Community **C**oordinated
Modeling **C**enter

Kameleon Conversion Software Version 6.01 January 2012

Nitesh Donti National Aeronautics and Space Administration **M** 2404497919 **E** ndonti21@gmail.com **W** http://ccmc.gsfc.nasa.gov/

Table of Contents

Adding a new Input Model to Kameleon

Existing Input Models

Currently, we support the following input models in version 6.0:

1. Enlil

2. SWMF Ionosphere

3. HDF5

In Development:

Open GGCM

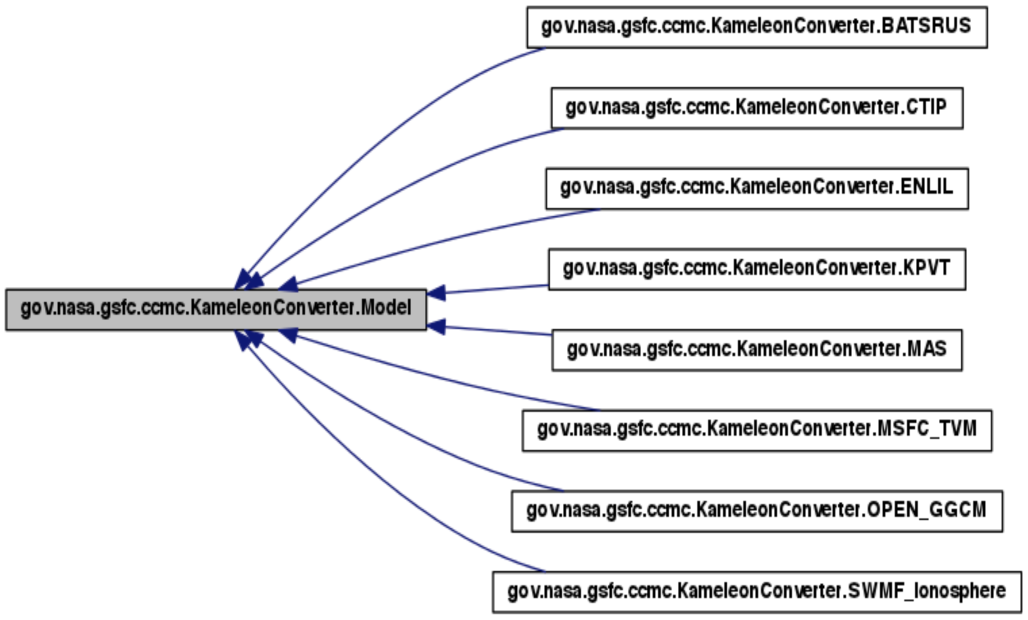
MAS

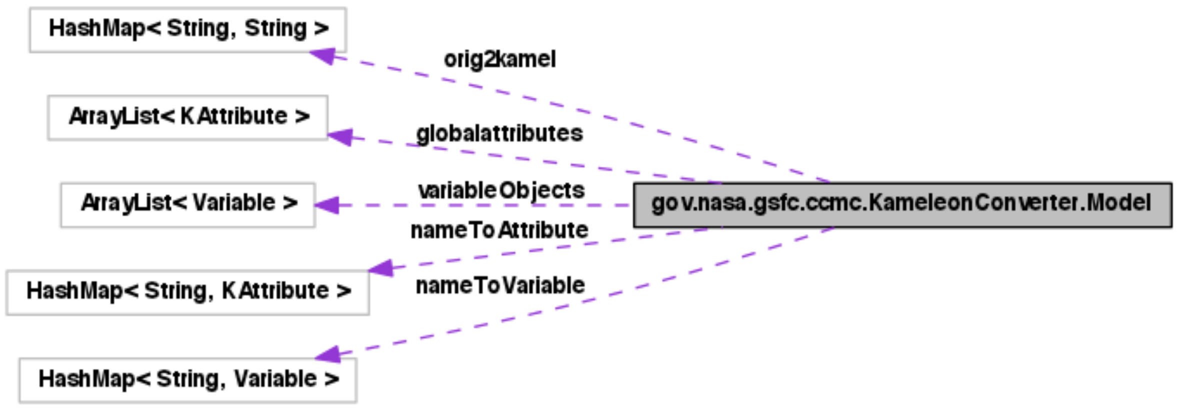
BATS-R-US

CTIP

KPVT

MSFC TVM


gov.nasa.gsfc.ccmc.KameleonConverter.Model
gov.nasa.gsfc.ccmc.KameleonConverter.BATSRUS
gov.nasa.gsfc.ccmc.KameleonConverter.CTIP
gov.nasa.gsfc.ccmc.KameleonConverter.ENLIL
gov.nasa.gsfc.ccmc.KameleonConverter.KPVT
gov.nasa.gsfc.ccmc.KameleonConverter.MAS
gov.nasa.gsfc.ccmc.KameleonConverter.MSFC_TVM
gov.nasa.gsfc.ccmc.KameleonConverter.OPEN_GGCM
gov.nasa.gsfc.ccmc.KameleonConverter.SWMF_Ionosphere


HashMap< String, String >
orig2kamel
ArrayList< KAttribute >
globalattributes
ArrayList< Variable >
variableObjects
gov.nasa.gsfc.ccmc.KameleonConverter.Model
nameToAttribute
HashMap< String, KAttribute >
nameToVariable
HashMap< String, Variable >

Finding the correct input files to use

The directory that holds the input files with all of the data should be entered as the argument for Input Directory, in the Command Line Interface.

Within this directory, you must **find which files are necessary** (contain information about the global attributes, variables, variable attributes, and variable data).

Typically, these files will all have the **same extension or prefix**. So, a very **simple parser** should be able to parse through the files in the directory and seek out the desired ones.

When the files are found, add their abstract pathnames (using getPath()) to the ArrayList pathholder, which will be turned into an array of Files called XFiles, from which all of the variables, etc. will be made.

Make sure to use the new model_key that was assigned to this new input model and place all the parsing code, etc. in a special case within the switch statement.

Customizing the Timestep name for the InputModel

```
//Assigning the command line arguments to the new model object
```

A for-loop will go through all of the files from which the converter will take information. The first thing to do here is create a new Object and assign it to "instance", which represents the InputModel Object for each timestep.

In order to name each timestep (which will be shown in the name of the newly converted file), **create a simple line parser** and set their part of the filename to the timestep of the InputModel.

Please see the example below:

```
case 4:
        instance = new ENLIL();
        start = Xfiles[i].toString().indexOf(".");
        end = Xfiles[i].toString().lastIndexOf(".");
        instance.setTimestep(Xfiles[i].toString().substring(start+1,end));
        break;
```

In this example, if the pathname were "/Users/John_Smith_Enlil/tim.0056.nc", then the name of the timestep for this Object would be "0056". The purpose of the timestep name is to differentiate among different files that are converted within the same project.

Also:

```
case 8:
        instance = new SWMF_Ionosphere();
        start = Xfiles[i].toString().lastIndexOf("/");
        end = Xfiles[i].toString().indexOf(".");
        instance.setTimestep(Xfiles[i].toString().substring(start+1,end));
        break;
```

Here, if the filename for one timestep were "/Users/John_Smith_SWMF/it_20110123_153034_0000.tec", then the timestep name would be "it_20110123_153034_0000".

Creating the necessary XML documents for the new input model

Create an XML file entitled “InputModel.xml”.

Make a start tag entitled “Model” with an attribute “name = InputModel”:

```
<Model name = "enlil">
```

The **first kind of Element** will be “attribute”. These elements will be the global attributes specific to each input model type. Some global attributes that will always need to exist are the README, model_name, and run_type.

The “attribute” element’s content will include **four child elements**: “name” (name of the global attribute), “description” (a description of what the global attribute represents), “dataType” (a string denoting what the data type of the value will need to be) and “value” (the value of the global attribute).

Please see the example below:

```
<attribute>
        <name>output_type</name>
        <description>Define the type of output that is contained in
the file (e.g. Global Magnetosphere model with Ionosphere output)</
description>
        <dataType>String</dataType>
        <value>Heliosphere</value>
</attribute>
```

The **second kind of Element** will be “variable”. These elements have to do with an input model’s variables. From the input file, each variable will come with a name (that we call the “original name”). However, oftentimes when converting the information to a different format, we give each variable a Kameleon CCMC standard name.

The “variable” element’s content will include **two child elements**: “OriginalName” (the name of the variable from the input file) and “KameleonName” (the name the CCMC wishes to call the variable). The child elements will be mapped together in a dictionary.

Please see the example below:

```
<variable>
        <OriginalName>V2</OriginalName>
        <KameleonName>utheta</KameleonName>
</variable>
```

Lastly, finish the XML document with an end-tag like so:

```
</Model>
```

Registering a new Variable with the Kameleon Software

If your new Input Model comes with variables that are not yet registered with the Kameleon Software, then you must update the existing records. In order to check if this is the case, go to the “variables.xml” and **see if all necessary variables** that come from the input file **exist** in this XML document. If not, please add to the document.

The Element is entitled “Variable” and its child elements have the names of the CCMC Standard Variable Attributes, in addition to the group “category”.

Please see the example below:

```
<Variable>
   <category>density</category>
   <name>Tn</name>
   <dataType>float</dataType>
   <classification>data</classification>
   <valid_min>0.0</valid_min>
   <valid_max>1000000.0</valid_max>
   <units>K</units>
   <grid_system>grid_system_1</grid_system>
   <mask>FLOAT_MASK</mask>
   <description>Neutral Temperature</description>
   <is_vector>false</is_vector>
   <position_grid_system>grid_system_1</position_grid_system>
   <data_grid_system>grid_system_1</data_grid_system>
</Variable>
```

Registering a new Variable Attribute with the Kameleon Software

In VariableCCMCAttributeNames.xml, **add newattribute as an element** like so:

```
<attribute>
     <name>newattribute</name>
     <description>description of what newattribute is </description>
     <datatype>datatype of the attribute’s value </datatype>
</attribute>
```

In variables.xml, **add a new child element to each Variable** element like so:

```
<newattribute>value of newattribute specific to this variable</newattribute>
```

Go to VariableXMLParser.java and **add newattribute to String[] tags**.

Registering a new Global Attribute with the Kameleon Software

If the global attribute will be **the same for each InputModel Object**, then go to the XML document entitled “InputModel.xml” and add an element in the following pattern:

```
<attribute>
    <name> </name>
    <description> </description>
    <dataType> </dataType>
    <value> </value>
</attribute>
```

Each new instance of the InputModel will have this new global attribute with the values you entered in the XML document.

If the global attribute will be **different for each InputModel Object**, then go to the XML document entitled “GlobalCCMCAttributeNames.xml”, and add an element like so:

```
<attribute>
    <name> </name>
    <description> </description>
    <datatype> </datatype>
</attribute>
```

Then, make sure you assign this global attribute a value somewhere in your code. Most global attribute values will come from the Database Info file, which is parsed by DatabaseInfoParser.java.

Auxiliary Files

Some input model types come with an auxiliary file, used to supplement the main data and attributes. Some even require multiple auxiliary files. Generally, if any are necessary, then the Command Line Interface will allow the user to specify a directory or file from which to get the desired information.

Currently, the following input models require an auxiliary file(s):

- Open GGCM

The following do not require any auxiliary files:

- Enlil (previously required auxiliary files)
- SWMF

In order to specify whether a model requires any auxiliary files, go to the createModel() method in CommandLineInterface.java.

Creating the custom read() method for a new Input Model

Before you begin:

- Create a class called InputModel.java.
- Create a new toString() method in the form `return "InputModel Object"`
- Create a new constructor with a body of `super`(“`InputModel.xml`”)
- Update CommandLineInterface to mention InputModel when it speaks about input models.
    - Update the key value reference table (assign InputModel a switch number)

Now you can start the read() method. The read() method for each new Input Model will be very unique. However, each read routine is required to extract the same three pieces of information from the file: the **global attributes**, the **variables**, and the **data of the variables**. In many cases, a fourth piece of information – certain **specified or updated attributes of the variables** – will also be collected. For this reason, be sure to add the original variable attributes to the variables first (which come from the InputModel.xml file that you must create), and then overwrite them with the more updated versions from the input files.

To begin, call `super.read();`

The next order of business is to figure out how to read all of the information stated above (in bold) from the InputModel’s Files.

If the file is a text file, you must create a method to parse all of the required data. Usually, **Scanner** is a helpful class to use to parse text. Some methods include .hasNext()/.hasNextLine(), which checks to see if there is another token/line available. To get the next token, use .next(), or use something else to grab the next token as a specific data type (e.g. .nextFloat(), .nextInt()) without making any casts.

If the file is unique, you must research how to access the necessary information through a website, guide, or Java API. Each case will be different.

```
//mapping original variable names to kameleon variable names
```

When you get the **names of the variables from the input file**, be sure to map them to the accepted Kameleon names with the HashMap o2k.

If new variables that are read-in that have not already been updated in the variables.xml and the InputModel.xml documents, then assign each name to "kameleon_identity_unknown_"+i, where i is an integer from 1 – 20, separate for each unknown variable.

Please see the example below:

```
if(o2k.get(varName)!=null)
       this.addVariableObject(new Variable(varName,o2k.get(varName)));
else
       this.addVariableObject(new Variable(varName,"kameleon_identity_unknown_"+i));
```

Without knowing what the registered kameleon names are, the interpolator will not know how to handle the data and cannot move the Kameleon process any further.

```
//adding Variable Attributes to each variable
```

In order to get the variable attributes and attribute values for standard CCMC variables, invoke the `getCCMCstandardattrs(Variable var)` method within your read method for each variable. This will set the values that have been registered in "Variables.xml". If your file comes with its own attributes and attribute values, make sure that these overwrite or add on to the original versions.

```
//additional required fields for each variable
```

Certain **fields** that you **need to add to each Variable Object** while reading in:

- **String dt**: the data type of the values of the variables (in String form, e.g. “float”)
- **int numElem**: the number of elements of data
- **int numDim**: the number of dimensions in the original file (data will be flattened later)
- **int[] dimSizes**: an int array of the number of spots in each dimension (not a long[])
- And of course, you need to read in to the **Object dataValues**. Since this is specified as merely an Object, as opposed to a float[] for example, you must use casting appropriately when assigning values and creating your information. At the end, however, it will in fact be an array of a primitive data type.

When new OutputFormats are created, they may need even more pieces of information from all InputModels’ read methods. Keep an eye out for this change when new OutputFormats are introduced.

Finally, when adding variables and global attributes to the InputModel object, remember that you are adding these to one timestep at a time. So in your code for InputModel.java, you may write `this.addGlobalAttribute` and `this.addVariableObject`.

Also, keep in mind that some InputModels will have to create some of their own global attributes, as opposed to (or in addition to) gathering them from the original input files.

The most important thing to remember about the read() methods is that each InputModel will read in and create the necessary information differently.

Adding a new Output Format to Kameleon

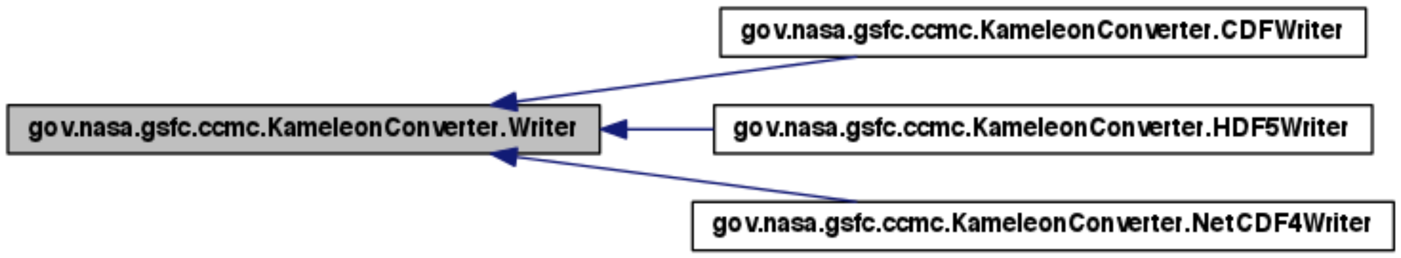
Existing Output Models

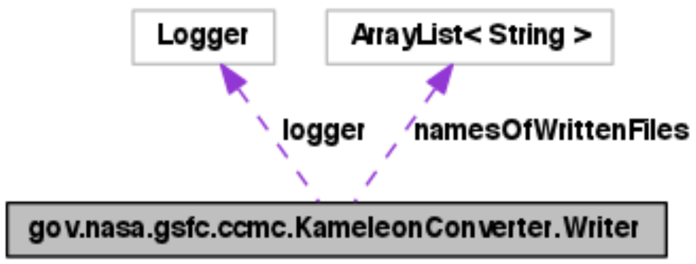
Currently, we support the following output formats:

1. CDF (Common Data Format)

2. HDF5 (HDF Group)

3. NetCDF4 (Network Common Data Format)


gov.nasa.gsfc.ccmc.KameleonConverter.CDFWriter
gov.nasa.gsfc.ccmc.KameleonConverter.Writer
gov.nasa.gsfc.ccmc.KameleonConverter.HDF5Writer
gov.nasa.gsfc.ccmc.KameleonConverter.NetCDF4Writer



Creating a the custom convert() method for a new Output Format

Before you begin:

- Create a class called “OutputFormatWriter.java”.
- Create a new toString() method in the form `return "OutputFormat File";`
- Create a new String extension() method in the form `return ".<file_extension>";`
    - For example, a NetCDF4 file ends in “.nc”, and an HDF5 file in “.h5”.
- Update CommandLineInterface to mention OutputFormat when it speaks about output formats.
    - In the “//checking --format argument for validity” section, follow the pattern to include OutputFormat. Note that when checking for the name, oftentimes the full name is not looked for. For example, if the format name is “PumpkinBox”, look for “pump” or “box.”
    -

Now you can start the convert(Model m) method.

First, call the `super.convert(m);`

Next, create a new OutputFormat file with the name `newfilename`, a field that comes from the Writer superclass.

Before beginning write, but after reading in your first variable, check to see that all 5 required pieces of information about each variable exist (dt, dimSizes, numDim, numElem, dataValues).

When writing the convert(Model m) method, keep in mind that **the goal is to write the following pieces of information** to the new OutputFormat file:

1. Variables
    a. Variable Attributes (CCMC Standard and Model Specific)
    b. Data Values
2. Global Attributes (CCMC Standard and Model Specific)

If given the option to create groups, create a group called Variables, in which you will place all of the newly created variables. The global attributes should be placed with the entire file, so place them in the root group.

To **retrieve the global attributes** of Model m, loop through its list of global attributes, and then **add them to the root group** of your new OutputFormat file. Use if-statements in order to find out the data type of each global attribute if you need to find out before adding them.

```
//if you need more information before you can write to the file
```

When writing your Variables, **use any of the fields from the model to give specifications** like Number of Dimensions of Number of Elements. If there is any information that you still need in order to write to your new OutputFormat, then you must create a field in the Variable.java class to hold that information, and then go to all of the subclasses of Model.java and write code to read in or create that information.

When writing your variables to file, add the variable, add the variables attributes, and add the data values. This process is different for each OutputFormat so make sure to look into how to do this. Order oftentimes will differ among OutputFormat types.

When adding the variable attributes, make sure that the **Model Specific attributes take precedence over the CCMC Standard attributes when there are duplicates**. In some cases, the client will attach a variable attribute to a variable, but its value will be different than the value that the CCMC has for that variable.

Writing the data to each variable can be the most difficult part. Some OutputFormats require lots of information to pass in as parameters for this action, so make sure to follow the examples from the API of the new OutputFormat.

Frequently Asked Questions (FAQ)

What script do I use to run the program through Command Line?

java -Xmx1024m -cp ./bin:lib/log4j-1.2.15.jar:lib/slf4j-api-1.5.6.jar:lib/slf4j-log4j12-1.5.6.jar:lib/netcdf-4.2.jar:lib/jargs.jar:lib/cdfjava.jar gov.nasa.gsfc.ccmc.KameleonConverter.CommandLineInterface -d [pathname of the Database Info File] -a [(conditional)pathname of the directory of input *auxiliary* files] -f [output *file* format] -m [input *model* format] -o [pathname of directory in which to place the converted *output* files] -i [pathname of the directory of *input* files]

If you have trouble making this work, simply run the code on your IDE with the appropriate program arguments as shown above and the VM arguments as shown here:

-Xmx1024m -cp ./bin:lib/log4j-1.2.15.jar:lib/slf4j-api-1.5.6.jar:lib/slf4j-log4j12-1.5.6.jar:lib/netcdf-4.2.jar:lib/jargs.jar:lib/cdfjava.jar

What else do I have to do when I first get this project onto my computer?

You have to add everything that is in the “lib” folder onto your Build Path. You can do this easily via Eclipse. Just right-click on the jar file and “Build Path” > “Add to Build Path”.

Also, find the Nujan jar file and add that to the Build Path as well.

Where do I state the mapping of the original Variable names to the Kameleon Variable names?

Go to the XML document entitled “[modelname].xml”. Put in an entry for <variable>. If any other variable names are found, they should map to “kameleon_identity_unknown”+i, where i is an integer from 1 to 20.

Where do I put the global variables that have a different standard value for each model, like the README or the model_name?

Go to the XML document entitled “[modelname].xml”. Put in an entry for <attribute>.

Where do I put the global variables that have the same value for every single model?

There should not be any, except for the Terms of Usage. If you need to add one, go to the Model() constructor and manually add it in there.

How do I read in data from an input file?

This process is different for every model. In some cases, you will have to find documentation online and other support in order to figure out how to access the information that you want. In other cases, you will simply have to parse a text file and put the information in the appropriate places. There is no general answer for the complicated reading process.

How can I convert just the first few files in the input directory as opposed to all of them, for testing purposes?

When initially entering arguments into the Command Line Interface, enter a "-t" followed by an integer representing the number of conversions you would like executed. The default situation converts all of the eligible files that are in the input directory. There is no support for selecting which of the eligible files to convert; the program will simply start with the first eligible file and keep going until it has reached your target number of files.

What are the Attribute XML files for?

The files that are entitled [Model]Attributes.xml just contain some information that was used in the previous Kameleon Converter. This information is not used at this point, but could be useful in the future. So for now, they do not serve any purpose but to store old information.

How do I view CDF and NetCDF files?

CDF is NASA's Common Data Format for the storage of vector, scalar and multidimensional data. Download NCBrowse or some other third-party application in order to view and manipulate raw data.

How do I use the Logger?

Implement a logger system of printing to the screen. To do this, write the following at the beginning of every class:

```
static Logger logger = Logger.getLogger(xxxxxxxxx.class);
```

- To print to the screen, write logger.info("...");.
- To print to the screen for debugging purposes, write logger.debug("...");.
- To print to the screen to report an error, write logger.error("...");.
- Import needed: `import org.apache.log4j.Logger;`

Any other notes?

- Minimize hardcoding information into specific classes. Try instead to keep the information in an XML document and then parse it in the code, so that the actual information is easier to change. If you see an opportunity to use XML, go for it.
- Follow the patterns you see. For example, if you are creating a model class, look at other model classes to see commonalities and patterns. The same goes for writer classes.
- Use the HDFView App to access and manipulate HDF5 output data.
- Make sure to update the Modification History in each java class you modify/add!

Credits

Name: Kameleon Conversion Software

Version: 6.01

Author: Nitesh Donti, Cornell University

NASA/GSFC/CCMC

Intern, Code 587

ndonti21@gmail.com

(240) 449 7919

Dates: June 1st, 2011 – August 5th, 2011; December 19th, 2011 – January 13th, 2012; December 17th, 2012 – January 11th, 2013

Acknowledgements:

David Berrios – Original Kameleon Software Writer and Mentor

Marlo Maddox – Original Kameleon Software Writer

Dr. Michael Hesse – Chief, CCMC

Community Coordinated Modeling Center, NASA Goddard Space Flight Center

Kameleon Conversion Software Version 6.0

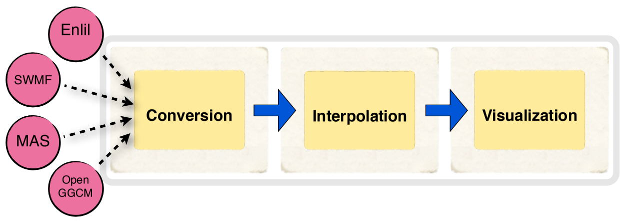
Nitesh Donti, Affiliation – Cornell University, GSFC Code 587
Community Coordinated Modeling Center, NASA Goddard Space Flight Center
Contact Information: nvd5@cornell.edu

Introduction

The Community Coordinated Modeling Center (CCMC) at NASA Goddard Space Flight Center (GSFC) has been developing the Kameleon Conversion Software over the past decade to address the difficulty in analyzing and disseminating the varying output formats of space weather model data. Written in C, a low-level programming language, the original software can convert model files from 7 different input formats to the standardized Common Data Format (CDF).

The software suite includes the conversion software, the data access and interpolation library, and the visualization and data analysis tools. After input models are converted, the interpolator reads the standardized files and arranges them so that the visualization tools can produce images and graphs to display the information.

File Flow



The Conversion Software is a key player in the CCMC's Runs On Request (RoR) system, which clients and researchers use to run space weather model simulations. The model runs themselves produce and generate the data that scientists use in their research.

Project Overview

Challenges

As the number of users grows, the number of requested models to convert and the number of requested formats to which to convert those models grow. Therefore, efforts need to be taken to make the existing software more flexible and more able to support new input models and output formats.

Goals

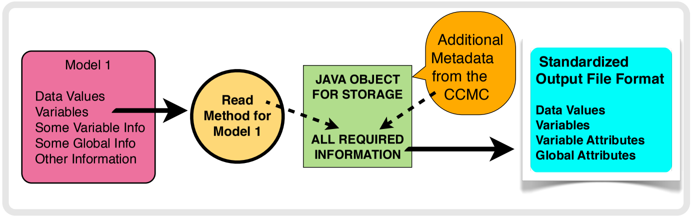
- Recreate the software using Java instead of C.
- Implement an Object-Oriented Programming model in order to facilitate the addition of new models and formats.
- Begin development of newer input model formats due to the increase in the number of models and the increase in usage of those models.
- Begin development of additional science data output formats for greater flexibility, as per customer requests.
- Allow programmers to easily read, write, and update model attributes and other stored information.

Complete Kameleon Conversion Redesign

Since the Original Kameleon, there has been little advancement of the converter, and no new formats have been supported. This is mostly due to the fact that the program was not easy to add on to. Not many developers, whether internal or external, wished to work with the C code.

In this new software, all of the code is written in Java, a high-level object-oriented programming language. Setting up, compiling, and running Kameleon on Java is much easier than on C, especially when on multiple servers. This saves a lot of time and effort when distributing the software amongst the members of the CCMC.

The conversion process consists of 3 steps:
1. Read in data from an input model file .
2. Store the data in a Java object and package it with additional metadata from the CCMC, in order to maintain uniformity.
3. Write the newly packaged data to a standardized output file.



Since the conversion is not directly from one format to another, adding writers and adding readers can occur independently of one another, which makes progression much simpler for developers.

Supported Input Models and Output Formats in Kameleon Conversion Software Version 6.0

Each of the models and writers is its own Java class, as seen in the image below.

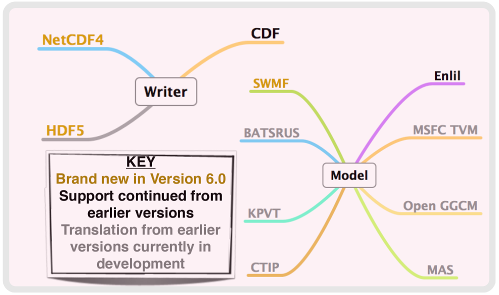
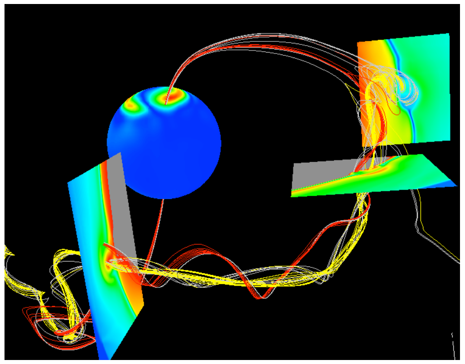

NetCDF4
CDF
Writer
HDF5
SWMF
BATSRUS
Enlil
MSFC TVM
Model
KPVT
Open GGCM
CTIP
MAS
KEY
Brand new in Version 6.0
Support continued from earlier versions
Translation from earlier versions currently in development

Image of a Visualized Model



New XML Document Implementation

In order to facilitate the updating/modifying of information about certain models, formats, and attributes, the new Kameleon Conversation Software utilizes XML documents to store this information.

XML is a textual data format with strong support for many programming languages. XML is human readable and easily parsable. Its main advantage is that it can separate the content from the code.

A client or developer can change a small piece of information (metadata) about an attribute or format without even touching the Java code. The program simply accesses the information in the XML document with a parser. Therefore, any changes in the XML document would not necessitate recompilation of the program, thus saving time and effort.

For example, one could take a variable called "Velocity" and modify an attribute called "units" via the XML document without writing any code.

All clients who wish to introduce new models, variables, attributes, etc. will be required to provide us with certain information to put into XML.

Advantage of the Standardized Model Files over the Original Input Model Files

Each original input model has a unique format. Oftentimes the model can be difficult to read and use. In addition, each contains slightly different information. When we convert these input models to standardized formats, we also append additional metadata from the CCMC and package the file into its final, standardized format. After conversion, all files will follow the same pattern and have the same structure. Every file will hold all of the required information needed to interpolate and visualize the data.

Acknowledgements: David Berrios – Program Mentor & Original Kameleon Software Writer
Marlo Maddox – Original Kameleon Software Writer
Dr. Michael Hesse – Director, CCMC
Community Coordinated Modeling Center, GSFC
Funding Support from New York Space Grant Consortium