INL/EXT-09-17168

MRDAP User/Developer Documentation

Joshua Cogliati
Michael Milvich

September 2009

INL/EXT-09-17168

MRDAP User/Developer Documentation

Joshua Cogliati
Michael Milvich

September 2009

Idaho National Laboratory
Reactor Physics Analysis & Design
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Nuclear Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

DISCLAIMER

Table of Contents

1. User Manual

2. Introduction

The Multi-Reactor Design and Analysis Platform (MRDAP) is designed to simplify the creation, transfer and processing of data between computational codes. MRDAP accomplishes these objectives with three parts:

— allows each integrated code, through a plugin, to specify the required input for execution and the required output needed,

— creates an interface for execution and data transfer,

— enables the creation of Graphical User Interfaces (GUI) to assist with input preparation and data visualization.

Reactor physics calculations, or in general many design calculations, often requires the use of multiple analysis codes. In many cases, data generated by one code is subsequently used as input by another code. Several methods have been used to transfer the data between multiple codes. The most rudimentary method is to simply transfer the data manually by copying certain sections of the output file and inserting them in the input to the next code. This is time consuming, inefficient and prone to errors. Furthermore, the manual transfer of data makes tasks that require repetitive execution and/or making small changes to the input (such as in the case of parametric studies) time consuming and unreliable. An alternative approach is to use shell scripts to automate sequential code execution and input preparation. While this simplifies the transfer of data and enables the user to prepare input more readily, the nature of shell scripts constraints their applicability to other analysis codes. This is due to the fact that, typically, the paths to certain files and libraries are hard coded into the script. Also, due to the fact that scripts primarily execute a series of predefined commands, if used by an inexperienced user, it can fail in several ways when applied outside its predefined scope. The lack of robustness of a scripting system can also waste resources, such as in the case of a failed code execution that generates necessary input for subsequent codes, in which case the automated scripting may very well continue, or may fail later giving the wrong location for the error. Second, if some of the upstream data has not changed, the script will still recalculate the data at that location. While scripts can be created that do not have those problems, robust scripts are time consuming, complex to create and usually not user-friendly. The aforementioned methods of connecting codes lack the robustness that one can achieve by implementing modern practices, languages, and data management systems to automate sequential input preparation and code execution.

MRDAP embodies an attempt to achieve these goals by simplifying reactor physics calculations by streamlining the transferring of data between codes. MRDAP also provides ways for GUIs to be created that can be used to generate or view data for codes. These capabilities can be added even when using existing codes, allowing easier processing and manipulation of data.

Ultimately, the main motivation of this work is enable analysts (who perform reactor physics calculations routinely), by providing a tool that increases efficiency and minimizes the potential for errors or failed executions.

3. Concepts

4. Concept Overview

5. Purpose of Project

- Connecting different nuclear codes
- Providing an archival format for simulations of nuclear designs
- Providing a GUI for existing nuclear codes
- Automating data transferring
- Simplifying using multiple nuclear codes

6. Methodology

- Separation of code running and GUI editing (allows workflow to be run without interaction)
- Flexible backend storage of data (currently Postgresql and XML supported)
- System handled data transfer between different codes
- Extensive use of plugins. Calculation engines, data editors and data types are plugins.

7. Glossary

Workflow – describes the plugin used to calculate some data. The workflow describes how the input data is transferred between computational plugins so it can be transformed into the output data.

Data plugin – a plugin that defines a data type.

Computational plugin – a plugin that is used in the workflow to compute output data from input data.

Core System – everything that is not a plugin.

Unit – a plugin that can be used in the workflow to do some type of processing. These include computational plugins as well as other types such as data creators and the database handling units.

Transfer lines – lines connecting units in the workflow.

8. Data System

The MRDAP data system handles the transferring and storing of data. The data is stored using a data source. This currently can be either an XML file or a Postgresql database. All the data that is stored is placed in this data source including the workflows and the data store information. Computational plugins can have their own data such as cross sections that are separate from the data system.

Other back ends including other database systems or other data systems such as HDF are possible, and could be done with only changing the data store, instead of the entire system.

Since the system knows which inputs are required, sometimes it can parallelize the computation. For example, if there are two computational plugins which have all their inputs satisfied, the system will run both at once.

Inside each datastore multiple projects can be created. Inside each project the different types of data are stored such as workflows and other data. Data from workflows can be stored using the datastore saver unit. This can be used in other workflows by using the datastore loader unit.

9. Plugin System

MRDAP allows many parts to be extended by the addition of plugins. New units can be created to allow different types of data to be created, or to allow new computations to be done. Plugins can be created to allow different types of data to be saved. Details on creating plugins are found in the developers documentation. Plugins are loaded when the program starts up.

10. Design Concepts

One of the goals is that the system stores all the information required to rerun any workflow. This allows the data to be used as a self contained archive format. This ability can be violated if for example plugins are used that read extra data from the disk.

A key concept is the separation of the GUI editors and GUI-less computation modules. This allows the computation to proceed independently of user interaction. It also allows the computation to be done with a command line tool instead of the GUI (this is weakly supported with the current code).

GUIs can be created via plugins for any type of data that the system handles. Currently, most data is still created with just string editors.

11. Workflow Editor

Workflows are created and run using the workflow editor. New workflows can be created with the new object button once a project has been opened. The workflow can be edited by double clicking on the name in the object browser.

The units that can be used are listed on the right. They can be dragged to the panel where they can be connected with transfer lines to other units. Double clicking on a unit opens up it editor. This allows various parameters for the unit to be set (not all units have editors). Right clicking on the unit gives various options. These include the run log that stores details about how the unit ran, and options to rename and remove the unit.

Inputs are marked with a arrow, outputs are just a plain line. To create a transfer line, drag from an output, to an input. Multiple transfer lines can start from one output, for these the output is duplicated, but an input can only have one connection The outputs and input locations relative to the unit can be changed by clicking and dragging with the control key held down.

The transfer lines can be edited. Right clicking on a line brings up a menu that allows a point to be inserted. Clicking on the line will highlight the line and indicate where each point is in the line. The points can then be dragged. This allows the lines to change direction and be routed around other objects to keep them neat.

The view of the workflow can be zoomed in and out with either the mouse wheel or the drop down menu at the side.

Multiple units can be selected by using the shift key while clicking. They can also be selected by dragging a box around them. These units can then be moved as a group.

Workflows can be run when they have been opened for editing. When this is done, the program chooses units that have all the data they need to run, and runs them. Then, with the additional data new units are chosen and they are run. When none of the units have all of their inputs satisfied, the program finishes. This usually results in data being stored so it can be examined.

Each workflow can have multiple configurations. Each configuration is a set of parameters that are used for each unit. Configurations allow one workflow to be used for multiple cases. Each case has it's own separate data storage and checkpoints. The configurations can be cloned and individual units can copy their data from a different configuration. There is a default configuration that other configurations will use if they don't have a parameter overridden. Configurations allow the units to have different context values. (So for example a integer expression could have different numbers and a string creator could have a different string.) Which units are used and all the connections that are used need to be the same for all the different configurations.

The workflow has a document size, that determines how big the space to work is. This can be changed by right clicking on the workflow's editing area, and choosing Properties, and then entering the width and height of the workflow.

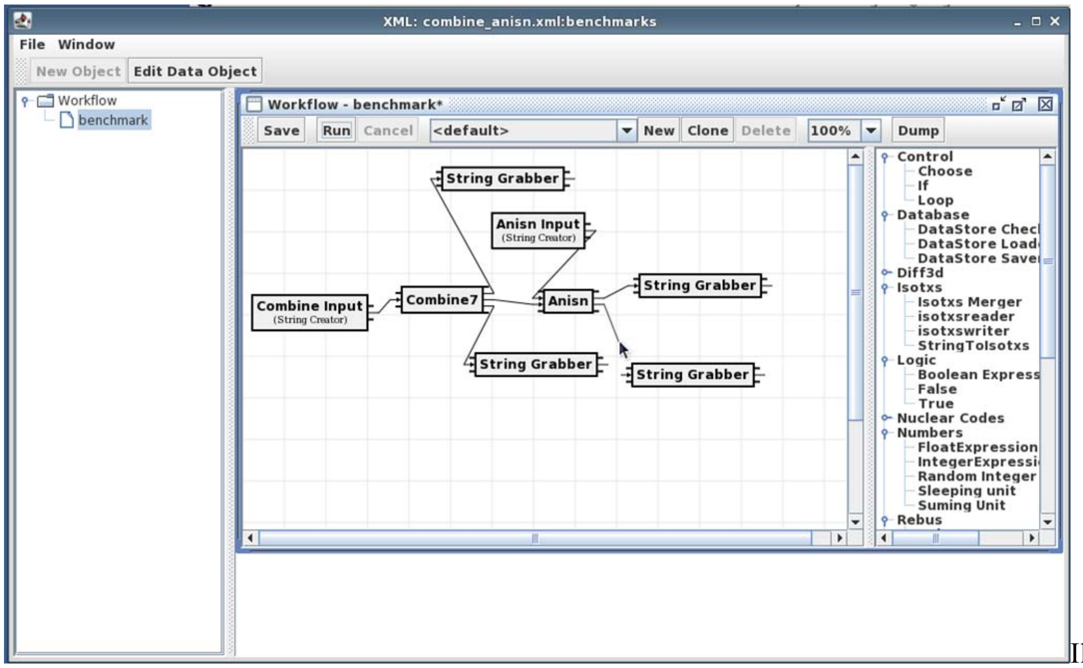

Illustration 1: Editing a workflow

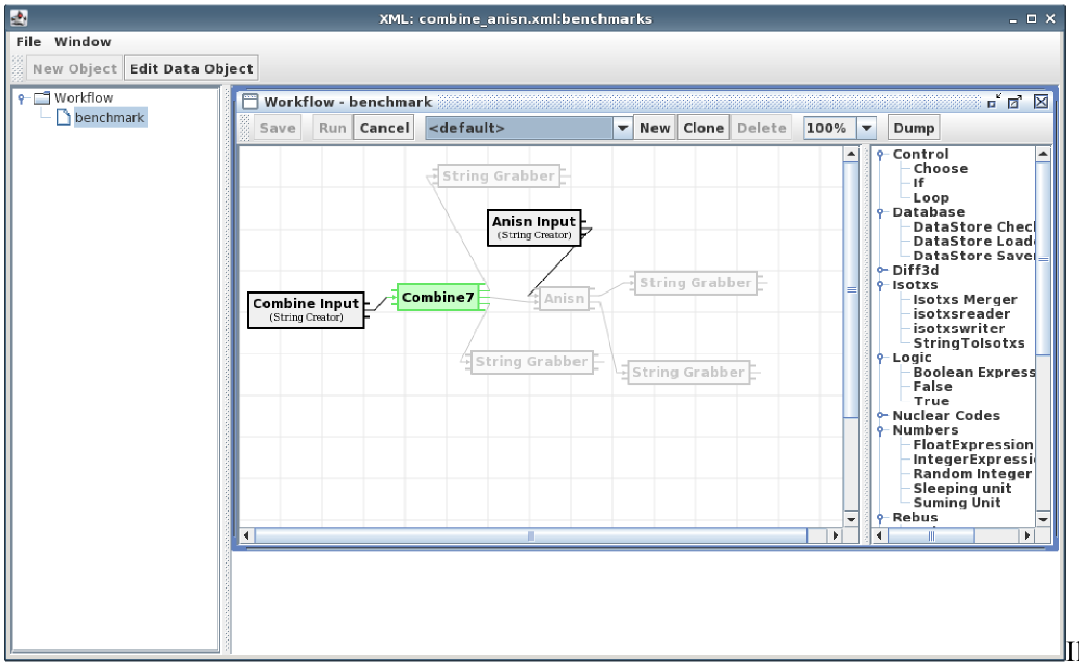

Illustration 2: Running a Workflow

12. Core plugins

13. Loop

Loops allow non-linear data transfers to occur. Basically, they map their input to some of their outputs, and then those outputs can be remapped back to the loops inputs. How this works depends on the mapping that the loop has.

Loops can be used to do the things that looping structures in programming languages are used for. A loop plugin can be used to create a Do loop, a For loop or a While loop.

Loops can use a conditional to terminate. There is an input pin (the last one) that takes a boolean, true means to continue to run, false to stop.

Loops can use a max time constraint. Select the max number of seconds, minutes, or hours to run. This will not terminate any internal looping units, just that the next time around to the loop if the duration has exceeded the time constraint, the loop will end.

Loops can terminate after running a fixed number of times.

The constraints are optional, so you have to enable the which constraints you want to use. You can use both at the same time if you want.

There can be a terminating reason output. It comes after the external outputs, and is an enumeration with CONDITIONAL, MAX_ITERATIONS, MAX_TIME. The inputs and outputs for the conditional, counter, and terminating reason can be hidden or shown. It is not possible to hide the optional inputs or outputs if there are existing connections to them. In order to hide them the links in the workflow must must be manually disconnected first.

The GUI for the data mapping has the following abilities:

• It can create mappings by dragging from the inputs to the outputs.

• It can create new inputs and new outputs.

• An output can have a connection from one external input, and one internal input, but not two external, or two internal.

• On an output the external input can be overridden with an internal input (drag lines from the external and internal), the first time through (or more exactly, anytime the internal input is null) the output will contain the external input, all the other times it will have the internal input.

• An internal output can be made that has no external input, the first time through its value will be null, each time after it will have the internal input value.

• The inputs and outputs can be renamed by right clicking on the input/output name and selecting "Rename..."

• An input or an output can be removed by right clicking and selecting remove.

• The mapping can be removed by clicking on the output pin and dragging off the mapping.

14. Checkpoint

Takes in one input and returns it. This also caches the input, so if it is needed again, and nothing changes upstream, it will return the input again. Double clicking on it will allow the stored data to be viewed.

This can be used to allow a lengthy calculation to be saved in one workflow, so that the next time the workflow is run it will not redo the calculation, unless one of the inputs changes for the data that is passed into the checkpoint.

Checkpoint *cannot* be used inside of a loop.

15. Database load and Save

16. DataStore Loader

Loads in data from the datastore. This allows data to be retrieved from a different workflow.

17. DataStore Saver

Saves data to the datastore. This allows data to be saved so it can be used in a different workflow. If the saver is located in a loop, the editor has a checkbox “Save Copies in Loop”, which when selected will make a folder of the items instead of a single item. The folder will contain all the different values that the loop generates, instead of just saving the last one.

18. Numerical Tools

19. Float Expression

Takes in Number variables, and outputs a single double. The editor allows you to input an expression, which is parsed to determine what variables are needed. Example expressions: '1' '2*((3)+1)' '2+3*5--1' '(foo – bar)*2' '1+sin(0.5+0.25*2)' 'sqrt(value)'. Allowed operators are '/','*','-' and '+' as well as parentheses for grouping. Allowed functions are 'abs', 'acos', 'asin', 'atan', 'ceil', 'cos', 'exp', 'floor', 'log', 'sin', 'sqrt', and 'tan'. Log is base e, and the trigonometry functions take or return values in radians.

20. Integer Summing Unit

Takes as inputs as many integer inputs as there are available, and sums them together and outputs the result.

21. Integer Experssion

Takes in Number variables, and outputs a single integer. The editor allows you to input an expression, which is parsed to determine what variables are needed. Example expressions: '1' '2*((3)+1)' '2+3*5--1' '(foo - bar)*2'. Allowed operators are '/','*','-' and '+' as well as parentheses for grouping.

22. String Tools

23. String Creator

String creator allows a string to be generated for use with other units. The output is the string that is edited by double clicking on the unit. The string can be viewed with a mono space or a proportional font.

24. String Concatenator

String concatenator takes as many inputs as are connected, and combines them and outputs them as a single string. This can be used to generate a combined string such as a new input for another code where most of the data is constant, but some needs to be the result of another computation.

25. String Grabber

Grabs a string and allows it to be viewed when the run is finished.

26. String To Double

Converts a string to a double value.

27. String To Integer

Converts a string to an integer value.

28. Object to String

Converts a object to a string. This can be used to convert many types of data to a string including numbers.

29. Logical Tools

30. Choose

Takes in a conditional and two banks of inputs. Depending on the conditional it will choose either the true bank or the false bank.

31. False

Always outputs a false boolean.

32. If

Takes a boolean input, and as many other inputs as connected. If the boolean is true, then it passes the outputs to the true bank, otherwise passes the outputs to the false bank.

33. True

Always outputs a false boolean.

34. Shell Unit Plugin

The shell unit plugin is used for running shell commands on either local or remote computers. This plugin is not directly used, but instead it can be used by other plugins. When the shell plugin is used, hosts can be defined and used with it in the preferences dialog. Then, the hosts can have plugins added to the host, and this defines where the plugin will be run. In the default case, the plugins will be run on the local host. However, the commands can also be run on a remote host by sshing to the remote computer and running them remotely. This is done either by running ssh command, or by using the Ganymed SSH library (distributed separately at http://www.cleondra.ch/ssh2/). The SSH method to use can be selected in the preferences.

35. FortranRunner

The FortranRunner unit allows a single fortran file to be run. When edited, a fortran program can be inputed. When it runs, it compiles the program, and then runs it. If there are comments of the form '!MRDAPInputs' followed by a list of files, it will take that list and read it in as inputs. If there are comments of the form '!MRDAPOutputs' followed by a list of files, it will pass those as outputs. For example:

```
!MRDAPInputs file1 file2

!MRDAPOutputs file_out

!MRDAPInputs file3
```

would have inputs file1, file2 and file3 and outputs file_out. There are also inputs for standard in, and outputs for standard out and standard error.

36. Mini Tutorial

This tutorial will show some workflow and database capabilities. This includes creating a workflow, running a workflow and saving and loading from the datastore. It also includes a use of a loop.

Start up MRDAP. A new XML file to store the data needs to be created. In the “select Data Source” dialog, choose the XML tab. Click the Create button to create a new XML file. Then, choose a filename and location and click on New. Back in the XML tab, choose the newly created file and then on Open. This will open the “Open/Create Project” dialog. Here, use the New button to create a new project, type in the ID as `tutorial`, then click on the open button to open it.

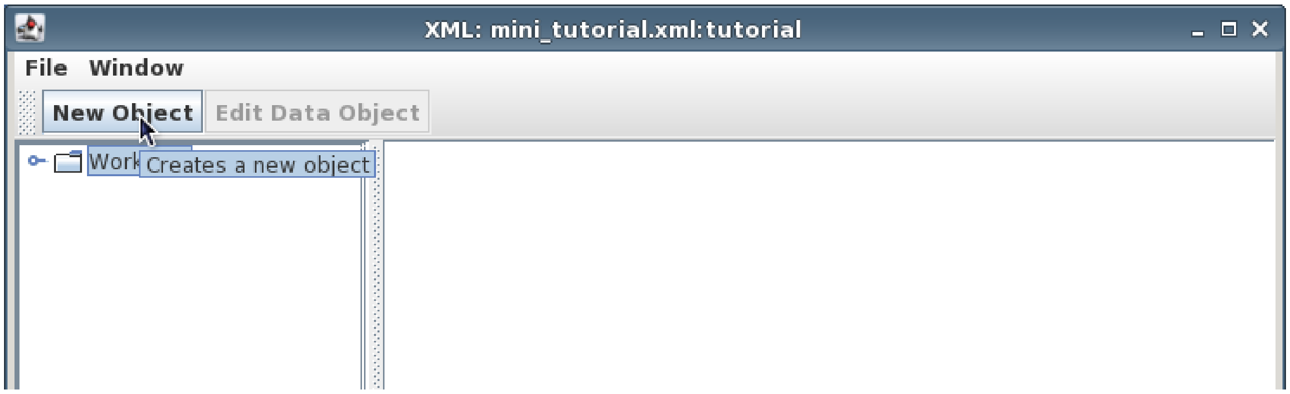

Illustration 3: New Object Button

Now that a project has been created and is open, a workflow needs to be created. Click on the workflow, and then on the “New Object” button to create a new workflow. Type in the name `saver` and then click on Okay to create it. The new workflow should now be listed in the Workflow folder, if it is not, double click on the Workflow folder to open it the folder.

Now, the workflow can be edited. Open the workflow, either by clicking on it and then on “Edit Data Object” or by double clicking on it. When the workflow opens there is a list of units on the right. Drag a “String Creator” from the Strings group to the layout panel. Drag a “DataStore Saver” from the Database group to the layout panel. Connect the string creator to the datastore saver by dragging a transfer line from the string creator's output (the plain line) to the datastore's input (the line with the arrow). Double click on the datastore saver and input the data id `data_to_save` into the dialog. Double click on the string creator and add some text (Such as “Hello”) to the dialog. Exit out of the string editor by clicking Okay. Next Run the workflow by clicking on the Run button (This will prompt you to save the workflow). Running the workflow will turn the workflow's icon into a folder, and under Runs/<default>/Results the saved text can be found. The next task to be done is to create a second workflow that uses this saved data.

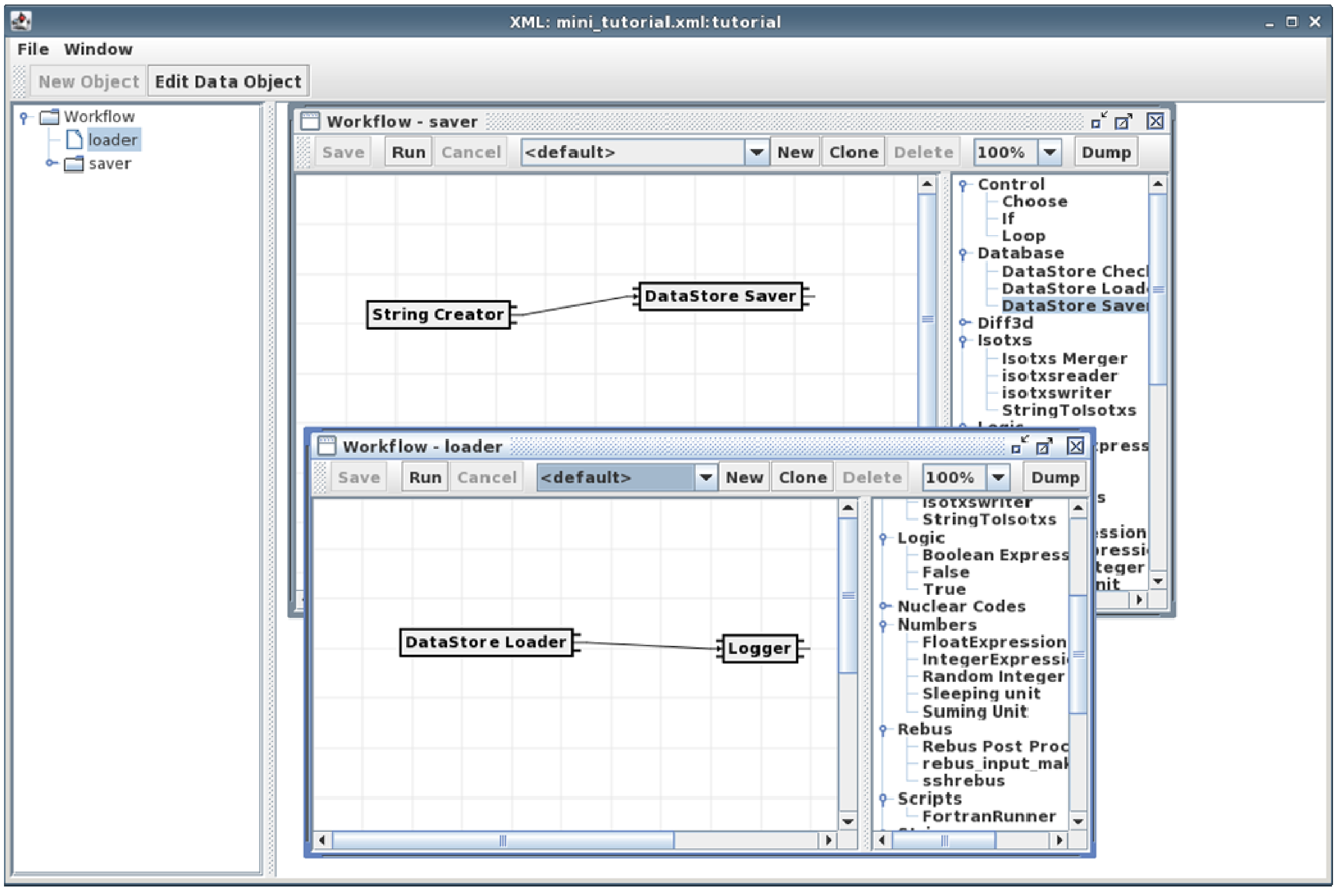

Illustration 4: MRDAP with loader and saver workflows.

Now a second workflow will be created. Create a new workflow with the “New Object” button and give it the id `loader`. Edit it, and inside it drag out a “Datastore Loader” from the Database group. Double click to edit the datastore loader, and inside choose source Project, type String, and then choose the results string `saver/Runs/<default>/Results/data_to_save` that the first workflow run created. Next drag out a Logger from the Utitilites group, and connect the datastore's output to the logger's input. Then run the workflow. Right click on the logger, and click on “Show Run Log”. If everything is hooked up correctly, then the execution log will show the original string typed into the String creator in the workflow `saver`.

The next workflow is to calculate a square root. Note that this is not the best way to calculate a square root in MRDAP, but it demonstrates the loop control. (A better way would be to just use the sqrt function inside of a Float Expression.) This will use the formula to iterate to determine the square root of value. To begin this create a new workflow `sqrt`. Edit the workflow. Drag out a Loop from the Control group. Edit the loop. First, in the Limits tab, enable maximum iterations and set it to 10 maximum iterations. Next go to the I/O Mapping tab in the Loop editor. Make a New Input “`value`” and make it external. Make a new internal input “`x_n`”. Now make a new external output “`sqrt`”. Make two new internal outputs “`x_n`” and “`value`”. Now drag a link from the external input “`value`” to the internal output “`value`”. Drag a second link from the external input “`value`” to the internal output “`x_n`”. Basically, this uses the value input as both an initial condition for the iteration, and as the value to be used for calculating the square root from. Next, drag a link from the internal input “`x_n`” to the internal output “`x_n`”. This will ensure that each iteration, the new “`x_n`” number is used. Lastly, drag the internal input “x_n” to the external output “`sqrt`”. This causes the last `x_n` number to be used for the final output of the loop. The mapping should look like Illustration 5. With this done, the loop editor can be closed.

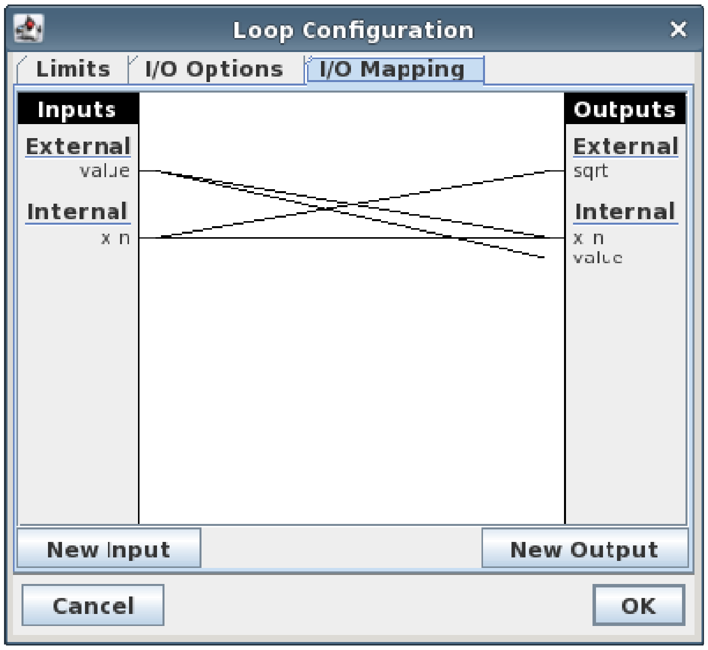

Illustration 5: Square Root Loop Connections

Now, drag a FloatExpression unit from the Numbers group to the workflow to use as the number that will have its square root calculated. Edit the float expression and put in a number such as 2.0. Then create a link from this float expression to the loop's `value` input (When the mouse is over the input, a tooltip will be displayed that gives the name of the input). Now create a second FloatExpression unit, and put the expression `0.5*(x_n+value/x_n)` inside it. Drag the `x_n` output from the loop to the new FloatExpression's `x_n` input. Repeat with the `value` input. Next drag the new FloatExpression's output to the loop's `x_n` input. Now, connect a logger to the new Float Expression's output. Run the loop. When the loop is done running, right click on the logger and go to the Run Log. With 2.0 this will have the numbers 1.5, 1.4166666666666665, 1.4142156862745097 and so on as the numbers converge to the square root.

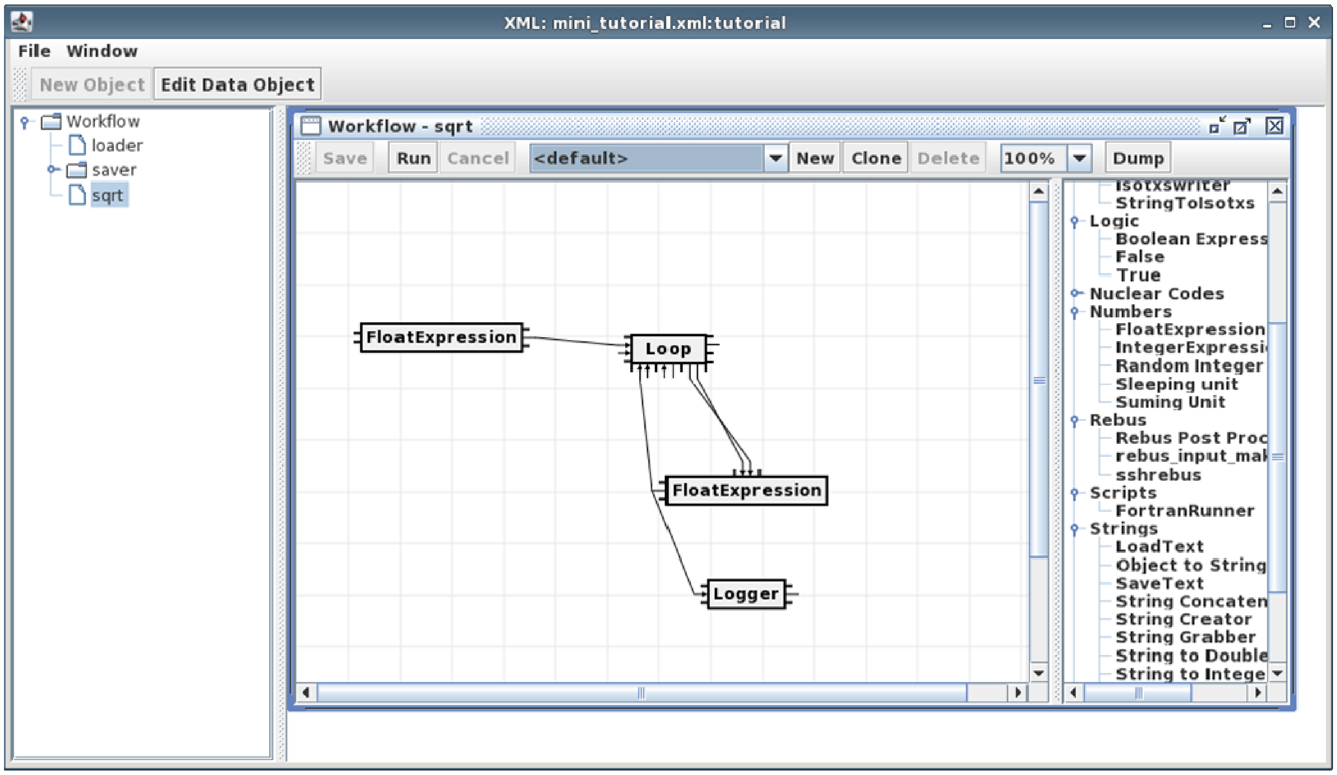

Illustration 6: First version of square root workflow

Next, convergence criteria can be added. First create a FloatExpression to calculate the difference from the final answer and the current answer with the expression `x_n*x_n - value`. Connect the inputs for this float expression to the x_n output and value output of the loop. Next create a Boolean Expression unit (found in the logic group) and put the expression `diff > 1.0e-9 or diff < -1.0e-9` in it. Connect the new FloatExpression's output to the new Boolean Expression's input and connect the Boolean expression's output to the conditional input of the loop. Now, rerun the workflow and notice how the loop terminates when the Boolean expression becomes false.

Lastly, a new configuration will be added. Click on the new button in the workflow next to the drop down configuration menu (that says <default>). Type in a name in the New Configuration dialog box. Now the configuration drop down menu will show the new name instead of <default>. Then edit the initial FloatExpression and change the number. Now run the configuration and notice that the new value is used. Then switch back to the <default> configuration and the old values are there. The configuration feature allows multiple parameter values to be used with the same workflow.

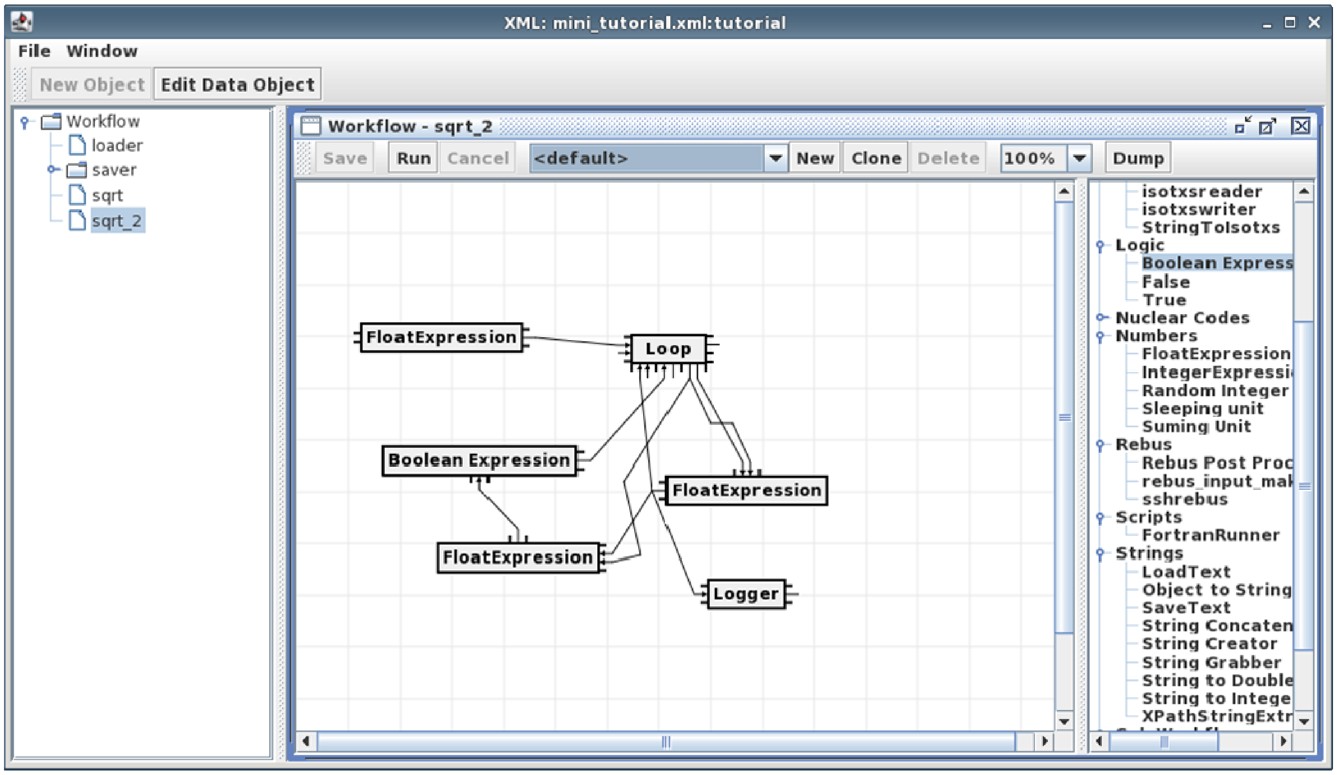

Illustration 7: Second version of square root workflow

37. Developer Manual

38. Introduction

Development for MRDAP typically consists of creation of plugins that add functionality to the system. This document describes the features of MRDAP that are relevant for further development of MRDAP and for development of plugins. Note that this document generally uses the older name GRAMPS as opposed to MRDAP. The classes in MRDAP generally are in the gramps namespace, since that was the name used at the time they were created.

39. Project Layout

The GRAMPS project is broken up into several sub projects. The main sub project gramps is located in its own folder. The gramps project contains all of the base code and the frameworks that make up GRAMPS. Plugins into the GRAMPS project are stored within the plugins folder. The layout of the top level folder is shown in Table 1 and the layout of both the gramps sub project and of the plugins is shown in Table 2.

Directory/File	Description
/	This is the project folder.
/build.xml	The ant build script for the entire project.
/build-common.xml	Contains common build targets for the plugins and gramps project.
/logging.conf	A configuration file to control the amount of logging GRAMPS performs.
/buildlibs	Contains libraries needed by ant to run the build script.

/dist	A self contained GRAMPS package that can be distributed.
/gramps	This contains the main gramps project.
/plugins	Contains the plugins for the gramps project.

Table 1: Main Project File Layout

Directory/File	Description
/	This is the sub project / plugin folder.
/build.properties	A properties file that can override some of the behaviors of the build script.
/manifest.mf	An optional manifest for the plugin / sub project resulting JAR file.
/build.xml	The build file for the sub project / plugin.
/bin	Contains the build files (*.class) files.
/lib	Contains libraries needed by the sub project / plugin.
/src	A src folder that contains the source for the sub project / plugin.
/src/info.xml	An XML file that describes the plugin.
/tests	A folder to contain any test cases needed to test the sub project / plugin.

Table 2: Sub Project (Plugin) File Layout

40. Build System

GRAMP uses ant to build and has the build scripts split into two parts. The main build script located in the root of the project doesn't actually do any of the building. Instead it delegates the building actions to the build scripts located within the subprojects and the plugins. Currently there is only one subproject the gramps project that forms the core of GRAMPS. The plugins are located in the plugins folder and are auto-discovered. Simply placing a plugin in the plugins folder will cause it to be included in any of the build actions.

41. Targets

Table 3 lists the targets and the actions that will be taken. Those actions change depending on if the main project's build script is being run, or one of the subproject or plugin build script is being run.

Target	Project	Subproject/Plugin
build	Calls build on all of the subproject / plugin's build scripts.	Builds all of the source files within the src folder and places the generated class files into the bin folder. In addition any non-java source files within the src folder is copied into the bin folder.
dist	Sets the distribution folder to /dist and calls dist on all of the subproject / plugin's build scripts.	Jars the contents of the bin folder to the distribution folder. If this is a plugin the jar is placed into the distribution's plugin folder. In the case of a subproject it is placed in the root of the distribution folder. In addition, if the subproject or plugin contains any libraries within the lib folder, they are copied to the distribution's lib folder.
dist-info	Same as dist, but calls dist-info instead.	Only creates a jar file with the plugin's

Target	Project	Subproject/Plugin
		info.xml file. This is only useful for debugging where the bin folders of each plugin has been manually set in the CLASSPATH. The jar file with the info.xml is enough to get GRAMPS to see the plugin, but the actual code will come from the CLASSPATH rather then from the jar file. This eliminates the need to create a new jar for every change made during development and debugging.
run	Sets up all the environment variables needed to run GRAMPS and then launches the GRAMPS application. It can take two defines, cmd_arguments and jvm_arguments . So to set the maximum memory the following can be used: 'ant run -Djvm_arguments=-Xmx1G ' Another example: ant run -Djvm_arguments='-Xmx1G -Dxml_data_source=/home/jjc/mrdap/misc_test.xml	N/A
test	Calls test on all of the subproject / plugin's build scripts.	By default, nothing. If a subproject / plugin wants to add something to the testing target, the test target can be overridden to perform the tests.
buildTestReport	Collects the results of the unit tests generated during the test target and builds an html page that summarizes the results.	Collects the results of the unit tests generated during the test target and builds an html page that summarizes the results.
nodeprun	Runs without checking dependencies. This should be run only after a ant dist has been done, since it will just use the files in the dist directory. Because of this, it will take less time to start the program than using run. This takes the same arguments as run does.	N/A
special-dist	Builds a special jar that contains the run time dependencies. This jar can be run with `java -jar MRDAP_full.jar`	N/A
create-zips	Creates zip files that can be distributed. The zips/MRDAP_core.zip contains the core MRDAP software and core plugins. The zips/MRDAP_full.zip contains what MRDAP_core does, as well all the nuclear plugins.	N/A

Table 3: Build targets

42. Plugin Build Scripts

Each core plugin has a build.xml build script located in the root of their directory structure. In addition the plugin's folder is placed into the main GRAMP's plugin folder. This will enable the auto-detection of the main build script to locate the plugin and forward actions to it. Out of tree plugins can be complied by other methods. The example unit in Chapter 2.6 is compiled manually.

Usually the plugin's build script only needs to import the build-common.xml script and specify a project name. The build-common.xml script contains all of the required targets. If the defaults are not suitable the plugin's developer does have the option of creating their own build script, but they must implement all of the targets described in Table 3. By default the project name is used to create the Jar file. A sample build.xml for a plugin is shown in Listing 1. For the common case the only change a plugin developer needs to make is to change the project name.

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="build" name="PluginNameGoesHere">
 <import file="../../build-common.xml"/>
</project>
```

Listing 1: Sample Plugin build.xml file.

43. Build Customizations

Many of the defaults used by the build system can be customized and are listed in Table 17. These can be overridden by providing a build.properties file, or by overriding them within the actual build.xml file.

Option	Main/Plugin	Description	Default
failOnError	m	If true, then the build stops on the first error.	false
jvm_arguments	m	Arguments to pass to the JVM when using the run target.	<empty>
cmd_arguments	m	Arguments to pass to GRAMPS when using the run target.	<empty>
build.dir	p	The directory to place the built class files in.	bin
src.dir	p	The directory where the source files are.	src
dist.dir	p	Where the distribution folder is.	dist
lib.dir	p	Where the lib folder is.	lib
info.path	p	Where the info.xml file is located	${src.dir}/info.xml
gramps.dir	p	Where the top level gramps directory is.	../../
test.dir	p	Where the test folder is.	tests
test.src.dir	p	Where the test source folder is.	${test.dir}/${src.dir}
test.lib.dir	p	Where the test lib folder is.	${test.dir}/${lib.dir}
test.build.dir	p	Where to store the class files for the test cases.	${test.dir}/${bin.dir}
test.report.dir	p	Where to store the test reports.	${test.dir}/report
test.plugins.dir	p	The plugin folder for the test runs.	${test.dir}/plugins
javac.debug	p	Controls the debug flag to javac.	true
jar.path	p	Sets where the plugin should be saved relative to the distribution folder.	${plugins.dir}/${ant.project.name}.jar

manifest.path	p	Where the plugin's manifest file is. If needed.	manifest.mf

Table 4: Customizable build properties.

44. Caveats

The ant optional packages need to be installed. On fedora this is called ant-nodeps. Sometimes the CLASSPATH environmental variable needs to be changed to include the files in buildlibs.

45. Data System

GRAMPS has two native file formats, an XML based file format and an SQL database file format. XML is used during development and for transporting data between two or more GRAMP installations. It will also form the “public” file format that can be used transfer data between GRAMPS and other external tools. Currently only XML and postgresql are supported. More general SQL support could be added. SQL should provide faster access to the data and is less memory intensive, especially with larger data sets.

The data system is an semi-automatic Java object to XML/SQL converter. The Java objects that are to be stored in the data system need to be annotated in order to control how the objects get saved into the data system.

Each data object is identified by the class of the object and the ID of the object. So it is possible to have data objects of different classes and have the same ID.

The GRAMPS data system is also hierarchical, as data objects can contain child objects. For example the main top level object is the Project object, and it can contain multiple Workflow objects, which can contain additional objects.

46. Annotations

The annotations are located in the gramps.io.annotations package. See prototype/plugins/IsotxsData for an example of a data plugin.

47. @ID

This annotation is used to identify a field or a method as being the ID field. Every top level object is required to have only one field or method flagged as being the ID. Sub objects may have ID fields, but it is not required.

48. @FieldAttributes

The @FieldAttributes annotation is used to override the defaults. GRAMPS will use the reflection APIs to extract out the name and types of the fields. If for some reason isn't satisfactory the @FieldAttributes allows the developer to customize them.

Parameter	Description	Default
name	The name of the field. Used for the database column and xml tag.	Dynamically lookup the name of the field.
readOnly	The field will only be read when saving the file. The stored value is ignored when reading back from the data store.	false
optional	The field is optional and when writing will be ignored if the value of the field is null. If not found while reading this field value will not be changed and will be whatever the class default specifies	false

customClass	Changes the class of the field. This is useful if the field is declared with a super class, but a specific subclass is used. This really only matters when loading objects where an instance of the class must be made.	Dynamically lookup the type of the field.

49. @Transient

Flags a field as being transient. This causes the data system to ignore this field. Fields marked with the Java transient keyword are also ignored. This should only be used if the Java transient keyword isn't appropriate. For example, if the field should be serialized, but not saved into the data system.

50. @Dynamic

Originally was used as a hint to the data system that the field could contain objects that were subclasses of the field's declared type. This is no longer needed as the data system now always stores the class name of an object if it differs from the declared type.

51. @Hidden

Used as a hint to the data browser that it should not show objects of this type. Classes that are only used for internal support and should not be shown to the user should be flagged with this annotation. For example, the metadata stored with each object is flagged as being hidden so that it does not clutter up the user interface.

52. @Mapping

This annotation is used to construct circler references so that a child object can have references back to the parent object. For an example of its use, look at the gamps.data.workflow.Workflow.

Parameter	Description	Default
mappedBy	The parent object must set this parameter to a unique identifier.	“”
name	The child object must set this to the name of a parent object that was set using the mappedBy parameter. The value of the field tagged with this annotation will be set to the instance of the parent object identified by the mappedBy parameter.	“”
type	Used to identify the type of mapping. Not currently used.	MappingTypes.NONE

53. @Root

Used to control how data objects integrate into the data browser. This annotation is used to specify what data objects can be created under it. For example, the Project class allows for sub workflows. So the Project class declares itself a Root and uses the Root annotation to specify that a workflow can be created under it.

54. @IsSubObjectOf

This is the inverse of the Root annotation and allows classes to specify that they are under another class. If I class wishes to be created under another classes they need to use this annotation and specify which classes they want to be created under.

55. DataSource Details

The DataSource is the public interface of the data system. All actions relating to data (loading, saving, listing, etc...) are funneled through the DataSource. The DataSourceManager is used to create a data source. Data sources are created using a URI. This URI can point to a file in the case of the XML format, or can be a jdbc connect string, in the case of a database.

56. Hierarchical Data Access

The GRAMPS data system is a hierarchical data system and objects can be stored under any other object. To access the sub objects (or to create sub objects) a sub data source must be created. This is performed by using an existing data source and opening an existing object as a new root object. This will return a new data source object that works like the original one, only its data operations will only work on sub objects of the root object.

57. DataBackingStore

The data backing store handles the details of saving and loading the data into and out of where ever it is put. Their are two current backing stores, one which saves into an XML file, and another which saves into PostgreSQL database. The backing store is an implementation of gramps.io.DataBackingStore and is responsible for fixing up the links between different classes (see gramps.io.DataContext and the isMappingOwner function in gramps.io.dataDescription.FieldDesc).

There is only one DataBackingStore per data resource, but there can be multipule DataSources using one DataBackingStore. As such the DataBackingStore must be created in a way that it does not rely on storing state between calls and must be thread safe, as any of its methods can be called from any thread.

The backing store saves Java instances. These need to be converted to some representation that can be loaded again. Each instance is identified by a String id. Classes can be nested when storing, as in if you have a String with and id Fred, under String:Fred you can have a Integer and another String.

58. Asynchronous Data Access

In order to facilitate asynchronous programming there is an AsyncDataStore that will perform the data operations on a background thread. The AsyncDataStore implements the same methods as the DataStore except it always returns an IOFuture. The IOFuture will eventually contain the result of the data operation. Clients can be notified when the data operation finishes by registering a listener with the IOFuture.

59. DataPaths

GRAMPS is a hierarchical data system so each data object is referenced by a data path. The data path is a simple list of class and ID pairs that uniquely identify an object. In general the data paths are not exposed to the user and the DataStore takes care of constructing the data paths before calling on the DataBackingStore.

60. Plugins

Plugins form the majority of real functionality of the GRAMPS system. Plugins are stored as jar files with all the java classes and resources need by that plugin. At launch the plugin system will load all the plugins jars that are found in dist/plugins directory, or in the plugins directory that is below the MRDAP_full.jar file. Each jar file can contain multiple plugins. In order to prevent collisions each plugin is loaded into its own namespace.

61. Creating a plugin

Each jar file is required to have a info.xml file that describes the plugins contained within the jar file. The info.xml file contains a list of the plugin defines. The plugin element has a mandatory `id` attribute and `type` attribute. The type attribute is used to identify what type the plugin is and is used by the system to find certain type of plugins. For example, the workflow editor will query the plugin system for all plugins with a type of "gramps.unit" in order to figure out what units are available. Different types of plugins have different requirements for what is in a plugin is and some plugin types must implement a certain interface.

Plugins can have dependencies and are checked at runtime. If the dependencies cannot be met the plugin will not be loaded. Each plugin is loaded in its own namespace, any listed dependencies are also loaded into the namespace with the plugin.If the plugin's type is gramps.pref it can have a preference editor associated with it.

Places to look for further information:
prototype/gramps/src/gramps/plugins/ contains the source for the plugin manager among other things
prototype/plugins/TestUnits/src/info.xml is an example info.xml file
prototype/plugins/Anisn for a plugin that has a dependency.

62. Extending the Data System

For the data system to work it needs to know what data classes are available. In order to allow plugins to add additional data types the plugins are scanned as they are loaded and any data classes defined within the Info.xml's <dataClasses> tag are automatically added to the data system.

63. Creating a Unit plugin

Unit plugins have type gramps.unit They need a text description of the plugin, an optional editor, an optional controller, and a runner. The editor allows the unit to provide a GUI that can be used to customize the unit's behaviour . They need inputs and outputs, which specify the class of the data that the plugin takes in and the output classes that are created by the plugin. The runner class takes the input, does its computations, and outputs the output. The preference editor class edits preferences that are common to all the units of a give type.

See Chapter 2.6 for two examples of unit plugins.

The runner usually extends gramps.units.AbstractUnitRunner. It needs to override runImpl to run its task. It gets input from the getInput(String inputName) where the name is what is declared in the XML description. It then returns its outputs using setOutput(String OutputName, Object output) using the name declared in the XML description. Optional inputs should be marked optional="true" in the input description in the XML file or with the InputDescription class. The unit plugin should check optional inputs to see if they are null.

If a unit needs more complicated input and outputs than a fixed number of inputs and a fixed number of outputs, it needs to provide a custom controller. It should extend gramps.units.DefaultUnitController. It then needs to provide the inputs and outputs that it needs, and can update them as well.

If the unit should be able to run even when some of it's non-optional inputs are not satisfied, then it can override isReadyToRun. This is used for control type units.

64. Important Classes

interface gramps.io.DataSource – Allows access to load and store data into the system. Currently, there is an gramps.io.xml.XMLDataSource that implements this using a XML file.

class gramps.ui.document.Document -- The Document class is intended to act as a controller between a data class and the GUI. In normal usage this should be subclassed by an editor of some type of data. Any methods that act on the data should be implemented within the Document subclass. Part of the responsibilty of those methods is to send out notifications when the data changes. The GUI classes will rely on the notifications to update the GUI.

65. Logging From Unit Runners

class gramps.units.AbstractUnitLoggerInit contains methods for logging. When these methods are used the data will be available to the system and can be displayed to the user besides just on the terminal output.

OutputStream getLoggerOutputStream()

PrintStream getLoggerPrintStream()

Logger getLogger()

66. Context

Each unit used in a workflow has a context, typically of class gramps.data.workflow.DefaultUnitContext. This allows the unit to store data that is specific to this particular use of the plugin and is persistent as it is stored within the workflow. The DefaultUnitContext allows different types of data to be stored by string keys. String values can be set with setStrValue(String key, String value) and retrieved with getStrValue(String key, String default). Context's can be either writable or read only. Typically, read only contexts are used in runners and writable are used in editors. When asking for a writable context the context is cloned to ensure that the original is not modified. This is done to prevent the context from changing under the feet from any other users. The getContext(boolean writable) can be used in both runners and writers to access the current context.

67. ShellUnit

The ShellUnit plugin provides the ability to run shell commands remotely and locally. It keeps track of hosts and allows plugins that use its services to run the commands on the remote hosts. To use it, the runner should extend gramps.units.shell.AbstractUnitShellRunner and then use the provided shell to do commands instead of regular java commands to run processes.

68. Example Units

The following show the use of the features to create two units. One uses the ShellUnits to create a uppercase converter unit, and the other uses pure java to create a line numbering unit.

To run, they need to be put in a jar file with the following layout:

```
     info.xml
     linenumber/Editor.class
     linenumber/JavaRunner.class
     linenumber/ShellRunner.class
```

The info.xml file specifies the input and output of the runner classes, and which editors to use, and that the ShellUnit is a dependency.

```
     <?xml version="1.0" encoding="UTF-8"?>
     <plugins>
       <plugin id="example.LineNumberExample" name="Line Number Example" 
type="gramps.unit" version="1.0">
         <description>Two line numbering examples.</description>
         <dependencies>
           <dependsOn id="org.inl.gramps.shellunits"/>
         </dependencies>
         <units>
           <unit id="example.ShellUppercase" name="ShellUpperCase"
          category="Utilities"
                 runner="linenumber.ShellRunner"
```

```
                isShellUnit="yes">
            <description>Uses the Shell to convert a String to
Uppercase.</description>
      <inputs>
        <input name="Input" id="input" class="java.lang.String"/>
      </inputs>
      <outputs>
               <output name="Uppercase" id="output" class="java.lang.String"/>
      </outputs>
           </unit>
           <unit id="example.JavaLineNumber" name="JavaLineNumber"
          category="Utilities"
                runner="linenumber.JavaRunner"
          editor="linenumber.Editor">
            <description>Uses the Shell to Number Lines.</description>
      <inputs>
        <input name="Input" id="input" class="java.lang.String"/>
      </inputs>
      <outputs>
               <output name="Numbered Lines" id="output"
class="java.lang.String"/>
      </outputs>
           </unit>
         </units>
       </plugin>
     </plugins>
```

The linenumber/ShellRunner.java file runs the shell command dd to convert a string to uppercase.

```
     package linenumber;

     import java.util.logging.Logger;
     import java.io.File;
     import java.io.IOException;

     import gramps.units.exceptions.UnitInvalidInputException;
     import gramps.units.exceptions.UnitRuntimeException;
     import gramps.units.shell.AbstractUnitShellRunner;
     import gramps.units.shell.ExecCommand;
     import gramps.io.IOUtilities;
     import gramps.units.shell.Shell;

     /**
      * Example shell runner using class.
      */
     public class ShellRunner extends AbstractUnitShellRunner {

         @Override
         protected void runImpl(Shell shell) throws UnitInvalidInputException,
UnitRuntimeException {
             Logger logger = getLogger();

             //Get the input to the unit
             String input = (String) getInput("input");
             try {
                 //Get temporary directory created by AbstractUnitShellRunner
                 File workingDirectory = shell.getWorkingDir();
```

```
                //Write out input file.
                logger.info("Input:" + input);
                File inputFile = new File(workingDirectory, "input");
                IOUtilities.writeStringAndClose(shell.putFile(inputFile),
input, "ISO-8859-1");
                File outputFile = new File(workingDirectory, "output");

                //Execute the dd command. Note that you can also just pass
files.
                ExecCommand cmd = shell.exec("dd", "if=input", "of=output",
"conv=ucase");
                cmd.waitForExit();

                String errorOutput =
IOUtilities.readAllAsString(cmd.getStderr());
                if (errorOutput.length() > 0) {
                    logger.warning("Program Error Output: " + errorOutput);
                }

                String output;
                if (shell.doesFileExist(outputFile)) {
                    output =
IOUtilities.readAllAsString(shell.getFile(outputFile), "ISO-8859-1");
                } else {
                    output = "";
                    logger.severe("No Output found from running program");
                }

                //Output the output from the unit
                setOutput("output", output);
            } catch (IOException io) {
                throw new UnitRuntimeException(io);
            } catch (InterruptedException ie) {
                throw new UnitRuntimeException(ie);
            }
        }
    }
```

The linenumber/JavaRunner.java class puts a prefix and a line number in front of each line on the input.

```
    package linenumber;

    import gramps.data.workflow.DefaultUnitContext;
    import gramps.units.AbstractUnitRunner;
    import gramps.units.exceptions.UnitInvalidInputException;
    import gramps.units.exceptions.UnitRuntimeException;
    import java.util.logging.Logger;

    /**
     * Example runner using pure java
     */
    public class JavaRunner extends AbstractUnitRunner {

        @Override
        protected void runImpl() throws UnitInvalidInputException,
UnitRuntimeException {
```

```
            Logger logger = getLogger();

            //Get the input to the unit
            String input = (String) getInput("input");
            //Get preference
            String prefix = getContext(false).getStrValue(Editor.PREFIX_KEY, 
"");

            StringBuffer output = new StringBuffer();
            int count = 1;
            for (String line : input.split("\n")) {
                output.append(prefix + Integer.toString(count) + " " + line + 
"\n");
                count++;
            }
            setOutput("output", output.toString());
        }

        @Override
        public DefaultUnitContext getContext(boolean writable) {
            return (DefaultUnitContext) super.getContext(writable);
        }
    }
```

The linenumber/Editor.java class provides an editor to change the prefix in for each line.

```
    package linenumber;

    import gramps.data.workflow.DefaultUnitContext;
    import gramps.ui.undo.UndoManager;
    import gramps.units.AbstractUnitEditor;
    import java.awt.Component;
    import javax.swing.JOptionPane;

    /**
     * Example editor class
     */
    public class Editor extends AbstractUnitEditor {

        protected static final String PREFIX_KEY = "prefix";

        public boolean showGUI(Component parent, UndoManager undoManager) {
            String origPrefix = getContext(false).getStrValue(PREFIX_KEY, "");
            Object selectedValue = JOptionPane.showInputDialog(parent,
                    "Choose line numbering Prefix", "Line Numbering Editor",
                    JOptionPane.INFORMATION_MESSAGE, null,
                    null, origPrefix);
            if (selectedValue == null) {
                return false;
            }
            String newPrefix = selectedValue.toString();
            if (!origPrefix.equals(newPrefix)) {
                DefaultUnitContext context = 
getContext(true).getUndoableContext(undoManager);
                context.setStrValue(PREFIX_KEY, newPrefix);
                return true;
            }
            return false; //No change to the context.
        }

        @Override
        public DefaultUnitContext getContext(boolean writable) {
            return (DefaultUnitContext) super.getContext(writable);
        }
    }
```

These can be converted to a plugin with the following commands (PATH needs to be adjusted to point to the correct directory where the distribution jar files are):

```
    CLASSPATH=PATH/MRDAP_full.jar:PATH/plugins/ShellUnits.jar javac
linenumber/*.java
    jar -cvf PATH/plugins/TestJar.jar info.xml linenumber/*.class
```

Alternatively, if the source is available the regular ant script can be used.

69. Acknowledgments

This work was supported as an LDRD project at the Idaho National Laboratory, under the auspices of the Department of Energy, Office of Nuclear Energy, Science, and Technology, under DOE Idaho Operations Office Contract DE-AC07-99ID13727. The following people helped with the development of MRDAP: Mike Ferrer, Charles Wemple, and Sam Bays. Daniel E Wessol came up with the original idea for MRDAP.

The following is documentation of the nuclear plugins that are in the MRDAP software package. These are less developed than the core plugins.

70. Nuclear Plugins

71. Anisn

Takes in an isotxs and an input as a string, and returns an error output and a standard output.

72. Combine7

Takes in the combine7 input, and creates an isotxs. Also outputs error output and standard output.

73. Dragon 4 Runner

Runs dragon on the input, and returns the results file. Also, the editor allows you to give a list of isotxs and other files that should be copied out and provided as extra outputs. Note that dragon runner expects a custom shell script to be in the dragon directory.

74. sshdiff3d

Ssh's to a remote computer and runs diff3d on it using the diff3d input and the isotxs file. Outputs the diff3d output and also output from running ssh.

75. Isotxs Merger

Merges together the isotopes in all the isotxs files that come in. Note that it is pretty stupid right now and does not check much. For example, if the energy groups do not match it will incorrectly output the merged isotxs file without giving any warning.

76. isotxsreader

Takes in the filename to use, and loads that filename from disk and outputs it as a isotxs type. In general it would be better to use a string creater and StringToIsotxs to create the isotxs, since then it is saved in the workflow instead of depending on it being on disk.

77. isotxswriter

Takes in the filename to use, and the isotxs data to save. Writes it to the disk as a text isotxs. This is useful for debugging.

78. MCNP5 Runner

Runs MCNP5 on the given input, and outputs the output file. The XSDIR input can be used to provide a custom xsdir file. Other files can be added with the optional input and output files in the editor. It assumes that mcnp5 is on the PATH.

79. NJOY 99 Runner

Runs NJOY on the given input. Grabs the specified tapes afterward. Input cross sections should be stored in the data directory for the plugin. Then they can be specified as “data/t404.gz as tape20,data/t511.gz as tape21” and they will be copied so that xnjoy can run them. The njoy binary xnjoy should be in the path.

80. Rebus Post Processor

Runs Ben Forget's post processor. Takes in the rebus output, the post processor input and an integer to specify the calculation type. Outputs a bunch of stuff depending on the input file. You can use the Random Integer Generator to set the calculation type (somewhat insane, but it works).

81. rebus_input_maker

Takes in a mcc template, a mcc fp template and a rebus template. Outputs a rebus input, and some pseudo mcc inputs. Uses Nick's gui program to create the data for these.

82. sshmc2

Ssh's to a remote computer and runs mc2 on it using the given inputs and an optional isotxs file. Outputs the new isotxs file (and does the copy back to the host computer) and also the standard outputs and error outputs from running mc2.

83. sshrebus

Ssh's to a remote computer and runs rebus on it using the rebus input and the isotxs file. Outputs the rebus output and also output from running ssh.

84. StringToIsotxs

Converts a string to isotxs data so it can be used by other units. Takes in a text isotxs file.