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.

```
<?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.

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 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. The HDDM package distribution directory contains a simple example in `models/exam1x.hddm`. Use the following commands to build the python module that you can use to read the contents of this file in the form of python objects.

```
$ 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("examx.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('examx2.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 examx2.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 examx2.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("examx2.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('exam2.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 **??** Advanced features below.