Tessella Project Number: 7596

**Tessella plc** 26 The Quadrant, Abingdon Science Park, Abingdon, Oxfordshire, OX14 3YS, UK

T: +44 (0)1235 555511 | F: +44 (0)1235 553301 | E: info@tessella.com | www.tessella.com

Client: RAL & ORNL

Mantid

Nested History Detailed Design Document



Document Control

This document is under document control. All members of the distribution list will receive updated copies whenever alterations are made. All other copies are uncontrolled.

Document Owner

|  |  |  |
| --- | --- | --- |
| Name | Role | Company |
| Nicholas Draper | Owner | Tessella |

Document Approval

The following are required to approve this document. Their approval signifies that the document is fit for purpose and approved for use. .

|  |  |  |
| --- | --- | --- |
| Name | Approval Role | Company |
| Martyn Gigg | Subject Expert | RAL |
| Michael Reuter | Subject Expert | ORNL |

Distribution

|  |  |  |
| --- | --- | --- |
| Name | Role | Company |
| Mantid GIT repository | Filing | GitHub |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Author | Issue | Date | Description |
| Nicholas Draper | V1.R1.M0 | 03-Jan--2013 | 1st version written |
| Samuel Jackson | V1.R1.M0 | 28-Mar-2014 | Updated with proposed changes. |

References

|  |  |  |  |
| --- | --- | --- | --- |
| Document | Ref | Date | Details & Issue |
| Architectural Design Document | [ADD] | 11-Nov-2009 | V1.R2.M1 |

Table of Contents

[1Introduction 3](#__RefHeading__814_1984247398)

[1.1Definition of Terms 3](#__RefHeading__816_1984247398)

[2Problem: 4](#__RefHeading__818_1984247398)

[3Proposed solution 4](#__RefHeading__820_1984247398)

[3.1Refactoring the history structure to store nested history records 4](#__RefHeading__822_1984247398)

[3.1.1In memory storage 4](#__RefHeading__824_1984247398)

[3.1.2Nexus File Storage 4](#__RefHeading__826_1984247398)

[3.2Recording the nested history records correctly 5](#__RefHeading__828_1984247398)

[3.2.1Changes to the Algorithm and DataProcessorAlgorithm class 5](#__RefHeading__830_1984247398)

[a)Python Algorithms 6](#__RefHeading__832_1984247398)

[3.2.2Workspace properties changes 6](#__RefHeading__834_1984247398)

[3.2.3Vector property changes 6](#__RefHeading__836_1984247398)

[3.2.4Removing History 6](#__RefHeading__838_1984247398)

[3.3Displaying and scripting the Nested history records 7](#__RefHeading__840_1984247398)

[3.3.1WorkspaceHistory Display 7](#__RefHeading__842_1984247398)

[3.3.2Python Scripting 7](#__RefHeading__844_1984247398)

# Introduction

Purpose of this Document:

This document describes the detailed design to implement nested algorithm history records within the Mantid program.

It is based on the design specified in the Architectural Design Document [ADD]

It will form the basis of the development of this aspect of the framework and act as a guide for maintaining the system.

Scope of this Document:

These requirements cover the development of the nested algorithm history aspect of the Mantid Framework.

Context of this Issue:

This is the first draft of the EW-DDD derived from the ADD and after internal review will be updated and used as a basis for the development of the system.

## Definition of Terms

|  |  |
| --- | --- |
| ADD | The Architectural Design Document (this document), the high level design document for the entire system. |
| URD | The User Requirements Document, records the users’ requirements for the system. |
| SRD | The Software Requirements Document, specifies the behaviour of the software system. |
| API | Application Programming Interface, defines the interface through which two programs may interact. |

# Problem:

People want to be able to see what Mantid is doing beneath the surface. Particularly as part of complex workflow algorithms. They would like to be able to script out the workspace history at different levels of depth (or see what a workflow algorithm has done in detail, and be able to reproduce it) in some way.

# Proposed solution

The three main areas that need to be resolved here are as follows:

* Refactoring the history structure to store nested history records
* Recording the nested history records correctly
* Displaying and scripting the Nested history records

These suggested changes clearly affect core functionality and therefore this will have to follow the [Core Functionality procedures](http://www.mantidproject.org/Working_on_Core_Functionality).

## Refactoring the history structure to store nested history records

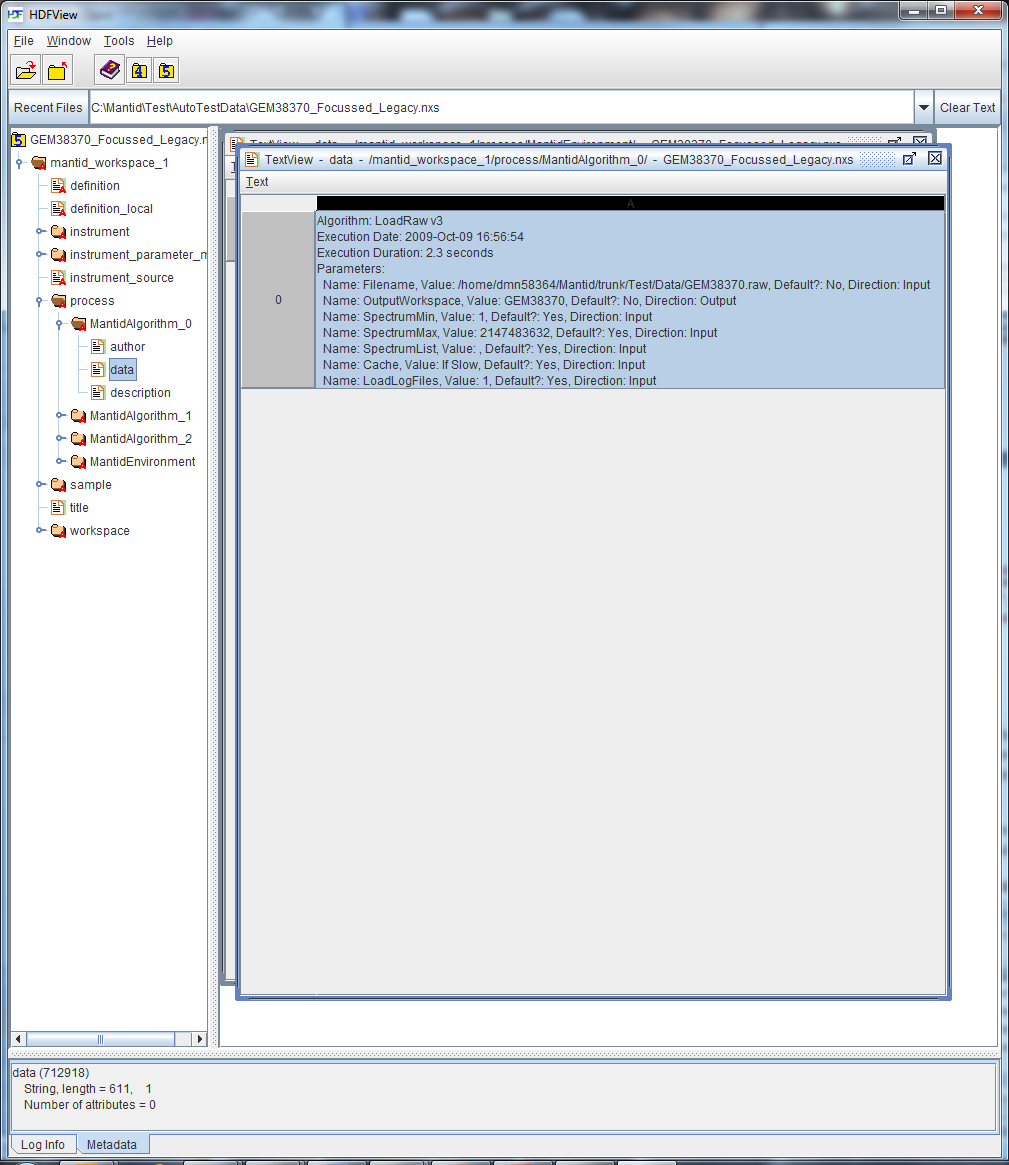
The current history record structure is a flat list of algorithm history records. This is also used when storing the data into intermediate nexus files, and anything we do will need to be compatible with previous nexus files.

### In memory storageA description...

The AlgorithmHistory class will be extended to allow nested records, that is an AlgorithmHistory record can also own within it other AlgorithmHistory records in an internal container. They should be able to be filled within the constructor, but getters and setters should also be added to allow full access.

Based on trial runs on simplified models of the algorithm history structure, the suggested STL container to use would be a set as this proved to be the quickest method (out of a set, list, and boost graph) of storing child history records in order according to their execution count. A set also allows us to keep our current strategy for merging history based on the execution count.

This should be done by adding overloads and methods to the existing class, and avoid changing existing methods wherever possible. It is suggested that at minimum the following changes be made to the AlgorithmHistory class:

* An addChildHistory() method is added that takes an AlgorithmHistory object and adds it to the internal container.
* A getChildHistories() method is added that returns a constant reference to the AlgorithmHistories object contained in the class.
* A setAlgorithmDuration() method is added so that the duration time can be set after child algorithms have finished execution. (See section 3.2.1)
* The constructor is overloaded to provide an option to not pass the duration as a parameter (for the same reason as above). But maintaining the existing constructor for loading AlgorithmHistory objects from file.

Additionally, I suggest that the PropertyHistory class is modified with another constructor which accepts a property object. This is purely to simplify construction of PropertyHistory objects.

### Nexus File Storage

The History records are stored within Nexus files, and as such we need to update the loading and saving of this data to support nested records. The overall structure of the records is shown to the right, with one NXnote for each algorithm. In order to preserve the integrity of previous files and keep things simple in the majority of simple cases we will keep a similar structure, so that there will be one NXnote entry for each top level parent algorithm in the History. Subsequent child algorithm histories will be stored in as additional NXNotes within the parent algorithms group, forming a tree structure of NXNotes within the nexus file.

Using a basic model of the WorkspaceHistory, AlgorithmHistory, and PropertyHistory structure to test loading and saving performance showed that the fastest results were achieved when using the same string representation that is currently used over saving out the proper types for each of the properties. This seemed to be down to the overhead of having to write out a new record for every property, rather than converting that data to a string and doing a single write operation. Additionally, using direct calls to the HDF5 library gave better performance over using Nexus library calls.

It is therefore suggested that we keep the current string representation of algorithm history information and potentially use direct calls to the HDF5 library instead of the Nexus library if speed becomes too much of an issue for a Nexus based implementation to handle.

An example of a previous algorithm NXNote, and a good example of an algorithm history:

Algorithm: LoadRaw v3

Execution Date: 2009-Oct-09 16:56:54

Execution Duration: 2.3 seconds

Parameters:

Name: Filename, Value: Data/GEM38370.raw, Default?: No, Direction: Input

Name: OutputWorkspace, Value: GEM38370, Default?: No, Direction: Output

Name: SpectrumMin, Value: 1, Default?: Yes, Direction: Input

Name: SpectrumMax, Value: 2147483632, Default?: Yes, Direction: Input

Name: SpectrumList, Value: , Default?: Yes, Direction: Input

Name: Cache, Value: If Slow, Default?: Yes, Direction: Input

Name: LoadLogFiles, Value: 1, Default?: Yes, Direction: Input

Presently the code for writing out workspace history is stored within WorkspaceHistory.saveNexus() and simply loops over the flat list of AlgorithmHistory entries using the AlgorithmHistory.printSelf() method to create the output for each AlgorithmHistory object, as shown above. The proposed change is that we overload this method with an extra parameter for a set of AlgorithmHistory objects which is called recursively on each set of AlgorithmHistory objects to write out their children.

The code for parsing AlgorithmHistory text is in the WorkspaceHistory.loadNexus() method. Again, this should be extended in a similar way to algorithm saving, using an overloaded method to read in an AlgorithmHistory entry then make a recursive call to the same function to read in each of the child algorithm entries.

I would also suggest that a new option be added to SaveNexusProcessed called “SaveHistory” which is set by default to be true and complements the option in LoadNexusProcessed called “LoadHistory”, so we have an option to skip writing out history to a file.

## Recording the nested history records correctly

Currently Algorithms calls a FillHistory() method at the end of successful execution, just before the workspace is stored. Prior to these changes child algorithms only store history if the recordHistoryForChild flag is set.

It is important to differentiate between workflow algorithms (which do not alter a workspace using anything other than other child algorithms) and other algorithms (that mix processing with possibly using other child algorithms). While it is desirable to include the child AlgorithmHistory records for workflow algorithms it is not advisable to do the same for all algorithms that use child algorithms.

Take the LoadRaw algorithm for example: This uses two child Algorithms, LoadLog and LoadInstrument, however it also performs a lot of other changes to the workspace itself. If the included the child algorithm calls for this algorithm uses might expect they can run the detailed child algorithms to reproduce the effect of the parent algorithm, but it would fail badly and cause confusion. Therefore we will only include Child Algorithms for workflow algorithms.

### Changes to the Algorithm and DataProcessorAlgorithm class

Workflow algorithms will be identified by inheriting from the DataProcessorAlgorithm abstract class that itself inherits from Algorithm. I propose the following changes to the Algorithm and DataProcessorAlgorithm classes in order to correctly record the algorithm histories. In the Algorithm class the following additions/modifications should be made:

* The createChildAlgorithm() function should be made virtual so that it can be overridden in DataProcessorAlgorithm
* A new function called trackAlgorithmHistory() should be added which takes a boolean setting the state of recordHistoryForChild and a reference to the parent algorithms’ AlgorithmHistory object. Instead of creating the AlgorithmHistory object at the end of the algorithm’s execution, it should create it at the start; setting all everything but the duration of the AlgorithmHistory (which has to be set at the end). This is because the child algorithm needs some reference to where to store its history object.
* Another new function called trackingHistory() should be created which returns a Boolean value indicating if history should be stored. This would encapsulate the check for if the algorithm is a child or has recordHistoryForChild set to true. This function should always return true when the algorithm is a DataProcessorAlgorithm.
* The fillHistory() method should be modified choose between two actions depending on state:
  + If the algorithm is not a child it stores its history in the WorkspaceHistory object. This is how the function currently works.
  + If recordHistoryForChild is true, append the history object for the this algorithm to the parent AlgorithmHistory object passed in by trackAlgorithmHistory().

In the DataProcessorAlgorithm class we propose the following changes:

* A new member function called createChildAlgorithm() is written that overrides the same function in the Algorithm class. This function calls the base class’s implementation of createChildAlgorithm() but also adds a call to trackAlgorithmHistory() which passes the child algorithm a reference to the parent AlgorithmHistory object.

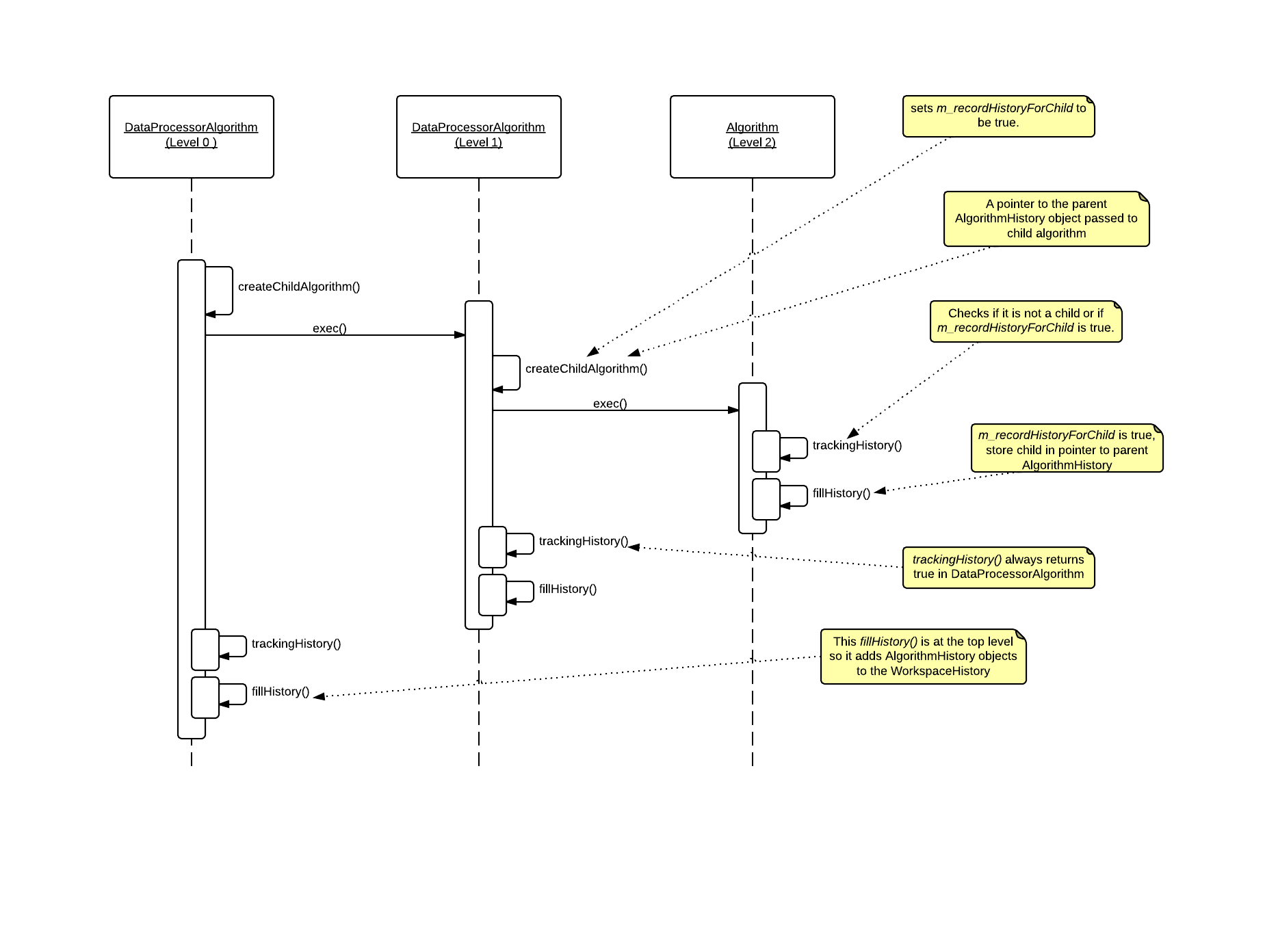
The diagram below shows a sequence diagram example of the program execution after the proposed changes have been made.

Figure Shows the proposed program flow for three nested algorithms. The level 0 algorithm’s history is added directly to the workspace's history.

Unfortunately, we cannot simply use a overridden version of fillHistory() in DataProcessorAlgorithm to collect the entries for any child algorithms because the we cannot access the child algorithms after they have been created as they’re weak pointers. We must therefore pass a pointer to the child’s fillHistory() method so the child knows whether to put the its history in the workspace or in add it to the parent algorithm after it has finished processing.

#### Python Algorithms

Currently Python Algorithms within other algorithms are not run as “proper” child algorithms, but are created as unmanaged algorithms. This will be changed to use the Algorithm.createChildAlgorithm() method. With this approach it should no longer be necessary to use the createChildAlgorithm() method directly within Python itself, instead any algorithms used within a python algorithm will automatically be created as child algorithms. However the method will be retained for historic reasons and to allow more detailed progress linkage.

### Workspace properties changes

Currently there are two ways of setting a workspace property, one uses the name of a workspace in the ADS, and dynamically finds and provides a shared pointer to the workspace. This approach is fine, works well with history records and will not be changed.

The second approach is where you set the pointer to the workspace to be used, and do not provide a name (normally because it is not stored in the ADS). This approach is frequently used in child algorithms. At present this leaves the History record without any value for this property, and scripts will not run. We should change this to provide a temporary name for the workspace. I suggest a naming strategy as follows:

“\_\_TMP” + the memory address of the workspace (sptr.get())

These names should just be used as a default name which can be used to satisfy the python validators. There should be no need to store the workspaces in the ADS which should be avoided to prevent processing overhead associated with it. This should allow the underlying scripts of workflow algorithms to be run in their stead.

### Vector property changes

Another consideration that needs to be addressed is the representation of vector properties where long sequences of regularly spaced numbers are being stored as a comma separated string. This is very impractical, especially in the case of algorithms such as CreateWorkspace where an array the size of the data is passed as a property. This increases the size of the history within the Nexus files and is awkward for users to look at in generated scripts. I propose that the we can shorten this into a more sensible representation by modifying the toValue() and fromValue() methods in the PropertyWithValue class to convert regularly spaced sequences to a more condensed format. For example, the sequence:

0,0.5,1,1.5,2,2.5,3

Could be represented as the string:

0:0.5:3

Where the first number is the first number in the sequence, the second is the step between numbers in the sequence and the third number is the final number in the sequence. This approach could also potentially be applied just to parts of properties such as converting the sequence:

0,1.3,5,1,2,3,4,5,10,13,48

To be:

0,1.3,5,1:1:5,10,13,48

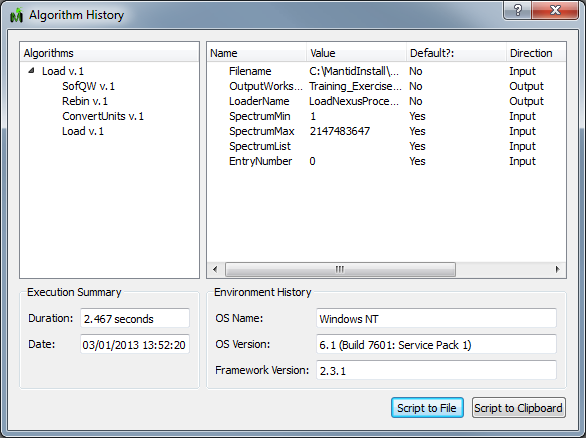
Which could help condense the size of properties which are mostly regular but that contain some data that does not follow the sequence.

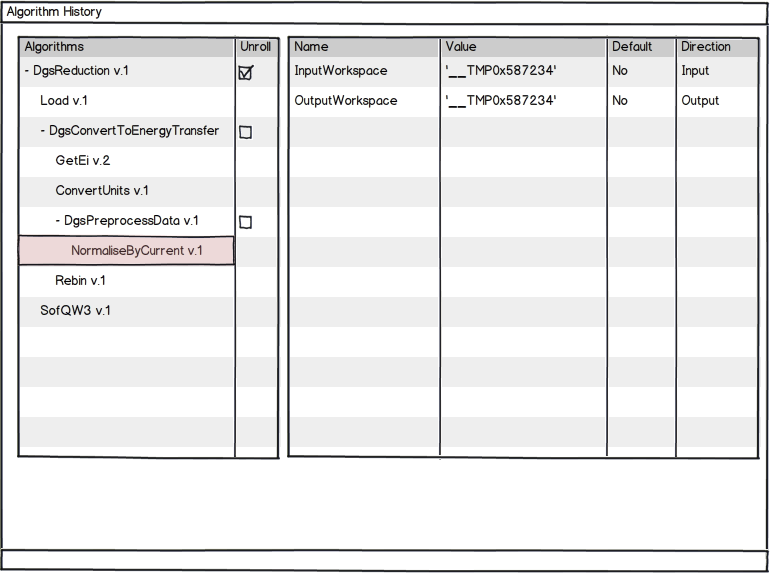
### Removing History

Finally, I suggest that we should have an additional algorithm called RemoveWorkspaceHistory that allows a workspace to be cleared of any history records it has regardless of whether that record is from a workflow algorithm or a regular algorithm or whether or not it is a child.

## Displaying and scripting the Nested history records

### WorkspaceHistory Display

This is the current Workspace History display.

The following mock-up outlines the proposed changes to the workspace history GUI to reflect the changes in the history structure.

The majority of the interface is to stay the same, with most of the changes centred on the algorithm history tree. Specifically, change the algorithm display in the tree to:

* Remove the artificial nesting under the most recent algorithm
* Allow expansion under algorithms with child records.
* Reverse the current order of entries in the list such that the most recent algorithm is at the bottom.
* Upon opening the dialog should have only show the top level algorithms.
* Provide a checkbox option to “Unroll” a workflow algorithm. This option allows the user to generate a script with all of the children of this algorithm, instead of just the algorithm itself.

### Python Scripting

Currently only parent algorithms are recorded in the Algorithm History. As such they never appear within the python scripts generated by the GeneratePythonScript algorithm. This will need to be changed so that the GeneratePythonScript algorithm can unroll algorithms to different levels in the hierarchy

This is a current generated python script:

######################################################################

#Python Script Generated by GeneratePythonScript Algorithm

######################################################################

Load(Filename='Data/CNCS\_7860\_event.nxs',OutputWorkspace='CNCS\_7860\_event')

ConvertUnits(InputWorkspace='CNCS\_7860\_event',OutputWorkspace='CNCS\_7860\_event',Target='DeltaE',EMode='Direct',EFixed='3')

Rebin(InputWorkspace='CNCS\_7860\_event',OutputWorkspace='CNCS\_7860\_event',Params='-2.9,0.03,2.9')

SofQW(InputWorkspace='CNCS\_7860\_event',OutputWorkspace='w1',QAxisBinning='0.2,0.2,2',EMode='Direct',EFixed='3')

Load(Filename='C:\MantidInstall\data\Training\_Exercise3a\_SNS.nxs',OutputWorkspace='Training\_Exercise3a\_SNS')

With the new changes we propose that when the algorithm has been unrolled to add spacing around the algorithm and a comment indicating what algorithm these child algorithms are part of. In the example below, the SofQ algorithm has been unrolled.

######################################################################

#Python Script Generated by GeneratePythonScript Algorithm

######################################################################

Load(Filename='Data/CNCS\_7860\_event.nxs',OutputWorkspace='CNCS\_7860\_event')

ConvertUnits(InputWorkspace='CNCS\_7860\_event',OutputWorkspace='CNCS\_7860\_event',Target='DeltaE',EMode='Direct',EFixed='3')

Rebin(InputWorkspace='CNCS\_7860\_event',OutputWorkspace='CNCS\_7860\_event',Params='-2.9,0.03,2.9')

# Child Algorithms of SofQW #

ChildAlg1(InputWorkspace='CNCS\_event',OutputWorkspace='CNCS\_event')

ChildAlg2(Workspace=”\_\_TMP0x12345AE3456EF”)

# End of Child Algorithms of SofQW #

Load(Filename='C:\MantidInstall\data\Training\_Exercise3a\_SNS.nxs',OutputWorkspace='Training\_Exercise3a\_SNS')

This should not however alter the output for examples without workflow algorithms containing child algorithms.