Ceda Data Model

David Barrett-Lennard

16 Oct 2016

# The Ceda Data Model

The Ceda Data Model (CDM) refers generally to the form of data modelling used in ceda.

A CDM schema refers to a particular schema (a set of ceda type definitions).

A CDM database is a ceda database that conforms to some CDM schema. A Ceda Database Management System (CDBMS) refers generally to a DBMS that supports the CDM.

The CDM is motivated by the following

* It is a pure data model, meaning that it concerns the modelling of encoded mathematical values, and not the modelling of Finite State Machines (FSMs) in the manner of OO.
* It supports a simple and direct mapping to data structures in C++
* There is an emphasis on static typing
* There is an emphasis on efficient binary encoding of data
* Compatibility with Operational Transform (OT), and unconstrained concurrent editing, branching and merging of the entire database.
* Indirect support for strong integrity constraints.
* Compatibility with storing the data in a multi-terrabyte Persistent Object Store (POS).
* Support for semi-structured data, such as needed for text markup.
* Supporting data models that encode the information in an abstract fashion that is decoupled from downstream processing or presentation.

# Reflection

Typically the CDM data types are reflected, meaning there is data recorded about the type definition which is accessible at run time. This has many purposes in the system, for example it is needed to support persistence, replication and bindings to other languages such as Python.

The reflection system is described in [1].

# Use of Xcpp

Currently a CDM schema must be specified in eXtended C++, meaning Xcpp is needed to translate the type definitions into C++ code.

The build system for Xcpp is described in [2].

Typically the Xcpp language extensions involve a ’$’ prefix character. For example there are keywords like $enum, $functor, $function, $class, $struct, $model, $interface, $mixin and $typedef.

Typically a ‘+’ character is used to enable both exporting from the library (so it is accessible to other libraries) and reflection.

# CDM schema

The available types are

|  |  |
| --- | --- |
| bool | Boolean type with the possible values false / true |
| int8 | Signed 8 bit integer. Range: -27 to 27-1 |
| int16 | Signed 16 bit integer . Range: -215 to 215-1 |
| int32 | Signed 32 bit integer. Range: -231 to 231-1 |
| int64 | Signed 64 bit integer. Range: -263 to 263-1 |
| uint8 | Unsigned 8 bit integer . Range: 0 to 28-1 |
| uint16 | Unsigned 16 bit integer. Range: 0 to 216-1 |
| uint32 | Unsigned 32 bit integer. Range: 0 to 232-1 |
| uint64 | Unsigned 64 bit integer. Range: 0 to 264-1 |
| float32 | 32 bit (single precision) floating point number |
| float64 | 64 bit (double precision) floating point number |
| char8 | For UTF-8 encodings of strings |
| char16 | For UTF-16 encodings of strings |
| string8 | Equivalent to xvector<char8> |
| string16 | Equivalent to xvector<char16> |
| T[size] | Fixed size array of mutable elements of type T. The size is fixed in the schema. |
| T[] | Similar to T[size] except that the initial size is specified when the instance is first constructed. |
| cref<C> | A pointer to an object of class C |
| pref<I> | A pointer to an object through an interface I |
| movable<cref<C>> or  movable<pref<I>> | A cref<C> or pref<I> with non-lossy movement (instead of assignment semantics which is the default) under OT |
| assignable<T> | Designates that the given type T is treated atomically and therefore only supports assignment operations on the entire variable. |
| offsetable<T> | Designates that the given integral type T has delta operation semantics under OT.  The integral types can be made to support delta semantics (instead of default which is assignment semantics) under OT. This means that positive or negative offsets to the value are accumulated (modulo 2n where n depends on the size of the integral type).  Note that floating point types do not support delta semantics because it is impossible to guarantee convergence at quiescence under OT due to round off errors. |
| xvector<T> | A dynamically resizeable array using a zero based index for *immutable elements* of type T.  Since the elements are immutable the only supported mutative operations are insertions and deletions on the vector itself.  At quiescence all sites will converge to the document state containing the union of all insertions minus the union of all the deletions. The only remaining ambiguity is the order for concurrent insertions at exactly the same position. This ordering is resolved using the total ordering on the site identifiers. By keeping the union of all insertions, these operations on the vector aren’t lossy. |
| xset<T> | A dynamically resizable set of *immutable elements* of type T.  Since the elements are immutable the only supported operations are insertions and deletions on the set. The elements do not have identity so therefore move semantics is not supported.  Under OT we get the unique result achieved by first removing the union of all the deletions then inserting the union of all the insertions. Note therefore that insertions take precedence over deletions. |
| xmap<K,V> | map from immutable key of type K to mutable element of type V |

# Schema evolution

Schema evolution refers to changes to the CDM schema over successive releases of the software system, which in principle can require all kinds of changes to the defined data types. This must be done carefully in a large distributed system to be sure the changes are compatible with the existing databases. Methodologies to allow for schema evolution are discussed in this document.

A goal of the supported schema evolution is to allow sites to be independently upgraded with new versions of software. It is probably impractical to atomically upgrade a large distributed system.

# Interfaces

The following declares an interface named ‘Shape’ with two polymorphic methods

$interface+ Shape

{

float64 GetArea() const;

float64 GetPerimeter() const;

};

Interfaces are described in [3]. They are relevant to persistent data types for the purpose of defining static types of pointers to objects that persist in the database. For example a pref<Shape>means a pointer to an object that implements the Shape interface.

There are two important interfaces named IObject and IPersistable. The latter must be implemented by objects that can persist in the database.

# Enumerated types

An enumerated type is specified in a similar manner to a C/C++ enumerated type. For example

$enum+ class FontStyle

{

Normal,

Oblique,

Italic

};

defines an enumerated type named FontStyle. The identifiers Normal, Oblique and Italic are called tags. In C++ code an expression such as FontStyle::Oblique is a literal which denote a value of type FontStyle.

By default the first tag appearing in the definition is the default value for a variable of type FontStyle. So the following code won’t trip the assertion.

FontStyle f1; // default constructor

assert(f1 == FontStyle::Normal);

The default value can be overridden using the ‘default’ keyword. For example

$enum+ class FontStyle

{

Normal,

default Oblique,

Italic

};

A variable of type FontStyle can be constructed with a given value. For example:

FontStyle f2(FontStyle::Oblique); // copy constructor

A variable of type FontStyle can be assigned a given value. For example:

f1 = FontStyle::Oblique; // assignment

FontStyle::LEN is a compile time constant which equals the number of tags (in this case 3). It can for example be used to declare the size of an array.

int A[FontStyle::LEN]; // array A has size 3

The tags of an enumerated type have associated integer values that by default begin at 0 for the first tag that appears in the definition and increment from there. An enumerated type supports an implicit conversion to type int. For example

int x = FontStyle::Oblique; // Same as setting x = 1

All 6 comparison operations (< <= == != > >=) are defined on an enumerated type according to the integral values of the tags. E.g.

assert(FontStyle::Normal < FontStyle::Oblique);

Values of the tags can be specified. For example:

$enum+ class FontWeight

{

Thin = 100,

Ultralight = 200,

Light = 300,

Book = 380,

default Normal = 400,

Medium = 500,

Semibold = 600,

Bold = 700,

Ultrabold = 800,

Heavy = 900,

Ultraheavy = 1000

};

### Schema evolution of enumerated types

The expected way to allow for schema evolution is to:

* Only append new tags at the end of the enumerated type definition
* Never delete tags, only mark them as deprecated

This ensures the values of tags do not change over time, and also because tags are never deleted they are available when dealing with old versions of objects in the database.

A tag is marked as deprecated with a minus character. For example

$enum+ class FontStyle

{

Normal,

-Oblique,

Italic

};

# Models

A model means a data structure which is directly mapped to a corresponding C/C++ struct. For example

$model+ Point

{

float32 x,y;

};

$model+ Circle

{

Point centre;

float32 radius;

};

$model+ Triangle

{

Point vertices[3];

};

$model+ Rectangle

{

float32 x1,y1,x2,y2;

};

### Model schema evolution

There is support for schema evolution of a model by allowing for attributes to be added or dropped over time. This information is related back to release numbers of the library in which the model definition appears.

Remarkably schema evolution can be deployed on sites incrementally – i.e. without needing to upgrade all sites at the same time (which may not be practical in large complex systems).

Here is a rather contrived example where a z coordinate was added, then dropped again and then the representation changed from Cartesian to polar coordinates:

$model+ Point

{

float64 r;

float64 theta;

$dropped

{

float64 x;

float64 y;

float64 z;

}

$schema

{

1: +x +y // Release #1 : (x,y)

2: +z // Release #2,3 : (x,y,z)

4: -z // Release #4 : (x,y)

5: -x -y +r +theta // Release #5- : (r,theta)

}

$evolve

{

r(x,y) { r = sqrt(x\*x + y\*y); }

theta(x,y) { theta = atan2(y,x); }

}

};

### Model constructors

A model may contain a constructor allowing for initialisation of the members. For example

$model+ Point

{

float64 x;

float64 y;

// constructor, has C++ syntax except $$ is used for

// the class name

$$() :

x(0.0),

y(0.0)

{

}

};

# Metadata on types

Xcpp supports embedding metadata in reflected types as described in [1]. The meaning of the metadata is application defined.

# Typedefs

As in C/C++ a typedef provides a convenient alias for a type expression. For example

$typedef+ float32 radiosity\_t : [range(0,1), fixed, precision(3)];

$model+ ColourRGB

{

radiosity\_t R;

radiosity\_t G;

radiosity\_t B;

};

$typedef+ assignable<ColourRGB> Colour;

# Namespaces

The normal C++ syntax can be used to define namespaces for the various type definitions. For example

namespace geom

{

$enum+ class Colour

{

red,

green,

blue

};

$model+ Point

{

float32 x,y;

};

} // namespace geom

## Immutable elements

The elements of an xvector<T> or xset<T> variable don’t in turn behave like variables because they are immutable and are never the target of operations. An xvector<T> or xset<T> only changes through insert and delete operations, not through operations on the elements themselves.

This is also the case for the keys (but not the mapped values) of an xmap<K,V>.

If it seems more appropriate for the element type T of an xvector<T> or xset<T> to be the target of update operations then the solution is to instead introduce an indirection with a pointer to an allocated object. It is common and useful for the elements of an xvector or xset to be immutable pointers to mutable objects.

## Objects

The data model encompasses an Object Oriented Database (OODB) by allowing for persistent objects to be created which reference each other with pointers. However, the pointers are not intended to be used to create arbitrary graphs. Instead the objects must form a hierarchical structure. This is interpreted as a means to represent values

An object references another object using a cref<C> where C is a class or a pref<I> where I is an interface.

There are two supported semantics on these reference types. By default they have assignment semantics, which means an assignment to the cref/pref replaces the previous object with a new object. The previous object is lost. Objects are also lost under merging of concurrent assignments.

The alternative is move semantics. An operation represents a re-parenting of an object, i.e. the transfer of a child object from the old parent to a new parent. This operation is non-lossy, even when merging concurrent operations.

A parent object references movable child objects using one of the following

* xvector<movable<cref<C>>>
* xvector<movable<pref<I>>>
* xset<movable<cref<C>>>
* xset<movable<pref<I>>>

The objects in these collections can be the target of operations independently of the parent, and under merging operations that target an object are unaffected by reparenting.

In this sense move operations allow moveable elements to be moved between different collections without losing their identity. Note that an element can be moved from either an xvector or xset to either an xvector or xset. The only constraint is that the collection type of the destination collection is compatible with the type of object.

Since moveable elements have identity, operations on objects are decoupled from operations that move the objects. This orthogonality helps to preserve the intentions of concurrent edits when they are merged.

### Implementation note

In the physical implementation the objects are always heap allocated and persist in the Log Structured Store with uniquely assigned OIDs. Therefore move operations are very efficient because they only involve movement of OID values between collections. Note that OIDs are always moved and not copied so it is never possible for two collections to reference or alias the same element.

This physical implementation also allows for lazy loading of the objects. ie these elements are loaded from disk on demand. Also they can be evicted on an LRU basis (say) to make appropriate use of system memory. This allows for efficiently processing of multi-terabyte databases. Furthermore in certain cases these elements can be downloaded on demand over the wire.

## Variants

### Closed variants

In the literature a variant is also referred to as a tagged union or discriminated union. See for example the Wikipedia article <http://en.wikipedia.org/wiki/Tagged_union>

A C++ programmer can think of a variant as combining the features of an enum and a union.

A (closed) variant is a value-type that can represent exactly one value from a specified set of value-types. A closed variant definition is very similar to a tuple definition except that instead of defining a set of attribute-type + attribute-name pairs, it defines a set of tag-type + tag-name pairs. Furthermore unlike a tuple definition a tag-type is optional.

For example

$variant+ Shape

{

Circle circle;

Triangle triangle;

Rectangle rectangle;

};

A value of type Shape can represent either a circle, triangle or rectangle value. The identifiers ‘circle’, ‘triangle’ and ‘rectangle’ are tag-names and any valid C/C++ identifier may be used. The types of the tag-names (Circle, Triangle and Rectangle) are called tag-types.

The tag-type declaration that precedes a given tag-name is optional. For example, the following example is equivalent to a simple enumerated type in C/C++. There is no additional state associated with any of the tags.

$variant+ PrimaryColour

{

red;

green;

blue;

};

It is straightforward to map a variant into an efficient grammar for a binary encoding. Serialisation of a variant involves writing the tag (eg using an int32) then the additional state for that tag if any. Note that the serialisation is very efficient.

Schema evolution is supported in a similar fashion to tuples. Conceptually new tags may only be added to the end, and dropped tags are only marked as dropped. [todo: arguably tags can never be dropped]

Anonymous tuple definitions are often useful within variant definitions. For example, Shape could instead be written as follows

$variant+ Shape

{

tuple { Point<float32> centre; float32 radius; } circle;

tuple { Point<float32>[3] vertices; } triangle;

tuple Rectangle { float32 x1,y1,x2,y2; } rectangle;

};

## Arrays

Arrays are fixed in size and therefore are not compatible with moveable elements (that can only appear in the dynamic collections mvector and mbag).

Since arrays do not support insert or erase operations the array elements are always located at the same index position. Therefore under OT there is no need to transform array index positions.

A variable of type T[] has subvariables of type T for each of its elements. By contrast an atomic<T[]> has no subvariables.

Arrays can nest arbitrarily. For this reason multidimensional arrays are achieved using nested one dimensional arrays.

The size of an array variable can be specified statically (ie in the CDM schema) or else it can be specified at run time at the point when the variable is first created. Once the array variable is created the array size is immutable.

# References

|  |  |
| --- | --- |
| [1] | Xcpp Reflection |
| [2] | Xcpp Build System |
| [3] | Xcpp Interfaces |
| [4] | Xcpp Macro Preprocessor |
| [5] | Xcpp Mixins |