Inter Model Data Exchange of Type Information via a Common Type Hierarchy

Andrew Smith and Peter McBrien

Dept. Computing, Imperial College London, London SW7 2AZ

Abstract. Data exchange between heterogeneous schemas is a difficult problem that becomes more acute if the source and target schemas are from different data models. The data type of the objects to be exchanged can be useful information that should be exploited to help the data exchange process. So far little has been done to take advantage of this in inter model data exchange. Using a common data model has been shown to be effective in data exchange in general. This work aims to show how the common data model approach can be useful specifically in exchanging type information by use of a common type hierarchy.
Keywords: data exchange, model management, data modelling

1 Introduction

Fagin *et al.* define **data exchange** as 'the problem of taking data structured under a **source schema** and creating an instance of a **target schema** that reflects the source data as accurately as possible' [1]. Data exchange may occur within a single data model, for example when building a relational data warehouse from a number of relational databases. However, it is often necessary to perform inter model data exchange in which two or more different data models are involved, such as when XML is used as data exchange format between a number of databases, which may use relational, XML, or other models for their internal storage of data.

One of the most challenging aspects of data exchange is how to transform constraints on the source schema to the target schema. This is particularly difficult if the source and target schemas come from different data models. Type information forms part of these constraints [2]. The inter model transformation of *integrity* constraints, like primary and foreign keys, has been investigated [3, 4] and data types have received attention in the field of database programming languages [2], but little has been written about how simple types, i.e. integer, float, string etc., should be transformed between models in a data exchange environment.

The solution we present is a graph based logical type hierarchy that can represent the data types of multiple data models and that links similar data types from different models by defining bi-directional mappings from the high level data types to a **common, extensible hierarchy** of data types. Cardelli [5] points out that the purpose of type systems is to prevent execution errors in

the running of a program. We present a system that attempts to prevent data errors during data exchange.

The contributions this paper makes are as follows:

- Offer a way of improving the expressiveness and safety of inter model transformations in a data exchange environment by adding type information.
- Provide a way of transforming data between schemas whose data types support different values.
- Show when transformations are illegal, when they need to be checked for data errors and when they do not, based on the data types of the source and target constructs.

The rest of the paper is organised as follows. Section 2 provides motivation by describing some of the difficulties inherent in data type exchange. Section 3 describes the formal definition of our type system. Section 4 introduces the AutoMed automatic mediation system which we have used to test our approach. Section 5 describes how the new type system can be added to AutoMed along with the definition of a number of operators that can be applied to the data types of objects in AutoMed. Section 6 describes a data exchange scenario in AutoMed using the new type system. Section 7 describes some other approaches to transformation of data types in a data exchange system. Finally some conclusions are drawn and some of our on-going work is described.

2 Motivation

In an inter model data exchange system with a typed target schema it is necessary to transform the data type of source schema objects to the target schema. Current solutions make use of simple one to one matching between data types in different models [6, 7]. This approach can lead to some problems:

- The data type of the source object may allow a greater range of values than that of the target object. This could lead to run-time errors when the materialised target schema is populated with data from the source schema.
- Types representing an identical concept may support a different range of values. For example, a boolean in one data modelling language may represent true as 1 while another may represent it as T. Again populating the target schema would cause errors. Some way of mapping the data values between the source and target is required.

Our solution is to represent the data types in a data exchange system as a **directed acyclic graph** (**DAG**) and to define a number of operations on the graph that can be used to overcome the problems described above. Hierarchies exist in some well known type systems, for example, Figure 1 is a portion of the XML Schema [8] type hierarchy. However, for data exchange purposes, the familiar tree hierarchy of Figure 1 has several short comings due to the hierarchy not fully modelling the domain of data values in XML. In particular:

– Figure 1 does not distinguish between types which are disjoint in their extent, such as positiveInteger (representing all positive integer values) and negativeInteger (representing all negative integer values), and those which overlap, such as int (representing $-2^{31}..2^{31}-1$) and unsignedInt (representing $0..2^{32}-1$).
  In data exchange, a mapping between source and target objects where the types are disjoint should be ruled illegal, unless an explicit conversion has been defined, but a mapping between objects that overlap should be allowed, with runtime range checking.
– Figure 1 fails to identify all isa associations in the hierarchy, shown as dashed lines in the figure. For example, all unsignedShort values (which are in the range $0..2^{16}-1$) are a subset of int values and all values in XML can be represented as string.
  In data exchange, if the source object type is a subset of the target object type, then the values may be cast without runtime range checking, since the cast will never fail.

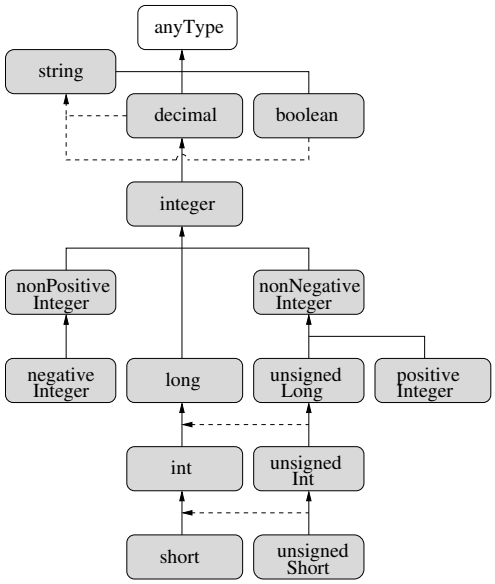

Fig. 1. A portion of the XML Schema type hierarchy [9]

A particular problem in data exchange is found when transferring between different type systems. Figure 2 shows a portion of a type hierarchy built for Postgres, showing some of the built-in types, and following the built-in casting rules, giving some isa relationships. When translating between the types of XML and Postgres the type systems need to be unified. Two difficulties arise from the fact that the modelling of the type systems might be quite different:

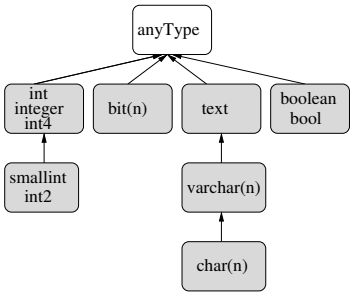

Fig. 2. A portion of the Postgres type hierarchy

- The types might not be named consistently. For example the Postgres integer (a 4-byte integer in the range $-2^{31}..2^{31}-1$) semantically corresponds to the XML int type, and not the XML integer type. Also the types might not be stored consistently: in XML all data is held as strings, whilst in Postgres the integer will be held as a twos complement binary number.
- There may be no equivalent type in the target model as compared with a source model type. For example, the bit(n) type of Postgres has no direct equivalent in XML.

Hence in the following sections we introduce our own definition of a type hierarchy that is suited to data exchange applications. In particular, it builds hierarchies solely on the range of data values a type may take. We then demonstrate how mapping rules between schemas in different modelling languages may be made type safe.

3 Data Exchange Type Hierarchy

To facilitate the precise representation of types in a data exchange environment, we introduce in Definition 1 a logical type hierarchy to be used to describe the types of a single or collection data modelling languages in a manner that allows us to address the problems discussed in the previous section. We will show that this type hierarchy can also be used to capture some semantics of types that are specific to a particular schema.

Definition 1. ***Type Hierarchy***
A type hierarchy TH_x is a tuple $\langle Types_x, Ext_x, \stackrel{t}{=}_x, \prec_x, \not\wedge_x, Mapping_x \rangle$ where:

- A finite set of type names $Types_x$. These will make up the nodes of the graph.
- An Ext_x function that returns a subset of the set of all possible data values $anyType_x$ of the hierarchy that is consistent with a type:
  $t \in Types_x \rightarrow Ext_x(t) \subseteq Ext_x(anyType_x)$
  The type $anyType_x$ represents the set of data values that a particular language (or collection of languages) may handle, and has the property that

$Ext_x(anyType_x) \subseteq C$ where C is the set of all data values that may be handled by the data exchange system.

- An **equality** relation, $\stackrel{t}{=}_x$, such that for $t, t' \in Types_x$
  $t \stackrel{t}{=}_x t' \iff Ext_x(t) = Ext_x(t')$
- A **partial ordering** relation $\prec_x$, such that for $t, t' \in Types_x$
  $t \prec_x t' \iff Ext_x(t) \subset Ext_x(t')$
  When all types $t, t' \in Types_x$ that have $t \stackrel{t}{=}_x t'$ are treated as a single node, the $\prec_x$ relation builds the types into a connected directed acyclic graph. $\stackrel{t}{=}_x$ and $\prec_x$ together make up the edges of the graph.
- A **disjoint** operator, $\not\frown_x$, such that for $t, t' \in Types_x$
  $t \not\frown_x t' \iff Ext_x(t) \cap Ext_x(t') = \phi$. This implies there is no pathway from t to t' in the graph and so we cannot cast between t and t'. If $t \not\frown_x t'$ then any subtype of t will also be disjoint from t'.
- A set $Mapping_x$ of mapping tables $\langle t_a, t_b, \{\langle s_a^1, s_b^1\rangle, \ldots, \langle s_a^n, s_b^n\rangle\}\rangle$, where $t_a, t_b \in Types_x$, $s_a^1, \ldots, s_a^n$ are subsets of $Ext(t_a)$ and are disjoint. Similarly $s_b^1, \ldots, s_b^n$ are subsets of $Ext(t_b)$ and are disjoint.
  We overload $Mapping_x$ to be used as a function with the following definition:
  $Mapping_x(t_a, t_b, v_a) = First(s_b^n) \mid$
  $\quad \langle t_a, t_b, map\rangle \in Mapping_x \wedge \langle s_a^n, s_b^n\rangle \in map \wedge v_a \in s_a^n$
  $Mapping_x^{-1}(t_a, t_b, v_a) = First(s_b^n) \mid$
  $\quad \langle t_b, t_a, map\rangle \in Mapping_x \wedge \langle s_b^n, s_a^n\rangle \in map \wedge v_a \in s_a^n$
  where *First* returns the first element of a set according to a sort order that is fixed for the system. We use $Mapping_x(type_a, type_b)$ to denote the specific mapping table that maps $type_a$ to $type_b$.

Examples 1 and 2 illustrate the values for the type hierarchy for the XML and Postgres type models.

Example 1. The type hierarchy for the XML Schema data modelling language, TH_{xml}, can be derived from Figure 1 as follows:

$$\begin{aligned}
Types_{xml} &= \{anyType_{xml}, short_{xml}, int_{xml}, nonPositiveInteger_{xml},\\
&\qquad negativeInteger_{xml}, string_{xml}, \ldots\}\\
Ext_{xml} &= \{boolean_{xml} \rightarrow \{0, 1, true, false\},\\
&\qquad short_{xml} \rightarrow \{\text{-32768, } \ldots \text{,32767}\}, \ldots\}\\
\stackrel{t}{=}_{xml} &= \{anyType_{xml} \stackrel{t}{=} string_{xml}\}\\
\prec_{xml} &= \{short_{xml} \prec int_{xml}, unsignedShort_{xml} \prec int_{xml},\\
&\qquad int_{xml} \prec long_{xml}, long_{xml} \prec integer_{xml},\\
&\qquad integer_{xml} \prec string_{xml} \ldots\}\\
\not\frown_{xml} &= \{negativeInteger_{xml} \not\frown positiveInteger_{xml}, \ldots\}\\
Mapping_{xml} &= \{\}
\end{aligned}$$

The Postgres type hierarchy in Example 2 is quite different. The integer and varchar branches of the hierarchy are quite distinct, since converting an integer to a varchar in Postgres requires a mapping function to be used.

Example 2. From the Postgres RDBMS type system illustrated in Figure 2, we derive the type hierarchy TH_{pg} as:

$$
\begin{aligned}
Types_{pg} &= \{anyType_{pg}, boolean_{pg}, bool_{pg}, integer_{pg}, int_{pg}, int4_{pg},\\
&\qquad smallint_{pg}, int2_{pg}, char(n)_{pg}, varchar(n)_{pg}, text_{pg} \dots\}\\
Ext_{pg} &= \{boolean_{pg} \rightarrow \{\text{'0','1','y','n','yes','no','t','f',true,false}\},\\
&\qquad smallint_{pg} \rightarrow \{\text{-32768}, \dots, 32767\}, \dots\}\\
\stackrel{t}{=}_{pg} &= \{boolean_{pg} \stackrel{t}{=} bool_{pg}, int_{pg} \stackrel{t}{=} integer_{pg}, int_{pg} \stackrel{t}{=} int4_{pg}, \dots\}\\
\prec_{pg} &= \{smallint_{pg} \prec integer_{pg}, integer_{pg} \prec bigint_{pg},\\
&\qquad bigint_{pg} \prec anyType,\\
&\qquad varchar(1)_{pg} \prec varchar(2)_{pg}, char_{pg} \prec varchar_{pg},\\
&\qquad varchar_{pg} \prec text_{pg}, text_{pg} \prec anyType_{pg}, \dots\}\\
\not\parallel_{pg} &= \{integer_{pg} \not\parallel varchar_{pg}, boolean_{pg} \not\parallel varchar, \dots\}\\
Mapping_{pg} &= \{\}
\end{aligned}
$$

Example 3. **Type Mapping for Postgres**

During a single model data exchange, it might be found that the boolean type in one Postgres database should map to 0 or 1 of type smallint in another Postgres database. This can be achieved by including the mapping table shown below:

$$
\begin{aligned}
&Mapping_{pg}(boolean_{pg}, smallint_{pg}) = \{\langle boolean_{pg}, int2_{pg},\\
&\quad \{\langle \{\text{'0','n','no','f',false}\}, \{0\}\rangle,\\
&\quad \langle \{\text{'1','y','yes','t',true}\}, \{1\}\rangle\}\rangle\}
\end{aligned}
$$

3.1 The Common Type Hierarchy

If the source and target schemas are defined in different data modelling languages we need a way of linking the type hierarchy defined for the source model, the **source type hierarchy**, to that defined for the target model, the **target type hierarchy**. To do this we introduce the extensible **common type hierarchy** (**CTH**).

Using the common hierarchy as an intermediary means that we only need to create one set of associations between the high level data types of a given model and those of the CTH. We do not need to define new associations each time we are faced with a new target model.

Definition 2. ***Common type hierarchy***

$$
\begin{aligned}
Types_c &= \{anyType_c, string_c, char_c, integer_c, short_c,\\
&\qquad float_c, real_c, boolean_c\}\\
Ext_c &= \{boolean_c \rightarrow \{true, false\}, integer_c \rightarrow \{-2^{31}, \dots, 2^{31}-1\},\\
&\qquad short_c \rightarrow \{-32768, \dots, 32767\}, \dots\}\\
\stackrel{t}{=}_c &= \{\}\\
\prec_c &= \{short_c \prec integer_c, integer_c \prec anyType_c, real_c \prec float_c,\\
&\qquad float_c \prec anyType_c, char_c \prec string_c,\\
&\qquad string_c \prec anyType_c, boolean_c \prec anyType_c\}\\
\not\parallel_c &= \{integer_c \not\parallel boolean_c, integer_c \not\parallel float_c, \dots\}\\
Mapping_c &= \{\}
\end{aligned}
$$

During an inter model data exchange the source and target type hierarchies will be merged with the CTH to form an inter model type hierarchy. Definition 3

gives a procedure for building such an inter model type hierarchy TH_{xc} from a data model's type hierarchy TH_x, and existing inter model type hierarchy TH_{im}, and a (possibly extended) CTH TH_c.

Definition 3. ***Inter model type hierarchy***

Merge(TH_x,TH_{im},TH_c,CM_{xc})
$Types_{xc} := Types_x \cup Types_{im}$
$Ext_{xc} := Ext_x \cup Ext_{im}$
$\not\frown_{xc} := \not\frown_x \cup \not\frown_{im}$
$\prec_{xc} := \prec_x \cup \prec_{im}$
$\stackrel{t}{=}_{xc} := \stackrel{t}{=}_x \cup \stackrel{t}{=}_{im}$
$Mapping_{xc} := Mapping_x \cup Mapping_{im} \cup CM_{xc}$

for_each $t_x \in Types_x$
$t_{xc} := anyType_c$
for_each $t_c \in Types_c$
if $Ext(t_x) \subseteq Ext(t_c) \wedge Ext(t_c) \subseteq Ext(t_{xc})$ then $t_{xc} := t_c$ endif
end
if $Ext(t_x) \subset Ext(t_{xc})$
$t'_c :=$ NewType(t_x)
$Types_c := Types_c \cup \{t'_c\}$
$\prec_c := \prec_c \cup \{t'_c \preceq t_{xc}\}$
$t_{xc} = t'_c$
for_each $t_c \in Types_c$
if $Ext(t_c) \not\frown Ext(t'_c)$ then $\not\frown_c := \not\frown_c \cup \{t_c \not\frown t'_c\}$ endif
end
end
$\stackrel{t}{=}_{xc} := \stackrel{t}{=}_{xc} \cup \{t_x \stackrel{t}{=}_{xc} t_{xc}\}$
end

return $\langle Types_{xc}, Ext_{xc}, \not\frown_{xc}, \stackrel{t}{=}_{xc}, \prec_{xc}, Mapping_{xc}\rangle$

The function NewType(t) *generates a new type t' such that $Ext(t') = Ext(t)$.*

Example 4. **Constructing an inter model type hierarchy**
Assume that XML Schema is our source data modelling language, Postgres is our target modelling language and we use the CTH in Definition 2. We will also assume the following reduced type hierarchies for XML and Postgres:

$TH_{xml} = \langle \{anyType_{xml}, boolean_{xml}, integer_{xml}, negativeInteger_{xml}\},$
$\{integer_{xml} \rightarrow \{-2^{31}, \ldots, 2^{31}-1\},$
$negativeInteger_{xml} \rightarrow \{-2^{31}, \ldots, 0\},$
$boolean_{xml} \rightarrow \{0, 1, true, false\}\}, \{\},$
$\{boolean_{xml} \preceq anyType_{xml}, integer_{xml} \preceq anyType_{xml},$
$negativeInteger_{xml} \preceq integer_{xml}\}, \{\}, \{\}\rangle$

$TH_{pg} = \langle\{anyType_{pg}, boolean_{pg}, int4_{pg}, smallint_{pg}\},$
$\{int4_{pg} \rightarrow \{-2^{31}, \ldots, 2^{31}-1\}, smallint_{pg} \rightarrow \{-32768, \ldots, 32767\},$
$boolean_{pg} \rightarrow \{\text{'0','1','y','n','yes','no','t','f',true,false}\}\}, \{\},$
$\{boolean_{pg} \preceq anyType_{pg}, int4_{pg} \preceq anyType_{pg}, smallint_{pg} \preceq int4_{pg}\},$
$\{boolean_{pg} \not\sqcap int4_{pg}\}, \{\}\rangle$

Using Definition 3 we first extend the CTH with the XML types to get a inter model type hierarchy TH_{im}

$TH_{im} = \mathsf{Merge}(TH_{xml}, TH_c, TH_c,$
$\langle boolean_{xml}, boolean_c, \{\langle\{0,\text{false}\}, \{\text{false}\}\rangle, \langle\{1,\text{true}\}, \{\text{true}\}\rangle\}\rangle)$

As a side effect, TH_c has the following changes:

$Types_c := Types_c \cup \{negativeInteger_c\}$

$\preceq_c := \preceq_c \cup \{negativeInteger_c \prec integer_c\}$

and the following inter model equalities are found in $\stackrel{t}{=}_{im}$:

$\stackrel{t}{=}_{im} = \{integer_{xml} \stackrel{t}{=} integer_c, negativeInteger_{xml} \stackrel{t}{=} negativeInteger_c, \ldots\}$

No additional disjoint relations are added. We can now add in the Postgres type hierarchy to expand the inter model hierarchy to $TH_{im'}$:

$TH_{im'} = \mathsf{Merge}(TH_{pg}, TH_{im}, TH_c, \langle boolean_{pg}, boolean_c,$
$\{\langle\{\text{'0','n','no','f',false}\}, \{\text{false}\}\rangle, \langle\{\text{'1','y','yes','t',true}\}, \{\text{true}\}\rangle\}\rangle)$

No new changes to TH_c occur, but the following additional inter model equalities are found in $\stackrel{t}{=}_{im'}$:

$\stackrel{t}{=}_{im'} = \{int4_{pg} \stackrel{t}{=} integer_c, smallint_{pg} \stackrel{t}{=} short_c, \ldots\}$

Note that the two mapping tables in $TH_{im'}$, $Mapping_{im'}(boolean_{xml}, boolean_c)$ and $Mapping^{-1}_{im'}(boolean_{pg}, boolean_c)$ allow us to map from a boolean in XML to a boolean Postgres.

3.2 Reducing constraint checking

The inter model type hierarchy described above can help us decide whether or not to add constraints. For example $short_{xml}$ and $smallint_{pg}$ are both equivalent to $short_c$. We can say with confidence that any value from a construct of type $short_{xml}$ can be stored in a construct of type $smallint_{pg}$. If we later expanded our system to include schemas from Microsoft SQL Server and discover that $smallint_{ms}$ also maps to $short_c$ we would know that values from all three modelling languages could be safely interchanged without checking.

This method can also highlight when checking is needed. Any time it is necessary to move down the merged inter model hierarchy to transform from one high level type to another we know that checking is necessary. For example assume we have an XML source schema that contains an element of type $integer_{xml}$, equivalent to $integer_c$ and a target of a Postgres column of type $smallint_{pg}$, equivalent to $short_c$. We know from Definition 2 that $short_c \prec integer_c$. To get from $integer_{xml}$ to $smallint_{pg}$ we need to go via $integer_c$ and then down the hierarchy to $short_c$. The checking is necessary here because we know from the definition that $Ext(integer_c) \supset Ext(short_c)$.

It may be necessary to find a common ancestor in the merged inter model hierarchy to be able to move between certain source and target data types.

In Example 4 we added $negativeInteger_c$ to the hierarchy. To transform from $negativeInteger_c$ to $short_c$ we need to go via a type in the merged hierarchy that is an ancestor of both types, in this case $integer_c$. If no such ancestor, other than the root node $anyType_c$, can be found the type transformation is invalid. As with the previous example, the step from $integer_c$ to $short_c$ is down the hierarchy so a constraint is generated.

Finally we can identify illegal castings. If our pathway from source to target includes data types t and t' such that $t \not\sqcap t'$ then the cast is illegal from Definition 1.

4 AutoMed

AutoMed [10] is a data integration and exchange system developed as a joint project between Imperial College London and Birkbeck College. It has been used successfully for data integration of schemas from a number of different data models including both XML and relational schemas.

AutoMed handles multiple data models defining the constructs of a higher level modelling language such as the relational model or XML in a lower level **hypergraph data model** (**HDM**) [11, 12].

Definition 4. ***HDM Schema***
Given a set of $Names$ that we may use for modelling the real world, an HDM **schema**, S, is a triple $\langle Nodes, Edges, Cons\rangle$ where:

- $Nodes \subseteq \{\langle\langle n_n\rangle\rangle \mid n_n \in Names\}$
  i.e. $Nodes$ is a set of nodes in the graph, each denoted by its name enclosed in double chevron marks.
- $Schemes = Nodes \cup Edges$
- $Edges \subseteq \{\langle\langle n_e, s_1, \ldots, s_n\rangle\rangle \mid n_e \in Names \cup \{\_\} \wedge s_1 \in Schemes \wedge \ldots \wedge s_n \in Schemes\}$
  i.e. $Edges$ is a set of edges in the graph where each edge is denoted by its name, together with the list of nodes/edges that the edge connects, enclosed in double chevron marks.
- $Cons \subseteq \{c(s_1, \ldots, s_n) \mid c \in Funcs \wedge s_1 \in Schemes \wedge \ldots \wedge s_n \in Schemes\}$
  i.e. $Cons$ is a set of boolean-valued functions (*i.e.* constraints) whose variables are members of $Schemes$ and where the set of functions $Funcs$ forms the HDM constraint language.

The extent of a node or edge is returned by the function $Ext_{S,I}$ where I is a specific instance of the schema S. Thus $Ext_{S,I}(\langle\langle \mathsf{n}\rangle\rangle)$ returns the values associated with node $\langle\langle \mathsf{n}\rangle\rangle$.

As detailed in [11, 12], the HDM may be used to model a wide range of higher level modelling languages, and we develop the techniques in [12] to handle types in translating between one higher level modelling language and another by using the HDM as a **common data model** (**CDM**).

Using the simple HDM constructs defined above, the extensional constructs of higher level modelling languages can be defined, based on the HDM. In [13],

higher level modelling language constructs were classified into (1) **nodal** constructs, which can exist independently, and map to nodes in the HDM, (2) **link-nodal** constructs, which must be attached to other constructs, but have some additional data, which map to a node and an edge in the HDM, and (3) **linking** constructs, which associate data values in existing constructs, and map onto edges in the HDM. For example, in an ER model, entities are nodal constructs, attributes are link-nodal, and relationships linking constructs. Once the constructs of the new modelling language have been defined in the HDM, we can define transformations that allow us to add, delete and rename these constructs.

4.1 Data Exchange in AutoMed

In AutoMed, data exchange is done in two phases: first the transformation pathways are set up, and second the data values are transformed using those pathways. This is roughly analogous to compile and run time for a conventional program. Setting up the pathways is a static operation whose complexity is linear in the number of constructs to be mapped. It can be done before data from the data source is imported into the system. The complexity of transforming the data values during run-time depends on the data involved and is less easy to quantify.

4.2 Example of Data Exchange without Type Information

Figures 3 and 4 show two schemas modeling the same Universe of Discourse (UoD), namely account debits. We wish to transform the XML Schema in Figure 3 in such a way that we can exchange data between it and the SQL table in Figure 4.

```
⟨xsd:element name="ledger"⟩
  ⟨xsd:complexType⟩
    ⟨xsd:sequence⟩
      ⟨xsd:element name="cash" type="xsd:negativeInteger"/⟩
      ⟨xsd:element name="complete" type="xsd:boolean"/⟩
    ⟨/xsd:sequence⟩
  ⟨/xsd:complexType⟩
⟨/xsd:element⟩
```

Fig. 3. XML Schema fragment 1

On the face of it the transformation is very simple. The values in the complete element in the XML schema will be added to those in the complete column in the relational table, and those in the cash element will be added to the cashIn column. However, this example highlights some problems we may face if we ignore type information when transferring data between XML and SQL.

```
CREATE TABLE money (
  cashIn smallint,
  complete boolean
)
```

Fig. 4. Postgres definition of table money

The XML Schema element cash with type negativeInteger can contain any negative integer whereas the largest negative number the Postgres column cashIn with type smallint is -32678. If the XML file had a value less than -32678 in it, attempting to put that into the Postgres table would cause an error. Similarly if we try to put the boolean value 0 from the complete element into the boolean column complete the operation will fail, since there are XML boolean values that cannot be stored in a Postgres boolean column and we have not specified how to map the XML boolean values into Postgres. These transformations are not type safe.

5 Adding the Type System to AutoMed

To add types to the AutoMed system, we extend the Definition 4 by adding a type hierarchy $TH = \langle Types, Ext, \stackrel{t}{=}, \prec, \emptyset\backslash, Mapping \rangle$ from Definition 1 to make a typed HDM schema be a tuple $\langle Nodes, Edges, Constraints, TH \rangle$. The scheme of each node will have an extra component, $t \in Types$ added, making the scheme of a node be $\langle\langle \mathsf{n}, \mathsf{t} \rangle\rangle$ The extent of a node can now be defined as a subset of the values represented by the node's type so for $x \in \langle\langle \mathsf{n}, \mathsf{t} \rangle\rangle \rightarrow x \in Ext(t)$, *i.e.* $\langle\langle \mathsf{n}, \mathsf{t} \rangle\rangle \in Ext(t)$. Note that edges connect other nodes and edges, and therefore their type can always be inferred from the base nodes they connect.

The primitive node operations described in [11, 12] are modified and expanded on to incorporate the new type information as follows.

1. A new operation getNodeType(q) returns the type, t, of a query q.

2. The primitive transformations add and delete are adapted slightly:
   - addNode($\langle\langle n, t \rangle\rangle, q$) returns a new schema by adding a node n of type t to the schema, where getNodeType(q) $\stackrel{t}{=} t$. The extent of the node is given by the value of the IQL (Intermediate Query Language) [14] query q.
   - deleteNode($\langle\langle n, t \rangle\rangle, q$) returns a new schema by deleting a node from the schema, where the value of the node may be recovered from q, and getNodeType(q) $\stackrel{t}{=} t$

It may be necessary to change the data type of a node during a transformation. We present two new primitive operations to allow this. These changes are done on the inter model type hierarchy.

Definition 5. changeNodeType($\langle\langle n, t_{old}\rangle\rangle, \langle\langle n, t_{new}\rangle\rangle$)
Condition: $Ext(t_{old}) \cap Ext(t_{new}) \neq \phi$, returns a new schema which differs only in that node $\langle\langle$n$\rangle\rangle$ has a different type. If $t_{old} \not\prec t_{new}$ this operation will generate the following constraint used during query processing:
$\forall\langle x\rangle \in \langle\langle \mathsf{n}, \mathsf{t_{old}}\rangle\rangle.x \in Ext(t_{new})$.

Definition 6. convertNodeType($\langle\langle n, t_{old}\rangle\rangle, \langle\langle n, t_{new}\rangle\rangle, Mapping_{im}(t_{old}, t_{new})$) defines a bidirectional type conversion from a node of type t_{old} to type t_{new}. Each value in the extent of $\langle\langle \mathsf{n}, \mathsf{t_{old}}\rangle\rangle$ is mapped to a value in the extent of t_{new} using the mapping tables and function from Definition 1.

This section has shown how our formalism can be applied to the AutoMed system by the addition of primitive operations to manipulate the types of the nodes in a schema. However, the method in Section 3 could equally well be applied to other data exchange systems.

6 AutoMed Transformation with Data Types

We can now define an AutoMed transformation pathway between the XML and SQL schemas from Section 4.2 using the operators defined above. The pathway is shown in Example 5.

The initial data type of each node matches the data type of the corresponding data source object. For example if we add a node representing the Postgres column complete from Figure 4 with type boolean, a node $\langle\langle \mathsf{complete}, \mathsf{boolean_{pg}}\rangle\rangle$ will be created using the addNode operator.

The extents of the high level model types are the set of allowable values as defined by the data source. For example the extent of $\mathsf{short_{xml}}$ as defined by the XML Schema standard in [9] will be the integers from -32768 to 32767.

We merge TH_{pg} of Example 2 and TH_{xml} of Example 1 with the CTH as shown in Example 4 to form $TH_{im'}$.

Example 5. **Type safe transformation pathway**
① changeNodeType($\langle\langle \mathsf{cash_{xml}}, \mathsf{negativeInteger_{xml}}\rangle\rangle, \langle\langle \mathsf{cash_{xml}}, \mathsf{negativeInteger_c}\rangle\rangle$)
② convertNodeType($\langle\langle \mathsf{complete_{xml}}, \mathsf{boolean_{xml}}\rangle\rangle$,
$\langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_c}\rangle\rangle, \mathsf{Mapping_{im'}}(\mathsf{boolean_{xml}}, \mathsf{boolean_c})$)
③ addNode($\langle\langle \mathsf{cashIn_{pg}}, \mathsf{negativeInteger_c}\rangle\rangle, \langle\langle \mathsf{cash_{xml}}, \mathsf{negativeInteger_c}\rangle\rangle$)
④ addNode($\langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_c}\rangle\rangle, \langle\langle \mathsf{complete_{xml}}, \mathsf{boolean_c}\rangle\rangle$)
⑤ deleteNode($\langle\langle \mathsf{complete_{xml}}, \mathsf{boolean_c}\rangle\rangle, \langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_c}\rangle\rangle$)
⑥ deleteNode($\langle\langle \mathsf{cash_{xml}}, \mathsf{negativeInteger_c}\rangle\rangle, \langle\langle \mathsf{cashIn_{pg}}, \mathsf{negativeInteger_c}\rangle\rangle$)
⑦ convertNodeType($\langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_c}\rangle\rangle$,
$\langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_{pg}}\rangle\rangle, \mathsf{Mapping_{im'}}(\mathsf{boolean_{pg}}, \mathsf{boolean_c})$)
⑧ changeNodeType($\langle\langle \mathsf{cashIn_{pg}}, \mathsf{negativeInteger_c}\rangle\rangle, \langle\langle \mathsf{cashIn_{pg}}, \mathsf{smallint_{pg}}\rangle\rangle$)

Two nodes $\langle\langle \mathsf{cash_{xml}}, \mathsf{negativeInteger_{xml}}\rangle\rangle$ and $\langle\langle \mathsf{complete_{xml}}, \mathsf{boolean_{xml}}\rangle\rangle$ representing the XML Schema elements along with their data types are created by the wrapping process. Before adding nodes to represent the Postgres columns

we change the type of the XML node to its equivalent in the CTH ① because we have $negativeInteger_{xml} \stackrel{t}{=} negativeInteger_c$. ② uses the mapping defined in Example 4.

In transformations ⑦ and ⑧ we convert the data types of the nodes from the HDM to the Postgres model. In ⑦ we change the type of $\langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_c} \rangle\rangle$ using the convertNodeType operation. We again use the mapping defined in Example 4, but this time the inverse of the function to map the values of the CTH boolean to the Postgres boolean. This overcomes the incompatibility of the XML Schema boolean and Postgres boolean data types and solves the second problem from the example in Section 4.2.

Finally in ⑧ we change the type of node $\langle\langle \mathsf{cashIn_{pg}}, \mathsf{negativeInteger_c} \rangle\rangle$ using changeNodeType. $smallint_{pg} \stackrel{t}{=} short_c$ but $short_c \not\prec negativeInteger_c$ so we need to find a common ancestor. The common ancestor is $integer_c$. Since we have to move down the hierarchy to get from $integer_c$ to $short_c$ the constraint:

$\forall x \in \langle\langle \mathsf{cash_{xml}}, \mathsf{negativeInteger_{xml}} \rangle\rangle . x \in Ext(smallint_{pg})$

is generated from Definition 5. The transformation is legal because there are no disjoint pairs of types in the pathway. We can check the constraint as we transfer data into the SQL table and so be guaranteed that we will not get any errors from the database. This solves the first problem we had in the example in Section 4.2.

After transformations ①–⑧ we have the nodes $\langle\langle \mathsf{cashIn_{pg}}, \mathsf{smallint_{pg}} \rangle\rangle$ and $\langle\langle \mathsf{complete_{pg}}, \mathsf{boolean_{pg}} \rangle\rangle$, that correspond to the columns in the Postgres table in Figure 4.

Using the type hierarchy has allowed us to identify a mapping between the XML and Postgres boolean data types. This was done automatically at compile-time, based on the types of the nodes involved. This could not have been done without using the type information. We have also identified a potential type casting problem that will need to be checked during run-time. Without the explicit identification of the latter problem a system would either need to check every transformation for type safety during the data value exchange or adopt a no-checking policy that could lead to unexpected problems.

7 Related Work

A number of papers have shown how to transform schemas from one model to another, most commonly XML to relational [15–17]. There are also systems that will exchange data between a number of different models [18, 12]. AutoMed falls into this category. Transforming integrity constraints from one model to another has also received some attention [3]. There does not, however, seem to have been much written about exchanging primitive type information between different models and in particular manipulating that information during the transformation process.

Some methods ignore the problem [15, 7] and make no explicit mention of how data type from one model are transformed into the other model. Rahm and Bernstein [6] suggest using a special synonym table to match data types

between different models to each other, an approach they adopt in Cupid [19]. This method is effective when mapping between two specific models but does not scale well to a system like AutoMed that supports multiple models.

A number of systems provide support for data types. TSIMMIS [20] allows data types to be stored as part of their Object-Exchange Model [21] but there is no mechanism to manipulate the data types. WOL [22] 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 [1, 23] 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 [2] and allowing type-incompatible transformations to be written. We have shown that there are times when data types are significant and should not be ignored.

8 Conclusions and Future Work

We have presented a method of improving the expressiveness of inter model data exchange between models that have constructs with associated data types. The method relies on converting the types from the source into a common type hierarchy capable of representing types from any data source, and from there into types from the target schema.

Types can be cast from one type to another and mappings can be defined between disjoint types. A formal definition of the type system has been provided and it has been shown how this can be included in the existing AutoMed system. An example inter model transformation with and without using the type system was presented to show the advantages of using the type system.

The type hierarchy described above has been implemented in AutoMed and has been used for data exchange from source schemas in a number of different models to an empty target schema in the relational and XML Schema models.

In the future we will extend the use of the mapping function to single model data exchange where for instance one Postgres database represents boolean values as single characters 't' and 'f' of type `char(1)` while another database may use a field of type `boolean`. We also hope to increase the efficiency of the method by defining direct transformation rules between certain well-known data models, bypassing the need to transform each data type into its HDM equivalent first.

References

1. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: Semantics and query answering. In: ICDT. (2003) 207–224
2. Atkinson, M.P., Buneman, P.: Types and persistence in database programming languages. ACM Comput. Surv. **19**(2) (1987) 105–190

3. Davidson, S.B., Fan, W., Hara, C., Qin, J.: Propagating XML Constraints to Relations. In: ICDE. (2003) 543–554
4. Calì, A., Calvanese, D., Giacomo, G.D., Lenzerini, M.: Data integration under integrity constraints. Inf. Syst. **29**(2) (2004) 147–163
5. Cardelli, L.: Type systems. ACM Comput. Surv. **28**(1) (1996) 263–264
6. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB J. **10**(4) (2001) 334–350
7. C. Liu, M. Vincent, J.L., Guo, M.: A virtual XML database engine for relational databases. XSYM 2003 (2003)
8. Henry S. Thompson et al. Eds: XML Schema part 1: Structures. http://www.w3.org/TR/xmlschema-1 (2001)
9. Paul V. Biron et al. Eds: XML Schema part 2: Datatypes second edition. http://www.w3.org/TR/xmlschema-2 (2004)
10. M. Boyd, S. Kittivoravitkul, C. Lazanitis, P.J. McBrien and N. Rizopoulos: AutoMed: A BAV Data Integration System for Heterogeneous Data Sources. In: CAiSE04. Volume 3084 of LNCS., Springer Verlag (2004) 82–97
11. McBrien, P., Poulovassilis, A.: A general formal framework for schema transformation. In: Data and Knowledge Engineering. Volume 28. (1998) 47–71
12. Boyd, M., McBrien, P.: Comparing and transforming between data models via an intermediate hypergraph data model. J. Data Semantics IV (2005) 69–109
13. McBrien, P., Poulovassilis, A.: A semantic approach to integrating XML and structured data sources. In: Advanced Information Systems Engineering. Volume 2068 of LNCS., Springer Verlag (2001) 330–345
14. Poulovassilis, A.: A tutorial on the IQL query language. AutoMed Technical Report http://www.doc.ic.ac.uk/automed/ (2004)
15. 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)
16. Ferández, M., Tan, W.C., Suciu, D.: Silkroute: Trading between relations and XML. In: Proceedings of the Ninth International World Wide Web Conference. (2000)
17. Jayavel Shanmugasundaram et al.: A general techniques for querying XML documents using a relational database system. SIGMOD Record **30**(3) (2001) 20–26
18. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. **30**(1) (2005) 174–210
19. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: VLDB. (2001) 49–58
20. Garcia-Molina, H., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J.D., Vassalos, V., Widom, J.: The tsimmis approach to mediation: Data models and languages. J. Intell. Inf. Syst. **8**(2) (1997) 117–132
21. Papakonstantinou, Y., Garcia-Molina, H., Widom, J.: Object exchange across heterogeneous information sources. In Yu, P.S., Chen, A.L.P., eds.: 11th Conference on Data Engineering, Taipei, Taiwan, IEEE Computer Society (1995) 251–260
22. Susan Davidson and A. Kosky: WOL: A Language for Database Transformations and Constraints. In: Proceedings of the International Conference of Data Engineering. (1997) 55–65
23. Miller, R.J., Haas, L.M., Hernández, M.A.: Schema mapping as query discovery. In: VLDB. (2000) 77–88