Handling Types in Inter-Model Data Transformations AutoMed Technical Report

Andrew Smith - acs203@doc.ic.ac.uk
Peter McBrien - pjm@doc.ic.ac.uk

22 July 2005

Abstract

Inter-model data transformation, for example from XML to the relational model, presents a number of challenges that do not exist in a single-model environment. How the constructs of the different models map onto each other has been the subject of extensive study. Something that has not been investigated in detail, however, is how any type definitions used in a source schema in one model relate to those in the target model, and how this information may be used to create safer and more efficient transformations. Using type information that can allow us to produce type safe transformations. i.e. transformations that do not attempt to relate constructs that have incompatible types. This can be of great benefit during query processing as it can help cut down data related runtime errors. Another key goal during inter-model data transformation is to maintain the expressiveness of the source schema in the target schema. Ignoring type information on a construct can lead to a loss of expressiveness. The method we present here helps to overcome both of these problems. It is based on an existing mediator system that makes use of a well defined common data model (CDM) as an intermediary for the transformations. This has the advantage that type castings need only be defined from the source language to the CDM rather than from each source to every other possible source language.

1 Introduction

How best to transform data from one data modeling language to another has been an aspect of data management that has been studied for a number of years. Bernstein [3] describes a number of scenarios where inter-model data transformation needs to be done. The problem has become particularly relevant in recent years as the growth of eCommerce has meant a lot of data needs to transformed from its more traditional relational format to more easily transportable formats like XML.

Data transformation is a component of the wider field of data integration. A substantial amount of work has been done in both areas. See [2, 9] for surveys on data integration. There has also been work done on inter-model transformation of integrity constraints [5, 4] but little has been written about how simple types, i.e. integer, float, string etc., that some view as a form of constraint [1], affect data transformations between constructs in different models. TSIMMIS [8] provides for type information to be stored in their Object-Exchange Model [16] but do not use this information to test the safety of transformations. WOL [18] is another language for database transformations that stores type information, however, the language is only able to describe transformations in relational and object-relational databases, not inter-model transformations. The Clio system [6, 15] defines a number of value based **source-to-target dependencies** that specify how and what source data should appear in the target. This data-centric approach does not make use of type information in the source schema to help map to the target schema. In ignoring type information these systems risk losing expressiveness during the transformations and allowing type-incompatible transformations to be written.

This work aims to address some of these problems by making the following contributions: It offers a way of improving the expressiveness and safety of inter-model transformations in a data integration environment by adding type information. The work also develops an existing data integration system called AutoMed, in that it adds a type system to AutoMed's common data model and also adds XML Schema as a new modeling language to the AutoMed mediator environment, allowing the XML Schema type information to be added to the existing XML model.

The rest of this paper is organised as follows: Section 2 provides a motivating example. Section 3 will briefly describe the AutoMed system and its common data model, the HDM. Section 4 describes the type system used in HDM, including a formal definition of the type hierarchy. Section 5 shows how this type information can be incorporated into an HDM schema. Section 6 gives an example of how higher level modeling languages can be represented in HDM. Section 7 shows how this work is applicable to inter-model transformations in HDM. Section 9 offers some conclusions and suggestions for future work.

2 Motivating Examples

Capturing type information during data transformations can be useful in a number of scenarios. The following example shows how enforcing type safe transformations and maintaining type expressiveness can be of use when creating a target schema in a different model from the source schema.

Transforming data from XML to the relational model and back again is a common task these days as data often needs to be moved between the Web and relational databases. We can imagine a scenario where a shoe manufacturer needs to send stock information to a number a retailers. The manufacturer has a database of information about the shoes including an `integer` field for the stock level sent to each retailer. This information is sent over the web as an XML document with an associated XML Schema. At first the schema document defined all the types of all the elements in the XML document to be `string`, for simplicity. Because there was no specific information about the type of the stock level field, some of the retailers stored this information in their databases as a `VARCHAR`. When it came time to return unsold stock to the manufacturer, data entry errors at some of the retailers meant that non-numerical data in the stock level field ended up causing problems for the manufacturer's database. Tracing where this came from and getting the correct information resent was a major headache. If the type of the original field had been maintained throughout the data transformation process these errors could have been detected locally and fixed there. Even if only the XML Schema had type information and transformations were forced to be type safe, the transformation from the local `VARCHAR` to `integer` would have failed.

Type information can also be useful in identifying incompatible constructs when transforming data between existing schemas. Assume we have a source schema with a field picture of type `integer` that is used to store the id of a picture in a database. In the target schema a field also called picture exists but this is a binary object and stores the picture itself. Looking at the types of these two fields would allow us quickly to decide they were incompatible without having to interrogate the data.

3 AutoMed and HDM

One way of transforming data from one model to another is to borrow the **mediator** approach from data integration. Wiederhold [19] defines a mediator as 'a software component that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer application.' Mediators provide the link between the local data sources and the user applications that will present the data. Garcia-Molina et al. [8] identify the following features that a mediator model should support:

1. A collection of structures rich enough to model data from numerous different sources.

2. Graceful handling of missing information in the source data.

3. Meta-information about the structures themselves.

AutoMed, a joint project between Imperial College London and Birkbeck College, is an example of a mediator system. It has show itself to be successful in addressing the first point [13].The second point is the subject of on-going research. This work partially addresses the third by adding meta-information about the type of data being transformed.

The mediator approach requires source schemas to be converted into **component** schemas in a common data model (CDM) able to express the semantics of all possible source schemas we may wish to exchange data between.

The CDM used by AutoMed is the Hypergraph Data Model (HDM) [12]. HDM has been used to model data from a wide variety of sources including relational data [13] and XML [14]. HDM is a low-level 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 HDM is also defined. Constructs and transformations in higher-level modeling languages are then defined in terms of these. The general nature of the language offers advantages over the model-specific techniques when coming to describe data from differing sources such as RDBMSs and XML Schema. Batini et al. [2] suggest that a simpler CDM, like HDM, has advantages over more complex models.

A schema in the HDM is defined in [12] as 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.

The HDM defines a number of primitive transformations that can be used together to create complex transformations. The reader is referred to [12] for a complete list of the primitive transformations.

4 The HDM Type System

A type system allows us to answer questions about the nature of the values in a node. In the context of transformations a type system can be used to answer specific questions about whether a transformation is **type safe**, i.e. has no mismatched types. A hierarchical system as described here can tell us how and if types in different data models can be recast to create valid transformations.

A type system also helps a transformation system to be more expressive. Atkinson and Buneman [1] note a lack of expressive power as a common failure in database programming languages.

The type system introduced here for HDM is based on those used in functional programming languages like Haskell [17]. It can be used to ensure that all transformations done in the HDM are type safe. This helps a user design transformations by flagging possible problems when the transformations are defined rather than having to back track though a number of transformations when a query fails or gives unexpected results.

Types can be said to describe values with common properties [10] and intuitively we can think of types as sets of those values.

4.1 The Type Hierarchy

The basis of the HDM type system is a canonical type hierarchy. A portion of this hierarchy is defined as follows:

$$T_{hdm} = \{any(string(integer), binaryObject, oid)\}$$

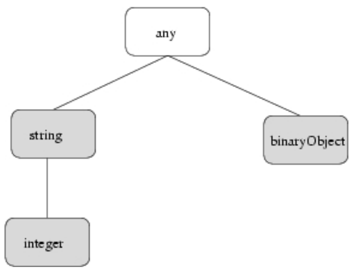

Figure 1: A portion of the HDM type hierarchy

Note that this is only an indicative portion of the hierarchy. The special *oid* type is used to represent object identifiers in non-key data models. An example of its use is given in Section 7.

There are two types of nodes in the hierarchy tree. Figure 1 shows a graphical representation of this where the shaded nodes represent atomic types and the non-shaded one is a generic root type with no specific type information associated with it. The type hierarchy for each language modeled in HDM will have one of these root nodes.

Examples of atomic types in some of the languages already modeled in HDM are:

Relational `VARCHAR`, `INTEGER`, `DATE`

XML Schema `anySimpleType`, `integer`, `string`

Each language modeled in AutoMed has a different type hierarchy geared to its specific needs. For example in a relational system, it is not normal to regard strings/varchars as more general than integers, but in XML it is. This is discussed in more detail in Section 6. The way each language's atomic types map to the HDM canonical types is also defined.

4.2 Formal definition

In keeping with the graphical nature of HDM, the type system is represented as a graph that is defined as follows:

Definition 1 *The components of the HDM type system* $(C, T, Ext, <)$ *are:*

- *A set of constant symbols* C *representing the data values that may occur in the universe of discourse*
- *A finite set of types* T
- *An* Ext *function that returns a subset of* C *that is consistent with a type:*
  $t \in T \rightarrow Ext(t) \subseteq C$
- *An order relation* $<$ *that builds the elements of* T *into a tree hierarchy, where if* $t, t' \in T$
  $t < t' \rightarrow Ext(t) \subseteq Ext(t')$
  $t < t' \rightarrow \neg t' < t$

The type casting operator, $\odot$, gives us a way of comparing types within a single model and between models. Within a single model a type may be casted to another if it is below that type in the type hierarchy. Between models, a type may be cast to another if its extent is a subset of the extent of the other type. The $\subseteq$ operator is used to represent this.

Definition 2 *The expression* $t \odot t'$ *indicates that* $Ext(t) \subseteq Ext(t')$*.*
Note that for intra-*model type casting, we can determine this simply by* $t < t' \rightarrow t \odot t'$*.*
For inter-*model casting we must detemine if* $Ext(t) \subseteq Ext(t')$*.*

Note the two types t, t' are equivalent to each other if $t \odot t'$ and $t' \odot t$. The following are some examples of the use of the operators defined above. Within the relational model we have:

$$int_{rel} \odot long_{rel} \iff int_{rel} < long_{rel}$$

Within XML Schema:

$$int_{xml} \odot long_{xml} \odot integer_{xml} \iff int_{xml} < long_{xml} < integer_{xml}$$

Between models we can say:

$$int_{xml} \odot int_{rel} \iff int_{xml} \subseteq int_{rel}$$

From the three operations above we can deduce that:

$$int_{xml} \odot long_{rel}$$

5 Type information in a schema

Type information is added to an HDM schema as a fourth component: T representing the type hierarchy. Each node in an HDM schema will have an extra component $t \in T$ that represents the type of the node, added to its scheme. The extent of the node will necessarily be a subset of the values represented by the type, i.e., $\langle\langle \mathsf{n}, \mathsf{t} \rangle\rangle \subseteq t$ and $x \in \langle\langle \mathsf{n}, \mathsf{t} \rangle\rangle \rightarrow x \in t$. Edges do not need to have type information added to them as the extent of an edge is a tuple made up of the extents of the constructs it links. The type of the tuple components comes from these constructs.

When a node from a source schema is added to AutoMed, its type is checked to see if it may be cast to an equivalent HDM type. It is important to map the new type to HDM type as far down the type hierarchy as possible to avoid problems like those mentioned in the motivating example.

If no suitable mapping exists a new type mapping from the source type to HDM can be defined. This may involve a 2-way function that explicitly maps the source schema type to an equivalent HDM type and back again. An example of this is given in Section 7.

Untyped nodes in the source schema can be can be added as nodes of root type if the target schema requires typed nodes. This has the disadvantage that transformations involving nodes with root type cannot be checked for type safety. User intervention may be necessary when choosing a type for an untyped node during a transformation. For example, when transforming unconstrained XML to the relational model there may be XML elements or attributes that are obviously integers. It may be possible to infer the type of a node from its extent. Work in this area is ongoing.

5.1 Primitive node operations

When an addNode operation is performed the type of the new node will be the type of the result of the query used to define the node's extent. This will generally be another node, so the type will come from that node.

The definition of addNode given in [12] is enhanced with type information as follows:

> addNode($\langle\langle \mathsf{name}, \mathsf{typeLookup(q)} \rangle\rangle, q$) adds a node name whose extent is given by the value of the query q. Conditions: name is not the name of an existing node. Assume the new node has type hierarchy T then typeLookup(q) returns $t \in T$ such that the type of q is equivalent to t.

typeLookup(q) is defined as follows:

1. Let $\langle\langle n_1, t_1 \rangle\rangle \cdots \langle\langle n_m, t_m \rangle\rangle, m > 1$ be nodes in query q.

2. If $t_1 \cdots t_m$ are all equal then:
   typeLookup(q) $= t_h \in T \mid t_1 \odot t_h$
   else
   typeLookup(q) $= t_h \in T \mid t_i \odot t_h \, \forall \, t_i \; 1 \leq i \leq m$.

3. If we cannot find such a t_h the addNode operation fails with a type incompatible error.

The deleteNode operation includes a query that allows the deleted node to be restored. The type of this query must be the same as the type of the node that is being deleted. The definition of the operation becomes:

> deleteNode($\langle\langle$name, t$\rangle\rangle, q$) removes a node name. Conditions: The query, q that defines how the extent of the deleted node can be restored, must return a value of type t.

We also need a new basic operation to change the type of a node.

> changeNodeType($\langle\langle$name, $\mathsf{t_{old}}\rangle\rangle$, $\mathsf{t_{new}}$) changes node type.
> Conditions: $\mathsf{t_{old}} < \mathsf{t_{new}}$. t_{old} and t_{new} must be in the same model.

This operation casts the node $\langle\langle$name$\rangle\rangle$ to a more general type.

6 High Level Modeling Languages in HDM

In general, any semantic modeling language consists of two types of construct: **extensional** and **constraint**. 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**: These link two other constructs. They cannot exist in isolation and 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

Constraint constructs are restrictions on the extents of the extensional constructs.

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.
- We need a del transformation for every construct in M, that reverses its add transformation.
- For those constructs that have a textual name a rename transformation is defined in terms of the HDM renameNode and renameEdge transformations.

6.1 XML and XML Schema

This section presents a definition of the XML model in terms of the HDM, as defined by an XML Schema document. Note that for clarity only the most basic XML Schema constructs are presented here. In particular no identity constraints have been described.

XML Schema [7], like DTDs, allows type information to be associated with XML documents. It describes the structure and type of the data that appears in an XML **instance** document conforming to the schema.

In AutoMed a transformation pathway is created based on the XML Schema structures. Any data in an XML instance document of that schema can be transformed along the same pathways. An XML Schema document can be viewed as being similar to an empty relational table. We can transform an XML Schema document into a relational table and then populate that table by taking data from an XML instance document of that schema.

Simple types in XML Schema form a hierarchy with the base type *anySimpleType* at the root. Any other simple types are defined as restrictions or extensions of this type. This is very similar to the way that types in HDM work.

An example of an XML Schema type hierarchy is shown in Figure 2 and can be defined as follows:

$$T_{xml} = \{anySimpleType(string(integer))\}$$

Some of the basic type castings between XML Schema and HDM are as follows:

Definition 3 *XML Schema to HDM type castings:*
$anySimpleType_{xml} \odot any_{hdm}$
$any_{hdm} \odot anySimpleType_{hdm}$
$integer_{xml} \odot integer_{hdm}$
$integer_{hdm} \odot integer_{xml}$

The various constructs in the XML Schema model are defined as follows:

1. An XML Schema **complexType** can exist on its own and is a **nodal** construct.
   The production rule is defined as follows:

   $$\mathsf{complexType}\langle\langle \mathsf{xs}{:}t, \mathsf{oid}\rangle\rangle \leadsto \langle\langle t\rangle\rangle$$

2. All XML **elements** in an instance document conforming to an XML Schema have a type as defined by that schema. They can exist independently and are defined as **nodal** constructs, represented in HDM by a node $\langle\langle \mathsf{xml}{:}e{,}t\rangle\rangle$.

   The production rule for elements is:

   $$\mathsf{element}\langle\langle \mathsf{xs}{:}e{,}t\rangle\rangle \leadsto \langle\langle e\rangle\rangle \iff t \in T$$

3. An XML **attribute** $\langle\langle \mathsf{xs}{:}e{:}a{,}t\rangle\rangle$ is associated, in its schema document, with a parent complex type. It is thus defined as a **nodal-linking** structure. Attributes also have a data type as defined by the schema.

   The production rule is as follows:

   $$\mathsf{attribute}\langle\langle \mathsf{xs}{:}e{:}a\rangle\rangle \leadsto \langle\langle pt : a, t\rangle\rangle, \langle\langle \_, pt, pt : a\rangle\rangle$$

4. Nesting of elements 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 $\langle\langle \mathsf{xs}{:}e\rangle\rangle$.

   The production rule for this construct is:

   $$\mathsf{complexTypeNest}\langle\langle \_, \mathsf{xs}{:}pt, \mathsf{xs}{:}e\rangle\rangle \leadsto \langle\langle \_, pt, e\rangle\rangle$$

6.2 The Relational Model

The HDM representation of the relational model is described in detail in [13]. Here we describe the type hierarchy and examples of the castings that have been added to the model.

The basic structure of the type hierarchy for the relational model is shown in Figure 1. We can define it formally as follows:

$$T_{rel} = \{root(varchar(date), integer, binaryObjects)\}$$

As the relational model is a keyed model, the type of all relations will be the type of their key field.

Example type castings to the HDM mode are defined below:

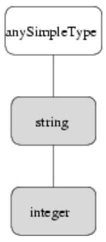

Figure 2: Sample HDM type hierarchy for XML Schema

Definition 4 *Relational to HDM type castings:*
$varchar_{rel} \odot string_{hdm}$
$string_{hdm} \odot varchar_{rel}$
$integer_{rel} \odot integer_{hdm}$
$integer_{hdm} \odot integer_{rel}$
$date_{rel} \odot string_{hdm}$

We see that *varchar* and *integer* map easily to *string* and *integer* respectively in the HDM model, however, *date* has no obvious HDM equivalent and so has been mapped to the more general *string* type.

7 Inter-Model Transformations

During a transformation from a source schema in one model to a target schema in a different model the type information for the target schema constructs is derived from the from the source constructs via the canonical HDM type system described above. Types are first converted from the source model into the canonical model and then from there into the target model. Using a common model as an intermediary for all inter-model transformations means that type castings only need to be defined between each model and the HDM rather than between every model used by AutoMed.

The following is an example of a schema transformation from XML, constrained by XML Schema, to the relational model. The transformations will maintain the expressiveness of the XML Schema types. Two examples will be given, one in which the XML Schema defines a `key` field and one in which there is no key field.

Consider the following XML fragment:

```
<class>
  <code>AI</code>
  <numStudents>25</numStudents>
</class>
```

Figure 3 shows an XML Schema fragment describing this.

As stated above, to transform this to HDM without losing expressiveness, we need to represent the types in HDM. If we did not there would be constraints left in the source schema with no equivalent in the target. As was mentioned in the introduction types can be viewed as a kind of constraint. This is a problem mentioned in [11] that can easily occur during inter-model transformations. For example, if we transform the XML Schema into a relational database we have no way of knowing that `numStudents` should be an `integer`.

A graphical representation of the schema is shown in Figure 4. Note that, in keeping with the type hierarchy graphs, the types of the shaded nodes are atomic types and the non-shaded nodes are non-atomic types, in this case an XML Schema `complexType`. In the first instance we will assume that the `code` element acts as a key. When this is the case the type of the non-shaded node becomes that of the key, in this case `string`, so the XML above could be represented by the following HDM schema:

```
<xsd:complexType name = "classType">
    <xsd:sequence>
        <xsd:element name = "code" type = "xsd:string" />
        <xsd:element name = "numStudents" type = "xsd:integer" />
    </xsd:sequence>
</xsd:complexType>

<xsd:element name = "class" type = "classType">
  <xsd:key name = "idCode">
    <xsd:selector xpath =  ".//class" />
    <xsd:field xpath =  "code" />
  </xsd:key>
</xsd:element>
```

Figure 3: XML Schema fragment

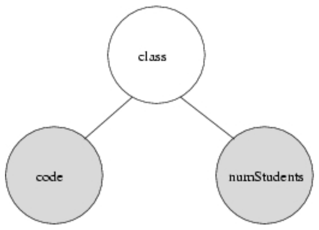

Figure 4: HDM representation of class XML Schema fragment

$S = \langle \{ \langle\langle \mathsf{class}, \mathsf{string} \rangle\rangle, \langle\langle \mathsf{code}, \mathsf{string} \rangle\rangle, \langle\langle \mathsf{numStudents}, \mathsf{integer} \rangle\rangle \},$
$\{ \langle\langle \_, \mathsf{class}, \mathsf{code} \rangle\rangle, \langle\langle \_, \mathsf{class}, \mathsf{numStudents} \rangle\rangle \},$
$\{ anySimpleType(string(integer), oid) \} \rangle \}$

If we do not have a key defined, the type of the class node becomes the special type, *oid*. This is a place-holder type that can be cast to any target atomic type. This may be necessary if the target schema is in a keyed model.

The transformations from the XML **elements** in Figure 3, which form the leaf nodes in Figure 4, to relational **columns** via the canonical HDM model are described below. We will not include those involving edges for sake of brevity, as edges do not have specific type information associated with them. We will also shorten typeLookup to tl.

$\mathsf{addNode}_{hdm}(\langle\langle \mathsf{code}, \mathsf{tl}(\langle\langle \mathsf{code}, \mathsf{string_{xml}} \rangle\rangle) \rangle\rangle, \langle\langle \mathsf{code}, \mathsf{string_{xml}} \rangle\rangle)$
Let $t = \mathsf{tl}$ be derived as follows:
Let $t_q \in T_{hdm}$ such that $t_q \odot t$
$t_q = string_{xml}$
$string_{xml} \odot string_{hdm}$ by Definition 3
therefore $t = string_{hdm}$

$\mathsf{addNode}_{hdm}(\langle\langle \mathsf{numStudents}, \mathsf{tl}(\langle\langle \mathsf{numStudents}, \mathsf{integer_{xml}} \rangle\rangle) \rangle\rangle,$
$\langle\langle \mathsf{numStudents}, \mathsf{integer_{xml}} \rangle\rangle)$
in a similar way to above $t = integer_{hdm}$

We now have the types in the canonical HDM form ready to transform into the relational model. The transformations for that are as follows:

$\mathsf{addNode}_{rel}(\langle\langle \mathsf{code}, \mathsf{tl}(\langle\langle \mathsf{code}, \mathsf{string_{hdm}} \rangle\rangle) \rangle\rangle, \langle\langle \mathsf{code}, \mathsf{string_{hdm}} \rangle\rangle)$
Let $t = \mathsf{tl}$ be derived as follows:
Let $t_q \in T_{rel}$ such that $t_q \odot t$.

$t_q = string_{hdm}$
$varchar_{rel} \odot string_{hdm}$ by Definition 4
therefore $t = varchar_{rel}$

$\mathsf{addNode}_{rel}(\langle\langle \mathsf{numStudents}, \mathsf{tl}(\langle\langle \mathsf{numStudents}, \mathsf{integer_{hdm}} \rangle\rangle) \rangle\rangle),$
$\langle\langle \mathsf{numStudents}, \mathsf{integer_{hdm}} \rangle\rangle)$
in a similar way to above $t = integer_{rel}$

Each relational column has a type associated with it when it is defined. The nodes from the final transformation above map easily to this, the types of the nodes becoming the types of the columns.

The class node, which is the root of the graph in Figure 4, will be transformed into a relation in the relational model. If the XML Schema has a key, as in our first instance, then that key is mapped to the primary key of the new relation along with its type. In this case, the $string_{xml}$ type from XML Schema maps to $varchar_{rel}$ as described above and so the node becomes $\langle\langle \mathsf{class}, \mathsf{varchar_{rel}} \rangle\rangle$ in the relational model.

If, however, there is no key and the XML type is *oid* then we need to make a decision about what type the primary key of the relation should have. From a computational point of view the simplest solution is to ask a data expert whether any of the fields in the source schema make a suitable key and if so use that. A more general way is to create a special oid attribute in the relational model that will act as the key. The extent of this attribute will be $1..n$ where n is the number of elements of that type in the source schema. The type of the attribute will be `integer`. In the context of our transformations the *oid* type from the HDM model will be cast to an `integer`. Formally we will have:

1. $\mathsf{addNode}_{rel}(\langle\langle \mathsf{oid}, \mathsf{integer_{rel}} \rangle\rangle, 1..n)$
2. $\mathsf{addNode}_{rel}(\langle\langle \mathsf{class}, \mathsf{integer_{rel}} \rangle\rangle, \langle\langle \mathsf{oid}, \mathsf{integer_{rel}} \rangle\rangle)$

The HDM in Figure 4, derived from the XML Schema with the key constraint in Figure 3, could thus be transformed into the following relational table as defined in SQL DDL.

```
CREATE TABLE class (
code VARCHAR,
numStudents INTEGER,
CONSTRAINT idCode PRIMARY KEY (code)
);
```

If we assume that we have no key in the XML Schema we would have the extra oid attribute that would become the primary key and we would have the following relational table:

```
CREATE TABLE class (
code VARCHAR,
numStudents INTEGER,
oid INTEGER
CONSTRAINT idCode PRIMARY KEY (oid)
);
```

In both cases the columns in the relations have the correct type information.

8 Implementation

This work has been conducted within the context of the AutoMed mediator environment. AutoMed allows transformations to be defined between a number of different data modeling languages. It has been implemented as a series of Java APIs and is freely available at http://www.doc.ic.ac.

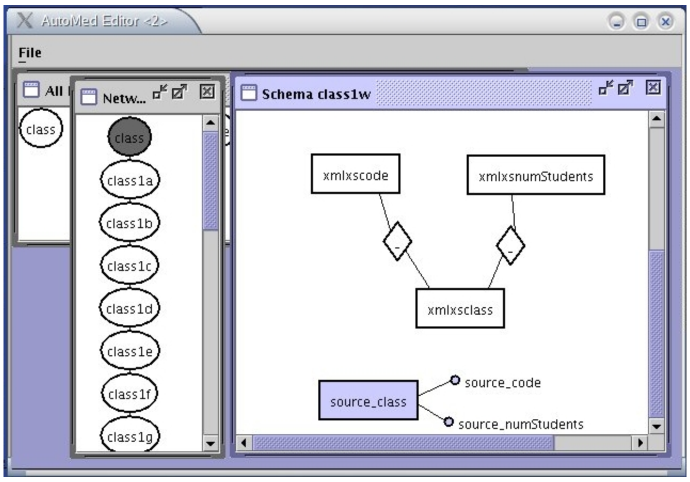

Figure 5: Screen shot of the AutoMed editor

uk/automed. There is a GUI that enables the user to perform various tasks and also provides a graphical view of the schemas in the system.

Figure 5 shows a screen shot from the AutoMed editor. The left hand window shows part of the sequence of transformations done on the class XML Schema to convert it to a relational schema. The right hand window shows a graphical representation of the XML on top and the relational tables that the XML has been transformed into underneath.

AutoMed is still undergoing development and part of this work is to add the type system described above to HDM, the CDM that AutoMed uses. A detailed definition of the type system has been created by colleagues working at Birkbeck College and will be included in the system shortly. At the moment a less rigorous implementation of the type system is being used to test the inter-model transformations described in this paper.

So far successful work has been done on transforming between XML Schema and the relational model, allowing us to the maintain the type information throughout the transformations. We aim to expand this to include other languages modeled by AutoMed in the future.

9 Conclusions and Future Work

In this paper we have shown how type information can be useful in inter-model data integration. We discussed how typing can be used to make sure that inter-model transformations are type-safe and to ensure that no expressiveness is lost when moving from a source to a target or global schema. This work has been done in the context of the AutoMed mediator system using its common data model, HDM. HDM was briefly introduced and then the new type system for HDM was described. A formal definition was given for the hierarchy and some of the primitive operations on HDM were redefined to take into account the new type system. It was shown how higher level modeling languages including their type information could be represented in HDM. An example of how type information could be preserved during inter-model transformations was given. Finally a brief description of the AutoMed system where we aim to implement this work was given.

Future work will focus on formalising the type castings between HDM and the other languages modeled in AutoMed. Some of this will focus on how structured types such as XML Schema complexTypes can best be included in the model. We will also expand the AutoMed system to fully include type information and to type check transformations. Work will also continue on the XML Schema model described briefly here.

References

[1] M. P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–190, 1987.

[2] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[3] P. A. Bernstein. Applying model management to classical meta data problems. In *CIDR*, 2003.

[4] A. Calì, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. *Inf. Syst.*, 29(2):147–163, 2004.

[5] S. B. Davidson, W. Fan, C. Hara, and J. Qin. Propagating XML Constraints to Relations. In *ICDE*, pages 543–554, 2003.

[6] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. In *ICDT*, pages 207–224, 2003.

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

[8] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The tsimmis approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2):117–132, 1997.

[9] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[10] M. Lenzerini. Type data bases with incomplete information. *Inf. Sci.*, 53(1-2):61–87, 1991.

[11] P. M. Boyd. Towards a semi-automated approach to intermodel transformation. AutoMed technical report No. 29 Version 2, December 2004.

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

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

[14] P. McBrien and A. 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.

[15] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[16] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In P. S. Yu and A. L. P. Chen, editors, *11th Conference on Data Engineering*, pages 251–260, Taipei, Taiwan, 1995. IEEE Computer Society.

[17] J. H. F. Paul Hudak, John Peterson. A gentle introduction to Haskell 98. http://www.haskell.org/tutorial/, 1999.

[18] Susan Davidson and A. Kosky. WOL: A Language for Database Transformations and Constraints. In *Proceedings of the International Conference of Data Engineering*, pages 55–65, April 1997.

[19] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, 1992.