C/C++ Application Programmer Interface for HDDM

Richard Jones and R.T. Jones
University of Connecticut
(Dated: November 1, 2010)

The Hall D data model (HDDM) is a language for specifying a data model for scientific data. The design is based on a hierarchical relationship graph where each node in the graph has a single parent node, multiple attributes, and an arbitrary number of child nodes, similar to a the elements in an xml document. The model is adapted specifically to the case of repetitive data models such as appear in the data stream from a high-energy physics experiment. The representation of the model in xml is an essential feature, although instantiation in memory does not involve the creation of explicit textual elements or a document object model (DOM). The HDDM toolkit includes tools to browse HDDM files in xml, check their validity, and serialize/deserialize them to data streams. The serialization library API is the subject of this article. Originally written in c, a new library in C++ is now being made available with significant extensions to the API. The motivation behind these extensions are explained and illustrated with use examples.

I. BASIC DESIGN

HDDM was designed to fill the need for a general i/o library to store and transform numerical data that are described by a flexible data model written in xml. The data model is a collection of metadata describing the meaning and structure of data, such as results from a scientific experiment or simulation. The metadata describes how the data are related to each other using a hierarchical graph whose nodes have attributes consisting of key-value pairs of numerical or string data, and connect to other nodes through a directed acylical graph.

The purpose of the library is to provide an API for users to be able to create in-memory instances of the data model in a way that guarantees conformity to the constraints of the model, and to serialize/deserialize the data in machine-independent self-describing byte streams. The hddm package provides tools for rendering hddm files in human-readable xml, for creating and managing xml data models, and for generating the libraries that support a fairly complete API for creating, reading, and writing hddm files.

II. C LANGUAGE API

The c language api has been in use for several years now for simulation data by the GlueX collaboration. It supports the basic data types listed in Table I as attributes. The names of the types have been chosen to coincide with the equivalent xml schema simpleTypes so that the data model can be easily written in terms of a xml schema against which any hddm file can be validated using a standard xml document validator. This correspondence with xml scheme types was not taken as an *a priori* requirement of the data model, but examination of the simpleType list indicated that these data types are reasonably complete and well-suited for the purposes of representing scientific data. The enumerated type `Particle_t` is an obvious extension to the basic set.

TABLE I: Basic data types supported by the c API.

type string	data type
`int`	32-bit signed integer
`long`	64-bit signed integer
`float`	IEEE 754 binary32 floating-point number
`double`	IEEE 754 binary64 floating-point number
`boolean`	0 (false) or 1 (true)
`string`	ordered sequence of UTF-8 bytes
`anyURI`	string conforming to the URI schema (see RFC 3305)
`Particle_t`	string literal found in the dictionary of particle names

```
<?xml version="1.0" encoding="UTF-8"?>
<HDDM class="x" version="1.0" xmlns="http://www.gluex.org/hddm">
  <student name="string" minOccurs="0">
    <enrolled year="int" semester="int" maxOccurs="unbounded">
      <course credits="int" title="string" maxOccurs="unbounded">
        <result grade="string" pass="boolean" />
      </course>
    </enrolled>
  </student>
</HDDM>
```

FIG. 1: Example of simple hddm data model template.

The c API consists of *makers* and an i/o suite. The makers are functions that construct an instance of the data model in memory and create new nodes to be added to it. Attaching nodes to a model instance and adding data to the nodes is done by the user by modifying the structures returned by the makers. How this works is clarified by the example in Fig. ??. The i/o suite consists of the following set of functions that provide facilities for reading, writing, and creating hddm files. In the following description, the symbol $ is used in the place of the user-defined identifier of the hddm family of related data models known as a *class.*

The meaning of most of these calls should be clear. The difference between `open_$_HDDM()` and `init_$_HDDM()` is that the former opens an existing hddm file for input, whereas the latter initializes a new hddm file for output, possibly overwriting an existing file of the same name. It is conventional, although not required, that hddm files end with the suffix *.hddm.* The custom type `$_iostream_t` is defined in the `hddm_$.h` header file that must be included in the user's c code in order to use the API.

The `read_$_HDDM()` function inputs the next fragment of the model data from the (previously opened with `open_$_HDDM()`) input hddm file and returns it in an in-memory data structure that can be directly accessed by the user code. If there is only one top-level element permitted in the data model (under the HDDM document element) then the entire file is read into memory at once, otherwise only the next instance of the top-level element together with its contents is read. Physics experimental data is typically modeled with repeating instances of a top-level object known as an *event.* If the data model allows for multiple instances of the top-level element then multiple calls to `read_$_HDDM()` are required to entirely read a hddm file.

The `flush_$_HDDM()` function frees the memory associated with the existing data model structure in memory. It is typically called between sequential reads in a processing loop in the user code. The `flush_$_HDDM()` function also doubles as a write function; when it is called with a non-zero *$_iostream_t** argument then the in-memory data structure is written to the (previously opened with `init_$_HDDM()`) output stream before being freed. It returns zero on success. The `skip_$_HDDM()` call allows events in the input file to be skipped without the overhead of constructing their representation in memory. The `$_iostream_t` i/o semantics are those of a stream, so `skip_$_HDDM()` should only be called with a positive skip count *nskip.* It returns the number of events successfully skipped.

A. c API data access semantics

The in-memory representation of the data model implemented by the hddm c-language API consists of nodes as independent data structures linked together using pointers. The data structure for each node consists of members with elementary types for the attribute data if any, followed by pointers to the child nodes if any. For node types that can appear multiple times inside a single parent node, list structures are created which consist of a repeat count followed by a list of pointers to the list elements. The construction of a populated data model instance is best illustrated by an example. Consider a simple data model described by the hddm template in Fig. 1.

The following code snippet could be used to create a brief transcript for one student, and write it to an output hddm file.

Running this program and then viewing the output file tscript.hddm with the tool hddm-xml produces the following printable view of the model document.

Note that string and anyURI attribute values must be malloc'ed before they can be inserted into the model data structures. They are automatically freed by the `flush_x_HDDM()` call. The remaining types are fixed-length and do not need space on the heap. The names of the structure members closely follow the names in the data model. Differences occur when an element can appear a variable number of times under

```
#include "hddm_x.h"

int main()
{
   x_iostream_t* fp;
   x_HDDM_t* tscript;
   x_Student_t*  student;
   x_Enrolleds_t* enrolleds;
   x_Courses_t* courses;
   x_Result_t* result;
   string_t name;
   string_t grade;
   string_t course;

   // first build the complete nodal structure for this record
   tscript = make_x_HDDM();
   tscript->student = student = make_x_Student();
   student->enrolleds = enrolleds = make_x_Enrolleds(99);
   enrolleds->mult = 1;
   enrolleds->in[0].courses = courses = make_x_Courses(99);
   courses->mult = 3;
   courses->in[0].result = make_x_Result();
   courses->in[1].result = make_x_Result();
   courses->in[2].result = make_x_Result();

   // now fill in the attribute data for this record
   name = malloc(30);
   strcpy(name,"Humphrey Gaston");
   student->name = name;
   enrolleds->in[0].year = 2005;
   enrolleds->in[0].semester = 2;
   courses->in[0].credits = 3;
   course = malloc(30);
   courses->in[0].title = strcpy(course,"Beginning Russian");
   grade = malloc(5);
   courses->in[0].result->grade = strcpy(grade,"A-");
   courses->in[0].result->pass = 1;
   courses->in[1].credits = 1;
   course = malloc(30);
   courses->in[1].title = strcpy(course,"Bohemian Poetry");
   grade = malloc(5);
   courses->in[1].result->grade = strcpy(grade,"C");
   courses->in[1].result->pass = 1;
   courses->in[2].credits = 4;
   course = malloc(30);
   courses->in[2].title = strcpy(course,"Developmental Psychology");
   grade = malloc(5);
   courses->in[2].result->grade = strcpy(grade,"B+");
   courses->in[2].result->pass = 1;

   // now open a file and write this one record into it
   fp = init_x_HDDM("tscript.hddm");
   flush_x_HDDM(tscript,fp);
   close_x_HDDM(fp);

   return 0;
}
```

FIG. 2: Example of simple hddm data writer using the c api.

```
<?xml version="1.0" encoding="UTF-8"?>
<HDDM class="x" version="1.0" xmlns="http://www.gluex.org/hddm">
  <student name="Humphrey Gaston">
    <enrolled semester="2" year="2005">
      <course credits="3" title="Beginning Russian">
        <result grade="A-" pass="1" />
      </course>
      <course credits="1" title="Bohemian Poetry">
        <result grade="C" pass="1" />
      </course>
      <course credits="4" title="Developmental Psychology">
        <result grade="B+" pass="1" />
      </course>
    </enrolled>
  </student>
</HDDM>
```

FIG. 3: Output from the simple example hddm data writer.

a particular parent, in which case a container structure is created to hold the list of daughters of the given type. The name of the list container is a pluralized form of the name of the daughter element.

Reading structures from a hddm file is illustrated by the following example, which reads from the file created by the example program above, and produces a summary.

III. C++ LANGUAGE API

There is interest in porting the hddm c-language API to c++ for several reasons. Any user wanting to use hddm structures in a c++ program would first wrap the hddm functions in classes, and then create objects from those classes that create and access information in the data model. The c++ API saves the user the trouble by providing the wrapper classes itself, with a basic set of creator and accessor methods that can be extended by the user's derived classes. The data encapsulation features of c++ can be exploited to better ensure the conformance of the data to the model than was possible with c, subject to the constraints of efficiency. Semantics for access to data in the model structures in c seems overly complex to programmers who are used to the traditional semantics of flat lists and tables. With the c++ API we offer alternative semantics using iterators that are much closer to those for flat tables, in addition to those provided by the c API.

One thing that can give a programmer a headache in accessing hddm data structures through the c API is the need to contruct deeply nested sets of loops to iterate over all of the instances of a given element. In the above example code that scans the data in transcript.hddm to add up the total number of credits earned by a student, a doubly-nested loop was needed. For more complex models, the nesting level can easily grow to as many as a dozen, which makes writing readable code more difficult. This comes about because of the hierarchical design of hddm data modeling which provides a top-down view of the data; in order to iterate over the leaves in the tree one must iterate over all of the branches that contain them. Traditional list processing takes the opposite bottom-up view of the data by treating all of the leaves of a given type as members of a single container, and using index tables to look up the parent attributes of a given leaf. The hddm c++ API introduces intelligent iterators that support a bottom-up view of the leaves in a hddm model structure by automatically walking the tree and finding all of the instances of a given element in a structure.

Unlike the c API where an element structures only carries information about its descendants, the element objects of the c++ API can also be queried for the attributes of their parents, in support of a bottom-up style. For an example of typical semantics for accessing hddm data structures in c++, see the following rewrite of the above c code for scanning the data in transcript.hddm.

Note how much simpler the code above is, using the c++ iterators than it was drilling down from the top-level structures using the c API in the prior example. Factory methods returning iterators for every element within the model are provided in the top-level class `$_HDDM`. See in the example how the enrolled attribute year is made available as an attribute of the course object pointed to by the `course_i` iterator. This illustrates how elements lower down in the hierarchy inherit the attributes of their parents in the tree, in addition to their own.

```
#include "hddm_x.h"

int main()
{
   x_iostream_t* fp;
   x_HDDM_t* tscript;
   x_Student_t* student;
   x_Enrolleds_t* enrolleds;
   int enrolled;
   x_Courses_t* courses;
   int course;
   int total_enrolled,total_courses,total_credits,total_passed;

 // read a record from the file
   fp = open_x_HDDM("tscript.hddm");
   if (fp == NULL) {
      printf("Error - could not open input file transcript.hddm\n");
      exit(1);
   }
   tscript = read_x_HDDM(fp);
   if (tscript == NULL) {
      printf("End of file encountered in hddm file transcript.hddm,"
             " quitting!\n");
      exit(2);
   }

   // examine the data in this record and print a summary
   total_enrolled = 0;
   total_courses = 0;
   total_credits = 0;
   total_passed = 0;
   student = tscript->student;
   enrolleds = student->enrolleds;
   total_enrolled = enrolleds->mult;
   for (enrolled=0; enrolled<total_enrolled; ++enrolled) {
      courses = enrolleds->in[enrolled].courses;
      total_courses += courses->mult;
      for (course=0; course<courses->mult; course++) {
         if (courses->in[course].result->pass) {
            if (enrolleds->in[enrolled].year > 1992) {
               total_credits += courses->in[course].credits;
            }
            ++total_passed;
         }
      }
   }
   printf("%s enrolled in %d courses.\n",student->name,total_courses);
   printf("He passed %d of them, earning a total of %d credits.\n",
          total_passed,total_credits);

   flush_x_HDDM(tscript,0);  // don't do this until you are done with tscript
   close_x_HDDM(fp);
   return 0;
}
```

FIG. 4: Example of simple hddm data model reader using the c api.

The c++ code to write the same output data file transcript.hddm is shown next.

Comparison of the c and c++ versions of the above code shows that the c++ code is more readable and less cluttered with memory management busywork. This is what motivated the writing of the new API for c++.

The hddm files produced using the c++ API are identical to those produced by the c API by default. In addition to the default, the c++ API provides optional on-the-fly compression and integrity checks which

```
#include "hddm_x.hpp"

int main()
{
   // create a new HDDM object and attach it to an input file
   hddm_x::HDDM tscript;
   if (! tscript.open("tscript.hddm")) {
      std::cout << "Error - could not open input file tscript.hddmn" << std::endl;
      exit(1);
   }

   // read a record from the file
   // note: any prior data held by tscript is implicity freed by read(),
   // no need to call flush()
   if (! tscript.read()) {
      std::cout << "Error - could not read from input file tscript.hddmn"
                << std::endl;
      exit(1);
   }

   // scan the contents and print a summary
   hddm_x::Courses_iterator course_i = tscript.courses_begin();
   hddm_x::Courses_iterator course_f = tscript.courses_end();
   int total_courses = course_f - course_i;
   int total_enrolled = 0;
   int total_credits = 0;
   int total_passed = 0;
   for (; course_i != course_f; ++course_i) {
      if (course_i->result()->pass()) {
         if (course_i.year() > 1992) {
            total_credits += course_i->credits();
         }
         ++total_passed;
      }
   }
   std::cout << course_i.name()
             << " enrolled in " << total_courses << " courses." << std::endl
             << "He passed " << total_passed
             << " of them, earning a total of " << total_credits
             << " credits" << std::endl;
   return 0;
}
```

FIG. 5: Example of simple hddm data model reader using the C++ api.

can be turned on by the user upon output, and is automatically detected on input. These features are provided using the xstream c++ library as the i/o layer within the c++ hddm package.

```
#include "hddm_x.hpp"

int main()
{
   // build the nodal structure for this record and fill in its values
   hddm_x::HDDM tscript;
   hddm_x::Student* student = tscript.create();
   student->name("Humphrey Gaston");
   hddm_x::Enrolleds_iterator enrolled = student->enrolleds.create();
   enrolled->year(2005);
   enrolled->semester(2);
   hddm_x::Courses_iterator course = enrolled->courses.create(3);
   course[0].credits(3);
   course[0].title("Beginning Russian");
   course[0].result()->grade("A-");
   course[0].result()->pass(true);
   course[1].credits(1);
   course[1].title("Bohemian Poetry");
   course[1].result()->grade("C");
   course[1].result()->pass(1);
   course[2].credits(4);
   course[2].title("Developmental Psychology");
   course[2].result()->grade("B+");
   course[2].result()->pass(true);

   // now open a file and write this one record into it
   tscript.init("tscript.hddm");
   tscript.flush();
   tscript.close();
   return 0;
}
```

FIG. 6: Example of simple hddm data model writer using the C++ api.

```
$_HDDM_t* read_$_HDDM($_iostream_t* fp);
int skip_$_HDDM($_iostream_t* fp, int nskip);
int flush_$_HDDM($_HDDM_t* this1,$_iostream_t* fp);
$_iostream_t* open_$_HDDM(char* filename);
$_iostream_t* init_$_HDDM(char* filename);
void close_$_HDDM($_iostream_t* fp);
```

FIG. 7: Prototypes for the i/o functions in the c API.