HDDM User's Guide

R.T. Jones
University of Connecticut
(Dated: February 9, 2021)

The HDDM (Hierarchical Document Data Model) is an xml schema for expressing the meaning and relationships of streaming data from scientific instruments. The design is based on a hierarchical network where each node in the graph has a single parent node, multiple key-value 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 construction of a Document Object Model (DOM) for the data. The HDDM toolkit includes tools to express HDDM streams in xml, check their validity against the schema, and serialize/deserialize from container objects in memory. Originally written in c, HDDM provides application programmer interfaces for C++ and python as well. In addition to its own native data format, applications that use HDDM to access their data can also read/write standard HDF5 files and ROOT trees.

I. INTRODUCTION

The HDDM toolkit provides the scientist with a means to format streaming data from a scientific instrument into a structured self-describing byte stream of binary data that is platform-independent and easy to browse, filter, transform extend, annotate, and validate using standard xml tools. The purpose of this User's Guide is to describe the use and operation of the HDDM tools, describing how to define a new data model or inspect an existing one, how to create new HDDM files or read data from existing files. HDDM tools automatically generate the object classes that represent the data described in the user's model, with methods to access the object data in memory as well as to serialize/deserialize themselves between memory and an external byte stream connected to an ordinary file on disk or to a network socket. The underlying implementation of the i/o library is in C++, so it provides good performance in terms of data rate to/from byte streams, with optional on-the-fly compression/decompression and data integrity verification. In addition to oridnary sequential access to data, random-access to records at an arbitrary offset in a stream is also provided for streams that support random seeks, without the need to read the entire stream.

II. TEMPLATES AND SCHEMAS

Every HDDM stream has an associated data model, expressed either in the form of a standard XML schema, or more compactly, in the form of a HDDM template. Tools are provided to translate between the schema and HDDM template description of the model. A HDDM template is a short xml document that describes the structure of one record in the HDDM stream. Every HDDM stream has a copy of its template in plain-text UTF-8 at the beginning, followed by a sequence of data records in a compact binary format. The template contains all of the information necessary to reconstruct the original data objects from the serialized records, together with their hierarchical arrangement. A simple example of a template is given in Fig. 1.

All of the records in the file represent repeats of this basic structure, with different values in the data fields. All actual data values are represented as attributes of tags. Attributes that are assigned type names ("string", "int", "long", "float", "double", "boolean", "anyURI", and "Particle_t") are user data. Any other values assigned to attributes other than these simple types are treated as annotations in the data model, eg. to specify the units assigned to physical values, and do not take up space in the file (other than in the template header). Some of these literal attributes function as metadata, eg. you might want to add an attribute unit="GeV" to document the units used for other attributes in a tag. Other special attributes like minOccurs/maxOccurs take special values that tell the data model whether a given element is always present in every record or may be omitted (minOccurs="0") or whether it may be repeated any number of times (maxOccurs="unbounded"), as in a standard xml schema. The top-level element is special in that it must always be named HDDM and have the attributes shown in Fig. 1. The class attribute of the `<HDDM>` element is any (preferrably short) string that you chose for the family of data models you are creating. Choose a short, unique name for your class because it is used in the type names of user objects that are defined written

```
<?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: A simple example of a HDDM data model template. The data stream would consist of a stream of `<student>` records of unbounded length, each with the same hierachical structure of contained data elements. Only the values designated by simple types, "int", "string", etc. are actually stored in the byte stream.

by the HDDM user library. Its purpose is to prevents collisions between different HDDM stream types that may coexist in a single application.

Templates provide an intuitive way of specifying the structure of a data record in a HDDM stream. For most users, this is all they need to know about in order to define their data models. For those familiar with XML schema validation, there is a more formal way to specify the structure of an xml document which is called a *xml schema*. HDDM uses schemas in two different ways. The first is to specify the structure of the templates themselves. The template shown in Fig. 1 conforms to a schema called `http://www.gluex.org/hddm`. This is not a URL to anywhere; it is a URI known as an *xml namespace*, as suggested by the name of the `xmlns` attribute in the HDDM tag of the template. The schema for this document type is found in hddm_schema.xsl in the schema directory of the distribution. The second use of schemas is related to the fact that every record in a HDDM stream is a valid xml fragment corresponding to a schema against which it can be validated. The HDDM toolkit provides a pair of tools *hddm-schema* and *schema-hddm* that convert back and forth between templates and schema. The two are equivalent ways of representing the same information about the structure of a HDDM record, with the schema being more complete and standards-based, while the template is shorter and more intuitive to most users. Schemas provide a much more general set of constraints that can be expressed for the data and relationships between them, but experience has shown that their practical use for this purpose is limited to special instances where standards-based data validation must be performed. The remainder of this document deals mainly only with templates.

III. HOW TO GET STARTED

The HDDM toolkit is distributed as a github repository `https://github.com/rjones30/HDDM`. Instructions for how to download and build HDDM are given in the INSTALL file provided at the top level of the download tree. The HDDM tools are installed by the installation procedure into the bin directory under the installation base. Before continuing to read this document, make sure that the basic HDDM tools including hddm-xml, xml-hddm, hddm-c, hddm-cpp, hddm-py, and xml-xml are in your shell PATH. These tools are not the HDDM libraries themselves, but the code generators you need to construct user-callable libraries from your HDDM template.

If you already have a HDDM data file that you want to read, you can generate the i/o user library that you can use to read from it and optionally to write a selection of the records to a new HDDM output file. The template that the code generators need to generate the user library is present in the header of the HDDM file that you want to read. Simply providing the data file as input to `hddm-c` will generate c header and implementation files that you need to include on the compiler command line together with your c application code for your project, and similarly, `hddm-cpp` in the case of C++ applications, or `hddm-py` to generate a python module. Of the three supported programming languages, the python implementation is the least verbose and most readable, so it is recommended as a starting point for someone experimenting with HDDM.

Independent of any user programs or language-specific API, the HDDM toolkit provides two tools that can be used to read and write HDDM files directly from the command line. The following command accepts any valid HDDM file as input and prints the contents of the file in plain-text xml to standard output.

```
$ hddm-xml [-n <count>] [-o <output.xml>] <datafile.hddm> [...]
```

The reverse action is provided by the `xml-hddm` tool.

```
$ xml-hddm [-n <count>] -t template.xml <input.xml> [...]
```

The full XML rendition of a data file with many records is highly verbose, which makes the plain-text xml rendering of a HDDM stream of limited practical interest, except as a means to visually browse the data, or to make small changes using a text editor. The reversibility of the conversion between xml and HDDM representations can be useful in cases where one might doubt the fidelity of the encoding being used by HDDM. These two tools do not require any compile-and-link step each time the template is changed, so they are very useful to quickly inspect the contents of a HDDM file. Keep them handy when working through the language-specific procedures below.

IV. HDDM IN PYTHON

If you have access to a HDDM file that was already written, copy it into your work directory and use as a template in building a python module to read the data into python list objects. Otherwise, the HDDM package distribution directory contains a simple example in `models/exam1x.hddm` that you can use for this purpose. Use the following commands to build the python module corresponding to your HDDM template.

```
$ hddm-cpp exam1x.hddm  # builds the underlying C++ library
$ hddm-py exam1x.hddm   # builds the python interface
$ python setup_hddm_x.py build -b build_hddm_x # creates the module hddm_x
```

In this example, I assigned 'x' as the HDDM *class* letter (see the HDDM tag in the template header). You should change it to whatever the class abbreviation you chose for your HDDM data model. The above steps should create a shared library that starts with `hddm_x` in your work directory. Copy that module to a directory in your PYTHONPATH where you usually place your private python modules.

Execute the following interactive python script to print the contents of the example HDDM file in plain text.

```
import hddm_x
for rec in hddm_x.istream("exam1x.hddm"):
   print(rec)
```

To see the same data printed out as a properly formatted xml document, replace the `print(rec)` in the above python HDDM reader with `print(rec.toXML())`'.

A. writing HDDM files in python

For this example, I continue using the same template as was used in the example python HDDM reader above. You should already have built and installed the hddm_x python module and installed it in your PYTHONPATH, using the build steps listed above. Execute the following in a fresh interactive python session to write a new output HDDM file from scratch, starting only from the template and some fake user data.

```
import hddm_x
ofs = hddm_x.ostream('exam2x.hddm')
xrec = hddm_x.HDDM()
student = xrec.addStudents()
student[0].name = 'Humphrey Gaston'
enrolled = student[0].addEnrolleds()
enrolled[0].year = 2005
enrolled[0].semester = 2
course = enrolled[0].addCourses(3)
course[0].credits = 3
course[0].title = 'Beginning Russian'
result = course[0].addResults()
result[0].grade = 'A-'
result[0].Pass = True
course[1].credits = 1
course[1].title = 'Bohemian Poetry'
result = course[1].addResults()
result[0].grade = 'C'
result[0].Pass = 1
course[2].credits = 4
course[2].title = 'Developmental Psychology'
result = course[2].addResults()
result[0].grade = 'B+'
result[0].Pass = True
ofs.write(xrec)
```

This generates a new HDDM file called exam2x.hddm. Now running the above 3-line python HDDM reader on exam2x.hddm should yield the following output.

```
HDDM
  student name="Humphrey Gaston"
    enrolled semester=2 year=2005
      course credits=3 title="Beginning Russian"
        result Pass=false grade="A-"
      course credits=1 title="Bohemian Poetry"
        result Pass=false grade="C"
      course credits=4 title="Developmental Psychology"
        result Pass=false grade="B+"
```

The structure of the output record you are writing is already known to the python module because it is configured with your template. All that the writer needs to do is to fill in the elements and assign the values of the defined attributes. You begin by creating an empty record by calling the HDDM() default constructor. Then you populate the structure top-down by calling addXXXs() methods for each tag XXX under that. The name XXXs is the name of the tag element in the template in a capitalized-plural form. The addXXXs() methods take a single optional int argument, which is the number of copies of that element that need to be added (default 1). They return a list that can be indexed in the usual python fashion to give access to the individual members of the list. Each of these has addXXXs() methods for each of its contents, and so on down the tree. You can omit whole branches of the tree by simply not calling the corresponding addXXXs() method, although xml rules require that you specify minOccurs="0" for the containing tag in the template if you plan to make that subtree optional. As soon as a new element list is created, you can fill in the values of its attributes using simple assignment semantics, as illustrated in the example. The names of the python data members are the same as the names of the attributes in the template.

B. reading HDDM files in python

For this illustration, I assume you have created the file exam2x.hddm using the writer described in the previous section. The following python program lets you open this file and extract bits of information from

the first record, writing a summary report at the end. Of course, in actual practice, a HDDM file would contain many records and the analysis would loop over many instances of student.

```
import hddm_x
ifs = hddm_x.istream("exam2x.hddm")
xrec = ifs.read()
total_enrolled = 0
total_courses = 0
total_credits = 0
total_passed = 0
for course in xrec.getCourses():
   total_courses += 1
   if course.getResult().Pass:
      if course.year > 1992:
         total_credits += course.credits
      total_passed += 1
   total_enrolled += 1
print(course.name, "enrolled in", total_courses, " courses",
       "and passed" , total_passed, "of them,\n",
       "earning a total of", total_credits, "credits.\n")
```

Running the above code should produce output like the following:

```
Humphrey Gaston enrolled in 3 courses and passed 3 of them,
earning a total of 8 credits.
```

In addition to each tag supporting the lookup (via getXXXs methods) of the tags immediately appearing under it in the template hierarchy, the top-level HDDM record provides global getXXXs methods for every tag throughout the hierarchy, and returns all instances of a given tag that appear anywhere in the record, in the order of their appearance. The istream object itself also functions as an iterable in python so the construct, `for rec in hddm_x.istream('exam2x.hddm'):` would look over all records in the input file, assigning the `rec` iteration variable to each record as it is read from the input stream. Likewise, each call to method getXXXs() returns a python list of tag element objects that is iterable using the usual python `for` semantics, as illustrated for xrec.getCourses() above. As before, the individual attributes of each tag instance are accessed as plain data members of their host object. The standard pythonb list functions (eg. len(list), str(list), repr(list)) all work as expected for these hddm tag list objects returned by getXXXs() method. These natural python iteration and accessor semantics provide a quick-and-simple prototyping framework for analysis of repetitve experimental data.

C. advanced features of the python API

See section VII Advanced features below.

V. HDDM IN C++

If you have access to a HDDM file that was already written, copy it into your work directory and use as a template in building a C++ library to read and write data to/from C++ objects. Otherwise, the HDDM package distribution directory contains a simple example in `models/exam1x.hddm` that you can use for this purpose. The following commands build the C++ library corresponding to your HDDM model.

```
$ hddm-cpp exam1x.hddm
$ mv hddm_x.cpp hddm_x++.cpp
$ g++ -c  hddm_x++.cpp -I $HDDM_INSTALL_DIR/include \
  $HDDM_SOURCE_DIR/XString.cpp $HDDM_SOURCE_DIR/XParsers.cpp \
  $HDDM_SOURCE_DIR/md5.c -I$XERCESCROOT/include \
  -L $XERCESCROOT/lib -l xerces-c \
  -L $HDDM_INSTALL_DIR/lib -lxstream -lz -lbz2
```

If the environment variables in this command are not defined in your shell environment, define them or replace them with the appropriate values. The rename step from `hddm_x.cpp` to `hddm_x++.cpp` is not strictly necessary at this point, but it is included here to avoid confusion between the files created in this section and those created later for use with the c API.

A. writing HDDM files in C++

This section turns once again to the example template `exam1x.hddm` used earlier under the python hddm writer section. Having already built a C++ library against this template, now it is time to write a user application that uses the library to create HDDM output according to the template. Open a new C++ source file in an editor and cut/paste the contents of the box below into it, then save it.

```
#include <fstream>
#include "hddm_x.hpp"
int main()
{
   // build the nodal structure for this record and fill in its values
   hddm_x::HDDM xrec;
   hddm_x::StudentList student = xrec.addStudents();
   student().setName("Humphrey Gaston");
   hddm_x::EnrolledList enrolled = student().addEnrolleds();
   enrolled().setYear(2005);
   enrolled().setSemester(2);
   hddm_x::CourseList course = enrolled().addCourses(3);
   course(0).setCredits(3);
   course(0).setTitle("Beginning Russian");
   course(0).addResults();
   course(0).getResult().setGrade("A-");
   course(0).getResult().setPass(true);
   course(1).setCredits(1);
   course(1).setTitle("Bohemian Poetry");
   course(1).addResults();
   course(1).getResult().setGrade("C");
   course(1).getResult().setPass(1);
   course(2).setCredits(4);
   course(2).setTitle("Developmental Psychology");
   course(2).addResults();
   course(2).getResult().setGrade("B+");
   course(2).getResult().setPass(true);

   std::ofstream ofs("exam2x.hddm");
   hddm_x::ostream ostr(ofs);
   ostr << xrec;
   xrec.clear();
   return 0;
}
```

Save this C++ program to a file named `write_exam.cpp` and compile it into an executable using a command like the following.

```
$ g++ -o write_exam write_exam.cpp hddm_x++.o -I. -I $HDDM_INSTALL_DIR/include \
  $HDDM_SOURCE_DIR/XString.cpp $HDDM_SOURCE_DIR/XParsers.cpp \
  $HDDM_SOURCE_DIR/md5.c -I$XERCESCROOT/include \
  -L $XERCESCROOT/lib -l xerces-c \
  -L $HDDM_INSTALL_DIR/lib -lxstream -lz -lbz2
```

The paths listed in the compilation command line may need to be customized for your own build environment. Once it completes successfully, will find the executable `write_exam` in the working directory. Run it as `./write_exam2` and it should create a new hddm file called `exam2x.hddm`. Running `hddm-xml write_exam2x.hddm` should produce output like the following.

```
HDDM
  student name="Humphrey Gaston"
    enrolled semester=2 year=2005
      course credits=3 title="Beginning Russian"
        result Pass=false grade="A-"
      course credits=1 title="Bohemian Poetry"
        result Pass=false grade="C"
      course credits=4 title="Developmental Psychology"
        result Pass=false grade="B+"
```

The structure of the output record you are writing is already known to the program because it has been built against the 'x' template. All that the writer needs to do is to add the data elements and assign the values of the attributes. You begin by creating an empty record by calling the `HDDM()` default constructor. Then you populate the structure top-down by calling `addXXXs()` methods for each tag `XXX` under that. The name `XXX`s is the name of the tag element in the template in a capitalized-plural form. The `addXXXs()` methods take a single optional int argument, which is the number of instances of that element to be added to the containing element (default is 1). They return a subclass of `std::list` that can be iterated over in the usual fashion, or indexed with `operator()(int)` to access the individual members of the list. Each of these has `addXXXs()` methods for each of its contents, and so on down the tree. You can omit whole branches of the tree by simply not calling the corresponding `addXXXs()` method, although xml rules require that you specify minOccurs="0" for the containing tag in the template if you plan to do that. As soon as a new element list is constructed, you can fill in the values of its object attributes using set<attname> methods, as illustrated in the example, where <attname> is a capitalized version of the names of the attribute in the template.

B. reading hddm files in C++

This section assumes that you have created the file exam2x.hddm using the procedure described in the previous section. The following C++ program opens this file and extracst bits of information from the first record, and writes a summary report. Of course, in actual practice such a data file would probably contain many records, and the analysis would loop over many instances of student.

```
#include <fstream>
#include "hddm_x.hpp"
int main()
{
   std::ifstream ifs("exam2x.hddm");
   hddm_x::HDDM xrec;
   hddm_x::istream istr(ifs);
   istr >> xrec;
   hddm_x::CourseList course = xrec.getCourses();
   int total_courses =course.size();
   int total_enrolled = 0;
   int total_credits = 0;
   int total_passed = 0;
   hddm_x::CourseList::iterator iter;
   for (iter = course.begin(); iter != course.end(); ++iter) {
      if (iter->getResult().getPass()) {
         if (iter->getYear() > 1992) {
            total_credits += iter->getCredits();
         }
         ++total_passed;
      }
   }
   std::cout << course().getName() << " enrolled in "
             << total_courses << " courses "
             << "and passed " << total_passed << " of them, " << std::endl
             << "earning a total of " << total_credits
             << " credits." << std::endl;
   return 0;
}
```

Running the above code should produce output like the following:

```
Humphrey Gaston enrolled in 3 courses and passed 3 of them,
earning a total of 8 credits.
```

See section VII Advanced features below.

VI. HDDM IN C

If you have access to a HDDM file that was already written, copy it into your work directory and use as a template in building a python module to read the data into c struct variables. Otherwise, the HDDM package distribution directory contains a simple example in `models/exam1x.hddm` that you can use for this purpose. The following commands build the c library that you will need to read and write HDDM streams that conform to this template.

```
$ hddm-c exam1x.xml
$ gcc -c hddm_x.c $HDDM_SOURCE_DIR/md5.c \
  -I$XERCESCROOT/include -L $XERCESCROOT/lib -l xerces-c \
  -L$HDDM_INSTALL_DIR/lib -lxstream -lz -lbz2
```

A. writing HDDM files in c

This example turns once again to the template `exam1x.hddm` that is included with the source distribution. Use the instructions in the previous section to build the `hddm_x` c API librar, then create a new main program source file and cut/paste the code below into it, then save it.

```
nclude "hddm_x.h"

int main()
{
   x_iostream_t* fp;
   x_HDDM_t* exam2;
   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
   exam2 = make_x_HDDM();
   exam2->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("exam2.hddm");
   flush_x_HDDM(exam2,fp);
   close_x_HDDM(fp);

   return 0;
}
```

Save this c program to a file called `write_exam2.c` and compile it into an executable using a command like the following.

```
$ gcc -o write_exam2 write_exam2.c hddm_x.o -I. -I $HDDM_INSTALL_DIR/include \
  -I$XERCESCROOT/include -L $XERCESCROOT/lib -l xerces-c \
  -L $HDDM_INSTALL_DIR/lib -l xstream -lbz2 -lz
```

The shell environment variables containing the package installation paths in the above compile command may need to be customized for your own environment. Once it completes successfully, you will find the executable `write_exam2` in the working directory. Run it as `./write_exam2` and it should create a new hddm file called `exam2.hddm`. Running `hddm-xml write_exam2.hddm` should produce output like the following.

```
HDDM
  student name="Humphrey Gaston"
    enrolled semester=2 year=2005
      course credits=3 title="Beginning Russian"
        result Pass=false grade="A-"
      course credits=1 title="Bohemian Poetry"
        result Pass=false grade="C"
      course credits=4 title="Developmental Psychology"
        result Pass=false grade="B+"
```

This example explains most of what you need to know to set up HDDM c-structs in memory, and write them to an output file. All storage for hddm data is allocated on the heap. Most of this allocation is carried out automatically by the `make_x_XXXs()` functions, although for strings (char arrays) the user needs to allocate initial storage for the values. Memory pointed to by the pointers returned by the `make_x_XXXs()` functions is owned by the user code until the pointer to it gets assigned to a HDDM struct member that is designated in the data model to hold it. After that, the memory is owned by the top-level HDDM containing record object, and should only be freed by calling the `flush_x_HDDM()` method. Calling `flush_x_HDDM(record, fp)` with its second argument (FILE*) open to an output file causes the record to be written to the output file. Calling it as `flush_x_HDDM(record,0)` causes it to bypass the output serialization step. Either way, `flush_x_HDDM()` frees all memory owned by the HDDM record, discarding its contents, before it returns.

The structure of the output record you are writing is already known to the program because the struct is specified according to the template. All that you need to do is to fill in the elements and assign the values of the attributes. You begin by creating an empty record by calling `make_x_HDDM()`. Then you populate the structure top-down by calling `make_x_XXXs()` for each tag `XXX` and assigning pointers to each one into the appropriate structure element of the parent element. The name `XXXs` is the name of the tag element in the template in a capitalized-plural form. The `addXXXs()` methods take a single optional int argument, which is the number of copies of that element that need to be added (default is 1). They return a pointer to an array of struct pointers which can be indexed in the usual c-fashion to access the individual members of the array. Each of the contained elements within a given host tag have a corresponding pointer in the host struct that must be assigned in the user code to the value returned by the `make_x_XXXs()` function, as illustrated. Any such pointers that are not assigned remain null (initialized by `make_x_XXXs`) and represent parts of the template tree that are missing from the record. This is a perfectly valid hddm record, but user code must check for the NULL pointer condition before trying to dereference it since c has no automatic checking of the validity of pointers. As soon as a new struct array element is created, you can fill in the values of its attribute members using direct assignment semantics, as illustrated in the example above. Any values that are not explicitly assigned remain at the default values, typically zero or null.

B. reading hddm files in c

This section assumes that you have created the file exam2.hddm using the instructions in the previous section. The following c program opens this file and extracts bits of information from the first record, writing a summary report at the end. Of course, in actual practice a HDDM file would probably contain many records, and the analysis would loop over many instances student.

```
#include "hddm_x.h"

int main()
{
   x_iostream_t* fp;
   x_HDDM_t* exam2;
   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("exam2.hddm");
   if (fp == NULL) {
      printf("Error - could not open input file exam2.hddm\n");
      exit(1);
   }
   exam2 = read_x_HDDM(fp);
   if (exam2 == NULL) {
      printf("End of file encountered in hddm file exam2.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 = exam2->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(exam2,0);  // don't do this until you are done with exam2
   close_x_HDDM(fp);
   return 0;
}
```

Running the above code should produce output like the following:

```
Humphrey Gaston enrolled in 3 courses and passed 3 of them,
earning a total of 8 credits.
```

Having read the section above on how to write HDDM records using the c interface, it should be easy to understand the meaning of the above code. The `read_x_HDDM()` call allocates all of the memory needed to stand up the full record hierarchy in memory. The `flush_x_HDDM(record,0)` call at the end of the loop ensures that all of this memory gets recycled to the heap before the next record is read in. Accessing leaf elements that are deep inside the hddm template hierarchy can only be achieved by traversing all of the nodes in the tree above, which makes a simple data mining operation somewhat verbose, as illustrated in the above example, although it still scales well because the model is hierarchical, not a linked list. If you are unsure about how to do something, browsing within the header file is probably not going to be very rewarding because all of the internal functionality of the logic that supports the serialization/deserialization of the data is exposed there. However, the user API is very simple. Access to the data-bearing attributes is through direct struct member access. Only the `make_x_XXXs` functions and the input/output functions (open, close, read, flush, skip) should be called by the user; all the rest are for internal use. As is the case for for all of the other API's, the template itself should be the only documentation you need to consult when writing code that interacts with HDDM data.

C. advanced features of the c API

The c API is no longer in active development. It is supported only for legacy applications that rely on it. The features described in the VII Advanced Features section below are not available using the c API. The only things that are ensured with regard to ongoing support of the c API is that it can read the streams that it writes based on any valid HDDM template, and that HDDM files written using the c-API can be read by applications built using any of the other API's. The converse of the last statement is not guaranteed to be true in all cases. If an input file is not readable by the c-API, it prints a polite error message reporting this fact and quietly exits.

VII. ADVANCED FEATURES

A. on-the-fly compression/decompresson

HDDM streams added support for on-the-fly compression on output (and decompression on input) with the introduction of the C++ API. Because the python API is a thin wrapper around the C++ classes, it also supports this feature. Compression can obviously only be controlled when the stream is being written. It can be switched on and off at any time after the stream is opened, either before the first record is written or any time thereafter. Whenever it is turned on or off, a small marker is written to the byte stream that tells the reader when to enable/disable decompression on the input stream. These transitions occur silently during input; no user action is needed, and no log messages are automatically generated. Two compression options are supported.

1. **bzip2 compression** - This option offers the best compression ratio, a factor of about 2.5 for particle physics experimental Monte Carlo data. It is also the most expensive in terms of cpu time needed during output. Cpu demand for decompression is much lower, more than an order of magnitude.
2. **zlib compression** - This option offers somewhat lower compression ratios, a factor of about 1.9 for particle physics experimental Monte Carlo data. However, it is also much less expensive in terms of cpu time than bzip2, by more than a factor 3. Cpu demand for decompression is much lower than compression, as is usually the case with codecs.

Both options are provided because each has its strengths and weaknesses in terms of cost/performance, and their relative behaviors may be quite different for different data models. Another factor to take into consideration when deciding which compression algorithm to use, if at all, is the implications of the compression block size on the efficiency for random access to records in the stream. For more information about random access, see the relevant section below. If the stream is uncompressed, random access to a particular record generates a read starting at the beginning of that specific record and only taking in the contents of that record, whereas if the stream is compressed, the entire compression block containing the record must be decompressed before the data for the desired record can be pulled in. The compression blocks for bzip2 compression are almost 1MB in size, whereas the zlib blocks are much smaller, around 32KB. There is no general answer to the question of which compression option is best. The person producing the data

should consider what the most likely scenarios are for reading the data, and weigh the costs and benefits of compression before making this decision.

In the C++ API, the HDDM namespaces have defined the following constants to distinguish different states of compression:

- `k_no_compression`
- `k_z_compression`
- `k_bz2_compression`

One of these three constans should be passed as mode to the `setCompression(mode)` method of the `hddm_x::ostream` class to initialize or change the compression state of any given output stream. All records written after this method is called will reflect the change. The present compression mode of either an input or output hddm stream can be queried by calling method `getCompression()`. The return value (int) can be compared with the three constants above to determine which of the three modes is presently enabled.

In python HDDM modules, stream objects of class `hdd_mx.istream` and `hddm_x.ostream` support selection and sensing of the current compression mode by exposing read/write attribute `compression`. The named constants listed above are defined within the `hddm_x` module namespace. Setting bz2 compression on an open ostream would look like, `fout.compression = hddm_x.k_bz2_compression`.

B. on-the-fly data integrity checks

HDDM streams added support for on-the-fly data integrity checks with the introduction of the C++ API. Because the python API is a thin wrapper around the C++ classes, it also supports this feature. Data integrity verification works by the writer computing a hash value on each output record and storing it as part of the output stream, which the reader then pulls off the stream and uses to verify the integrity of the data is reads from the stream. Two 32-bit hash algorithms are currently supported by hddm.

1. **CRC32** - the 32-bit cyclic redundancy check algorithm
2. **MD5** - the MD5 one-way hash algorithm

CRC is considered in cryptographic circles as an error detection algorithm, meaning that a single bit change in the data record will result in a change in the 32-bit code, and it is very rare that a combination of errors cancels out and generates the same crc as the original data. This is probably all we need for our data, and it is much faster to compute than MD5. MD5 is called a one-way hash in cryptographic jargon, which means that a single bit change in the data record will be reflected in a *vastly* different value for this 32-bit code, with approximately 50% of the bits changing in the hash as a result of a single bit-flip in the input. One might consider this marginally better for error detection in a random byte stream, but it is more expensive to compute than a CRC code. Neither MD5 nor CRC32 options result in any noticeable overhead in the context of any models tested so far.

In the C++ API, the hddm namespaces have defined the following constants to distinguish different states of data integrity checking:

- `k_no_integrity`
- `k_crc32_integrity`
- `k_md5_integrity`

One of these three values should be passed as mode to the `setIntegrityChecks(mode)` method of the `hddm_x::ostream` class to change the current state of the output stream. All records written after this method is called will reflect this change. The present integrity checking mode of either an input or output hddm stream can be queried by calling method `getIntegrityChecks()`. The return value (int) can be compared with the three constants above to determine which of the three integrity checking modes is presently enabled.

In python HDDM modules, stream objects of class `hddm_x.istream` and `hddm_x.ostream` support the same interface by exposing read/write attribute `integrity`. The named constants listed above are defined within the `hddm_x` module namespace. Setting CRC32 compression on an open ostream might look like, `fout.integrity = hddm_x.k_crc32_integrity`.

C. random access to hddm records

HDDM streams added support for random access on input with the introduction of the python API. Because the python API is a thin wrapper around the C++ classes, it is also supported by the C++ API. Random-access writing to hddm streams is not supported; the access point for output streams is always positioned after the end of the previous output record. Random-access reads are supported on any input stream that supports repositioning. To succeed, the random access position must have been generated by a previous call to the `getPosition()` method of the same HDDM stream, either during the initial phase when the stream was being written, or during a subsequent read pass over the same stream. The `getPosition()` query returns an opaque value representing a point in the stream either at the beginning of the first record, or immediately after the last valid record read or written on the stream. Random access to individual records in the input hddm stream can take place in any order, and involve displacements either forward or reverse from the position of last access.

Attempts to access a stream at an uninitialized position, or at a position that was generated on a different HDDM stream, will result in unpredictable behavior, most likely a segmentation fault upon the next attempt to read from the stream. The following three integer values are needed to define a stream position.

1. **block_start** (uint64_t) - absolute stream position (std::streampos value) of the beginning of the block containing the record
2. **block_offset** (uint32_t) - offset with the block to the start of the designated record, or 0 if compression is disabled
3. **block_status** (uint32_t) - complete state (compression, integrity, other information about the stream state at this position)

If a database were used to store a map of valid positions for a set of HDDM files, a minimum field width of 128 bits would be needed. Of course, you might want to save the name and creation date of the input file that the positions apply to, so that you do not accidentally try to apply them to a different file than they were created for. If the stream is uncompressed then `block_offset`=0, but still `block_start` and `block_status` would be needed. The `block_status` value is typically the same for all positions in a given file or dataset, so in most cases only a single value for that variable needs to be kept, together with a list of the starts and offsets for the given file.

The object class `hddm_x::streamposition` is used to hold stream position information. Public data members with the names listed above are exposed for members of the streamposition class. Both `hddm_x::istream` and `hddm_x::ostream` classes have `getPosition()` members that return a streamposition value. It can either be recorded by saving the values of its three data members, or by keeping the object in memory and passing it to the corresponding `istream::setPosition(streamposition)` method called on an istream that is (presumably) open for input on the same file. If the 3 values are stored, they can later be quickly turned back into a streamposition object using the constructor `streamposition(start,offset,status)`.

HDDM files that were written since this feature was introduced are marked with the capability to support random access. To check if a given file that has been opened for input on a `hddm_x::istream` supports random access, simply call method `getPosition()` within a try-catch block and catch the RuntimeError that is thrown if the input does not support this feature.

Support in the python API for random access follows closely the scheme described above for C++. The `hddm_x.istream` and `hddm_x.ostream` classes both have read/write data members called `position` that reference objects of type `hddm_x.streamposition`. These objects can be saved and then later assigned to an `hddm_x.istream` opened on the same file to seek to the same position in the input stream using a command like `fin.position = hddm_x.streamposition(start,offset,status)`. Until another position assignment is executed, reading proceeds in a serial fashion starting from the last record read from the stream.

VIII. REFERENCES