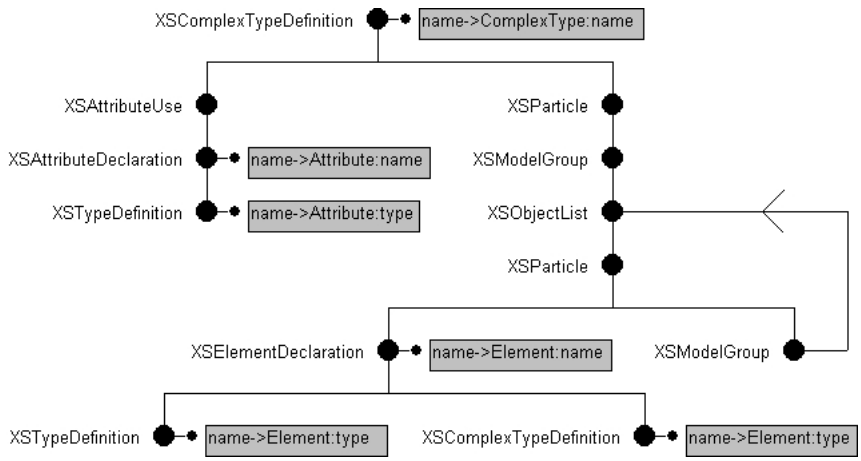

Figure 31: How XML Schema API components of a complex type map to XSDOM

9.5.1 Element

This class represents the XML Schema element construct. These can either be within a `ComplexType` or the root element which can exist on its own. Each element has a name and a type. It may also have a keyref to a key attribute in another complexType.

9.5.2 Attribute

This class represents the XML Schema attribute construct. Each attribute has a name and a type. As with an element an attribute may also have a keyref to a key attribute in another complexType.

9.5.3 ComplexType

This class represents an XML Schema complexType construct. It has 0 or more child elements and 0 or more child attributes. One of the attributes may be a key.

9.5.4 Document

This class represents a whole XML Schema document. It contains the root element of the document and a list of all the complexTypes in the document.

9.5.5 Converting from XML Schema API to XSDOM

The XML Schema is loaded into memory in two different ways depending on whether a XML document is being transformed or an XML Schema document. If we are transforming an XML document it is first parsed. The post schema validation info set (PSVI) is then used to get the schema information. If no such information is found an exception is raised and the program exits. If an XML Schema document is being transformed then DOM [15] methods are used to load the schema information.

The XML Schema API [25] provides methods to extract all the different components of an XML Schema document. Not all of the the XML Schema model is supported at present. An `XSModelGroup` can be a sequence, choice or all only sequences are supported. Attribute groups, simple type declarations, wildcards, and referenced elements are not supported. Only two of the built-in simple types will be supported, namely: `integer` and `string`. The components that have been captured and stored in the XSDOM model are those that were deemed most useful to the task at hand.

Figure 31 shows how the XML Schema API components of a complex type map to XSDOM objects. The methods to do the transformation are in the `XMLSchemaWrapper` class. Each gray box

contains a mapping from an XML Schema API field to the appropriate field in the XSDOM object. The components of XSDOM were designed to closely resemble the structure of the XML Schema objects they represent so the mapping from the XML Schema API to XSDOM is reasonably straightforward.

1. The XML Schema document model is stored in an `XSModel` object. The name of the root object becomes the name of the root element in the XSDOM document representing this XML Schema.

2. The complex type declarations all generate XSDOM `complexType` objects that are added to the XSDOM document. This is first pass of the mapping.

3. After the complex type object has been created any `XSAttributeUses` in the XML Schema complex type declaration are added as child `attribute` objects to the XSDOM complex type. The first `XSParticle` node in the right hand branch of the tree in figure 31 represents an XML Schema sequence. Any `XSElementDeclarations` in the sequence are added as child `elements` of the complex type parent. If the `XSObjectList` contains another `XSModelGroup` all the `XSElementDeclarations` in it are added to the XSDOM complex type too. The addition of the attributes and elements to the complex type objects constitutes a second pass in the mapping.

4. A final pass is done to add the XML Schema **key** and XML Schema **keyref** identity constraints. Keys are added as a `key` field in the complex type object they refer to. If an element or an attribute acts as a keyref then a `keyref` field in the element or attribute object is added.

9.5.6 XSDOM to Automed XML Schema

Methods in the `XMLSchemaWrapper` class use the XSDOM representation of the XML Schema to populate the Automed schema. The use of the XSDOM classes as an intermediary between the XML Schema API and Automed makes the mapping straightforward. It is also done in three passes.

1. First all the complex type objects generate XML Schema **type** constructs in Automed.

2. Second all the attribute and element children of the complex type generate **attribute** and **element** and **complexTypeNest** pair constructs.

3. Finally any complex type objects with a non-null key field generate a **key** construct and any elements or attributes with a non-null keyref field generate a **keyref** construct.

9.5.7 Automed XML Schema to XSDOM

This is exact reverse of the operation described in the previous section and is needed when a relational schema has been transformed in Automed to an Automed XML Schema. The translation is in the `XMLSchemaBuilder` class. The XSDOM representation can then be displayed with the BuildJTree class.

To allow maximum flexibility the relational schema is displayed in its most simple form with no key relations. This allows the user to choose the way the XML Schema will look.

9.5.8 XSDOM to JTree

The `BuildJTree` class contains methods to convert the XSDOM representation of the schema document to DefaultMutableTreeNode objects that can be displayed as a JTree. Using XSDOM as an intermediary instead of converting XML Schema API objects into DefaultMutableTreeNodes (DMTN) abstracts away much of the complexity of the XML Schema API. The DMTNs are instances of TransferableTreeNodes that have a name, keyref, a type and an icon.

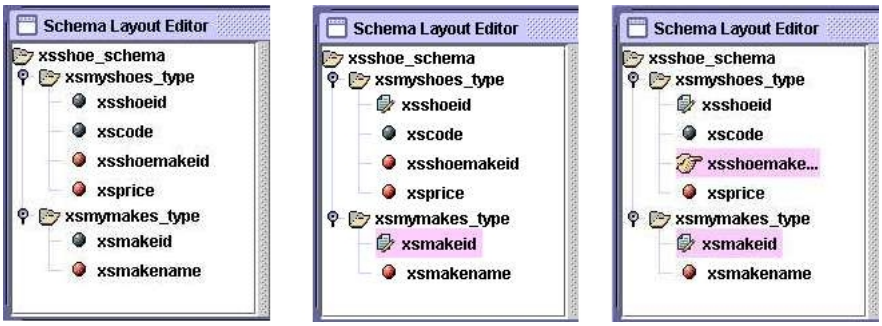

Figure 32: JTrees showing simple XML Schema transformations

The `createJTree` method creates a JTree representation of the XML Schema document. Figure 32 is a series of screen shots showing how a simple schema can have keys and keyrefs added to it. The left hand tree has no keys. The middle tree has had key constraints added to xsshoeid and xsmakeid. The right hand tree has had a keyref constraint added linking xsshoemakeid to the key xsmymakes_type: xsmakeid

9.6 GUI

The `XMLSchemaEditor` class is the main class for the GUI.

9.6.1 Operations in the GUI

There are a number of things one can do in the GUI. The formal transformations that are performed when these operations are done are described in the next section.

Firstly one can manipulate the schema. All the schema transformations are done on Automed XML Schema constructs. Hopefully this can be expanded to work with HDM in the future:

Drag and Drop One complex type can be nested inside another one by dragging it over one of the elements in the new parent complex type.

Clicking the **Transform** menu item will bring up the following options:

Make key Selecting a child element on the JTree and then clicking Transform/Make Key will create an XML Schema **key** construct and will transform the child element into an XML Schema **attribute** if it is an XML Schema **element** as all keys are defined to be attributes. The child icon will change.

Link This will create an XML Schema **keyref** construct. The element selected first will be made a key attribute so a **key** construct will be created for it.

XMLSchema to HDM Selecting this menu item will transform the JTree to HDM.

HDM to Relational Once an HDM representation of the schema has been created this menu item will transform the HDM to the Automed relational model.

Clicking the **Output** menu item will bring up the following options. They are described in detail in Section 9.8:

Output Relational This will create a relational schema from the JTree.

Output XML Schema This will create an XML Schema document and an XML instance document from the JTree. This can be done any number of times so lots of different schema documents can be created.

9.7 Transformations in the GUI

A number of the operations in the GUI transform the XML Schema objects represented in the JTree. The basic operations can be used together to define composite transformations.

9.7.1 Drag and Drop

The drag and drop operation can be used to add or remove nesting. For the drag and drop to result in a consistent model only a type object can be dragged and it must be dragged over an element whose extent is the same as that of the type being dragged. At the moment drags and drops are not checked for validity. This could be done in the future.

If for example a type object $\langle\langle \mathsf{xs}{:}dragged \rangle\rangle$ with parent type $\langle\langle \mathsf{xs}{:}pd \rangle\rangle$ is dragged onto a target element $\langle\langle \mathsf{xs}{:}target{:}t \rangle\rangle$ with parent $\langle\langle \mathsf{xs}{:}pt \rangle\rangle$ creating a new element $\langle\langle \mathsf{xs}{:}new{:}dragged \rangle\rangle$ we will have the following composite transformation. If an element of type $\langle\langle \mathsf{xs}{:}dragged \rangle\rangle$ already existed inside $\langle\langle \mathsf{xs}{:}pd \rangle\rangle$ it and the edge linking it to its parent are removed with two contract transformations.

$\mathsf{addNest}_{\mathsf{xs}}(\langle\langle \mathsf{dragged}, \mathsf{pt}, \mathsf{target} \rangle\rangle)$

The operations that make up this composite transformation are as follows:

$\mathsf{addElem}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}new{:}dragged \rangle\rangle, \langle\langle \mathsf{xs}{:}target{:}t \rangle\rangle)$
$\mathsf{addCTN}_{\mathsf{xs}}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}new{:}dragged \rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}target{:}t \rangle\rangle)$
$\mathsf{delCTN}_{\mathsf{xs}}(\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}target{:}t \rangle\rangle, \langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}new{:}dragged \rangle\rangle)$
$\mathsf{delElem}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}target{:}t \rangle\rangle, \langle\langle \mathsf{xs}{:}new{:}dragged \rangle\rangle)$
$\mathsf{contractCTN}_{\mathsf{xs}}(\langle\langle \_, \mathsf{xs}{:}pd, \mathsf{xs}{:}oldElement{:}dragged \rangle\rangle)$
$\mathsf{contractElem}_{\mathsf{xs}}(\langle\langle \mathsf{xs}{:}oldElement{:}dragged \rangle\rangle)$

9.7.2 Making an attribute a key

The transformation to make an attribute a with parent p a key called k is as follows:

$\mathsf{addKey}_{\mathsf{xs}}(\langle\langle \mathsf{k}, \mathsf{p}, \mathsf{a} \rangle\rangle)$

9.7.3 Adding a keyref

The transformation to make an attribute a with parent p a keyref called kr referring to a key called tf is as follows:

$\mathsf{addKeyRef}_{\mathsf{xs}}(\langle\langle \mathsf{kr}, \mathsf{p}, \mathsf{a}, \mathsf{tf} \rangle\rangle)$

9.8 Materialising the Automed XML Schema

Methods in XMLSchemaTreeBuilder take the JTree nodes and convert them into an XSDOM representation. Methods in the XMLSchemaOutputter class materialise the XSDOM.

Different classes of Automed XML Schema elements are defined below:

- An XML Schema element is a **parent** if it appears as the second element in a complexType-Nest scheme.
- A XML Schema element is a **child** if it appears as the third element in a complexTypeNest scheme.
- The set of **children-p** of a particular parent (p) are all those XML Schema child elements that have the same parent.

Rules to transform the Automed XML Schema model to an XML Schema document:

1. For an Automed schema called *AM_Schema* an XML Schema `element` named *AM_Schema* is created:

```
<element name="AM_Schema">
    <complexType>
    <sequence>
        <!-- Rest of the schema -->
    </sequence>
    </complexType>
</element>
```

2. For each parent element in an Automed XML Schema **complexTypeNest** construct an XML Schema `complexType` with name *nodeName_type* is created:

```
<complexType name = "nodeName_type">
    <sequence>
        <!-- child elements -->
    </sequence>
</complexType>
```

3. For each Automed XML Schema child **element** linked to it's parent by the a **complexTypeNest**:

   $\langle\langle$xs:*elementName*:*elementType*$\rangle\rangle$
   $\langle\langle$_, xs:*parentType*, xs:*elementName*:*elementType*$\rangle\rangle$

   an XMLSchema `element` is created inside the element sequence of the XML Schema parent complexType:

```
<complexType name = "parentType">
    <sequence>
    <element name = "elementName" type = "elementType" />
    <!-- Other child elements -->
    </sequence>
</complexType>
```

4. For each Automed XML Schema **attribute** construct

   $\langle\langle$xs:*parentType*:*attributeName*:*attributeType*$\rangle\rangle$

   an XML Schema `attribute` is created within the `parentType`:

```
<complexType name = "parentType">
    <!-- child element sequence -->
    <attribute name = "attributeName" type = "attributeType" />
</complexType>
```

5. For each Automed XML Schema **key** construct

   $\langle\langle$keyName, xs:*parentType*:*keyField*$\rangle\rangle$

   an XML Schema `key` element is created as shown below:

```
<key name = "keyName">
    <selector xpath="<!-- XPath to the parent of keyField -->" />
    <field xpath="@keyField" />
<key>
```

6. For each Automed XML Schema **keyref** construct

   $\langle\langle$keyrefName, xs:*parentType*:*refType*:*keyrefField*$\rangle\rangle\rangle$

   an XML Schema `keyref` element is created as shown below:

```
<keyref name = "keyrefName" refers = "refType">
    <selector xpath="<!-- XPath to the parent of keyrefField -->" />
    <field xpath="@keyrefField" />
<key>
```

9.9 Creating SQL tables

The `RelationalOutputter` class converts constructs from the SQL model into relational tables. The algorithm is as follows:

1. Each **table** construct and any **column** constructs associated with that table

   ⟨⟨sql:*tableName*⟩⟩
   ⟨⟨sql:*tableName*:*columnName1*:*columnType1*⟩⟩
   .
   .
   ⟨⟨sql:*tableName*:*columnNamen*:*columnTypen*⟩⟩

   generate an SQL table:

   ```
   CREATE TABLE tableName
       (columnName1 columnType1, ..., columnNamen columnTypen);
   ```

2. Once all the tables have been created any key constraints are added. Each **primaryKey** construct

   ⟨⟨sql:*tableName*:*keyColumn*⟩⟩

   generates a primary key:

   ```
   ALTER TABLE tableName
       ADD CONSTRAINT tableName_pk PRIMARY KEY (columnName);
   ```

3. Each **foreignKey** construct

   ⟨⟨sql:*tableName*:*refTableName*:*keyColumn*⟩⟩

   generates a foreign key:

   ```
   ALTER TABLE tableName
       ADD CONSTRAINT tableName_fk
           FOREIGN KEY (columnName) REFERENCES refTableName;
   ```

4. If the transformation was on an XML document then the relevant constructs are queried using IQL and SQL INSERT statements are issued to populate the new database.

As can be seen from the above once the Automed transformations have been done to create constructs closely related to each model, actually materialising the model is not difficult.

10 Conclusions

Relational databases are the dominant data storage model and XML the de facto standard for sending information over the Internet. Being able to transfer data between the two formats has become an important task. This project has presented a method of transferring data based on a low-level data modeling language thus abstracting away some of the differences between the models and highlighting the similarities.

The first part of this report discussed the difficulties in the translation process and provided an overview of current approaches to the problem. Some examples of how XML has been stored in relational databases were given and conversely examples were given of how relational data may be exported as XML. The advantages and disadvantages of the different approaches was discussed.

It was shown that having a good schema describing the XML aided the translation process. XML Schema was introduced as a widely accepted standard for describing these schemas.

A technique for creating an XML Schema document from a relational schema was presented. Two different approaches to creating a relational schema were also described. One that did not need an XML Schema and one that did. It was shown that the latter resulted in a far more efficient relational schema. The examples given throughout the report show that there are many different ways of representing the same data in both XML and the relational model. Choosing the most efficient is a very difficult task. Two automatic techniques were mentioned as well as the fact that some authors propose a level of user interaction in the choice.

The advantages of using a low-level language that to abstract away the differences between XML and RDBMSs was discussed HDM is one such language. A description was given as well as an example of how an XML document might be represented in the HDM.

Three major difficulties with the translation process were identified: transforming an XML schema into a relational one and visa versa, providing some way of assessing the suitability of the transformed schema and providing a way of querying the new schema. These problems were addressed as follows:

10.1 Transforming the Schemas

The first task in transforming the schemas was to identify a way of describing them. For the relational model SQL DDL statements are the accepted and well understood way of doing this. Deciding which schema language to use to describe XML was more difficult. The first problem is that many XML documents exist that have no schema document describing them at all. Florescu and Kossman's edge schema for storing XML documents that have no constraining schema document was presented. Far more efficient and accurate transformations can achieved if there is a schema. I decided to chose XML Schema as my schema language as it is powerful and is an accepted W3C standard. It also has the advantage over the older DTDs that the schema itself is an XML document making it easier to parse and extract information from.

I took the decision to do the XML to relational transformations via the HDM, within the Automed framework. This provided a number of advantages. Firstly using a low level abstraction of the data structures involved highlighted the differences and similarities between the two models. The graphical nature of the HDM further aided this. Another advantage of using HDM was that the relational model had already been described in terms of HDM structures. Related to that now that XML Schema has been described in terms of the simple HDM structures it should be easy to transform the model into other models already described in terms of the HDM such as the ER model. Finally Automed provided a sound theoretical basis for the transformations. On a practical level I was able to take advantage of the Automed APIs that already exist to create my demonstration program and use the Automed editor to view them.

Using the HDM meant that the first task was to define XML Schema constructs in the HDM model, i.e. how could things like complex types be described in terms HDM nodes, edges and constraints. Once this was done the primitive transformations on the XML Schema model were defined. A few more complicated composite transformations within the model were also defined. As the projects main focus was transforming from XML to the relational model a complete set of

transformations was not presented. This may be a task for the future. To further help with the generalisation of the model transformations were defined from the XML Schema oriented structures to general HDM oriented structures. These should be able to be used in transformations to any or the models supported by Automed. In this instance they were transformed into relational oriented structures. In practice this process involved writing an Automed wrapper to transform the XML Schema document into Automed schema constructs and a number of transformation methods using the Automed API. Once the relational oriented structures were created they were used to build up SQL DDL statements that actually created the database. Using an intermediate CDM additionally helped made each step in the transformation process more manageable.

Going in the other direction I took advantage of the SQL wrapper already in Automed to create a schema of relational oriented constructs. These were then also transformed into more general HDM oriented constructs and finally into XML Schema oriented constructs. The XML Schema constructs were then used to create an XML Schema document.

Writing my program within the sound theoretical framework of the Automed system often highlighted problems in my model as I realised that bits of my theory would not work in practice. This helped enormously in making my model more sound and consistent. If I had not had well established theory behind the creation of my model and transformations I believe they would have been of far less value and much more likely to have contained errors.

10.2 Choosing the Most Appropriate Schema

To help with the task of choosing the most appropriate schema a visual tool was created presenting the XML Schema document as a tree. This made it easy to see any nesting in the schema. Functionality was provided to allow a user to change the nesting of the schema by dragging and dropping nodes. Leaf nodes could also be turned into keys or keyrefs. Once the schema was what the user wanted it could be materialised as either an XML Schema or a relational database. The visualisation is XML Schema oriented. When translating from the relational model to XML Schema the relational constructs were first transformed into XML Schema oriented ones before being displayed in the tree.

10.3 Querying the Data

Another component of the Automed framework was used to move data between the two models. IQL was used to query the final schema along the transformation pathway. In this way XML Schema oriented queries were used to extract the relational data and relational oriented queries used to extract the XML data. The latter was done in two different ways. The visual tool allowed either an XML document constrained by an XML Schema to be loaded or an XML Schema document on its own. If an XML document was loaded the data in it was used to populate the relational database. If an XML Schema document was loaded into the tool the database was left empty. Creating the XML representation of the relational data involved creating an XML document conforming to the XML Schema already created and then populating the data elements and attributes with data from the database.

10.4 Future work

The XML Schema standard is extremely comprehensive and only the subset of it thought most relevant to data storage was can be imported into Automed using the XML Schema wrapper. Hence many XML Schema documents cannot be imported. An obvious extension to this project would be to improve the XML Schema wrapper so that all XML Schema documents could be imported. Another drawback to the method proposed is that an XML document do be imported must be constrained by an XML Schema. Zamboulis and Poulovassilis present a method for inferring a XML DataSource Schema from a XML document and importing that into Automed [33]. A combination of their approach and the one presented here would be able to import any XML document and be able to create an accurate an efficient model if an XML Schema did exist.

The transformations from XML Schema to relational model create a lot of relations: one for each complex type, one for each unbounded element and one for each element of complex type. These could almost certainly be optimised in the future. A way of automatically transforming XML Schema elements that have choice associated with them, either as part of a key or using the `choice` tags within a complex type was only discussed briefly. A more formal treatment of this problem is needed in the future.

Only a limited number of transformations on the XML Schema model were presented. These could be expanded to include a set of all legal transformations on an XML Schema document. The model could then be used for restructuring XML Schema documents in a provably consistent fashion.

The visual tool was designed as a proof-of-concept and has a number of shortcomings that could be improved with more time. Operations cannot be reversed. At present there is no support for multi-value keys or XML namespaces. Operations are not checked for validity. In spite of these shortcoming creating the tool within the Automed framework constantly highlighted problems in my model as I went along and was an invaluable to the whole process.

References

[1] ALTOVA. xmlspy. http://www.xmlspy.com/, 2004.

[2] M. Boyd and N. Tong. The Automed repositories and API. Technical report, AutoMed Project, 2001.

[3] J. Liu C. Liu, M. Vincent and M. Guo. A virtual XML database engine for relational databases. XSYM 2003, 2003.

[4] James Clark and Steve DeRose Eds. XML Path Language (XPath) Version 1.0. http://www.w3c.org/TR/xpath, November 1999.

[5] F. Chiu D. Lee, M. Mani and W. W. Chu. NeT & CoT: Translating relational schemas to XML schemas using semantic constraints. UCLA CS Technical Report, February 2002.

[6] C.J. Date. *An introduction to Database Systems*. Addison-Wesley Publishing Company, 1995.

[7] F. H. Lochovsky D.C. Tsichritzis. *Data Base Management Systems*. New York: Academic Press, 1977.

[8] Jayavel Shanmugasundaram et al. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB Conference*, 1999.

[9] Jayavel Shanmugasundaram et al. Efficiently publishing relational data as XML documents. *VLDB Journal: Very Large Data Bases*, 10(2–3):133–154, 2001.

[10] Jayavel Shanmugasundaram et al. A general techniques for querying XML documents using a relational database system. *SIGMOD Record*, 30(3):20–26, 2001.

[11] L. Popa et al. Mapping XML and relational schemas with Clio. http://www.almaden.ibm.com/cs/clio/papers/icde02demo.pdf.

[12] M. Carey et al. XPERANTO: Middleware for publishing object-relational data as XML documents. In *VLDB Conference*, pages 646–648, 2000.

[13] M. Wahl et al. Lightweight Directory Access Protocol (v3), RFC 2251. http://www.ietf.org/rfc/rfc2251.txt, December 1997.

[14] Phil Bohannon et al. From XML Schema to relations: A cost-based approach to XML storage. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, 2002.

[15] Philippe Le Hgaret et al. Document Object Model (DOM). http://www.w3.org/DOM/, April 2004.

[16] Brett McArthur et al. Eds. JDOM. http://www.jdom.org/, February 2004.

[17] Henry S. Thompson et al. Eds. XML Schema part 1: Structures. http://www.w3.org/TR/xmlschema-1, 2001.

[18] Scott Boag et al. Eds. XQuery 1.0: An XML Query Language. W3C Working Draft. http://www.w3c.org/TR/xquery, November 2003.

[19] Tim Bray et al. Eds. Extensible Markup Language (XML) Version 1.0 Third Edition. http://www.w3.org/TR/REC-xml, February 2003.

[20] D.C. Fallside. XML Schema part 0: Primer. http://www.w3.org/TR/xmlschema-0, 2001.

[21] Mary Ferández, Wang-Chiew Tan, and Dan Suciu. Silkroute: Trading between relations and XML. In *Proceedings of the Ninth International World Wide Web Conference*, 2000.

[22] D. Florescu and D. Kossman. Storing and querying xml data using an RDBMS. *Bulletin of the Technical Commitee on Data Engineering*, 22(3):27–34, September 1999.

[23] Jonathan Gennick. SQL in, XML out. http://otn.oracle.com/oramag/oracle/03-may/o33xml.html, 2003.

[24] Kohsuke Kawaguchi. W3C XML Schema made simple. http://www.xml.com/pub/a/2001/06/06/schemasimple.html, June 2001.

[25] Elena Litani. XML Schema API. http://www.w3.org/Submission/xmlschema-api/, March 2004.

[26] Peter McBrien. Automed in a nutshell. http://www.doc.ic.ac.uk/automed, 2004.

[27] Peter McBrien and Alexandra Poulovassilis. A general formal framework for schema transformation. In *Data and Knowledge Engineering*, volume 28, pages 47–71, 1998.

[28] Peter McBrien and Alexandra Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering*, volume 1626 of *LNCS*, pages 333–348. Springer Verlag, 1999.

[29] Peter McBrien and Alexandra Poulovassilis. A semantic approach to integrating XML and structured data sources. In *Advanced Information Systems Engineering*, volume 2068 of *LNCS*, pages 330–345. Springer Verlag, 2001.

[30] Dare Obasanjo. W3C XML Schema design patterns: Avoiding complexity. http://www.xml.com/pub/a/2002/11/20/schemas.html, November 2002.

[31] Alexandra Poulovassilis. The automed intermediate query language. Technical report, AutoMed Project, 2001.

[32] International Telecommunication Recommendation. Recommendation x.521 (02/01) article e21263. http://www.itu.int/rec/recommendation.asp?type=items&lang=e&parent=T-REC-X.521-200102-I, February 2001.

[33] Lucas Zamboulis and Alexandra Poulovassilis. XML data integration by graph restructuring. Technical report, 2003.