**ABSTRACT**

Different scientific applications use many different formats to store their data. Even if common, self-described data formats, such as HDF5 or NetCDF, are used, data organization (e.g., the structure and names of groups, datasets and attributes) differs between applications and experiments. This makes development of uniform visualization tools problematic and data comparison difficult. VizSchema is an effort to standardize metadata of common self-described data formats so that the entities needed to visualize the data can be identified and interpreted by visualization tools. These standards are expressed both as human-readable text, programmatically, and as an XML description. An HDF5 data reader and a plugin for the visualization tool VisIt implementing this standard has been developed. This plugin allows visualization of data from multiple applications that use notions of fields, particles, and geometries. The data that has been visualized comes from multiple domains: fusion and plasma physics simulations, accelerator physics, climate modeling, and nanotechnology.

**KEYWORDS**

Visualization, schema, HDF5, NetCDF, VisIt, standardization, metadata

**1. INTRODUCTION**

Visualization is extremely valuable in providing better understanding of scientific data generated by simulations and guiding researchers in designing more meaningful experiments. Scientific models need to be compared with each other and validated against experiments. Consequently, most computational scientists rely on visualization tools. However, visualization and data comparison is often made difficult by the fact that various simulations use very different data formats and visualization tools.

Self-describing data formats are increasingly being used for storage of data generated by simulations. Such formats allow the code to store and access data within a file by name. The file storage system then takes care of developing an index for the data. In addition, the data can be decorated with attributes describing the units, dimensions, and other metadata for a particular variable. The self-describing formats now in use also help to deal with binary incompatibilities. Because different machine architectures use different binary representations for numbers, a binary file written by one processor may not be readable by another processor. Self-describing data file formats and interfaces ensure that the data is written in a universal binary format on all processors, and that software reading the data translates it to the appropriate architecture-specific format.

The Hierarchical Data Format (current version is HDF5) [1] and the NetCDF [2] format are in common use in the fusion, accelerator and climate modeling communities. HDF5 allows one to create a multi-tiered data structure inside of a file, so that one can create nested structures of groups and datasets. The NetCDF format does not allow for arbitrary depth structure in the same way. Both formats have parallel I/O important for large-scale simulations.

Examples of HDF5 use include fusion and plasma physics codes such as M3D [3], a 3D MHD code developed at Princeton Plasma Physics Laboratory (PPPL), VORPAL [4], a 3D plasma simulation code...
developed at the University of Colorado and Tech-X, NIMROD [5], another large 3D MHD multiorganizational code, and the suite of codes in the recently-started fusion framework FACETS [6]. Some other important efforts (such as SWIM [7]) have built on NetCDF.

In spite of the fact that all these codes use self-describing data format, their files are organized very differently. They often do not share the node structure, do not agree on attributes, use different names for physically similar variables and store data in different structures. In other words, self-describing formats, though powerful, do not impose universally interpretable data structures. How does one recognize that a particular dataset represents a mesh and what kind of mesh is it? How does one indicate that a dataset is mapped to a particular mesh? Which data ordering is used (is it grouped by components or position indices)? Using some standards within these formats could resolve this problem.

Visualization tools used by different teams are also very non-uniform. For a long time, scientific community used IDL [8] (Interactive Data Language) and AVS/Express [9]. Lately, many teams are moving towards the freely available, open source, high-quality visualization tools VisIt [10] and ParaView [11].

Lately there has been a movement to standardize data and increase tool interoperability. For example, the recent U.S. decision to review its interest in the International Thermonuclear Experimental Reactor (ITER) and the upcoming Fusion Simulation Project [12] will require simulations to exchange data, communicate with each other, and be usable for researchers with a variety of post-processing and analysis tools. Climate modeling and space weather modeling are moving to comprehensive integrated modeling as well and feel the pressure to unify formats and approaches.

In this paper we present our efforts to develop such a visualization standard for computational applications dealing with field and particles data - fusion and accelerator modeling, climate and nanotechnology simulations. Our approach is based on first identifying the entities of interest to visualization, relationships between these entities and then defining intuitive and minimalistic ways to express them using metadata and common constructs used in self-described data formats: groups, datasets, and attributes. We call this formulation VizSchema (VS) and express it as a plain text specification and as XML [13].

![Fig.1. The architecture of VizSchema. The underlying is VS Data Model. VS Interface reflects the model and is used by visualization plugins to access visualization data from the native data. Readers implement the interface for particular file formats. Modules VS ParaView Plugin and VS NetCDF Reader are not implemented and shown to demonstrate future use of the standard.](image)

Based on the schema (see Fig. 1), we have defined a C++ interface (VS Interface) for reading visualization data into memory, implemented this interface for the HDF5 format (VS HDF5 Reader) and used it to develop a plugin (VS VisIt Plugin) for VisIt, which became very throughout the scientific community. The VS Interface can be reused for creating NetCDF and other readers and can be called from other visualization tools capable of using C/C++, for example ParaView or AVS/Express.
2. VIZSCHEMA DATA MODEL

2.1 Introduction

In this section we describe the elements of the Visualization Schema. These elements identify the data structures that one needs to expose in order to do visualization. They are not about HOW the visualization is performed (i.e. the type of light or position of the camera); instead, they are WHAT is being visualized (data and geometry). In designing the schema we use the following guiding principles:

- All the markup for the schema should be contained in the attributes so that users could choose the names of the data itself (typically contained in groups and datasets) as they please. The markup can be generated during I/O or added in a post-processing step.
- We expect these attributes to start with “vs”.
- VizSchema attributes can refer to other entities using their short or fully qualified name. If a short name is used, the reader will first search in the same space and then enlarge the search until the matching name is found.
- If variables have the same name (in different leaves of the file) they are considered one variable with the combined mesh.
- Each vs entity has an attribute vsType, which describes its category (“variable” or “mesh”, for example).
- Some entities have different kinds (i.e. subtypes), in which case a vsKind attribute specifies the kind (“uniform”, for example).

2.2 Variables and Variables With Meshes

All of the visual data is split into two types: a variable or a variable with mesh. A variable represents data which lives on a mesh described separately, while a variable with mesh contains spatial information within itself. In typical Particle-in-Cell simulation all electric and magnetic fields share the same mesh, so this mesh is described once and the values of the fields are described separately as variables. Particle data, on the other hand, typically places the momentum and position in the same dataset. This type of data does not need a separate mesh, so here the tool must generate a point mesh from the positional data within this dataset. So, particle data is a variable with mesh.

The following pseudo-code snippet shows markup for a variable in HDF5:

```hdf5
Dataset "phi" {
    Att vsType = "variable" // Required string
    Att vsMesh = "mycartgrid" // Required string
    DATASPACE [201, 301, 105] // Required float array
    Att vsCentering = "zonal" // Optional string, defaults to nodal
}
```

The vsType attribute in this example indicates that this dataset represents a variable which can be visualized on a separately defined mesh. The vsMesh attribute gives the name of the mesh that holds the position data for this dataset. The actual variable data is saved in the array “dataspace”. The optional attribute vsCentering instructs that the data should be interpolated to a zone. Other information, such as size of the dataset and data limits, can be derived from the above four attributes. Note that the name of the dataset, “phi”, is chosen by the user and is not restricted by VizSchema in any way.

For comparison the following snippet shows the markup for a variable with mesh:

```hdf5
Dataset "electrons" {
    Att vsType = "variableWithMesh" // Required string
    Att vsMesh = "mycartgrid" // Required string
    DATASPACE [201, 301, 105] // Required float array
    Att vsCentering = "zonal" // Optional string, defaults to nodal
}
```
Att vsNumSpatialDims = 3 // Required integer
DATASPACE [n0, n1] // Required float array
Att vsIndexOrder = "compMinorC" // Optional string
}

The vsType attribute specifies the type of the dataset as “variableWithMesh”. Instead of specifying a separate mesh with a vsMesh attribute, the mesh will be generated using the first 3 dimensions (specified by vsNumSpatialDims) of the given array “dataspace”. The remaining dimensions are considered to contain the actual variable data. In the case of particles, this is often a vector representing velocity (x, y, z), followed by any other values of interest to the researcher.

One also needs to describe the ordering of the data or the order of indices starting from the fastest-varying. For example, for the 3D case:

\[
\text{compMinorC} = (i0, i1, i2, ic) \\
\text{compMinorF} = (i2, i1, i0, ic) \\
\text{compMajorC} = (ic, i0, i1, i2) \text{ (same as compMinorF for 1D)} \\
\text{compMajorF} = (i2, i1, i0, ic) \text{ (same as compMinorC for 1D)}
\]

In component minor order, the indices \((i0, i1, i2, ic)\) are such that the component index, \(ic\), appears last. The C reference would be \(\text{array}[i0][i1][i2][ic]\), while the Fortran reference would be \(\text{array}(i0,i1,i2, ic)\). In component major, the indices \((ic, i0, i1, i2)\) are such that the component index, \(ic\), appears first. The C reference would be \(\text{array}[ic][i0][i1][i2]\), while the Fortran reference would be \(\text{array}(ic,i0,i1,i2)\).

When addressing the array in memory, two adjacent memory locations can differ by incrementing either the first index (Fortran) or the last index (C). Since the data is generally written to HDF5 files without changing the order, the component index must be specified. The default value of this attribute is \text{compMinorC}. This attribute is needed to reorder data as expected by a visualization tool.

Our system also allows user-defined expressions using regular mathematical symbols and the other variables in the file. Evaluating these expressions is left to the visualization code itself. Our VisIt plugin, for example, passes the expression directly to VisIt, which uses Python as its expression language. Therefore one could define a density of electric energy as follows:

```
Group anygroupname {
    ATT vsType = "variableDefinition" // Required string
    ATT vsDefinition = "elecEnergyDensity = (E_0*E_0+E_1*E_1+E_2*E_2)" // Required string
}
```

In defining this, we assume that the visualization tool can parse and evaluate such expressions. These assumptions are valid for our VisIt plugin implementation, which uses Python as its expression language.

### 2.2 Meshes

There is no uniform classification of meshes across tools and simulations. Based on our experience with several codes, we determined that the following mesh type categorizations are sufficiently general:

- Structured grid, which is defined by a list of points, each point defined by its coordinates
- Rectilinear grid, which is defined by the lists of increasing coordinate values for each axis and is a specialization of a structured grid
- Uniform grid (sometimes also called uniform Cartesian), which has constant distances between nodes in all directions and is a specialization of a rectilinear mesh
- Unstructured grid, which is defined by points and cells of various types.
The VizSchema markup for these mesh types is shown in the following examples. The first example describes a structured mesh with component-minor ordering. The dataset contains the mesh’s points as an array ordered in X, Y, and Z, with 3 values (x,y,z) at each mesh point, for a total of 4 array dimensions:

```plaintext
Dataset "mystructmesh" {
  Att vsType = "mesh" // Required string
  Att vsKind = "structured" // Required string
  Dataspace [n0][n1][n2][n3] // Required float array
  Att vsIndexOrder = "compMinorC" // Optional string
  Att vsStartCell = [0, 0, 0] // Required integer array if part of another mesh
}
```

The second example describes a 2D rectilinear mesh. It is a group containing 2 datasets, each of which contains the mesh points along one axis (X, Y). By default the points for the first axis are in a dataspace named “axis0” and the second axis in “axis1”, however there are optional vs attributes which allow the user to specify a different name for these arrays.

```plaintext
Group "myrectgrid" {
  Att vsType = "mesh" // Required string
  Att vsKind = "rectilinear" // Required string
  Dataset axis0[n0] // Required array – points on first axis
  Dataset axis1[n1] // Required array – points on second axis
}
```

The third example describes a 3D uniform mesh. Since all the mesh points are uniformly distributed, the coordinates of each point do not have to be provided. Instead, the VS attributes give the start and end position and number of cells along each axis, and the VS plugin will automatically generate the mesh.

```plaintext
Group "myunigrid" {
  Att vsType = "mesh" // Required string
  Att vsKind = "uniform" // Required string
  Att vsStartCell = [0, 0, 0] // Required integer array if part of another mesh
  Att vsNumCells = [200, 200, 104] // Required integer array
  Att vsLowerBounds = [-2.5, -2.5, -1.3] // Required float array
  Att vsUpperBounds = [2.5, 2.5, 1.3] // Required float array
}
```

The final example describes a 3D unstructured mesh. Such a mesh is generated from two arrays, one containing the coordinates of the mesh points, and the other giving the set of points that compose each cell in the mesh. By default, the coordinate array is named “points” and the cell array is named “polygons”. The optional attributes vsPoints and vsPolygons permit arrays with non-default names to contain this information.

```plaintext
Group "mypolymesh" {
  Att vsType = "mesh" // Required string
  Att vsKind = "unstructured" // Required string
  Att vsPoints = "points" // Optional string (default = "points")
  Att vsPolygons = "polygons" // Optional string (default = "polygons")
  Dataset points [n0_points][n1_points] // Required float array
  Dataset polygons [n0_polys][n1_polys] // Optional integer array
}
```
The list of supported kinds of meshes can be expanded as we encounter more kinds of simulation data. Some of them may be able to map to an already existing type, with the data translations implemented in the plugins.

2.3. Adding Markup and Using XML for VizSchema

One of the factors in adoption of any visualization markup for self-describing data formats will be how easy it is to markup existing simulation data files without the need to change the simulation codes that write the files. This implies an ability to post-annotate data files based on the set of structure rules for visualization structure prescribed by the schema. Rather than having each simulation team write code to do this annotation, we intend provide a tool to do this in a general way so that one can write a short text description of the annotations and run the tool to add the annotations to the binary data format (HDF, NetCDF, etc.). The description should hold for a given version of the simulation and is written once, whereas the tool may be run many times for different output datasets. This description will be validated against the rules expressed in XML. A snippet of the XML description is given below.

```
<Choice name="mesh">
  <SingleChoice name="vsKind" minOccurs="0" default="uniform">
    <Choice name="uniform">
      <IntegerVector name="vsStartCell" minOccurs="1"></IntegerVector>
      <IntegerVector name="vsNumCells" minOccurs="1"></IntegerVector>
      <RealVector name="vsLowerBounds" minOccurs="1"></RealVector>
      <RealVector name="vsUpperBounds" minOccurs="1"></RealVector>
    </Choice>
    <Choice name="rectilinear">
      <Text name="vsAxis0" default="axis0"></Text>
      <Text name="vsAxis1" default="axis1"></Text>
      <Text name="vsAxis2" default="axis2"></Text>
    </Choice>
  </SingleChoice>
</Choice>
```

3. STATUS AND EXAMPLES

As mentioned in the introduction, based on VizSchema data model, we defined and implemented a C++ reading interface (HDF5 Reader on Fig 1) which allows accessing variables, variables with meshes and meshes as void* arrays accompanied by their metadata including their name, kind (if applicable and the array’s shape). Next we developed a new plugin for VisIt (VisIt plugin on Fig. 1).

To facilitate the adoption of the schema, we prototyped a F90 library for marking up HDF5 files so that they become VS compliant and developed prototypes of VS data readers using Python and IDL.

Several codes adopted VizSchema and now provide the compliant output during I/O, and for some codes we provided tools for retrofitting their data after it has been generated. The plugin code was tested on Linux and OS X and is installed on such supercomputers as franklin.nersc.gov. Fig.2 show some examples of visualizations done using the VizSchema plugin for VisIt.
Fig. 2. Example of using VizSchema in various applications. The top left image shows a screen capture of OASCR Award for Scientific Visualization at the 2008 Scientific Discovery through Advanced Computation Conference (Seattle) for the video, “Visual Inspection of a VORPAL Modeled Crab Cavity” [14]. The top right image shows test particles in a monomer (data generated by Polyswift [15]). The lower left image shows electrons in a particle simulation (data generated by VORPAL. The lower right image depict electron density in a tokamak (data generated by two coupled codes of FACETS).

4. CONCLUSIONS AND FUTURE DIRECTIONS

This work showed the value of a standardized approach to visualization and a need to come up with a uniform classification of visual data (especially meshes). It also became clear, that to make our approach more useful, we need to make the VizSchema elements less HDF5-centric: map the schema using the NetCDF lingo (variables, dimensions and attributes) and consider developing a reader for the NetCDF data (VSNETCDF Reader in Fig. 2). In addition, one needs tools that could bring more applications into the developed standard. That is why we intend to develop a tool that would read an XML description of a data file and annotate HDF5 and NetCDF files in accordance with the schema - thus making them VS compliant.
and ready to use by the visualization plugins. These tools will be probably implemented in Python and use Python APIs for HDF5 and NetCDF. Finally, in order to accommodate large amounts of data we will need to implement the parallel version of the VisIt plugin.

5. RELATED WORK

There are several efforts that are trying to provide specific semantic layers on top of popular data formats. Examples include XDMF [16] data model, which separates the metadata about the structure of HPC data from the main data and express it in XML, NeXus [17], which abstracts the schema that is appropriate for the neutron science and provides an API for reading and writing HDF5 files obeying the schema, and ESML [18] which created a schema suitable for earth science applications and provides a uniform interface to several data formats.

ACKNOWLEDGEMENT

Multiple people from Tech-X Corporation, especially Sc. Sides, D. Smithe, A. Hakim and M. Miah contributed to this project. Special thanks the VisIt team, especially H. Childs, S. Ahern, J. Meredith and G. Weber, who were patient and helped us to use VisIt and debug our plugin.

REFERENCES

[17] NeXus: http://www.nexusformat.org/Main_Page