Final Report on Tech-X Phase II SBIR project "FSML – Fusion Simulation Markup Language,"

Grant No DE-FG02-04ER84101

Svetlana Shasharina - PI

1. Introduction

This report summarizes the final successes of the project but skips the descriptions of the routine work, as it was described in detail in our semi-annual reports.

VizSchema is a new name of the code and the schema (which used to be FSML). It reflects a fact that instead of using semantic fusion-specific schema, we moved to the syntactic schema (from the physics point of view), which is still semantic from the visualization point of view.

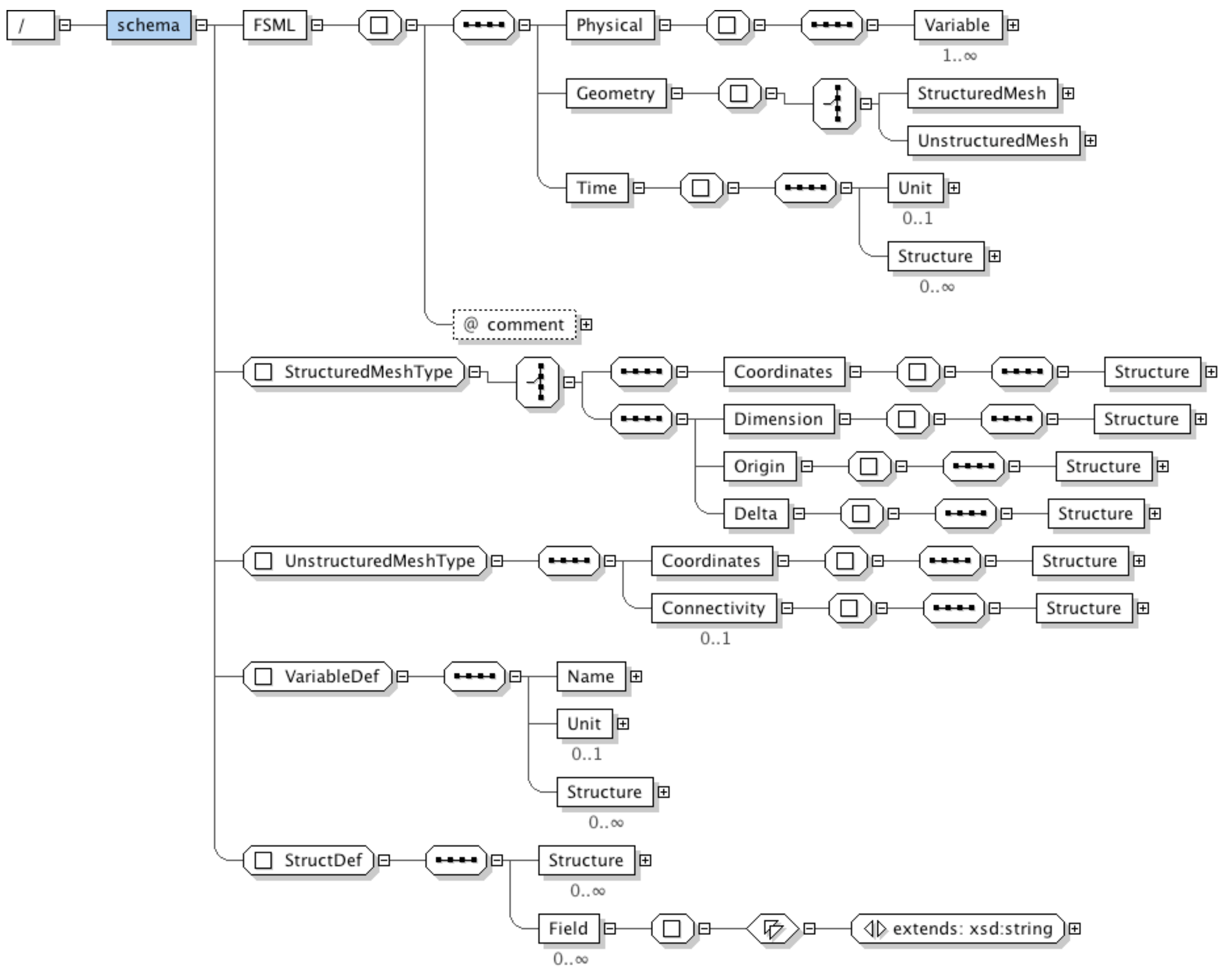


Figure 1. XML schema representing VizSchema.

VizSchema is an agreement about what visualization data is. This agreement can be expressed as XML schema (that was the original intention of the project) or can be reflected in the native data itself. In the next two sections we describe our approaches using XML and using HDF5 markup consistent with the schema.

2. XML approach

When XML schema is used, any particular type of a simulation output is then described by an XML instance, which maps the schema fields to the entities of the native data format (dataset, attribute or group). This XML approach proved to be useful for the applications, which output their data in the same way each time. Examples of such codes are NIMROD and M3D. The XML schema that we used is shown on Figure 1.

Based on this XML schema, we created a C++ library that reads data one-dimensional arrays accompanied by their metadata. This library was then used to create an AVS/Express module and a VisIt plugin that allowed importing NIMROD and M3D data into these tools (see Figure 2).

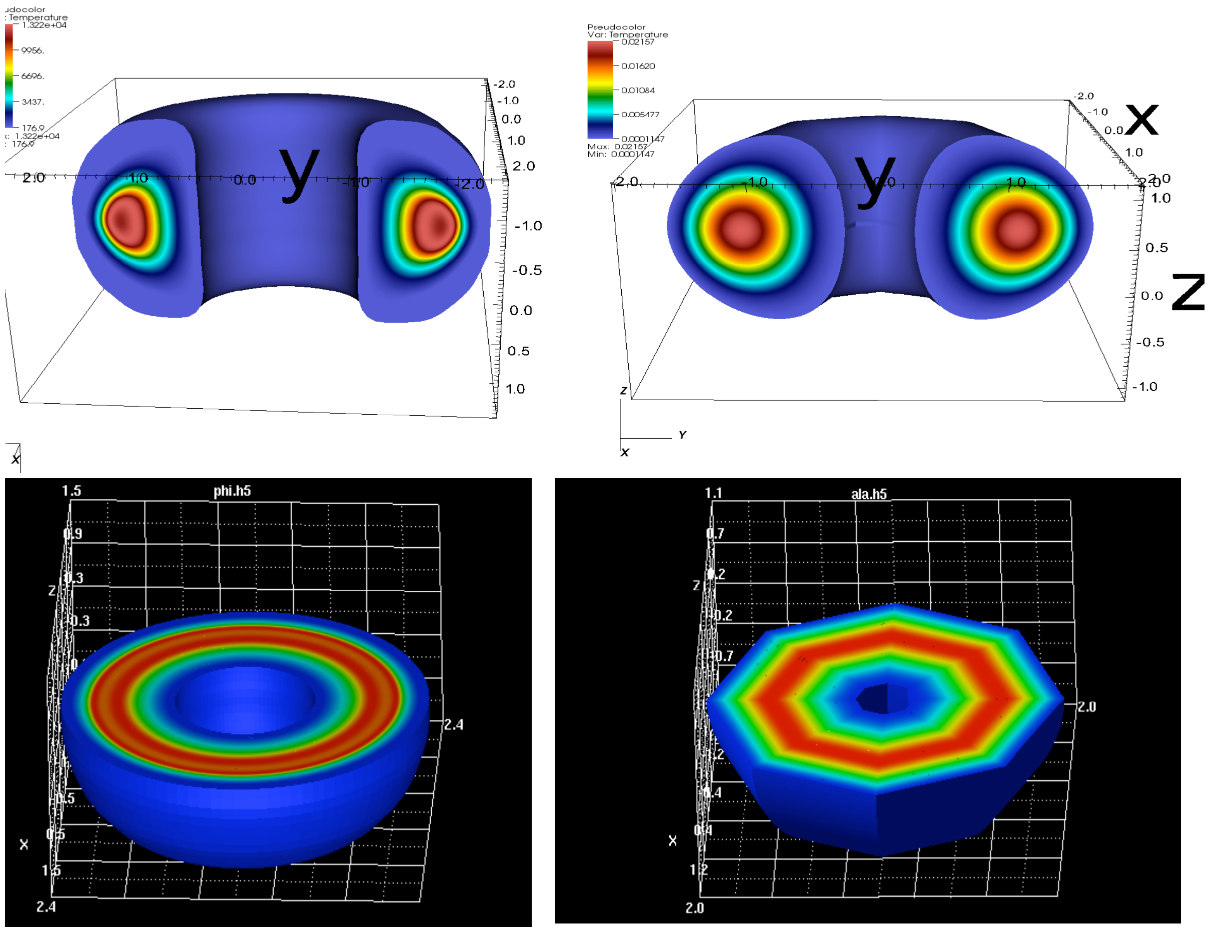


Figure 2. Comparative visualization of NIMROD and M3D data using XML schema. The top row shows VisIt results, the lower row shows AVS/Express visualizations.

3. Markup approach

Introduction

The XML approach briefly described above does not work for the applications which change the data organization from run to run. This means that it is impossible to have one XML instance per application. Examples of such applications are VORPAL and FACETS. These applications drove our effort to create an agreement on how one should markup data internally. The description of this agreement follows.

HDF5 markup

Overview

The data, which is supposed to be plotted, is called a *variable* in this document. VizSchema is a convention about the mark-up (attributes and data structure) of HDF5 files to define such variables so that they can be imported into our VSH5 plugin. All VizSchema attributes start with “vs”. Entities noticed by VizSchema always have attributes vsType, which is a string attribute that shows the category of the entity. For example, vsType can be equal to “mesh”. Some entities can be classified further, and in this case they use the vsKind attribute. For example, a mesh can be of kind “uniformCartesian.” We currently support three string values of vsType: “variable”, “variableWithMesh” and “mesh”.

In a file compliant with the VizSchema, default variables are represented by datasets, which must have the vsType attribute equal to “variable” or “variableWithMesh”. Such dataset can be either single- or multiple-component. This is figured out by comparing dimensions of the dataset and the mesh. By default, multi-component datasets are component-minor (meaning that fastest index is the component index). If a dataset is component-major, it should manifest this by a string attribute vsOrder equal to “compMajor”.

By default, all variables data is node-centered (meaning that the colors are linearly interpolated between the nodes). In order to change this, one should use optional attribute vsCellOffset and set it to “center”, so that the zonal look will be used. No other offset are currently supported for now.

If the dataset is single-component, it exposes one variable with the same name. If the dataset is multi-component, each component is exposed as a separate variable with the name composed of the name of the dataset and “_i”, where is the index of the component.

The “variable” data is assumed to be on a mesh, which can be a real N-dim mesh or 1-dim time sequence, with such mesh represented by an outside group. A typical example is electric field on a 3D grid. The “variable” data must have attribute vsMesh equal to the name of the mesh on which data resides.

A mesh is referenced by the variables using an unqualified name if it is in the same level. If the mesh group/dataset is not on the same level with the variable, then the full path should be provided. Mesh groups/datasets must have the vsType attribute equal “mesh” and a string attribute vsKind attribute defining the type of the mesh. The other mesh attributes and datasets structure of are not fully defined as they depend on the mesh type and are subject to further standardization.

Additional variables can be defined in the vsVars group with the vsType attribute equal to “vsVars”, which should use variables (and their component) to define expressions readable by VisIt.

The “variableWithMesh” data contains the mesh within itself. A typical example is particle data, which has locations usually in the same dataset with other variables such as momenta. Such entities are always multi-component and need to specify the integer value of the vsNumSpatialDims attribute, which allows the plugin to separate the data that needs to be visualized from the spatial information used to create the mesh.

Variables

This section is supposed to give exhaustive examples of how to define variables on meshes defined outside the data.

Single-component variables using one dataset

```
Group A {
  Dataset phi {
    Att vsType = “variable”         // Var on external mesh, required
    Att vsMesh = “grid0”            // Name of the mesh, required
  }
  Group myVars {
    vsType = “vsVars”
    Att tanphi = “sin(phi)/cos(phi)”       //Extra variable, optional
  }

  Group grid0 {
    Att vsType = “mesh”             // This is a mesh, required
    Att vsKind = “uniformCartesian” // Kind of the mesh, required
    Att totNumPhysCells = [50, 50, 50] // Attributes needed to define this mesh
    Att lowerBounds = [0., 0., 0.]  // More attributes
    Att upperBounds = [1.,1.,1.]    // More attributes
    Att startCell = [0, 0, 0]       // More attributes
  }//End of grid0
}
```

Assuming that there is just one component, this example defines phi as a variable on grid0 and adds an extra variable tanphi on the same grid. The group for the mesh data is in the group containing the variables using this mesh and that is just a name of the group if referenced in vsMesh, rather than the full path.

Multi-component variables using one dataset

```
Group A {
  Dataset b {
    Att vsType = “variable”         // Var on external mesh, required
    Att vsMesh = “grid0”            // Name of the mesh, required
    Att vsOrder = “compMajor”       // If absent, comp minor is assumed
   }//End of b

  Group vsVars {
    Att mag_pressure = "b_0*b_0+b_1*b_1+b_2*b_2"
    Att b  = "{b_0,b_1,b_2}"
  }

  Group grid0 {
    Att vsType = "mesh"             // This group is a mesh, required
    Att vsKind = "unstructured"     // Kind of the mesh, required
    Dataset coordinates {}          // Appropriate for this type data
    Dataset connectivity {}         // Appropriate for this type data
  }//End of grid0
}
```

Assuming that comparison of dimensions of the dataset b and dimensions of mesh "grid0" concluded that the number of components of b is 3, there will be 3 variables b_0, b_1 and b_2. In addition, there is a variable mag_pressure defined. All variables are on mesh named "grid0" which is on the same level as variables using it.

Single-component variables using many datasets

```
Group step50 {

  Group BVars {
    Dataset Bx {
      Att vsType = "variable"       // Required to be used and exposed
      Att vsMesh = "/step50/grids/grid0"          // Full path to grid, required
    }//End of Bx

    Dataset By {
      Att vsType = "variable"
      Att vsMesh = "/step50/grids/grid0"
    }//End of By

    Dataset Bz {
      Att vsType = "variable"
      Att vsMesh = "/step50/grids/grid0"
    }//End of Bz

    Dataset Br {
      Att vsType = "variable"
      Att vsMesh = "/step50/grids/grid1"
    }//End of Br

     Dataset Bphi {
      Att vsType = "variable"
      Att vsMesh = "/step50/grids/grid1"
    }//End of Bphi


  }//End of BVars

  Group ExtraVars {
    vsType = "vsVars"
    Att B = "{Bx, By, Bz}"
  }
  Group grids {
    Group grid0 {
      Att vsType = "mesh"                  // This is a mesh, required
      Att vsKind = "uniformCartesian"      // Kind of the mesh, required
      Att totNumPhysCells = [50, 50, 50]   // Appropriate for the kind attribs
      Att lowerBounds = [0., 0., 0.]       // Appropriate for the kind attribs
      Att upperBounds = [1.,1.,1.]         // Appropriate for the kind attribs
      Att startCell = [0, 0, 0]            // Appropriate for the kind attribs
    }//End of grid0

    Group grid1 {
      Att vsType= "mesh"                   // This is a mesh, required
      Att vsKind = "unstructured"          // Kind of the mesh, required
      Dataset coordinates {}               // Appropriate for the kind data
      Dataset connectivity {}              // Appropriate for the kind data
    }//End of grid1
  }//End of grids

}//End of step50
```

In this example, Bx, By, Bz, Br and Bphi are variables. In addition, a vector B is defined and added as a variable. Br and Bphi are defined on "grid1" while Bx, By, Bz and B are defined on "grid0". Since grid0 and grid1 are not in the same level as the variables, full path is used to specify them in the attribute vsMesh.

Multi-component variables using many datasets

```
Group EVars {
  Dataset E {
    Att vsType = "variable"         // Required
    Att vsMesh = "/grid1"           // Name of the grid
  }//End of E

  Dataset B {
    Att vsType = "variable"
    Att vsMesh = "/grid1"
  }

  //Extra variables
  Group yVarss {
    Att vsType = "vsVars"
    Att P = "E_0*E_0+E_1*E_1+B_0*B_0+B_1*B_1"
  }

}//End of EVars

Dataset grid1 {
  Att vsType = "mesh"               // This is a mesh, required
  Att vsKind = "unstructured"       // Kind of the mesh, required
  Dataset coordinates {}            // Appropriate for the type data
  Dataset connectivity {}           // Appropriate for the kind data
}//End of grid1
```

Comparison with the grid1 dimensions concluded that E and B have 2 components. Components E_0, E_1, B_0 and B_1 are variables, and an extra variable P is defined.

A time dependent variable as 1D plot

```
Group A {
  Dataset totalEnergy {
    Att vsType = “variable”         // This is a variable
    Att vsMesh = “timeSequence1”    // Name of the mesh, required
  }
  Group timeSequence1 {
    vsType = mesh                   // This is a mesh, required
    vsKind = “irregular”            // Kind of the mesh, required
    Att values = [t0, t1,.. tn]     // Appropriate for the kind data
  }
}
```

This example exposes a variable totalEnergy plotted against time with times defined in a group named “timeSequence1”. Note that, although physically timeSequence is a list of instances, for the viz purposes it is marked as a mesh.

VariableWithMesh

This section is supposed to give examples of how to define variables on meshes defined within the data.

VariableWithMesh in a dataset

```
Group C {
  Dataset electrons {
    Att vsType = “variableWithMesh”        // This is a variable containing mesh
    Att vsNumSpatialDims = 3               // Required
  }//End of Electrons

 Group vsVars {
    Att Vx = “electrons_0”
  }    // Extra variables

}//End of C
```

We are following VORPAL example for now. Such data is supposed to give 2 dimensions and always represents many components, which are found using the value of vsNumSpatialDims. If the data is component-minor (like in this example), first 3 columns are interpreted as coordinates of an unstructured mesh. The remaining columns are variables with the indices offset by the value of vsNumSpatialDims (for example, in our case electrons_0 is the 4th column of the dataset). Extra variables can be defined in the terms of these components (Vx in this example). In the case of component-major, we deal with rows, instead of columns and need to add the vsOrder = “compMajor.”

Multi-component vs single-component variables

VSH5 compares mesh dimensions to variable dimensions. Dimensions of variables are defined using HDF5's dataspaces. For example, the following variable has dimensions 200x200x104x2:

```
Dataset d3dVacVess {
      DATASPACE { ( 200, 200, 104, 2 ) / ( 200, 200, 104, 2 ) }
      Att vsKind = "varible"
      Att vsMesh" = mesh1
}
```

The mesh's dimensions can be defined in various attributes and depend on a kind of a mesh. For example, this mesh defines its dimensions in the attribute totalNumPhysCells to be 200x200x104:

```
Group mesh1 {
      Att vsType = "mesh"
      Att vsKind = "uniformCartesian"
      Att startCell = [0, 0, 0]
      Att totalNumPhysCells = [200,200, 104]
      Att lowerBounds = [-2.5, -2.5, -1.3]
      Att upperBounds = [2.5, 2.5, 1.3]
}
```

If the dimensions of such mesh is the same as dimensions of the variable, VSH5 treats the variable is a scalar quantity mapped to the mesh. If the variable's dimensions are a multiple of the mesh dimensions (2 in the case above), VSH5 defines an array and break out the components (2 in the case above).

The number of meshes of different kinds is defined differently for each kind. It should be added to getMeshDims method in VsH5Reader class.

The variable is only recognized as a vector (in the VisIt sense) if it is declared as such in a vsVars group. In other cases such as the variable size being less than the mesh size, or larger but not a multiple, the variable cannot be mapped to the mesh.

In the case of "variableWithMesh" variables, the value of the attribute vsNumSpatialDims indicates how many components of the variable's dataset contain spatial coordinates. The remaining components are treated as data components.

Meshes

We have specifications for three kinds of meshes: uniform Cartesian, structured and unstructured. A uniform Cartesian mesh is a specialized form of structured mesh in which the mesh divisions are located at regular intervals along Cartesian axes. Example of a uniform Cartesian mesh:

```
Group mesh1 {
      Att vsType = "mesh"
      Att vsKind = "uniformCartesian"
      Att startCell = [0, 0, 0]
      Att totalNumPhysCells = [200, 200, 104]
      Att lowerBounds = [-2.5, -2.5, -1.3]
      Att upperBounds = [2.5, 2.5, 1.3]
}
```

All the attributes are required.

A structured mesh has indexed vertices which are not necessarily regularly spaced or aligned with the coordinate system, and so is specified simply as an array of mesh points. The format of a 3D structured mesh:

```
Dataset mesh3 {
      DATASPACE [n0][n1][n2][3]
      Att vsType = "mesh"
      Att vsKind = "structured"
}
```

The values (n0, n1, n2) correspond to the mesh indices. All the attributes are required.

2D and 1D structured meshes have fewer indices, like so:

```
Dataset mesh2 {
      DATASPACE [n0][n1][2]
      Att vsType = "mesh"
      Att vsKind = "structured"
}
Dataset mesh1 {
      DATASPACE [n0][1]
      Att vsType = "mesh"
      Att vsKind = "structured"
}
```

An unstructured mesh has 2 required attributes (vsType, vsKind), 1 required dataset (points), and one or more datasets describing connectivity (for example, polygons, tetrahedral and hexahedra). For example, VORPAL specifies a mesh for some surfaces using this format:

```
Group poly {
       Att vsType = "mesh"                          // Required
       Att vsKind = "unstructured"                  // Required
       Dataset points [num_points][ndims]           // Required
       Dataset polygons [num_polys][num_verts]      // Optional
}.
```

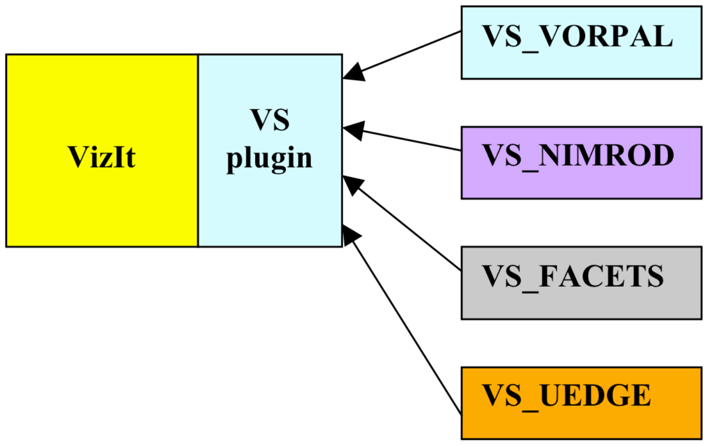


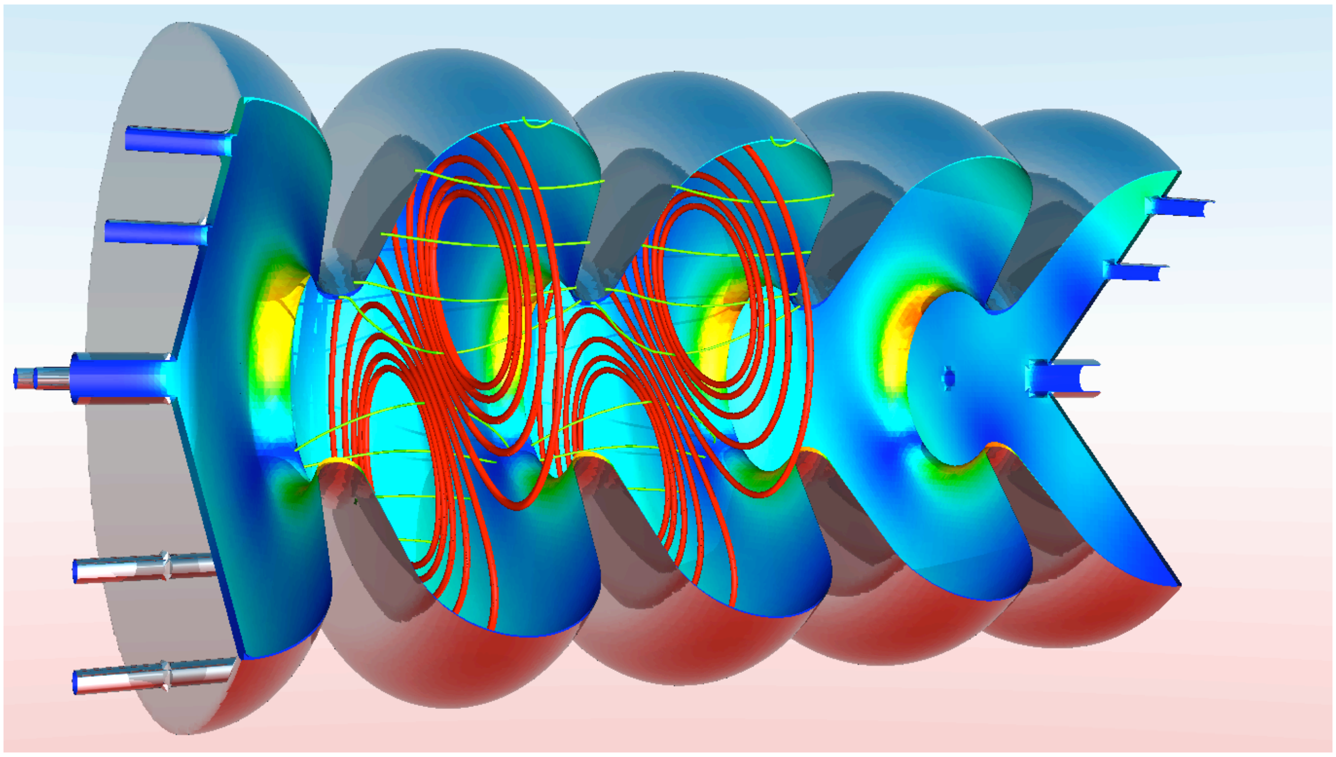
Figure 3. The latest version of VizSchema approach.

VizSchema libraries

Based on the markup, we developed a C++ library which reads visualization data into arrays. Based on this library we developed a new VisIt plugin called Vs (see Figure 3).

The markup described above was accepted by several applications: VORPAL (see Figures 4-5), FACETS (see Figure 6), UEDGE (see Figure 6), NIMROD (see Figure 7), MODAVE (see Figure 8) and the plugin was tried by several members of the VisIt team.

Figure 4. Simulation of the crab cavity using VORPAL data. The data was brought up into



VisIt using the Vs plugin and further rendered using POVRAY ray-tracing tool.

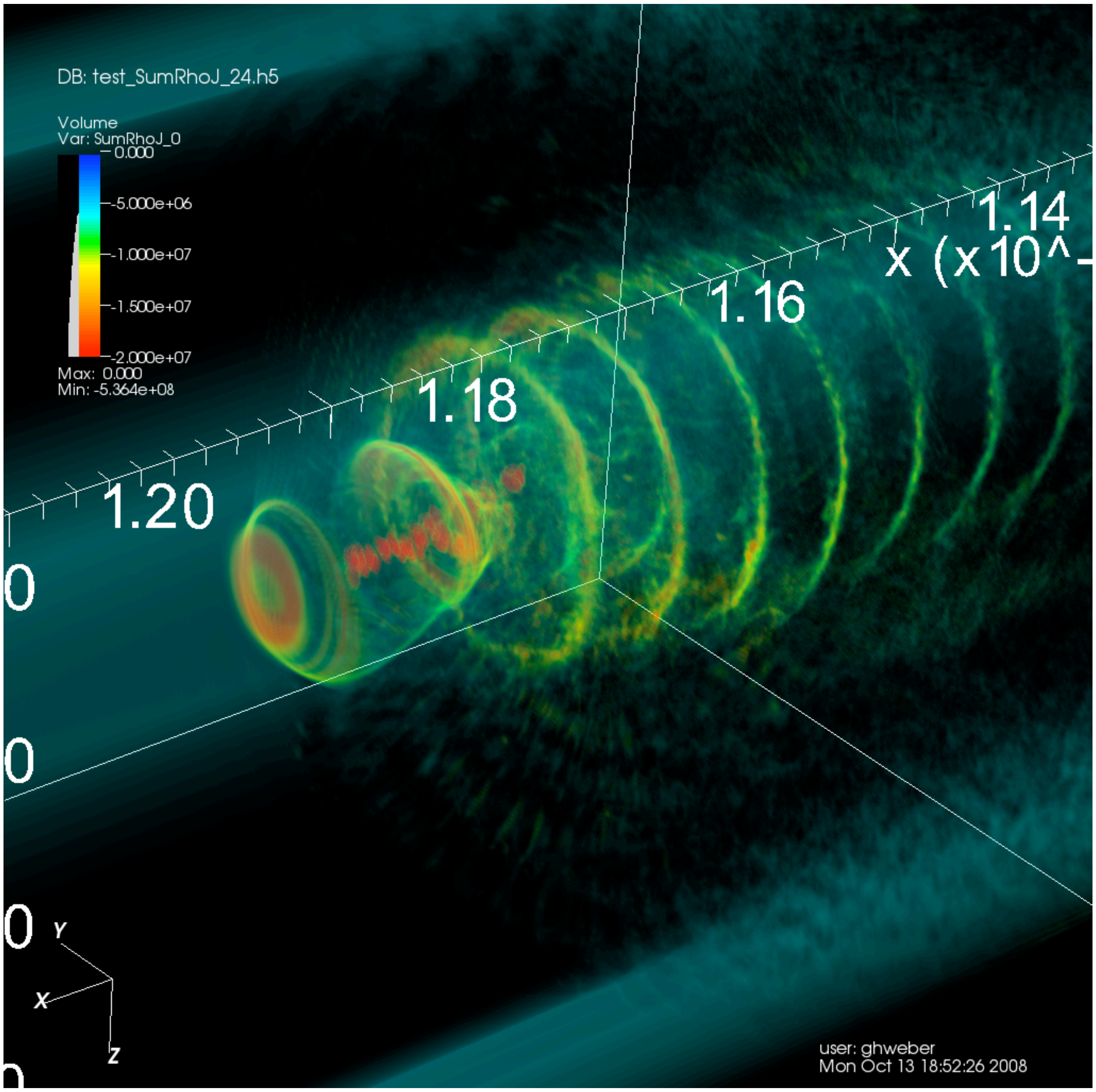


Figure 5. Wakefield visualization obtained using VORPAL simulation and Vs plugin (courtesy of Gunther Weber and Cameron Gedes).

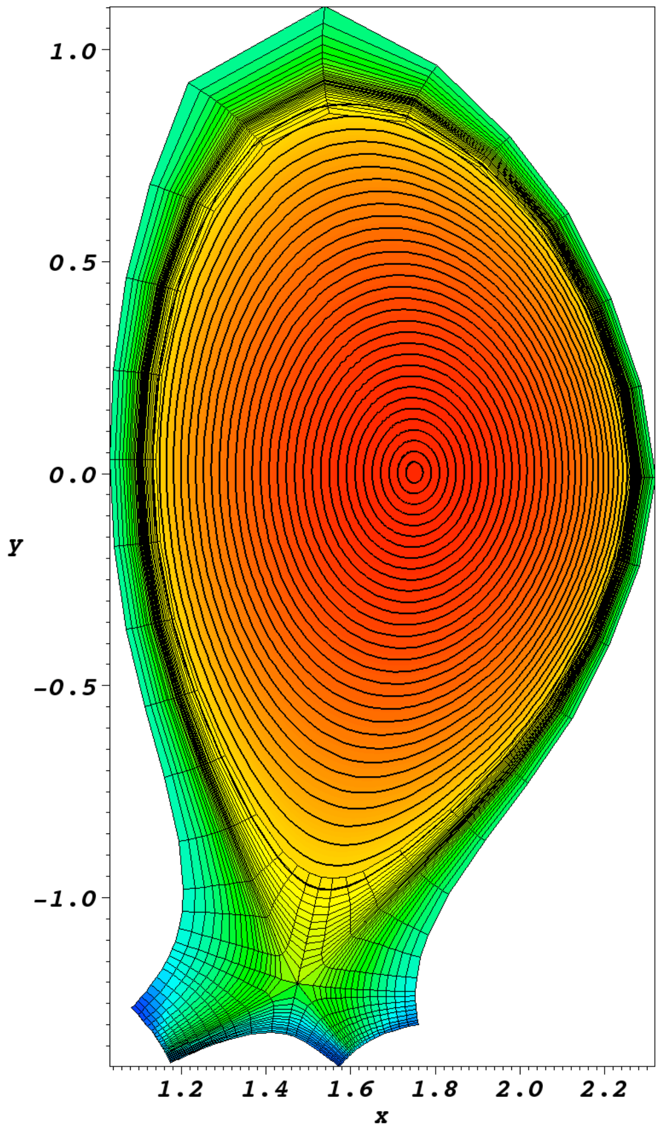


Figure 6. FACETS simulation of Core and Edge (UEDGE) components.

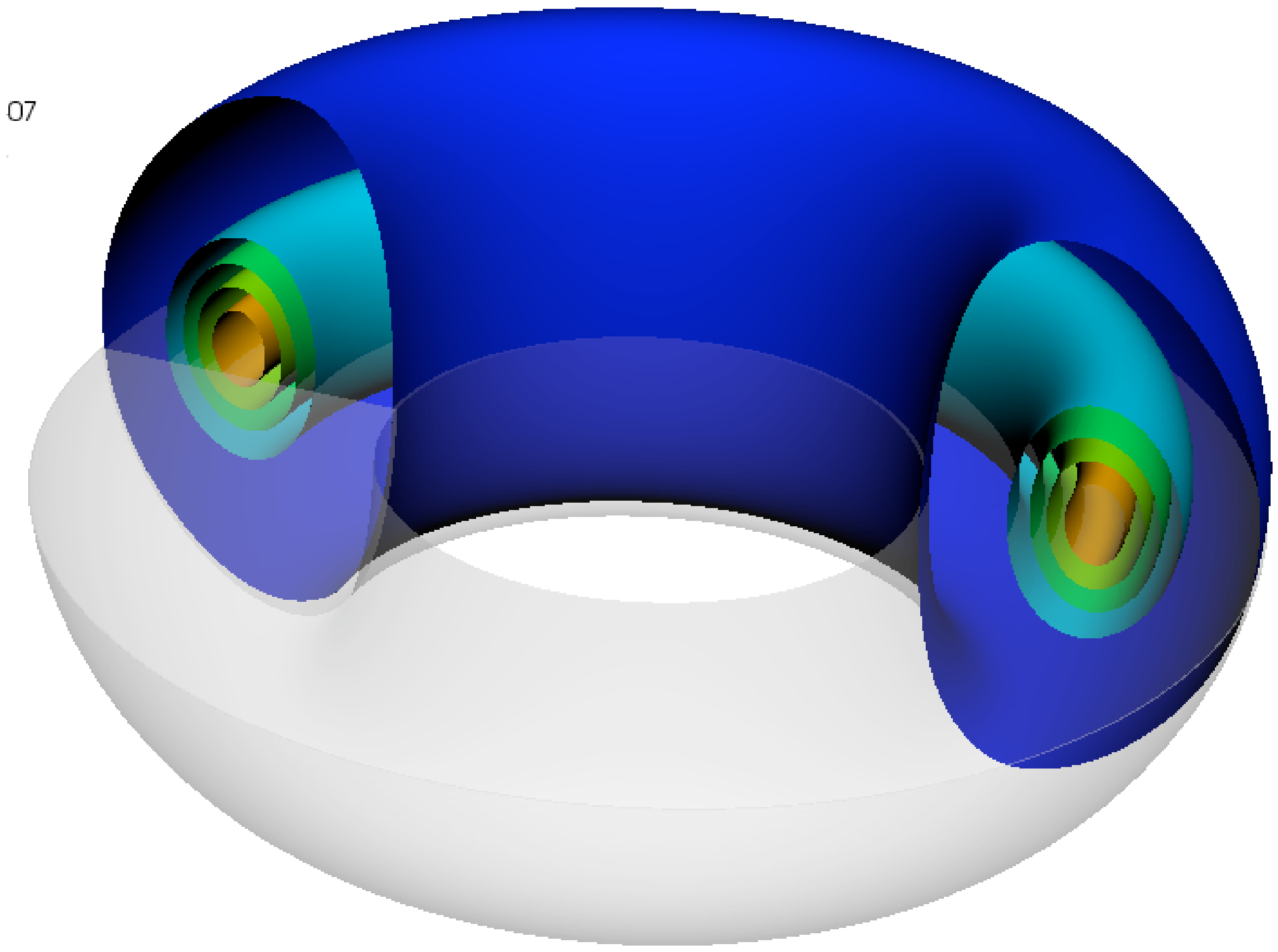


Figure 7. NIMROD's data visualized using the Vs plugin.

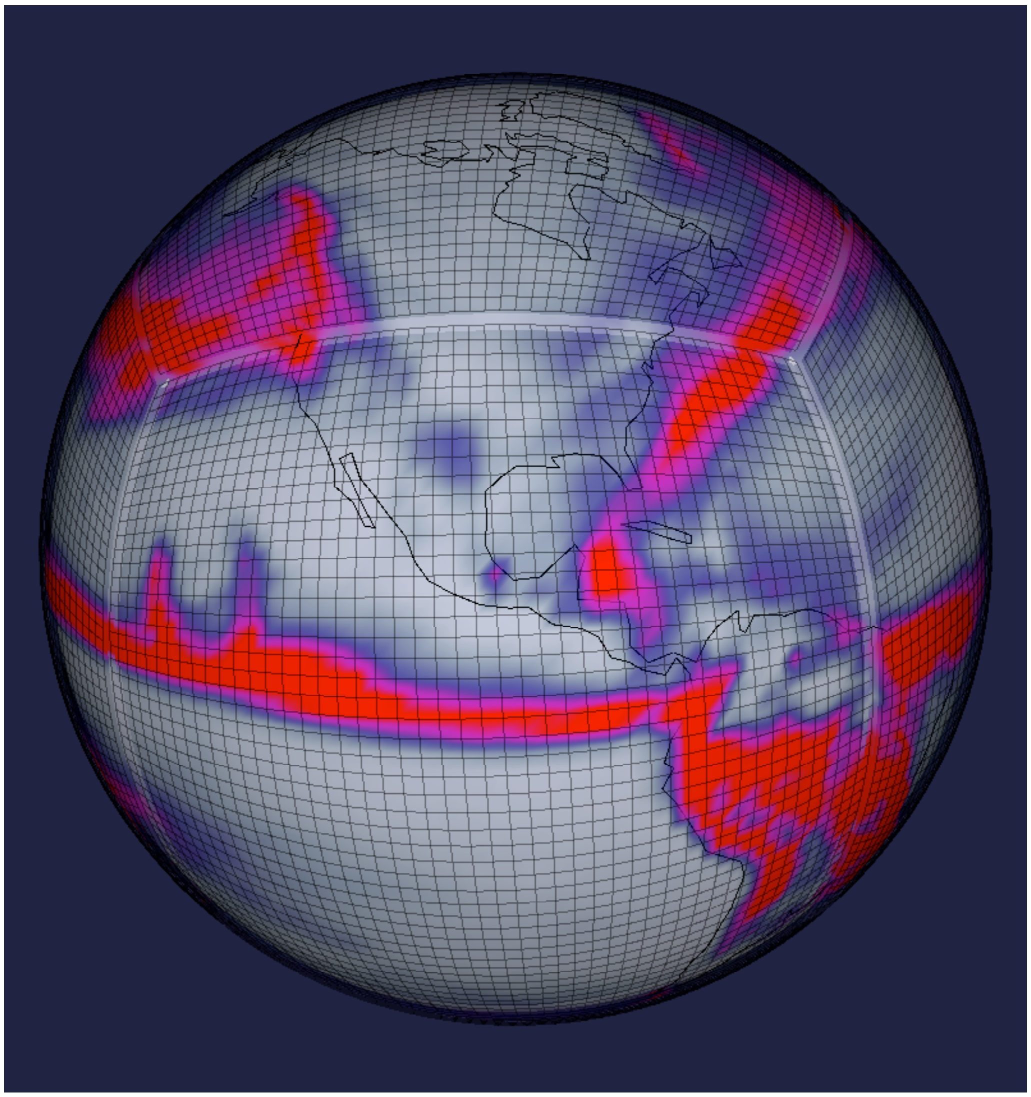


Figure 8. Precipitation data from a climate modeling code MODAVE. Visualized using the Vs plugin.

4. Conclusion

The software developed in this project (VisSchema) is being used by multiple teams: FACETS (Fusion SciDAC), COMPASS (HEP/NP SciDAC using VORPAL), VACETS (Visualization SciDAC) and CEMM (Fusion SciDAC).

Please contact Svetlana Shasharina (sveta@txcorp.com) for more details.