Australian Government
Department of Defence
Science and Technology

# The Janus C++ Library – An Interface Class for DAVE-ML Compliant XML-Based Flight Model Datasets

*Geoff Brian and Shane D Hill*

**Aerospace Division**
**Defence Science and Technology Group**

DST-Group–TN–1658

## ABSTRACT

The Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) is a syntactical language for exchanging flight vehicle dynamic model data. It has been developed in conjunction with the ANSI/AIAA S-119-2011 Flight Dynamics Model Exchange Standard prepared by the American Institute of Aeronautics and Astronautics (AIAA) Modeling and Simulation Technical Committee (MSTC). The purpose of DAVE-ML is to provide a framework to encode entire flight vehicle simulation data packages for exchange between simulation applications and the long-term archiving of model data. This document describes an application programming interface (API) to the DAVE-ML dataset structure that has been developed by the Defence Science and Technology (DST) Group. The API is known as 'Janus'.

*Published by*

*Aerospace Division*
*Defence Science and Technology Group*
*506 Lorimer St,*
*Fishermans Bend, Victoria 3207, Australia*

*Telephone:* *1300 333 362*
*Facsimile:* *(03) 9626 7999*


*AR-016-923*
*July, 2017*

***APPROVED FOR PUBLIC RELEASE***

# The Janus C++ Library – An Interface Class for DAVE-ML Compliant XML-Based Flight Model Datasets

## Executive Summary

The Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) is a syntactical language for exchanging flight vehicle dynamic model data. It has been developed in conjunction with the ANSI/AIAA S-119-2011 Flight Dynamics Model Exchange Standard, prepared by the American Institute of Aeronautics and Astronautics (AIAA) Modeling and Simulation Technical Committee (MSTC). The intended purpose of DAVE-ML is to provide a framework for encoding entire flight vehicle simulation data packages for exchange between simulation applications and the long-term archiving of model data. Such data packages are commonly used in research, engineering development, and flight training simulations. DAVE-ML is designed to provide a programming-language-independent representation of aerospace vehicle characteristics, such as the aerodynamics, mass, propulsion, navigation and control properties.

The Defence Science and Technology (DST) Group has developed an application programming interface (API) enabling flight modelling and simulation applications to directly interface with aerospace vehicle datasets encoded using the DAVE-ML syntax. The API is known as 'Janus' and is implemented as a C++ library. Janus enables an applications to read DAVE-ML datasets and interactively extract data. Furthermore, Janus enables DAVE-ML style datasets to be created using predefined data.

This document describes the usage and structure of the Janus API. It details the process of instantiating Janus within an application, together with public functions that enable the application to interact with information stored within DAVE-ML style datasets. The Janus API provides a capability to simplify the exchange of aerospace vehicle dynamic model data between simulation applications.

THIS PAGE IS INTENTIONALLY BLANK

# Authors


## Geoff Brian

Aerospace Division


Geoff Brian joined the Defence Science and Technology Group in 1989, after graduating from the University of New South Wales with a Bachelor in Aeronautical Engineering with first class honours. He has worked in the fields of fixed-wing aircraft flight dynamic and performance evaluation, aerodynamics, flight testing, aircraft modelling and simulation. This work has included supporting in-service military aircraft, including the F-111, F/A-18, A/P-3C, and PC-9/A; together with fighter, air-to-air refuelling, transport, and unmanned aerial vehicles acquisition projects. In 1998 Mr Brian was awarded a United Kingdom Chevening Scholarship and attended Loughborough University obtaining a MSc in Industrial Mathematical Modelling with honours. In 2010 he was awarded a Defence Science Fellowship and spent a year at the NASA Langley Research Center investigating aerodynamic parameter identification techniques using NASAs Generic Transport Model flight test aircraft, and developing modelling and simulation standards for the exchange of aerospace vehicle data

**Shane Hill**
Aerospace Division

Mr. Hill graduated in 1987 with a Bachelor of Engineering, (Aeronautical Engineering) with Honours from the University of Sydney. He commenced work at the Defence Science and Technology Group in 1988, working in the area of Aircraft Flight Dynamics. Significant achievements have included the development of a flight dynamic model for a joined wing aircraft, a six degree of freedom flight dynamic model of a spinning aircraft and contributed in the development of flight dynamic models for F-111C and F/A-18 aircraft. Many years of experience in aircraft incident investigation work lead Mr. Hill to develop the Graphical Replay System (GRS) and the Flight-Path Reconstruction (FPR), software that is widely used within Defence to reconstruct Black-Box data and visually replay the flight-paths of aircraft. Mr. Hill moved into the Aircraft Flight Loads area in 1995 where he has been involved with the calibration of F/A-18 and P3 aircraft to record in-flight loads. In 1999 Mr. Hill lead a small team of engineers at Lockheed Martin Tactical Aircraft Systems, Fort Worth, to review the F-111 structural parametric load equations, used to develop load spectrums F-111 wing fatigue tests aimed at determining wing life under Australian RAAF usage. Currently Mr Hill is the Science Team Leader (STL) for the Aircraft Modelling and Simulation group and is involved with the development of aircraft simulation and performance tools used by Australian Defence and international partners.

# Contents

# Notation

| | |
|---|---|
| **AD** | Aerospace Division |
| **ADF** | Australian Defence Force |
| **AIAA** | American Institute of Aeronautics and Astronautics |
| **API** | application programming interface |
| **APS** | Aircraft Performance and Survivability Branch |
| **DAVE-ML** | Dynamic Aerospace Vehicle Exchange Markup Language |
| **DoF** | degree of freedom |
| **DOM** | Document Object Model |
| **DST Group** | Defence Science and Technology Group |
| **DTD** | Document Type Description |
| **MathML** | Mathematical Markup Language |
| **MSTC** | Modeling and Simulation Technical Committee |
| **NaN** | Not-a-Number |
| **SI** | Systeme International d'Unites |
| **XML** | eXtensible Markup Language |

# 1 Introduction

## 1.1 Background

Defence Science and Technology (DST) Group Aircraft Performance and Survivability Branch (APS) has reviewed its flight model development and maintenance processes, in conjunction with the requirements of Defence flight model users [1], and decided to align its future flight model dataset structures with the American Institute of Aeronautics and Astronautics (AIAA) modelling and simulation standard [2] and the related Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) Document Type Description (DTD) [3], [4]. Ball Solutions Group was contracted to develop a programming library to provide an application programming interface (API) to the DAVE-ML dataset structure, [5], which is being used by DST APS for the development of performance, flight dynamic and other aircraft models. The library was implemented as a C++ class known as 'Janus.' As the DAVE-ML standard has developed, and as the needs of users have dictated, further development of the Janus library has been performed by DST.

## 1.2 Document Revision History

| Version | Effect | Date |
|---|---|---|
| 0.90 | Original Draft Release | 23 July 2004 |
| 0.91 | additional function documentation | 15 October 2004 |
| 0.92 | additional function documentation, including simple code examples | 14 December 2004 |
| 0.93 | modified API to include 3 types of output variables, replaced dependent variable access procedures with output variable access | 22 December 04 |
| 0.94 | multi-dimensional polynomial interpolation added | 11 January 2005 |
| 0.95 | ungridded interpolation added | 7 February 2005 |
| 0.96 | basic MathML logic capability added | 11 February 2005 |
| 0.97 | namespaces added | 12 March 2005 |
| 0.98 | file header processing and output | 29 March 2005 |
| 0.99 | string array handling added | 30 June 2005 |
| 1.00 | encrypted XML handling added | 18 July 2005 |
| 1.08 | updated to DAVE-ML Ver 2.0RC1 | 14 July 2008 |
| 1.09 | updated to build using Xerces-C version 2.x or 3.x | 17 April 2009 |
| 1.10 | converted to sub-class structure, updated to DAVE-ML Ver 2.0RC2, numerous minor improvements | 31 March 2010 |
| 2.00 | updated to DAVE-ML Ver 2.2, improved management of MathML capability and underlying code structure | 4 September 2014 |

The original version of this document was prepared by Ball Solutions Group, ABN 66 072 963 690, for DST Group, APS of Aerospace Division (AD), in accordance with Attachment A to Contracts No. 123394 and 139826 [5]. Amendments to the document and to the code were performed later by Quantitative Aeronautics, ABN 65 088 688 680, also under contract

to DST, APS of AD [6]. Subsequent amendments to the document and to the code were produced by DST, APS of AD.

## 1.3 Purpose

This document outlines the usage and structure of the Janus C/C++ programming library.

## 1.4 Scope

The elements of the Janus library described in this document are:

1. Methods of instancing the Janus class,
2. The related lower level classes, instanced within a Janus class, which contain and provide access to data extracted from a Document Object Model (DOM) based on the content of a DAVE-ML-compliant dataset,
3. Public types in the classes,
4. Public functions in the classes, and
5. Source file components applicable to each function.

## 1.5 Overview

The Janus library provides a flight modelling programmer with direct access to an eXtensible Markup Language (XML) dataset that conforms to the AIAA modelling standard as implemented under the DAVE-ML DTD version 2.2 [4], in the form of a C++ class. It was initially developed under Linux using the gcc 3.3.4 compiler, and has been tested and used under Microsoft Windows 2000, XP, Vista, and Windows 7 using the Visual C++ compiler, and in the Cygwin environment using the gcc 3.4.4 compiler. Development has since migrated to the newer versions of the gcc compiler under Linux. The class remains backwards compatible with earlier versions of both Linux and Microsoft environments. To load and parse an XML file, the Janus class makes use of the Pugi XML parser library, currently tested at version 1.5, which replaces the previously used Apache Xerces-C++ [1] validating XML parser library.

When initialised, which requires the calling program to supply an XML dataset file name, the library creates and loads a DOM from the file using the PugiXML [2] parser and then extracts numerical and string data from the DOM and stores it in vectors of classes for access from the calling program through the Janus interface. Depending on the dataset and the application, further initialisation may be required after first instantiation. During the initialisation process, problems with the XML file or its contents will cause a standard exception to be thrown, with a relevant message provided. If the calling program does not catch and process this error,

[1] https://xerces.apache.org/xerces-c/ - Apache Software License, Version 2.0
[2] http://pugixml.org/ - MIT license

execution will abort. After initialisation, the Janus library discards the DOM, and therefore, it only operates on the state of the dataset as at initialisation, or on changes made through the Janus interface.

With initialisation complete, the calling program can supply the current state values of relevant independent variables through the Janus and VariableDef interfaces and receive in return output variable values compatible with all the independent variables. The forms of the data and the interpolation, curve fitting or function evaluation required to generate the dependent variable values are controlled by the XML data file content, and are transparent to both the calling program and the user. For these functions, which may be called repeatedly during program execution, speed of execution is a priority and so limited error checking or notification is performed. Wherever error checking is performed, errors are notified using standard exceptions.

At any stage after initialisation, the Janus instance and its sub-classes may be queried for details of any variable or function, including units, names, descriptions, minima, maxima, and interpolation or extrapolation attributes.

The XML dataset may include details of uncertainty in all variables and functions, from which the library can compute the current uncertainty associated with any variable.

Check data included within an XML dataset may be used by the library to validate the processes it performs.

## 1.6 Data Types

To the modeller whose code uses a Janus instance to determine variable values, the underlying form of the XML dataset is irrelevant. However, the dataset developer needs to take account not just of the DAVE-ML DTD, which guides production of a well-formed valid dataset, but also of how Janus treats each data type. The three main data types that will be encountered are:

1. Gridded data, arranged in up to 32 dimensions on a regular grid, which can be interpolated or extrapolated using linear (the default), discrete or low order polynomial data fitting;
2. Ungridded data, a cloud of arbitrarily located data points forming a convex hull, which is partitioned using Delaunay triangulation [3] and interpolated multi-linearly; and
3. Functional representation in Mathematical Markup Language (MathML) [4] form, which is evaluated in accordance with the mathematical operators shown in the dataset. At present Janus implements only the more common operators and qualifiers defined in the MathML DTD. Other operators will be added on request from users. It deals only with real number data, but includes logical operators that return Boolean qualifiers within a calculation element.

For every dataset to be accessed, there will be a preferred data type based on the form of the data and its possible applications. In choosing how to represent a particular piece of

[3] http://www.qhull.org/ - Qhull License
[4] https://www.w3.org/Math/

data within the XML dataset, the modeller should consider how to best make use of Janus's capabilities. Where relevant to computational comparisons below, the software is considered as running on a 'typical' engineering-use PC circa 2009, under either Linux, Windows + Cygwin (Mingw) or Microsoft Visual C++. Some aspects that may be relevant are:

1. Gridded data using linear interpolation is generally the fastest to evaluate, with Janus performing several million evaluations per second on a representative aerodynamic dataset. As the number of degrees of freedom is increased for datasets of equivalent complexity, function evaluation speed typically reduces by forty percent for each additional degree of freedom.

2. Polynomial or spline interpolation of gridded data is typically about forty percent of the speed of linear interpolation of the same data.

3. Ungridded data is generally the slowest to evaluate. For one degree of freedom (DoF) data, an ungridded interpolation is typically an order of magnitude slower than gridded interpolation of the same data based on the same breakpoints. This is because of the added complexity of the barycentric coordinate computation used to weight the contributing data points. In addition, as the number of degrees of freedom increases for datasets of equivalent complexity, function evaluation speed typically reduces by an order of magnitude for each additional DoF.

4. Extrapolation of any form of data is inherently risky; however, gridded data extrapolation is much safer than ungridded data extrapolation. Because the ungridded data is processed in barycentric coordinates, not Cartesian coordinates, and checking Cartesian directions wastes processing time, Janus will only extrapolate such data if *all* independent variables of the function are set to be extrapolated in both directions.

5. MathML functions, including piecewise representations, are evaluated at speeds similar to gridded data of equivalent complexity. However, high order polynomial evaluation can be computationally costly. Luckily, high order polynomials are almost always a bad choice for representation of aeronautical data.

6. An extension to the DAVE-ML standard allows arrays of strings to be stored in and accessed from gridded tables. The applications of this are quite limited, and the related function documentation should be fully complied with for successful use.

7. An extension to the DAVE-ML standard allows scale factors to be applied to the outputs of functional computations. This was implemented in response to pressing user requests; however, it is not recommended in normal use.

8. Computation of uncertainty may be quite slow, since propagation of errors through the data structure requires repeated function evaluations and, in some cases, numerical differentiation. Where computation speed is an issue, it may not be appropriate to request uncertainty for each computation.

9. An extension to the DAVE-ML standard allows data to be managed as vectors or n-dimensional matrices, [9].

## 1.7 Example Usage

Examples of usage of the individual components of the Janus library are included throughout this report. The code below provides a minimalist example of usage of the complete Janus system to select an output, enter the required inputs, and compute the corresponding output data value.

```
#include <iostream>
#include "Janus.h"

using namespace std
using namespace janus;

int main( int, char**)
{
  Janus janus( "~/pika/pika.xml");

  VariableDef jAlpha = janus.getVariableDef("alpha_");
  VariableDef jCL    = janus.getVariabeDef( janus.getVariableDef("CL");

  double alpha = 5.0 * pi / 180.0;
  jAlpha.setValue( alpha);
  double CL = jCL.getValue();
  cout << "CL = " << CL << endl;

  return 0;
}
```

The "test" subdirectory in the Janus distribution also contains various complete code examples that can be compiled directly.

# 2 Module Documentation

This section presents functions to permit an application to interface with an DAVE-ML compliant XML file using the Janus API software. It details functions for instantiating an instance of the Janus API, together with high-level functions for interfacing with data stored within the instance.

A number of interface functions have been deprecated as the functionality of the Janus API has been revised and further developed. These functions are presented in Appendix A of this report. Support for these functions has been retained within Janus to provide backwards compatibility for applications that were developed using previous versions of the API. However, it is recommended that these functions should not be used when developing new applications, instead the various components should be accessed through the specific class interfaces.

## 2.1 Janus - Class Instantiation

### Enumerations

### Functions

- virtual void clear ()
- virtual size_t exportToBuffer (std::ostringstream &documentBuffer)
- virtual size_t exportToBuffer (unsigned char *&documentBuffer)
- virtual size_t exportToFile (const dstoute::aFileString &dataFileName)
- DomFunctions::XmlNode getDomDocument () const
- const char * getJanusVersion (VersionType versionType=HEX) const
- const dstoute::aFileString & getXmlFileName () const
- void initiateDocumentObjectModel (const dstoute::aString &documentType="DAVEfunc")
- bool isJanusInitialised () const
- Janus ()
- Janus (const dstoute::aFileString &documentName, const dstoute::aFileString &keyFileName="")
- Janus (unsigned char *documentBuffer, size_t documentBufferSize)
- Janus (const Janus &rhs)
- Janus & operator= (const Janus &rhs)
- virtual void setXmlFileBuffer (unsigned char *documentBuffer, const size_t &documentBufferSize)
- virtual void setXmlFileName (const dstoute::aFileString &documentName, const dstoute::aFileString &keyFileName="")
- virtual ~Janus ()

### 2.1.1 Detailed Description

The instantiation functions relate to the construction and destruction of a Janus instance. They perform XML initialisation using the pugiXML parser loading the supplied XML file or data buffer to a DOM structure and create vectors and numeric arrays based on the XML data.

The instantiation process will throw standard exceptions if the XML file or the data buffer do not load or parse successfully. If the calling program does not catch these exceptions, the program will abort. An example of exception handling, applicable to all forms of Janus instantiation, is:

```
try {
  prop.setXmlFileName( fileName );
}
catch ( exception &excep ) {
  cerr << excep.what() << " \n\n";
  return 1;
}
```

### 2.1.2 Enumeration Type Documentation

#### 2.1.2.1 enum ExportObjectType

This enum is used to indicate the Export Object options.

Enumerator

***FILE*** a data file
***BUFFER*** a data Buffer

#### 2.1.2.2 enum VersionType

This enum is used to indicate whether a short, long, or HEX library version description string is required.

Enumerator

***SHORT*** a short, purely numeric string, eg "0.97"
***LONG*** a longer, alpha-numeric string, eg "Janus V-0.97"
***HEX*** a hexadecimal, eg 0x000907

### 2.1.3 Function Documentation

#### 2.1.3.1 void clear ( ) [virtual]

Initialise all Janus member variables as per the empty constructor.

#### 2.1.3.2 virtual size_t exportToBuffer ( std::ostringstream & *documentBuffer* ) [inline], [virtual]

This function exports the contents of the Janus instance to an XML data buffer complying with the DAVE-ML syntax as defined by the DAVE-ML DTD.

This function may be overloaded by equivalent functions for projects that inherit Janus.

Parameters

| | |
|---|---|
| *documentBuffer* | an address to a ostringstream buffer that data will be written. This data is not '\0' terminated. |

Returns

The *size* of the buffer will be returned. If size is zero then the export of the buffer has failed.

#### 2.1.3.3 virtual size_t exportToBuffer ( unsigned char *& *documentBuffer* ) [inline], [virtual]

This function exports the contents of the Janus instance to an XML data buffer complying with the DAVE-ML syntax as defined by the DAVE-ML DTD.

This function may be overloaded by equivalent functions for projects that inherit Janus.

Parameters

| | |
|---|---|
| *documentBuffer* | an address to a unsigned char* buffer that data will be written. The memory allocated to the buffer within this function MUST be managed (freed) by the external application. The buffer will be '\0' terminated. |

Returns

The *size* of the buffer will be returned. If size is zero then the export of the buffer has failed.

#### 2.1.3.4 virtual size_t exportToFile ( const dstoute::aFileString & *dataFileName* ) [inline], [virtual]

This function exports the contents of the Janus instance to an XML text file complying with the DAVE-ML syntax as defined by the DAVE-ML DTD.

This function may be overloaded by equivalent functions for projects that inherit Janus.

Parameters

| | |
|---|---|
| *dataFileName* | the name of the file to which the data will be written. |

Returns

The value returned will be greater than zero if the data has been successfully exported to the file, else it will be zero.

#### 2.1.3.5 DomFunctions::XmlNode getDomDocument ( ) const [inline]

This function permits a calling routine or application to gain access to the parent document of an instantiated DOM.

Returns

a *DomFunctions::XmlDoc* pointer to the DOM document is returned

#### 2.1.3.6 const char * getJanusVersion ( VersionType *versionType* = HEX ) const

This function allows the calling program to retrieve the version number of the Janus library that is in use. It is particularly useful for dynamically linked programs that may use several different library versions.

Parameters

| | |
|---|---|
| *versionType* | determines whether a short, long or hexadecimal string is returned. |

Returns

a character pointer to the version description string.

#### 2.1.3.7 const dstoute::aFileString& getXmlFileName ( ) const [inline]

If the instance has been fully initialised, the fully-qualified name of the XML dataset file from which it was initialised is returned by this function.

Returns

The XML file name e.g. "~/pika/pika_prop.xml"

#### 2.1.3.8 void initiateDocumentObjectModel ( const dstoute::aString & *documentType* = *"DAVEfunc"* ) [inline]

This function is used to initiate a Document Object Model that stores data for exporting to an XML file.

Parameters

| | |
|---|---|
| *documentType* | this is a string indicating the document type definition of the read or exported XML file. The default document type is *DAVEfunc* indicating that the XML file is encoded using the DAVE-ML syntax as defined in the DAVEfunc.dtd. XML syntax and applications that build upon Janus may define alternative document types, and accompanying document type definitions, such as *THAMESfunc*, which is the Time History for Aircraft Modelling Exchange Syntax. |

#### 2.1.3.9 bool isJanusInitialised ( ) const [inline]

This function permits a calling program to determine if this instance of Janus has been instantiated or is empty.

Returns

a boolean indicating if the Janus instance has been instantiated.

#### 2.1.3.10 Janus ( )

The empty constructor can be used to instance the Janus class without supplying a name for the XML file from which the DOM is to be constructed, but in this state is not useful for any class functions. It will require an XML file name to be supplied before any further use of the instanced class. This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

setXmlFileName

#### 2.1.3.11 Janus ( const dstoute::aFileString & *documentName*, const dstoute::aFileString & *keyFileName* = "" )

The constructor, when called with the XML document name, does the XML initialisation and then loads the DOM structure. A minimal example is:

```
#include <string>
#include "Janus.h"

using namespace std;

int main (int argc, char* args[])
{
  char fileName[] = "~/pika/pika_prop.xml";
  Janus prop( fileName );

  return 0;
}
```

When the XML file name is supplied, either at instantiation or afterwards when using setXmlFileName, the DOM is parsed against *DAVEfunc.dtd* before data structures are set up. The

constructor calls private functions within the class to set up instances representing *fileHeader*, *variableDef*, *breakpointDef*, *griddedTableDef*, *ungriddedTableDef*, *function* and *checkData* DOM Level 1 elements.

Parameters

| | |
|---|---|
| *documentName* | is the XML file name, e.g. "~/pika/pika_prop.xml" |
| *keyFileName* | is name of a file containing a RSA public key. It is used for interacting with encrypted DAVE-ML data files. |

#### 2.1.3.12 Janus ( unsigned char * *documentBuffer*, size_t *documentBufferSize* )

The constructor, when called with the XML document name, does the XML initialisation and then loads the DOM structure. A minimal example is:

```
#include <string>
#include "Janus.h"

using namespace std;

int main (int argc, char* args[])
{
  unsigned char* buffer;
  ...
  Janus prop( buffer, bufferSize);

  return 0;
}
```

When the XML file name is supplied, either at instantiation or afterwards when using setXmlFileName, the DOM is parsed against *DAVEfunc.dtd* before data structures are set up. The constructor calls private functions within the class to set up instances representing *fileHeader*, *variableDef*, *breakpointDef*, *griddedTableDef*, *ungriddedTableDef*, *function* and *checkData* DOM Level 1 elements.

Parameters

| | |
|---|---|
| *documentBuffer* | is the XML buffer." |
| *documentBufferSize* | is the XML buffer size or length." |

#### 2.1.3.13 Janus ( const Janus & *rhs* )

The copy constructor may be used to duplicate the data stored within a Janus instance.

Parameters

| | |
|---|---|
| *rhs* | is a reference to the Janus instance being copied. |

See also

operator=

#### 2.1.3.14 Janus & operator= ( const Janus & *rhs* )

The assignment operator may be used to duplicate the data stored within a Janus instance.

Parameters

| | |
|---|---|
| *rhs* | is a reference to the Janus instance being copied. |

#### 2.1.3.15 void setXmlFileBuffer ( unsigned char * *documentBuffer*, const size_t & *documentBufferSize* ) [virtual]

An uninitialised instance of Janus is populated using a nominated XML data buffer through this function. The DOM for this Janus instance is loaded from the buffer, parsed against the *DAVEfunc.dtd* before data arrays are set up.

If an instance has previously been initialised then the check performed at load time throws a standard exception.

If the instance of Janus is not being inherited by another process then the data buffer is deleted after it has been parsed successfully.

Parameters

| | |
|---|---|
| *documentBuffer* | is the XML data buffer |
| *documentBufferSize* | is the size of the buffer in bytes |

#### 2.1.3.16 void setXmlFileName ( const dstoute::aFileString & *documentName*, const dstoute::aFileString & *keyFileName* = "" ) [virtual]

An uninitialised instance of Janus is populated using a nominated XML file through this function. The DOM for this Janus instance is loaded from the named file, parsed against the *DAVEfunc.dtd* before data arrays are set up.

If another XML file name is supplied to an instance that has already been initialised then the check performed at load time throws a standard exception.

Parameters

| | |
|---|---|
| *documentName* | is the XML file name, e.g. "~/pika/pika_prop.xml" |
| *keyFileName* | is name of a file containing a RSA public key. It is used for interacting with encrypted DAVE-ML data files. |

#### 2.1.3.17 ~Janus ( ) [virtual]

After deleting memory allocations, the parser instance is released. The destructor is called automatically when the instance goes out of scope.

## 2.2 Janus - Level 1 Elements

### Functions

- void displayCheckDataSummary (const CheckData &checkData)
- VariableDef * findVariableDef (const dstoute::aString &varID)
- BreakpointDefList & getBreakpointDef ()
- const CheckData & getCheckData (const bool &evaluate=true)
- const FileHeader & getFileHeader () const
- const FunctionList & getFunction () const
- Function & getFunction (size_t index)
- GriddedTableDefList & getGriddedTableDef ()
- PropertyDefList & getPropertyDef ()
- PropertyDef & getPropertyDef (size_t index)
- PropertyDef & getPropertyDef (const dstoute::aString &ptyID)
- UngriddedTableDefList & getUngriddedTableDef ()
- VariableDefList & getVariableDef ()
- VariableDef & getVariableDef (size_t index)
- VariableDef & getVariableDef (const dstoute::aString &varID)
- int getVariableIndex (const dstoute::aString &varID) const

### 2.2.1 Detailed Description

There are seven DAVE-ML DTD Level 1 elements that attach directly to the *DAVEfunc* root element and provide the basic structure of the DOM. These functions provide access to each of these elements, so that lower-level functions can access the raw data of the element contents. These functions are generally used within the Janus instance, and some of them are not normally called by external programs.

### 2.2.2 Function Documentation

#### 2.2.2.1 void displayCheckDataSummary ( const CheckData & *checkData* )

The DAVE-ML DTD permits an XML dataset to contain sets of input signal values and the corresponding output signal values (and, optionally, the corresponding internal values). These sets of data may be used to validate the functional processes represented by the remainder of the XML dataset. A dataset that includes one or more *checkData* elements can therefore be self-validating. This function provides a means for calling programs to display a summary of the check data contained within a Janus instance. Not all datasets contain a *checkData* element.

Parameters

| | |
|---|---|
| *checkData* | A reference to the CheckData instance within a Janus instance. |

#### 2.2.2.2 VariableDef * findVariableDef ( const dstoute::aString & *varID* )

This function searches the list of VariableDefs within a Janus instance for an entry corresponding to the *varID* variable identifier specified.

Parameters

| | |
|---|---|
| *varID* | is a C++ string containing the *varID* of the variableDef within the list of VariableDef instances to be returned from the Janus instance. A null pointer is returned if there is no corresponding variableDef for the *varID* specified. |

Returns

The pointer to the selected VariableDef is returned, otherwise 0.

#### 2.2.2.3 BreakpointDefList& getBreakpointDef ( ) [inline]

This function returns the list of the breakpoint definitions, *breakpointDef*, storing gridded table break point data. A *breakpointDef* contains identification and cross-reference data, as well as a set of independent variable values associated with one of the dimensions of a gridded table of data. A breakpoint definition may be used by more than one gridded table function. Provided it has been instantiated without error, each Janus instance contains one *breakpointDef* vector of instances. However, the vector may have zero length in a dataset that does not include gridded data.

Returns

A reference to the list of BreakpointDef instances is returned.

#### 2.2.2.4 const CheckData & getCheckData ( const bool & *evaluate* = *true* )

The DAVE-ML DTD permits an XML dataset to contain sets of input signal values and the corresponding output signal values (and, optionally, the corresponding internal values). These sets of data may be used to validate the functional processes represented by the remainder of the XML dataset. A dataset that includes one or more *checkData* elements can therefore be self-validating. This function provides a means for calling programs to access all check data contained within a Janus instance. Not all datasets contain a *checkData* element.

Returns

A reference to the CheckData instance within a Janus instance is returned.

#### 2.2.2.5 const FileHeader& getFileHeader ( ) const [inline]

This function permits a calling program to retrieve the header information for a DAVE-ML compliant XML dataset. Descriptive material is contained in the file header. This includes file authorship, modification records, and cross-references to source material. FileHeader and lower level class functions may be used through the returned reference to access the *fileHeader* contents. Provided it has been instantiated without error, each Janus instance contains one FileHeader instance.

Returns

A reference to the FileHeader instance is returned.

#### 2.2.2.6 const FunctionList& getFunction ( ) const [inline]

The *function* elements contained in a DOM that complies with the DAVE-ML DTD, indicate how an output value is to be computed from independent inputs and tabulated data, either gridded or ungridded. Each *function* has an optional description, optional provenance data, and either a simple table of input/output values or references to more complete (possibly multiple) input, output, and function data elements. In general, calling programs should access function-based data through *variableDef* procedures rather than directly through the Function instance and its lower-level procedures. Provided it has been instantiated without error, each Janus instance contains a vector of Function instances; however, the vector may have zero length.

Returns

A reference to the list of Function instances is returned.

#### 2.2.2.7 Function& getFunction ( size_t *index* ) [inline]

As well as accessing the complete vector of Function instances within a Janus instance, an individual Function may be accessed by index.

Parameters

| | |
|---|---|
| *index* | is the index of a required Function within the vector of Function instances contained in a Janus instance. It has a range from 0 to (getFunction().size() - 1) |

Returns

The selected Function is returned by reference.

#### 2.2.2.8 GriddedTableDefList& getGriddedTableDef ( ) [inline]

Within the DOM, a *griddedTableDef* contains points arranged in an orthogonal (possibly multi-dimensional) array, where the independent variables are defined by separate breakpoint vectors. This table definition may be specified within a function, or separately so that it may be used by multiple functions. Janus handles both forms similarly. The table data points are specified as comma-separated values in floating-point notation (0.93638E-06) in a single long sequence as if the table had been unravelled with the last-specified dimension changing most rapidly. Line breaks and comments are ignored by Janus. Provided it has been instantiated without error, each Janus instance contains one GriddedTableDef vector of instances; however, the vector may have zero length.

Returns

A reference to the list of GriddedTableDef instances is returned.

#### 2.2.2.9 PropertyDefList& getPropertyDef ( ) [inline]

The *propertyDef* elements contained in a DOM that complies with the DAVE-ML DTD, defines a descriptive parameter that may be used describe a property or object associated with an model. It can be used to encode non-numeric parameters. In general, calling programs should access property-based data through the PropertyDef instance and its lower-level procedures. Provided it has been instantiated and initialised without error, each Janus instance contains a ParameterDef vector of instances with zero or more entries.

Returns

A reference to the list of PropertyDef instances is returned.

#### 2.2.2.10 PropertyDef& getPropertyDef ( size_t *index* ) [inline]

As well as accessing the complete vector of PropertyDefs within a Janus instance, an individual PropertyDef may be accessed by index.

Parameters

| | |
|---|---|
| *index* | is the offset of a required PropertyDef within the vector of PropertyDef instances contained in a Janus instance. It has a range from 0 to (getPropertyDef().size() - 1) |

Returns

The selected PropertyDef is returned by reference.

#### 2.2.2.11 PropertyDef& getPropertyDef ( const dstoute::aString & *ptyID* )

As well as accessing the complete vector of PropertyDefs within a Janus instance, an individual PropertyDef may be accessed by ptyID.

Parameters

| | |
|---|---|
| *ptyID* | is a C++ string containing the *ptyID* of a required PropertyDef within the vector of PropertyDef instances contained in a Janus instance. Where no PropertyDef contains a *ptyID* matching the input, a standard exception will be thrown. |

Returns

The selected PropertyDef is returned by reference.

#### 2.2.2.12 UngriddedTableDefList& getUngriddedTableDef ( ) [inline]

Within the DOM, an *ungriddedTableDef* contains points that are not in an orthogonal grid pattern; thus, the independent variable coordinates are specified for each dependent variable value. The table data point values are specified as comma-separated values in floating-point notation. Provided it has been instantiated without error, each Janus instance contains one UngriddedTableDef vector of instances; however, the vector may have zero length.

Returns

A reference to the list of UngriddedTableDef instances is returned.

#### 2.2.2.13 VariableDefList& getVariableDef ( ) [inline]

The *variableDef* elements contained in a DOM that complies with the DAVE-ML DTD identify the input and output signals used by function blocks. They also provide MathML content markup to indicate that a calculation is required to arrive at the value of the variable, using other variables as inputs. The variable definition can include statistical information regarding the uncertainty of the values that it might take on, when measured after any calculation is performed. In general, calling programs should access variable-based data through the VariableDef instance and its lower-level procedures. Provided it has been instantiated and initialised without error, each Janus instance contains one VariableDef vector of instances, of length not less than one.

Returns

A reference to the list of VariableDef instances is returned.

#### 2.2.2.14 VariableDef& getVariableDef ( size_t *index* ) [inline]

As well as accessing the complete vector of VariableDefs within a Janus instance, an individual VariableDef may be accessed by index.

Parameters

| | |
|---|---|
| *index* | is the offset of a required VariableDef within the vector of VariableDef instances contained in a Janus instance. It has a range from 0 to (getVariableDef().size() - 1) |

Returns

The selected VariableDef is returned by reference.

#### 2.2.2.15 VariableDef& getVariableDef ( const dstoute::aString & *varID* )

As well as accessing the complete vector of VariableDefs within a Janus instance, an individual VariableDef may be accessed by varID.

Parameters

| | |
|---|---|
| *varID* | is a C++ string containing the *varID* of a required VariableDef within the vector of VariableDef instances contained in a Janus instance. Where no VariableDef contains a *varID* matching the input, a standard exception will be thrown. |

Returns

The selected VariableDef is returned by reference.

#### 2.2.2.16 int getVariableIndex ( const dstoute::aString & *varID* ) const [inline]

A variable's *varID* attribute is uniquely related to the *variableDef* and may be used as an index. This function is used by a calling program to establish the numeric index associated with a variable definition. If the *varID* is known for a variable definition then knowing the index provides a more efficient means of interfacing with the *variableDef* as it eliminates the need to perform string comparisons. The returned integer value may be used to address all *variable'-related* attributes, child nodes or data elements.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the variable of which it is an attribute. |

Returns

An *index* in the range from 0 to (getNumberOfVariables() - 1), corresponding to the variable whose *varID* matches the input *varID*. If the input does not match any dependent variable ID within the DOM, the returned value is -1.

# 3 Namespace Documentation

This section documents the *janus* namespace.

## 3.1 janus Namespace Reference

### Classes

- class Array
- class Author
- class Bounds
- class BreakpointDef
- class CheckData
- class CheckInputs
- class CheckOutputs
- class DimensionDef
- class FileHeader
- class Function
- class FunctionDefn
- class GriddedTableDef
- class InDependentVarDef
- class InternalValues
- class Janus
- class Model
- class Modification
- class PropertyDef
- class Provenance
- class Reference
- class Signal
- class SignalList
- class StatespaceFn
- class StaticShot
- class TransferFn
- class Uncertainty
- class UngriddedTableDef
- class VariableDef
- class XmlElementDefinition

### 3.1.1 Detailed Description

The Janus class, all functions within it, and various subordinate classes are included in the *janus* namespace. Because Janus is a library, it is expected to be used in conjunction with external classes and the namespace provides a way of avoiding possible naming clashes with those classes.

# 4 Class Documentation

This section documents the public interface functions for the various Janus API classes.

## 4.1 Array Class Reference

`#include <Array.h>`

Inherits XmlElementDefinition.

Inherited by Denominator, and Numerator.

### Public Member Functions

- Array ()
- Array (const DomFunctions::XmlNode &elementDefinition)
- void exportDefinition (DomFunctions::XmlNode &documentElement, const dstoute::aString &elementTag="")
- size_t getArraySize () const
- const dstoute::aStringList & getStringDataTable () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- void setStringDataTable (const dstoute::aStringList stringDataTable)

### 4.1.1 Detailed Description

An Array instance holds in its allocated memory alphanumeric data derived from an *array* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file.

It includes entries arranged as follows: Entries for a vector represent the row entries of that vector. Entries for a matrix are specified such that the column entries of the first row are listed followed by column entries for subsequent rows until the base matrix is complete. This sequence is repeated for higher order matrix dimensions until all entries of the matrix are specified.

The Array class is only used within the janus namespace, and should only be referenced through the Janus class.

### 4.1.2 Constructor & Destructor Documentation

#### 4.1.2.1 Array ( )

The empty constructor can be used to instance the Array class without supplying the DOM *array* element from which the instance is constructed, but in this state it is not useful for any

class functions. It is necessary to populate the class from a DOM containing an *array* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.1.2.2 Array ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to an *array* element within a DOM, instantiates the Array class and fills it with alphanumeric data from the DOM. String-based numeric data are converted to double-precision linear vectors.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *array* component node within the DOM. |

### 4.1.3 Member Function Documentation

#### 4.1.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement*, const dstoute::aString & *elementTag* = "" )

This function is used to export the *array* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |
| *elementTag* | a string defining the XML element tag name |

#### 4.1.3.2 size_t getArraySize ( ) const [inline]

This function returns the number of entries stored in an array.

Returns

The number of entries stored in an array.

#### 4.1.3.3 const dstoute::aStringList& getStringDataTable ( ) const [inline]

This function provides access to a vector of alphanumeric data stored in an Array instance. This vector contains the data strings in the same sequence as they were presented in the *dataTable* of the corresponding XML dataset.

Returns

The string list containing the alphanumeric content of the Array instance is returned by reference.

#### 4.1.3.4 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of Array is filled with data from a particular *array* element within a DOM by this function. If another *array* element pointer is supplied to an instance that has already been initialised, data corruption will occur and the entire Janus instance will become unusable.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *array* component node within the DOM. |

#### 4.1.3.5 void setStringDataTable ( const dstoute::aStringList *stringDataTable* ) [inline]

This function permits the string data table of the *array* element to be reset for this Array instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *stringDataTable* | a string list containing data table entries. |

The documentation for this class was generated from the following files:

- Array.h
- Array.cpp

## 4.2 Author Class Reference

`#include <Author.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- Author ()
- Author (const DomFunctions::XmlNode &authorElement)
- void exportDefinition (DomFunctions::XmlNode &documentElement, const dstoute::aString &authorTag="author")
- const dstoute::aStringList & getAddress () const
- const dstoute::aString & getAddress (const size_t &index) const
- size_t getAddressCount () const
- const dstoute::aString & getContactInfo (const size_t &index) const
- size_t getContactInfoCount () const
- const dstoute::aString & getContactInfoType (const size_t &index) const
- const dstoute::aString & getContactLocation (const size_t &index) const
- const dstoute::aString & getEmail () const
- const dstoute::aString & getName () const
- const dstoute::aString & getOrg () const
- const dstoute::aString & getXns () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- void setAddress (const dstoute::aStringList &address)
- void setContactInfo (const dstoute::aStringList &contactInfo)
- void setContactInfoType (const dstoute::aStringList &contactInfoType)
- void setContactLocation (const dstoute::aStringList &contactLocation)
- void setEmail (const dstoute::aString &email)
- void setName (const dstoute::aString &name)
- void setOrg (const dstoute::aString &org)
- void setXns (const dstoute::aString &xns)

### 4.2.1 Detailed Description

An Author instance holds in its allocated memory alphanumeric data derived from an *author* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance may describe an author of a complete dataset, or of a component of a dataset, or of a modification to a dataset. Author contact details may be expressed in either *address*

or *contactInfo* forms. The *contactInfo* form is newer, more flexible and generally preferred. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Author class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader, Modification or Provenance classes.

Typical usage might be:

```
Janus test( xmlFileName );
int nAuthors = test.getFileHeader().getAuthorCount( );
cout << "Number of authors : " << nAuthors << "\n\n";
for ( size_t i = 0 ; i < nAuthors ; i++ ) {
  Author author = test.getFileHeader().getAuthor( i );
  cout << " Author " << i << " : Name                  : "
       << author.getName( ) << "\n"
       << "          Organisation            : "
       << author.getOrg( ) << "\n"
       << "          Email                   : "
       << author.getEmail( ) << "\n\n";
  for ( size_t j = 0 ; j < author.getAddressCount() ; j++ ) {
    cout << "          Address " << j << "               : "
         << author.getAddress( j ) << "\n\n";
  }
  for ( size_t j = 0 ; j < author.getContactInfoCount() ; j++ ) {
    cout << "          Contact " << j << " type          : "
         << author.getContactInfoType( j ) << "\n"
         << "                    location      : "
         << author.getContactLocation( j ) << "\n"
         << "                    content       : "
         << author.getContactInfo( j )
         << "\n\n";
  }
}
```

### 4.2.2 Constructor & Destructor Documentation

#### 4.2.2.1 Author ( )

The empty constructor can be used to instance the Author class without supplying the DOM *author* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing an *author* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.2.2.2 Author ( const DomFunctions::XmlNode & *authorElement* )

The constructor, when called with an argument pointing to an *author* element within a DOM, instantiates the Author class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *authorElement* | is an address of an *author* component node within the DOM. |

### 4.2.3 Member Function Documentation

#### 4.2.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement*, const dstoute::aString & *authorTag* = `"author"` )

This function is used to export the *author* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |
| *authorTag* | a string specifying the tag to use when exporting the *Author* element. The default tag is *author*, with other an alternative being *pilot* when used by higher-level applications such as *Thames*. |

#### 4.2.3.2 const dstoute::aStringList& getAddress ( ) const `[inline]`

This function returns the *address* list from an Author instance.

Returns

The *address* list is passed as a reference to string list of *address* entries.

#### 4.2.3.3 const dstoute::aString& getAddress ( const size_t & *index* ) const `[inline]`

This function returns a selected *address* component from an Author instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getAddressCount() - 1 ), and selects the required *address* component. An attempt to access a non-existent *address* will throw a standard out_of_range exception. |

Returns

The selected *address* string is passed by reference.

#### 4.2.3.4 size_t getAddressCount ( ) const [inline]

This function returns the number of addresses listed in an Author instance. An instance can have no, one or multiple *address* components. The *address* and *contactInfo* components are mutually exclusive alternatives. If the instance has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

See also

getContactInfoCount

#### 4.2.3.5 const dstoute::aString& getContactInfo ( const size_t & *index* ) const [inline]

This function returns the content of a selected *contactInfo* component from an Author instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getContactInfoCount() - 1 ), and selects the required *contactInfo* component. An attempt to access a non-existent *contactInfo* will throw a standard out_of_range exception. |

Returns

The selected *contactInfo* content string is passed by reference.

#### 4.2.3.6 size_t getContactInfoCount ( ) const [inline]

This function returns the number of *contactInfo* components listed in the referenced Author instance. An instance can have no, one or multiple *contactInfo* components. The *contactInfo* and *address* components are mutually exclusive alternatives. If the instance has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

See also

getAddressCount

#### 4.2.3.7 const dstoute::aString& getContactInfoType ( const size_t & *index* ) const [inline]

This function returns the *contactInfoType* of a selected *contactInfo* component from an Author instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getContactInfoCount() - 1 ), and selects the required *contactInfo* component. An attempt to access a non-existent *contactInfo* will throw a standard out_of_range exception. |

Returns

The selected *contactInfoType* string is passed by reference.

#### 4.2.3.8 const dstoute::aString& getContactLocation ( const size_t & *index* ) const [inline]

This function returns the *contactLocation* of a selected *contactInfo* component from an Author instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getContactInfoCount() - 1 ), and selects the required *contactInfo* component. An attempt to access a non-existent *contactInfo* will throw a standard out_of_range exception. |

Returns

The selected *contactLocation* string is passed by reference.

#### 4.2.3.9 const dstoute::aString& getEmail ( ) const [inline]

This function returns the author's *email* attribute from the referenced Author instance. The *email* attribute contains the author's email address. This is an optional attribute.

Returns

The *email* string is passed by reference. If the Author instance has not been initialised or does not contain an *email* attribute, an empty string is returned.

#### 4.2.3.10 const dstoute::aString& getName ( ) const [inline]

This function returns the author's *name* from the referenced Author instance. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is passed by reference.

#### 4.2.3.11 const dstoute::aString& getOrg ( ) const [inline]

This function returns the author's *org* attribute from the referenced Author instance. The *org* attribute is a descriptive string identifying the author's employing organisation. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *org* string is passed by reference.

#### 4.2.3.12 const dstoute::aString& getXns ( ) const [inline]

This function returns the author's *xns* attribute from the referenced Author instance. The *xns* attribute is a descriptive string containing the author's eXtensible Name Service identifier. This is an optional attribute.

Returns

The *xns* string is passed by reference. If the Author instance has not been initialised or does not contain an *xns* attribute, an empty string is returned.

#### 4.2.3.13 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of Author is filled with data from a particular *author* element within a DOM by this function. If another *author* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *author* component node within the DOM. |

#### 4.2.3.14 void setAddress ( const dstoute::aStringList & *address* ) [inline]

This function permits the *address* vector of the *author* element to be reset for this Author instance. An alternative is to populate the *contactInfo* entries of the Author instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *address* | a string list containing address entries for the named author. |

#### 4.2.3.15 void setContactInfo ( const dstoute::aStringList & *contactInfo* ) [inline]

This function permits the vector of *contactInfo* of the *author* element to be reset for this Author instance. The element content is set through this function, with the type and location attributes populated using the *setContactInfoType()* and *setContactLocation()* functions. An alternative is to populate the *address* entries of the Author instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *contactInfo* | a string list containing contact information for the named author. |

#### 4.2.3.16 void setContactInfoType ( const dstoute::aStringList & *contactInfoType* ) [inline]

This function permits the vector of *contactInfoType* data of the *author* element to be reset for this Author instance. These data are an attribute of the *contactInfo* element.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *contactInfoType* | a string list containing contact type information (as defined in the DAVE-ML dtd) for the named author. |

#### 4.2.3.17 void setContactLocation ( const dstoute::aStringList & *contactLocation* ) [inline]

This function permits the vector of *contactLocation* data of the *author* element to be reset for this Author instance. These data are an attribute of the *contactInfo* element.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *contactLocation* | a string list containing contact location information (as defined in the DAVE-ML dtd) for the named author. |

#### 4.2.3.18 void setEmail ( const dstoute::aString & *email* ) [inline]

This function permits the author's *email* attribute of the *author* element to be reset for this Author instance. The *email* attribute contains the author's email address. This is an optional attribute.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *email* | a string detailing the address to be stored in the author's email attribute. |

#### 4.2.3.19 void setName ( const dstoute::aString & *name* ) [inline]

This function permits the *name* attribute of the *author* element to be reset for this Author instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *name* | a string detailing the author's name. |

#### 4.2.3.20 void setOrg ( const dstoute::aString & *org* ) [inline]

This function permits the *org* attribute of the *author* element to be reset for this Author instance.

If the instance has not been initialised from a DOM then this function permits it to be set

before being written to an output XML based file.

Parameters

| | |
|---|---|
| *org* | a string detailing the name of the author's organisation. |

#### 4.2.3.21 void setXns ( const dstoute::aString & *xns* ) [inline]

This function permits the *xns* attribute of the *author* element to be reset for this Author instance. The *xns* attribute is a descriptive string containing the author's eXtensible Name Service identifier. This is an optional attribute.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *xns* | a string detailing the author's xns attribute. |

The documentation for this class was generated from the following files:

- Author.h
- Author.cpp

## 4.3 Bounds Class Reference

`#include <Bounds.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- Bounds ()
- Bounds (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- void exportDefinition (DomFunctions::XmlNode &documentElement)
- double getBound (const int &functionIndex=-1) const
- const dstoute::aString & getVarID () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- virtual ~Bounds ()

### 4.3.1 Detailed Description

A Bounds instance holds in its allocated memory alphanumeric data derived from a *bounds* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The element contains some description of the statistical limits to the values the citing parameter element might take on. This can be in the form of a scalar value, a *variableDef* that provides a functional definition of the bound, a *variableRef* that refers to such a functional definition, or a private table whose elements correlate with those of a tabular function defining the citing parameter. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Bounds class is only used within the janus namespace, and should only be referenced indirectly through the Uncertainty class or through the variable functions within the Janus class.

One possible usage of the Bounds class might be:

```
Janus test( xmlFileName );

for ( int i = 0 ; i < test.getNumberOfVariables() ; i++ ) {
  Uncertainty::UncertaintyPdf pdf = test.getVariableDef().at( i ).
    getUncertainty().getPdf( );
  if ( Uncertainty::NORMAL_PDF == pdf ) {
    double bound = test.getVariableDef().at( i ).getUncertainty().
      getBounds().getBound();
    cout << " Gaussian bound = " << bound << "\n";
  }
  else if ( Uncertainty::UNIFORM_PDF == pdf ) {
    const vector<Bounds>& bounds = test.getVariableDef().at( i ).
      getUncertainty().getBounds();
    if ( 1 == bounds.size() ) {
      double symmetricBound = bounds.at( 0 ).getBound();
      cout << " Uniform symmetric bound = " << symmetricBound << "\n";
    }
    else {
      double lowerBound = bounds.at( 0 ).getBound();
      double upperBound = bounds.at( 1 ).getBound();
      cout << " Uniform bounds range = [ " << lowerBound << " to "
           << upperBound << " ]\n";
    }
  }
}
```

### 4.3.2 Constructor & Destructor Documentation

#### 4.3.2.1 Bounds ( )

The empty constructor can be used to instance the Bounds class without supplying the DOM *bounds* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *bounds* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.3.2.2 Bounds ( const DomFunctions::XmlNode & *elementDefinition,* Janus * *janus* )

The constructor, when called with an argument pointing to a *bounds* element within a DOM, instantiates the Bounds class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *bounds* component node within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to evaluate bounds with a functional dependence on the instance state. It must be passed as a void pointer to avoid circularity of dependencies at build time. |

#### 4.3.2.3 ~Bounds ( ) [virtual]

Destructor required to free the locally allocated memory of the variableDef

### 4.3.3 Member Function Documentation

#### 4.3.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *bound* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.3.3.2 double getBound ( const int & *functionIndex* = *-1* ) const

This function returns the current value of the bound defined by this Bounds instance, based if necessary on the current state of all variables within the parent Janus instance. It will perform whatever computations are required to determine the bound. If the bound can not be determined for any reason, a NaN will be returned.

Parameters

| | |
|---|---|
| *functionIndex* | is an optional argument, only necessary for tabular bounds included in either GriddedTableDef or UngriddedTableDef instances. It refers to the Function instance making use of the table. |

Returns

A double precision representation of the current *bound* value.

#### 4.3.3.3 const dstoute::aString& getVarID ( ) const [inline]

If the bound is expressed in terms of a *variableDef* or *variableRef*, this function allows the *varID* attribute of the bound's variable to be determined. If the instance has not been populated, or if the bound is not expressed in terms of a *variableDef* of *variableRef*, an empty string will be returned.

Returns

The *varID* attribute string of a *variableDef* that expresses this bound functionally is passed by reference.

#### 4.3.3.4 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

An uninitialised instance of Bounds is filled with data from a particular *bounds* element within a DOM by this function. If another *bounds* element pointer is supplied to an instance that has already been initialised, the instance is re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *bounds* component node within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to evaluate bounds with a functional dependence on the instance state. It must be passed as a void pointer to avoid circularity of dependencies at build time. |

The documentation for this class was generated from the following files:

- Bounds.h
- Bounds.cpp

## 4.4 BreakpointDef Class Reference

`#include <BreakpointDef.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- BreakpointDef ()
- BreakpointDef (const DomFunctions::XmlNode &elementDefinition)
- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const dstoute::aString & getBpID () const
- const std::vector< double > & getBpVals () const
- const dstoute::aString & getDescription () const
- const dstoute::aString & getName () const
- size_t getNumberOfBpVals () const
- const dstoute::aString & getUnits () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- void setBpID (const dstoute::aString &bpID)
- void setBpVals (const std::vector< double > bpVals)
- void setDescription (const dstoute::aString &description)
- void setName (const dstoute::aString &name)
- void setUnits (const dstoute::aString &units)

#### 4.4.1 Detailed Description

A BreakpointDef instance holds in its allocated memory alphanumeric data derived from a *breakpointDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes numeric break points for gridded tables, and associated alphanumeric identification data.

A *breakpointDef* is where gridded table breakpoints are defined; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table. These are separate from function data, and thus they may be reused. The *independentVarPts* element used within some DAVE-ML *functionDefn* elements is equivalent to a *breakpointDef* element, and is also represented as a BreakpointDef within Janus.

The BreakpointDef class is only used within the janus namespace, and should only be referenced through the Janus class.

Janus exists to handle data for a modelling process. Therefore, in normal computational usage it is unnecessary (and undesirable) for a calling program to be aware of the existence of this class. However, functions do exist to access BreakpointDef contents directly, which may be useful during dataset development. A possible usage might be:

```
Janus test( xmlFileName );
const vector<BreakpointDef>& breakpointDef = test.getBreakpointDef();
for ( int i = 0 ; i < breakpointDef.size() ; i++ ) {
  cout << " bpID = " << breakpointDef.at( i ).getBpID()
       << ", units = " << breakpointDef.at( i ).getUnits() << "\n";
  cout << " values = [ " << breakpointDef.at( i ).getBpVals() << " ]\n";
}
```

### 4.4.2 Constructor & Destructor Documentation

#### 4.4.2.1 BreakpointDef ( )

The empty constructor can be used to instance the BreakpointDef class without supplying the DOM *breakpointDef* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *breakpointDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.4.2.2 BreakpointDef ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *breakpointDef* element within a DOM, instantiates the BreakpointDef class and fills it with alphanumeric data from the DOM. The string content of the *bpVals* element is converted to a double precision numeric vector within the instance.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *breakpointDef* component node within the DOM. |

### 4.4.3 Member Function Documentation

#### 4.4.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *breakpointDef* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.4.3.2 const dstoute::aString& getBpID ( ) const [inline]

This function provides access to the *bpID* attribute of a *breakpointDef*. This attribute is used for indexing breakpoints within an XML dataset. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *bpID* string is passed by reference.

#### 4.4.3.3 const std::vector< double>& getBpVals ( ) const [inline]

This function provides access to the breakpoint values within a BreakpointDef instance. The breakpoints are a vector of monotonically increasing values used as the independent terms in function based on a gridded table. The function is not generally accessed directly by users, but is employed by the Janus class in performing function evaluations.

Returns

A reference to the *bpVals* vector from this BreakpointDef instance.

#### 4.4.3.4 const dstoute::aString& getDescription ( ) const [inline]

This function provides access to the *description* child of the *breakpointDef* element represented by this BreakpointDef instance. A *breakpointDef's* optional *description* child element consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description. If no description is specified in the XML dataset, or the BreakpointDef has not been initialised from the DOM, an empty string is returned.

Returns

The *description* string is passed by reference.

#### 4.4.3.5 const dstoute::aString& getName ( ) const [inline]

This function provides access to the *name* attribute of the *breakpointDef* element represented by this BreakpointDef instance. The *name* attribute is optional. If the instance has not been initialised from a DOM, or if no *name* attribute is present, an empty string is returned.

Returns

The *name* string is passed by reference.

#### 4.4.3.6 size_t getNumberOfBpVals ( ) const [inline]

This function provides the number of breakpoint values within a BreakpointDef instance. The breakpoints are a vector of monotonically increasing values used as the independent terms in function based on a gridded table. The function is not generally accessed directly by users, but is employed by the Janus class in performing function evaluations.

Returns

The number of breakpoint values stored in the *bpVals* vector from this BreakpointDef instance.

#### 4.4.3.7 const dstoute::aString& getUnits ( ) const [inline]

This function provides access to the *units* attribute of a *breakpointDef* represented by this BreakpointDef instance. A breakpoint array's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by AD APS [7] in accordance with Systeme International d'Unites (SI) and other systems. The *units* attribute is optional. If the instance has not been initialised from a DOM, or if no *units* attribute is present, an empty string is returned.

Returns

The *units* string is passed by reference.

#### 4.4.3.8 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of BreakpointDef is filled with data from a particular *breakpointDef* element within a DOM by this function. The string content of the *bpVals* element is converted to a double precision numeric vector within the instance. If another *breakpointDef* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *breakpointDef* component node within the DOM. |

#### 4.4.3.9 void setBpID ( const dstoute::aString & *bpID* ) [inline]

This function permits the *bpID* attribute of the *breakpointDef* element to be reset for this BreakpointDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *bpID* | The breakpoint identifier *bpID* string. |

#### 4.4.3.10 void setBpVals ( const std::vector< double > *bpVals* ) [inline]

This function permits the breakpoint values vector (*bpVals*) element of the *breakpointDef* element to be reset for this BreakpointDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *bpVals* | The vector of numerical breakpoint values. |

#### 4.4.3.11 void setDescription ( const dstoute::aString & *description* ) [inline]

This function permits the *description* element of the *breakpointDef* element to be reset for this BreakpointDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *description* | The *description* string. |

#### 4.4.3.12 void setName ( const dstoute::aString & *name* ) [inline]

This function permits the *name* attribute of the *breakpointDef* element to be reset for this BreakpointDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *name* | The *name* string. |

#### 4.4.3.13 void setUnits ( const dstoute::aString & *units* ) [inline]

This function permits the *units* attribute of the *breakpointDef* element to be reset for this BreakpointDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *units* | The *units* identifier string. |

The documentation for this class was generated from the following files:

- BreakpointDef.h
- BreakpointDef.cpp

## 4.5 CheckData Class Reference

`#include <CheckData.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- CheckData ()
- CheckData (const DomFunctions::XmlNode &elementDefinition)
- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const Provenance & getProvenance () const
- StaticShotList & getStaticShot ()
- const StaticShot & getStaticShot (const size_t &index) const
- size_t getStaticShotCount () const
- const bool & hasProvenance () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- void setStaticShot (const StaticShotList staticShot)

### 4.5.1 Detailed Description

Check data is used for XML dataset content verification. A CheckData instance holds in its allocated memory alphanumeric data derived from a *checkData* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It will include static check cases, trim shots, and dynamic check case information. At present only static check cases are implemented, using *staticShot* children of the top-level *checkData* element. The functions within this class provide access to the raw check data, as well as actually performing whatever checks may be done on the dataset using the *checkData*. The provenance sub-element is now deprecated and has been moved to individual staticShots; it is allowed here for backwards compatibility.

The CheckData class is only used within the janus namespace, and should normally only be referenced through the Janus class.

Typical usage of the checking functions:

```
Janus test( xmlFileName );
CheckData checkData = test.getCheckData();
int nss = checkData.getStaticShotCount( );
cout << " Number of static shots = " << nss << endl;
for ( int j = 0 ; j < nss ; j++ ) {
  StaticShot staticShot = checkData.getStaticShot( j );
  int nInvalid = staticShot.getInvalidVariableCount();
  if ( 0 < nInvalid ) {
    for ( int k = 0 ; k < nInvalid ; k++ ) {
      string failVarID =
        staticShot.getInvalidVariable(  k );
      cout << "  Problem at varID : " << failVarID << endl;
    }
  } else {
    cout << "  No problems from static shot " << j << " ... " << endl;
  }
}
```

### 4.5.2 Constructor & Destructor Documentation

#### 4.5.2.1 CheckData ( )

The empty constructor can be used to instance the CheckData class without supplying the DOM *checkData* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *checkData* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.5.2.2 CheckData ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *checkData* element within a DOM, instantiates the CheckData class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address to a *checkData* component node within the DOM. |

### 4.5.3 Member Function Documentation

#### 4.5.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *checkData* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.5.3.2 const Provenance& getProvenance ( ) const [inline]

This function provides access to the *provenance* or *provenanceRef* element contained in a DAVE-ML *checkData* element. The element is deprecated in this location; however, access through this function is retained for compatibility with older datasets. There may be zero or one of these elements attached to the *checkData* element in a valid dataset. If the instance has not been initialised or the *checkData* element has no provenance, an empty Provenance instance is returned.

Returns

The Provenance instance is returned by reference.

See also

Provenance

#### 4.5.3.3 StaticShotList& getStaticShot ( ) [inline]

This function provides access to the *staticShot* elements referenced by a DAVE-ML *checkData* element. There may be zero, one or many *staticShot* elements within the *checkData* component of a valid XML dataset.

Returns

A list containing the StaticShot instances is returned by reference.

See also

StaticShot

#### 4.5.3.4 const StaticShot& getStaticShot ( const size_t & *index* ) const [inline]

This function provides access to a *staticShot* element referenced by a DAVE-ML *checkData* element. There may be zero, one or many *staticShot* elements within the *checkData* component of a valid XML dataset.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getStaticShotCount() - 1 ), and selects the required StaticShot instance. Attempting to access a StaticShot outside the available range will throw a standard out_of_range exception. |

Returns

The requested StaticShot instance is returned by reference.

See also

StaticShot

#### 4.5.3.5 size_t getStaticShotCount ( ) const [inline]

This function allows the number of *staticShot* elements referenced by a *checkData* element to be determined. If the CheckData instance has not been populated from a DOM, zero is returned. Because future *checkData* may include other cases than static shots, a *checkData* element without any *staticShot* components may still be valid.

Returns

An integer number, zero or more in a populated instance.

See also

StaticShot

#### 4.5.3.6 const bool& hasProvenance ( ) const [inline]

This function indicates whether the *checkData* element of a DAVE-ML dataset includes either *provenance* or *provenanceRef* children. For DAVE-ML version 2.0RC3 and subsequent releases, the use of *provenance* or *provenanceRef* at the *checkData* level is deprecated.

Returns

A boolean indication of the presence of *checkData's* provenance.

#### 4.5.3.7 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of CheckData is filled with data from a particular *checkData* element within a DOM by this function. If another *checkData* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address to a *checkData* component node within the DOM. |

#### 4.5.3.8 void setStaticShot ( const StaticShotList *staticShot* ) [inline]

This function permits the staticShot instance vector of the *checkData* element to be reset for this CheckData instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *staticShot* | The list of staticShot instances. |

The documentation for this class was generated from the following files:

- CheckData.h
- CheckData.cpp

## 4.6 CheckInputs Class Reference

`#include <CheckInputs.h>`

Inherits SignalList.

### Public Member Functions

- CheckInputs ()
- CheckInputs (const DomFunctions::XmlNode &checkInputsElement)

### 4.6.1 Detailed Description

A CheckInputs instance functions as a container for the Signal class through the use of the Signals class. It provides the functions that allow a calling StaticShot instance to access the *signal* elements that define the input values for a check case. A *checkInputs* element must contain *signal* elements that include *signalName* and *signalUnits* elements.

The CheckInputs class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot class.

Typical usage:

```
Janus test( xmlFileName );
const CheckData& checkData = test.getCheckData();
size_t nss = checkData.getStaticShotCount( );
for ( size_t j = 0 ; j < nss ; j++ ) {
  const StaticShot& staticShot = checkData.getStaticShot( j );
  const CheckInputs& checkInputs = staticShot.getCheckInputs();
  size_t ncinp = checkInputs.getSignalCount();
  cout << " staticShot[" << j << "] : " << endl
```

```
      << "      Name                      = "
      << staticShot.getName( ) << endl
      << "      Number of check inputs    = " << ncinp << endl;
  for ( size_t k = 0 ; k < ncinp ; k++ ) {
    cout << "  checkInputs[" << k << "] : " << endl
         << "    signalName              = "
         << checkInputs.getName( k ) << endl
         << "    signalUnits             = "
         << checkInputs.getUnits( k ) << endl
         << "    signalValue             = "
         << checkInputs.getValue( k ) << endl
         << endl;
  }
}
```

### 4.6.2 Constructor & Destructor Documentation

#### 4.6.2.1 CheckInputs ( ) [inline]

The empty constructor can be used to instance the CheckInputs class without supplying the DOM *checkInputs* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *checkInputs* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

setCheckInputsFromDom

#### 4.6.2.2 CheckInputs ( const DomFunctions::XmlNode & *checkInputsElement* ) [inline]

The constructor, when called with an argument pointing to a *checkInputs* element within a DOM, instantiates the CheckInputs class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *checkInputsElement* | is an address to a *checkInputs* component node within the DOM. |

The documentation for this class was generated from the following file:

- CheckInputs.h

## 4.7 CheckOutputs Class Reference

`#include <CheckOutputs.h>`

Inherits SignalList.

### Public Member Functions

- CheckOutputs ()
- CheckOutputs (const DomFunctions::XmlNode &checkOutputsElement)

### 4.7.1 Detailed Description

A CheckOutputs instance functions as a container for the Signal class through the use of the Signals class. It provides the functions that allow a calling StaticShot instance to access the *signal* elements that define the output values for a check case. A *checkOutputs* element must contain *signal* elements that include *signalName* and *signalUnits* elements.

The CheckOutputs class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot class.

Typical usage:

```
Janus test( xmlFileName );
CheckData checkData = test.getCheckData();
size_t nss = checkData.getStaticShotCount( );
for ( size_t j = 0 ; j < nss ; j++ ) {
  StaticShot staticShot = checkData.getStaticShot( j );
  CheckOutputs checkOutputs = staticShot.getCheckOutputs();
  size_t ncout = checkOutputs.getSignalCount();
  cout << " staticShot[" << j << "] : " << endl
       << "      Name                        = "
       << staticShot.getName( ) << endl
       << "      Number of check outputs   = " << ncout << endl;
  for ( size_t k = 0 ; k < ncout ; k++ ) {
    cout << "  checkOutputs[" << k << "] : " << endl
         << "    signalName                = "
         << checkOutputs.getName( k ) << endl
         << "    signalUnits               = "
         << checkOutputs.getUnits( k ) << endl
         << "    signalValue               = "
         << checkOutputs.getValue( k ) << endl
         << "          tol                 = "
         << checkOutputs.getTolerance( k ) << endl
         << endl;
  }
}
```

### 4.7.2 Constructor & Destructor Documentation

#### 4.7.2.1 CheckOutputs ( ) [inline]

The empty constructor can be used to instance the CheckOutputs class without supplying the DOM *checkOutputs* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *checkOutputs* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.7.2.2 CheckOutputs ( const DomFunctions::XmlNode & *checkOutputsElement* ) [inline]

The constructor, when called with an argument pointing to a *checkOutputs* element within a DOM, instantiates the CheckOutputs class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *checkOutputsElement* | is an address to a *checkOutputs* component node within the DOM. |

The documentation for this class was generated from the following file:

- CheckOutputs.h

## 4.8 DimensionDef Class Reference

`#include <DimensionDef.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- DimensionDef ()
- DimensionDef (const DomFunctions::XmlNode &elementDefinition)
- void exportDefinition (DomFunctions::XmlNode &documentElement, const bool &isReference=false)
- size_t getDim (const size_t &index) const
- size_t getDimCount () const
- const dstoute::aString & getDimID () const
- size_t getDimTotal () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- void setDimID (const dstoute::aString &dimID)
- void setDimRecords (const std::vector< size_t > &dimRecords)

### 4.8.1 Detailed Description

A DimensionDef instance holds in its allocated memory alphanumeric data derived from a *dimensionDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source

file. It includes descriptive, alphanumeric identification and cross-reference data.

The DimensionDef class is only used within the janus namespace, and should only be referenced through the Janus class.

### 4.8.2 Constructor & Destructor Documentation

#### 4.8.2.1 DimensionDef ( )

The empty constructor can be used to instance the DimensionDef class without supplying the DOM *dimensionDef* element from which the instance is constructed, but in this state it not useful for any class functions. It is necessary to populate the class from a DOM containing a *dimensionDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.8.2.2 DimensionDef ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *dimensionDef* element within a DOM, instantiates the DimensionDef class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an addressto a *dimensionDef* component node within the DOM. |

### 4.8.3 Member Function Documentation

#### 4.8.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement*, const bool & *isReference* = `false` )

This function is used to export the *dimensionDef* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |
| *isReference* | a boolean flag indicating the dimensionDef element should be treated as a reference. |

#### 4.8.3.2 size_t getDim ( const size_t & *index* ) const [inline]

This function returns a reference to the selected dimension record within the DimensionDef instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getDimCount() - 1 ), and selects the required dimension record. An attempt to access a non-existent *dimension* record will throw a standard out_of_range exception. |

Returns

The requested dimension record is returned by reference.

#### 4.8.3.3 size_t getDimCount ( ) const [inline]

This function returns the number of dimension records listed in a DimensionDef. If the instance has not been populated from a DOM element, zero is returned.

Returns

An unsigned integer (size_t) number, one or more in a populated instance.

#### 4.8.3.4 const dstoute::aString& getDimID ( ) const [inline]

This function provides access to the *dimID* attribute of the *dimensionDef* element represented by this DimensionDef instance. An dimensionDef's *dimID* attribute is normally a short string without whitespace, such as "matrix_3x3", which uniquely defines the dimensionDef. It is used for indexing dimension tables within an XML dataset, and provides underlying cross-references. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *dimID* string is returned by reference.

#### 4.8.3.5 size_t getDimTotal ( ) const [inline]

This function returns the combined total of the dimensions defined for the DimensionDef instance. This is the multiple of each of the elements.

Returns

An unsigned integer (size_t) number, multiplying each dimension value in a populated instance.

#### 4.8.3.6 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of DimensionDef is filled with data from a particular *dimensionDef* element within a DOM by this function. If another *dimensionDef* element pointer is supplied to an instance that has already been initialised, data corruption may occur.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address to a *dimensionDef* component node within the DOM. |

#### 4.8.3.7 void setDimID ( const dstoute::aString & *dimID* ) [inline]

This function permits the dimension identifier *dimID* of the *dimensionDef* element to be reset for this DimensionDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *dimID* | a string identifier representing the instance of the *dimensionDef*. |

#### 4.8.3.8 void setDimRecords ( const std::vector< size_t > & *dimRecords* ) [inline]

This function permits the dimension records of the *dimensionDef* element to be reset for this DimensionDef instance.

If the instance has not been initialised from a DOM then this function permits them to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *dimRecords* | a vector of unsigned integers (size_t) representing the sizes of each dimension defined for a vector or matrix array. |

The documentation for this class was generated from the following files:

- DimensionDef.h

- DimensionDef.cpp

## 4.9 DomFunctions Class Reference

`#include <DomFunctions.h>`

### 4.9.1 Detailed Description

This class contains common functions for interacting with a Document Object Model (DOM) containing data from a DAVE-ML compliant XML dataset source file.

The DomFunctions class is only used within Janus.

The documentation for this class was generated from the following file:

- DomFunctions.h

## 4.10 ExportMathML Class Reference

`#include <ExportMathML.h>`

### 4.10.1 Detailed Description

This class contains functions for exporting mathematics procedures defined using the MathML syntax to a DOM. Data detailing each MathML operation is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

The ExportMathML class is only used within the Janus.

The documentation for this class was generated from the following files:

- ExportMathML.h
- ExportMathML.cpp

## 4.11 FileHeader Class Reference

`#include <FileHeader.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- FileHeader ()
- FileHeader (const DomFunctions::XmlNode &elementDefinition)
- const AuthorList & getAuthor () const
- const Author & getAuthor (const size_t &index) const
- size_t getAuthorCount () const
- const dstoute::aString getClassification () const
- const dstoute::aString & getCreationDate () const
- const dstoute::aString getDataAssumptions () const
- const dstoute::aString & getDescription () const
- const dstoute::aString & getFileVersion () const
- const ModificationList & getModification () const
- const Modification & getModification (const size_t &index) const
- size_t getModificationCount () const
- const dstoute::aString & getName () const
- const ProvenanceList & getProvenance () const
- const Provenance & getProvenance (const size_t &index) const
- size_t getProvenanceCount () const
- const ReferenceList & getReference () const
- const Reference & getReference (const size_t &index) const
- size_t getReferenceCount () const
- const dstoute::aString getTag () const
- const dstoute::aString getType () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)

#### 4.11.1 Detailed Description

A FileHeader instance holds in its allocated memory alphanumeric data derived from the *fileHeader* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. There is always one FileHeader instance for each Janus instance. It requires at least one author, a creation date and a version indicator; optional content are description, references and modification records. The class also provides the functions that allow a calling Janus instance to access these data elements.

The FileHeader class is only used within the janus namespace, and should only be referenced indirectly through the Janus class.

A typical usage is:

```
Janus test( xmlFileName );
FileHeader header = test.getFileHeader();
int nAuthors = header.getAuthorCount( );
cout << "Number of authors : " << nAuthors << "\n\n";
for ( int i = 0 ; i < nAuthors ; i++ ) {
  Author author = header.getAuthor( i );
  cout << " Author " << i << " : Name                  : "
       << author.getName( ) << "\n"
       << "          Organisation            : "
       << author.getOrg( ) << "\n"
}
cout << " File creation date               : "
     << header.getCreationDate() << "\n"
     << " File version                     : "
     << header.getFileVersion() << "\n"
     << " File description                 : "
     << header.getDescription() << "\n"
     << " Number of reference records      : "
     << header.getReferenceCount() << "\n"
     << " Number of modification records   : "
     << header.getModificationCount() << "\n";
```

### 4.11.2 Constructor & Destructor Documentation

#### 4.11.2.1 FileHeader ( )

The empty constructor can be used to instance the FileHeader class without supplying the DOM *fileHeader* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *fileHeader* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.11.2.2 FileHeader ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *fileHeader* element within a DOM, instantiates the FileHeader class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address to the Level 1 element within a DOM that is tagged as a *fileHeader*. There should always be one such element. |

### 4.11.3 Member Function Documentation

#### 4.11.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *FileHeader* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address pointer to the parent DOM node/element. |

#### 4.11.3.2 const AuthorList& getAuthor ( ) const [inline]

This function returns a reference to the list of authors defined within the FileHeader instance.

Returns

The list authors is returned by reference.

See also

Author

#### 4.11.3.3 const Author& getAuthor ( const size_t & *index* ) const [inline]

This function returns a reference to the selected Author instance within the FileHeader instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getAuthorCount() - 1 ), and selects the required Author instance. An attempt to access a non-existent *author* will throw a standard out_of_range exception. |

Returns

The requested Author instance is returned by reference.

See also

Author

#### 4.11.3.4 size_t getAuthorCount ( ) const [inline]

This function returns the number of primary authors listed in a FileHeader. If the instance has not been populated from a DOM element, zero is returned.

Returns

An integer number, one or more in a populated instance.

See also

Author

#### 4.11.3.5 const dstoute::aString getClassification ( ) const [inline]

The *classification* element is an optional document identifier defining the security classification for the information stored with the XML dataset. This function returns the *classification* element of the referenced file header, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *classification* string is returned by reference.

#### 4.11.3.6 const dstoute::aString& getCreationDate ( ) const [inline]

This function returns the *creationDate* element of the *fileHeader* element (*fileCreationDate* is a deprecated alternative). The format of the dataset string is determined by the XML dataset builder, but DAVE-ML recommends the ISO 8601 form ("2004-01-02" to refer to 2 January 2004). If the FileHeader has not been populated from a DOM element , the function returns an empty string.

Returns

The *creationDate* string is returned by reference.

#### 4.11.3.7 const dstoute::aString getDataAssumptions ( ) const [inline]

The *dataAssumptions* element is an optional identifier documenting assumptions associated with the information stored with the XML dataset. This function returns the *dataAssumptions* element of the referenced file header, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *dataAssumptions* string is returned by reference.

#### 4.11.3.8 const dstoute::aString& getDescription ( ) const [inline]

This function returns the *description* from a *fileHeader* element, if one has been supplied in the XML dataset. The description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description string. Since description of a file is optional, the returned string may be blank.

Returns

The *description* string is returned by reference.

#### 4.11.3.9 const dstoute::aString& getFileVersion ( ) const [inline]

The *fileVersion* element is an optional document identifier for a *fileHeader*. The format of the version string is determined by the XML dataset builder. This function returns the *fileVersion* element of the referenced file header, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *fileVersion* string is returned by reference.

#### 4.11.3.10 const ModificationList& getModification ( ) const [inline]

This function provides access to the *modificationRecord* elements contained in a DAVE-ML *fileHeader* element, through the Modification class structure.

Returns

The list of modification records is returned by reference.

See also

Modification

#### 4.11.3.11 const Modification& getModification ( const size_t & *index* ) const [inline]

This function provides access to the *modificationRecord* elements contained in a DAVE-ML *fileHeader* element, through the Modification class structure.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to ( getModificationCount() - 1 ), and selects the required *modificationRecord*. An attempt to access a non-existent *modificationRecord* will throw a standard out_of_range exception. |

Returns

The requested Modification instance is returned by reference.

See also

Modification

#### 4.11.3.12 size_t getModificationCount ( ) const [inline]

This function returns the number of *modificationRecord* records at the top level of the *fileHeader* component of the XML dataset. A *fileHeader* can include no, one or multiple *modificationRecord* components. If the FileHeader has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

See also

Modification

#### 4.11.3.13 const dstoute::aString& getName ( ) const [inline]

This function returns the optional *name* attribute of the FileHeader instance, if one has been supplied in the XML dataset. If not, or if the instance has not been initialised from a DOM, it returns an empty string.

Returns

The *name* string is passed by reference.

#### 4.11.3.14 const ProvenanceList& getProvenance ( ) const [inline]

This function provides access to the *provenance* elements contained in a DAVE-ML *fileHeader* element, through the Provenance class structure.

Returns

The list of provenence records is returned by reference.

See also

Provenance

#### 4.11.3.15 const Provenance& getProvenance ( const size_t & *index* ) const [inline]

This function provides access to the *provenance* elements contained in a DAVE-ML *fileHeader* element, through the Provenance class structure.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to ( getProvenanceCount() - 1 ), and selects the required *provenance* record. An attempt to access a non-existent *provenance* will throw a standard out_of_range exception. |

Returns

The requested Provenence instance is returned by reference.

See also

Provenance

#### 4.11.3.16 size_t getProvenanceCount ( ) const [inline]

This function returns the number of *provenance* elements contained in a DAVE-ML *fileHeader* element. It does NOT include provenance elements contained in other elements of the dataset. There may be zero or more of these elements in a valid file header.

Returns

The integer count of provenance elements contained in a *fileHeader* element is returned. Possible values are zero or more.

See also

Provenance

#### 4.11.3.17 const ReferenceList& getReference ( ) const [inline]

This function provides access to the *reference* records contained in the XML dataset file header, through the Reference class structure.

Returns

The list of references is returned by reference.

See also

Reference

#### 4.11.3.18 const Reference& getReference ( const size_t & *index* ) const [inline]

This function provides access to the *reference* records contained in the XML dataset file header, through the Reference class structure.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to ( getReferenceCount() - 1 ), and selects the required *reference* record. An attempt to access a non-existent *reference* will throw a standard out_of_range exception. |

Returns

The requested Reference instance is returned by reference.

See also

Reference

#### 4.11.3.19 size_t getReferenceCount ( ) const [inline]

This function returns the number of *reference* elements listed in a *fileHeader* element. A *fileHeader* can include no, one or multiple *reference* components. If the FileHeader has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

See also

Reference

#### 4.11.3.20 const dstoute::aString getTag ( ) const [inline]

The *tag* element is an optional identifier that is used to identify the several DAVE-ML compliant XML dataset source files as being part of the same version of an aircraft model. This is similar to a *tag* used in source code version control.

This function returns the *tag* element of the referenced file header, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *tag* string is returned by reference.

#### 4.11.3.21 const dstoute::aString getType ( ) const [inline]

The *type* element is an optional parameter that is used to identify different types of DAVE-ML compliant XML dataset source files by aircraft type; for example, fixed wing verses rotary wing.

This function returns the *type* element of the referenced file header, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *type* string is returned by reference.

#### 4.11.3.22 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of FileHeader is filled with data from the *fileHeader* element within a DOM by this function. If a *fileHeader* element pointer is supplied to an instance that has

already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address to the Level 1 element within a DOM that is tagged as a *fileHeader*. There should always be one such element. |

The documentation for this class was generated from the following files:

- FileHeader.h
- FileHeader.cpp

## 4.12 Function Class Reference

`#include <Function.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- Function ()
- Function (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- const std::vector< double > & getData () const
- const dstoute::aString & getDefnName () const
- const size_t & getDependentDataColumnNumber () const
- int getDependentVarRef () const
- const dstoute::aString & getDescription () const
- size_t getIndependentVarCount () const
- const InDependentVarDefList & getInDependentVarDef () const
- ExtrapolateMethod getIndependentVarExtrapolate (const size_t &index) const
- InterpolateMethod getIndependentVarInterpolate (const size_t &index) const
- const double & getIndependentVarMax (const size_t &index) const
- const double & getIndependentVarMin (const size_t &index) const
- int getIndependentVarRef (const size_t &index) const
- const dstoute::aString & getName () const
- const Provenance & getProvenance () const
- int getTableRef () const
- const ElementDefinitionEnum & getTableType () const

- const bool & hasProvenance () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- const bool & isAllInterpolationLinear () const

### 4.12.1 Detailed Description

A Function instance holds in its allocated memory alphanumeric data derived from a *function* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Each function has optional description, optional provenance, and either a simple input/output values or references to more complete (possible multiple) input, output, and function data elements.

The Function class is only used within the janus namespace, and should only be referenced through the Janus class.

Where a *function* is defined directly using *dependentVarPts* and *independentVarPts*, these are converted during initialisation to externally-defined gridded tables and breakpoints respectively. Any data tables defined within the *functionDefn* are also converted to external tables. Because of these processes, a Function instance never contains primary data, only references to external tables, breakpoints and variables. However, because it is possible (but heavily discouraged) to apply output scale factors to tabulated data with Janus, a copy of the relevant external table with current scale factors applied is maintained within each Janus Function instance.

Janus exists to abstract data form and handling from a modelling process. Therefore, in normal computational usage, it is unnecessary and undesirable for a calling program to even be aware of the existence of this class. However, functions do exist to access Function contents directly, which may be useful during dataset development. A possible usage might be:

```
Janus test( xmlFileName );
const vector<Function>& function = test.getFunction();
for ( int i = 0 ; i < function.size() ; i++ ) {
  cout << " Function " << i << " :\n"
       << "   name        = " << function.at( i ).getName() << "\n"
       << "   description = " << function.at( i ).getDescription() << "\n";
  cout << "   Number of independent variables = "
       << function.at( i ).getIndependentVarCount() << "\n";
  for ( int j = 0 ; j < function.at( i ).getIndependentVarCount() ; j++ ) {
    cout << "   Input variable " << j << " varID = "
         << test.getVariableDef().
              at( function.at( i ).getIndependentVarRef( j ) ).getVarID()
         << "\n";
  }
}
```

### 4.12.2 Constructor & Destructor Documentation

#### 4.12.2.1 Function ( )

The empty constructor can be used to instance the Function class without supplying the DOM *function* element from which the instance is constructed, but in this state is not useful for any

class functions. It is necessary to populate the class from a DOM containing a *function* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.12.2.2 Function ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

The constructor, when called with an argument pointing to a *function* element within a DOM, instantiates the Function class and fills it with alphanumeric data from the DOM. String-based cross-references as implemented in the XML dataset are converted to index-based cross-references to improve computational performance.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *function* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. It must be passed as a void pointer to avoid circularity of dependencies at build time. |

### 4.12.3 Member Function Documentation

#### 4.12.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *Function* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.12.3.2 const vector< double > & getData ( ) const

This function returns the tabular data for the table associated with a Function instance. For a gridded table this represents all the data as a continuous sequence, while for an ungridded table this represents the dependent data column of the table.

Returns

The data for the table is returned as a reference to a vector of double numeric values.

#### 4.12.3.3 const dstoute::aString& getDefnName ( ) const [inline]

This function provides access to the optional *name* attribute of the *functionDefn* that is a child of a *function*. If the function definition has no name attribute or has not been initialised from a DOM, an empty string is returned.

Returns

The *functionDefn name* string is returned by reference.

#### 4.12.3.4 const size_t& getDependentDataColumnNumber ( ) const [inline]

This function returns the column number associated with the dependent data of an ungridded table, that has been defined for the *functionDefn* instance using an ungridded table reference. This parameter may be non-zero if the ungridded table has multiple dependent data columns.

Returns

The column index (size_t) of the particular dependent data parameter within the the list of dependent data of an ungridded table associated with the *functionDefn* instance.

#### 4.12.3.5 int getDependentVarRef ( ) const

Each Function instance involves one dependent variable and one or more independent variables. Within the Function, the dependent variable is referenced by an index into the vector of VariableDef instances within the encompassing Janus instance.

Returns

An integer index to the dependent variable of the referenced Function within the encompassing Janus instance.

#### 4.12.3.6 const dstoute::aString& getDescription ( ) const [inline]

This function provides access to the optional *description* of the *function* element represented by this Function instance. A *function's description* child element consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description. If no description is specified in the XML dataset, or the Function has not been initialised from the DOM, an empty string is returned.

Returns

The *description* string is returned by reference.

#### 4.12.3.7 size_t getIndependentVarCount ( ) const [inline]

This function returns the number of *independentVarRef* or *independentVarPts* elements used in a *function*. If the instance has not been populated from a DOM, zero is returned. In all other cases, there must be one or more independent variables.

Returns

An integer number, one or more in a populated instance.

#### 4.12.3.8 const InDependentVarDefList& getInDependentVarDef ( ) const [inline]

This function provides access to the independent variable definitions instances that have been defined for the function instance. An empty vector will be returned if the Function instance has not been populated from a DOM. In all other cases, the vector will contain at least one independent variable instance.

Returns

A list of independent variable definitions instances.

See also

InDependentVarDef

#### 4.12.3.9 ExtrapolateMethod getIndependentVarExtrapolate ( const size_t & *index* ) const [inline]

The *extrapolate* attribute of an independent variable describes any allowable extrapolation in the independent variable's degree of freedom beyond a function's tabulated data range. The *extrapolate* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then the enum representing its value within the Function instance defaults to NEITHER.

When the returned value is NEITHER, MINEX, or MAXEX, constraining the independent variable at both ends, the maximum, or the minimum respectively, the constrained independent variable value used for the *function* evaluation will be the more limiting of:

| Min Constraints | Max Constraints |
|---|---|
| lowest independentVarPts *or* lowest breakpoint | highest independentVarPts *or* highest breakpoint |
| *min* attribute | *max* attribute |

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getIndependentVarCount() - 1), and selects the required independent variable. Attempting to access an independent variable outside the available range will throw a standard out_of_range exception. |

Returns

An Extrapolation enum containing the extrapolation constraint on the independent variable selected, determined as tabulated above.

See also

getIndependentVarExtrapolateFlag

#### 4.12.3.10 InterpolateMethod getIndependentVarInterpolate ( const size_t & *index* ) const [inline]

The *interpolate* attribute of an independent variable describes the form of interpolation applicable to that variable's degree of freedom within the range of the tabulated dataset. The *extrapolate* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then the enum representing its value within the Function instance defaults to LINEAR.

Parameters

| | |
|---|---|
| *index* | is an integer in the range from 0 through ( getIndependentVarCount() - 1 ), and selects the required independent variable. Attempting to access an independent variable outside the available range will throw a standard out_of_range exception. |

Returns

An Interpolation enum containing the interpolation technique applicable to the independent variable selected.

#### 4.12.3.11 const double& getIndependentVarMax ( const size_t & *index* ) const [inline]

The *max* attribute of a *function's* independent variable describes an upper limit for the independent variable's value during computation of the output. This function makes that limit available to the calling program. The *max* attribute is optional for all degrees of freedom for a *function*, and if it is not set for any particular degree of freedom then the data may be extrapolated upwards without limit in that degree of freedom unless the *extrapolate* attribute indicates otherwise.

Note that a variable may be an independent input for multiple *functions*, and may have a different *max* in each such *function*. Also, the *max* need not coincide with the maximum *independentVarPts* or breakpoint ( $x_{\max}$ ) for its degree of freedom.

The value ( $x$) of an independent variable used for evaluation of a function is never greater than *max*, no matter what the input value is or what other constraints are applied. Within this constraint, the *max* attribute interacts with both the highest available value for its variable and the variable's *extrapolate* attribute (see getIndependentVarExtrapolate()), to define the input value used in a function evaluation. Whenever a constraint is activated during a function evaluation, the extrapolation flag for that degree of freedom is changed, and can

be checked by the calling program (see getIndependentVarExtrapolateFlag()). The various possible combinations of constraining attributes and data limits are:

| *extrapolate* attribute | $x$ relative values | $x$ used in computation | extrapolation flag after computation |
|---|---|---|---|
| any value | $x < max < x_{\max}$ | $x$ | NEITHER |
| | $x < x_{\max} < max$ | $x$ | NEITHER |
| neither / min | $max < x < x_{\max}$ | $max$ | MAXEX |
| | $max < x_{\max} < x$ | $max$ | MAXEX |
| | $x_{\max} < x < max$ | $x_{\max}$ | XMAX |
| | $x_{\max} < max < x$ | $x_{\max}$ | XMAX |
| max / both | $max < x < x_{\max}$ | $max$ | MAXEX |
| | $max < x_{\max} < x$ | $max$ | MAXEX |
| | $x_{\max} < x < max$ | $x$ | XMAX |
| | $x_{\max} < max < x$ | $max$ | MAXEX |

If a *max* limit has not been specified for a variable, this function returns DBL_MAX.

Parameters

| | |
|---|---|
| *index* | is an integer in the range from 0 through ( getIndependentVarCount() - 1 ), and selects the required independent variable. Attempting to access an independent variable outside the available range will throw a standard out_of_range exception. |

Returns

A double precision value for the selected variable's maximum limit.

#### 4.12.3.12 const double& getIndependentVarMin ( const size_t & *index* ) const [inline]

The *min* attribute of a *function's* independent variable describes a lower limit for the independent variable's value during computation of the output. This function makes that limit available to the calling program. The *min* attribute is optional for all degrees of freedom for a *function*, and if it is not set for any particular degree of freedom then the data may be extrapolated downwards without limit in that degree of freedom unless the *extrapolate* attribute indicates otherwise.

Note that a variable may be an independent input for multiple *functions*, and may have a different *min* in each such *function*. Also, the *min* need not coincide with the minimum *independentVarPts* or breakpoint ( $x_{\min}$ ) for its degree of freedom.

The value ( $x$) of an independent variable used for evaluation of a function is never less than *min*, no matter what the input value is or what other constraints are applied. Within this constraint, the *min* attribute interacts with both the lowest available value for its variable and the variable's *extrapolate* attribute (see getIndependentVarExtrapolate()), to define the input value used in a function evaluation. Whenever a constraint is activated during a function evaluation, the extrapolation flag for that degree of freedom is changed, and can be checked

by the calling program (see getIndependentVarExtrapolateFlag()). The various possible combinations of constraining attributes and data limits are:

| *extrapolate* attribute | $x$ relative values | $x$ used in computation | extrapolation flag after computation |
|---|---|---|---|
| any value | $x_{\min} < min < x$ | $x$ | NEITHER |
| | $min < x_{\min} < x$ | $x$ | NEITHER |
| neither / max | $x_{\min} < x < min$ | $min$ | MINEX |
| | $x < x_{\min} < min$ | $min$ | MINEX |
| | $min < x < x_{\min}$ | $x_{\min}$ | XMIN |
| | $x < min < x_{\min}$ | $x_{\min}$ | XMIN |
| min / both | $x_{\min} < x < min$ | $min$ | MINEX |
| | $x < x_{\min} < min$ | $min$ | MINEX |
| | $min < x < x_{\min}$ | $x$ | XMIN |
| | $x < min < x_{\min}$ | $min$ | MINEX |

If a *min* limit has not been specified for a variable, this function returns -DBL_MAX.

Parameters

| | |
|---|---|
| *index* | is an integer in the range from 0 through ( getIndependentVarCount() - 1 ), and selects the required independent variable. Attempting to access an independent variable outside the available range will throw a standard out_of_range exception. |

Returns

A double precision value for the selected variable's minimum limit.

#### 4.12.3.13 int getIndependentVarRef ( const size_t & *index* ) const

This function provides access to the *independentVarRef* or *independentVarPts* elements used in a *function*. Within the Function, these variables are referenced by indices into the vector of VariableDef instances within the encompassing Janus instance.

Parameters

| | |
|---|---|
| *index* | is an integer in the range from 0 through ( getIndependentVarCount() - 1 ), and selects the required independent variable. Attempting to access an independent variable outside the available range will throw a standard out_of_range exception. |

Returns

An integer index to the selected independent variable of the referenced Function within the encompassing Janus instance.

#### 4.12.3.14 const dstoute::aString& getName ( ) const [inline]

This function provides access to *name* attribute of a *function*. If the function has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is returned by reference.

#### 4.12.3.15 const Provenance& getProvenance ( ) const [inline]

This function provides access to the Provenance instance associated with a Function instance. There may be zero or one of these elements for each function in a valid dataset, defined either directly or by reference.

Returns

The Provenance instance is returned by reference.

See also

Provenance

#### 4.12.3.16 int getTableRef ( ) const

This function provides access to a table forming the basis for evaluation of a function. Within the Function, the table is referenced by an index into the vector of table definition instances encompassed within the Janus instance.

Returns

An integer index to the table used by the referenced Function encompassed within the Janus instance.

#### 4.12.3.17 const ElementDefinitionEnum& getTableType ( ) const [inline]

This function returns the type of the table that is associated with the Function instance, being either a *gridded* table or an *ungridded* table. This functionality is used internally when instantiating a Janus instance and returning data from a Function instance.

Returns

An enumeration defining the table type associated with the Function instance.

#### 4.12.3.18 const bool& hasProvenance ( ) const [inline]

This function indicates whether a *function* element of a DAVE-ML dataset includes either *provenance* or *provenanceRef*.

Returns

A boolean variable, 'true' if the *function* includes a provenance, defined either directly or by reference.

See also

Provenance

#### 4.12.3.19 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

This function populates a Function instance based on the corresponding *function* element of the DOM, defines the cross-references from the Function to variables and breakpoints, and sets up arrays which will later be used in run-time function evaluation. If another *functionElement* pointer is supplied to an instance that has already been initialised, data corruption will occur and the entire Janus instance will become unusable.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *function* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. It must be passed as a void pointer to avoid circularity of dependencies at build time. |

#### 4.12.3.20 const bool& isAllInterpolationLinear ( ) const [inline]

This function indicates whether the referenced Function instance requires linear or lower order interpolation in all independent variables. It is a convenience function, saving checking and speeding up the interpolation process in the most common case.

Returns

A boolean variable, "true" if linear interpolation is required in all degrees of freedom.

The documentation for this class was generated from the following files:

- Function.h
- Function.cpp

## 4.13 FunctionDefn Class Reference

`#include <FunctionDefn.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- FunctionDefn ()
- FunctionDefn (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- const size_t & getDependentDataColumnNumber () const
- const dstoute::aString & getName () const
- int getTableIndex () const
- const dstoute::aString & getTableReference () const
- const ElementDefinitionEnum & getTableType () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- void setTableIndex (int tableIndex)
- void setTableReference (const dstoute::aString &xReference)
- void setTableType (const ElementDefinitionEnum &tableType)

### 4.13.1 Detailed Description

A FunctionDefn instance holds in its allocated memory alphanumeric data derived from a *functionDefn* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Each function stores function data elements.

The FunctionDefn class is only used within the janus namespace, and should only be referenced through the Janus class.

Janus exists to abstract data form and handling from a modelling process. Therefore, in normal computational usage, it is unnecessary and undesirable for a calling program to even be aware of the existence of this class. However, functions do exist to access FunctionDefn contents directly, which may be useful during dataset development. A possible usage might be:

```
Janus test( xmlFileName );
const vector<Function>& function = test.getFunction();
for ( int i = 0 ; i < function.size() ; i++ ) {
  cout << " Function " << i << " :\n"
       << "   name        = " << function.at( i ).getName() << "\n"
       << "   description = " << function.at( i ).getDescription() << "\n";
  cout << "   Number of independent variables = "
       << function.at( i ).getIndependentVarCount() << "\n";
  for ( int j = 0 ; j < function.at( i ).getIndependentVarCount() ; j++ ) {
    cout << "   Input variable " << j << " varID = "
         << test.getVariableDef().
              at( function.at( i ).getIndependentVarRef( j ) ).getVarID()
         << "\n";
  }
}
```

### 4.13.2 Constructor & Destructor Documentation

#### 4.13.2.1 FunctionDefn ( )

The empty constructor can be used to instance the FunctionDefn class without supplying the DOM *function* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *functionDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.13.2.2 FunctionDefn ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

The constructor, when called with an argument pointing to a *functionDefn* element within a DOM, instantiates the FunctionDefn class and fills it with alphanumeric data from the DOM. String-based cross-references as implemented in the XML dataset are converted to index-based cross-references to improve computational performance.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *functionDefn* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

### 4.13.3 Member Function Documentation

#### 4.13.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *functionDefn* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.13.3.2 const size_t& getDependentDataColumnNumber ( ) const [inline]

This function returns the column number associated with the dependent data of an ungridded table, that has been defined for the *functionDefn* instance using an ungridded table reference. This parameter may be non-zero if the ungridded table has multiple dependent data columns.

Returns

The column index (size_t) of the particular dependent data parameter within the the list of dependent data of an ungridded table associated with the *functionDefn* instance.

#### 4.13.3.3 const dstoute::aString& getName ( ) const [inline]

This function provides access to *name* attribute of a *functionDefn*. If the functionDefn has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is returned by reference.

#### 4.13.3.4 int getTableIndex ( ) const [inline]

This function returns an index to the table forming the basis of the *functionDefn* instance. This index identifies the particular table within the list of tables encoded within the Janus instance. This function is used internally within Janus when evaluating a Function instance.

Returns

An integer index to the table used to evaluate a Function instance.

#### 4.13.3.5 const dstoute::aString& getTableReference ( ) const [inline]

This function returns a reference identifier for the table forming the basis of the *functionDefn* instance. The reference is the table identifier, being either the gtID attribute of a gridded table, or the utID attribute of an ungridded table. This reference is used internally within Janus to identify the particular table within the list of tables encoded within the Janus instance.

Returns

A string representing the tables identifier.

#### 4.13.3.6 const ElementDefinitionEnum& getTableType ( ) const [inline]

This function returns an enumeration defining the type of data table associated with the *functionDefn* instance. The enumeration will differentiate the data table as being either gridded or ungridded.

Returns

An enumeration defining the form of the data table.

#### 4.13.3.7 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

This functionDefn populates a FunctionDefn instance based on the corresponding *functionDefn* element of the DOM, defines the cross-references from the Function to variables and break-points, and sets up arrays that will later be used in run-time function evaluation. If another *functionElement* pointer is supplied to an instance that has already been initialised, data corruption will occur and the entire Janus instance will become unusable.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *function* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

#### 4.13.3.8 void setTableIndex ( int *tableIndex* ) [inline]

This function is used to set the index identifying the table from within the list of tables encoded in the Janus instance. This function is used internally within Janus while instantiating a DAVE-ML compliant XML dataset source file.

Parameters

| | |
|---|---|
| *tableIndex* | is the integer index from the list of tables encoded wthin the Janus instance of the table associated with the *functionDefn* instance |

#### 4.13.3.9 void setTableReference ( const dstoute::aString & *xReference* ) [inline]

This function is used to set the reference identifier of the table forming the basis of the *functionDefn* instance. The reference is the table identifier, being either the gtID attribute if a gridded table, or the utID attribute of an ungridded table. This function is used internally within Janus while instantiating a DAVE-ML compliant XML dataset source file.

Parameters

| | |
|---|---|
| *xReference* | is the reference identifier of the table associated with the *functionDefn* instance. |

#### 4.13.3.10 void setTableType ( const ElementDefinitionEnum & *tableType* ) [inline]

This function is used to set the form of the data table associated with the *functionDefn* instance using a enumeration. This function is used internally within Janus while instantiating a DAVE-ML compliant XML dataset source file.

Parameters

| | |
|---|---|
| *tableType* | is a enumeration defining the form of the data table associated with the *functionDefn* instance. |

The documentation for this class was generated from the following files:

- FunctionDefn.h
- FunctionDefn.cpp

## 4.14 GriddedTableDef Class Reference

`#include <GriddedTableDef.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const std::vector< size_t > & getBreakpointRef () const
- const std::vector< double > & getData () const
- const dstoute::aString & getDescription () const
- const dstoute::aString & getGtID () const
- const dstoute::aString & getName () const
- const Provenance & getProvenance () const
- const dstoute::aStringList & getStringData () const
- Uncertainty & getUncertainty ()
- const dstoute::aString & getUnits () const
- GriddedTableDef ()
- GriddedTableDef (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- const bool & hasProvenance () const
- const bool & hasUncertainty () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- bool isDataTableEmpty () const

- bool isStringDataTableEmpty () const
- void setBreakpointRefs (const std::vector< size_t > breakpointRef)
- void setDescription (const dstoute::aString &description)
- void setGtID (const dstoute::aString &gtID)
- void setJanus (Janus *janus)
- void setName (const dstoute::aString &name)
- void setTableData (const std::vector< double > dataPoints)
- void setUnits (const dstoute::aString &units)

### 4.14.1 Detailed Description

A GriddedTableDef instance holds in its allocated memory alphanumeric data derived from a *griddedTableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes points arranged in an orthogonal, multi-dimensional array, where the independent variable ranges are defined by separate breakpoint vectors. The table data point values are specified as comma-separated values in floating-point notation (0.93638E-06) in a single long sequence as if the table had been unravelled with the last-specified dimension changing most rapidly. Gridded tables in DAVE-ML and Janus are stored in row-major order, as in C/C++ (Fortran, Matlab and Octave use column-major order). Line breaks and comments in the XML are ignored. Associated alphanumeric identification and cross-reference data are also included in the instance.

The GriddedTableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

Janus exists to abstract data form and handling from a modelling process. Therefore, in normal computational usage, it is unnecessary and undesirable for a calling program to even be aware of the existence of this class. However, functions do exist to access GriddedTableDef contents directly, which may be useful during dataset development. A possible usage might be:

```
Janus test( xmlFileName );
const vector<GriddedTableDef>& griddedTableDef = test.getGriddedTableDef();
for ( int i = 0 ; i < griddedTableDef.size() ; i++ ) {
  cout << " Gridded table " << i << " :\n"
       << "   name        = " << griddedTableDef.at( i ).getName() << "\n"
       << "   gtID        = " << griddedTableDef.at( i ).getGtID() << "\n"
       << "   units       = " << griddedTableDef.at( i ).getUnits() << "\n"
       << "   description = " << griddedTableDef.at( i ).getDescription()
       << "\n";
  const vector<int>& breakpointRef =
    griddedTableDef.at( i ).getBreakpointRef();
  for ( int j = 0 ; j < breakpointRef.size() ; j++ ) {
    cout << " Breakpoint " << j << " bpID = "
         << test.getBreakpointDef().at( breakpointRef.at( j ) ).getBpID()
         << "\n";
  }
  const vector<double>& dataTable = griddedTableDef.at( i ).getData();
  if ( 0 == dataTable.size() ) {
    const vector<string>& stringDataTable =
      griddedTableDef.at( i ).getStringData();
    for ( int j = 0 ; j < stringDataTable.size() ; j++ ) {
```

```
      cout << stringDataTable.at( j ) << "\n";
    }
  }
}
```

### 4.14.2 Constructor & Destructor Documentation

#### 4.14.2.1 GriddedTableDef ( )

The empty constructor can be used to instance the GriddedTableDef class without supplying the DOM *griddedTableDef* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *griddedTableDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.14.2.2 GriddedTableDef ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

The constructor, when called with an argument pointing to a *griddedTableDef* element within a DOM, instantiates the GriddedTableDef class and fills it with alphanumeric data from the DOM. String-based numeric data are converted to double-precision linear vectors.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *griddedTableDef* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

### 4.14.3 Member Function Documentation

#### 4.14.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *GriddedTableDef* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address pointer to the parent DOM node/element. |

#### 4.14.3.2 const std::vector< size_t>& getBreakpointRef ( ) const [inline]

This function provides access to the vector of breakpoint indices associated with a GriddedTableDef instance. The vector contains one integer for each relevant *breakpointDef*, representing the position of the relevant *breakpointDef* in the vector of BreakpointDefs within a Janus instance.

Returns

The vector of breakpoint indices is returned by reference.

See also

BreakpointDef

#### 4.14.3.3 const std::vector< double>& getData ( ) const [inline]

This function provides access to a vector of numeric data stored in a GriddedTableDef instance. This vector contains the double precision variables in the same sequence as they were presented in the *dataTable* of the corresponding XML dataset.

Returns

The vector of double-precision numeric content of the GriddedTableDef instance is returned by reference.

#### 4.14.3.4 const dstoute::aString& getDescription ( ) const [inline]

This function provides access to the *description* child of the *griddedTableDef* element represented by this GriddedTableDef instance. A *griddedTableDef's* optional *description* child element consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description. If no description is specified in the XML dataset, or the GriddedTableDef has not been initialised from the DOM, an empty string is returned.

Returns

The *description* string is returned by reference.

#### 4.14.3.5 const dstoute::aString& getGtID ( ) const [inline]

This function provides access to the *gtID* attribute of a *griddedTableDef*. This attribute is used for indexing gridded tables within an XML dataset. Where a *griddedTableDef* within the DOM does not contain a *gtID* attribute, or where a *griddedTable* or *dependentVarPoints* have been placed in the GriddedTableDef structure, a *gtID* string is generated and inserted in the DOM at initialisation time. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *gtID* string is returned by reference.

#### 4.14.3.6 const dstoute::aString& getName ( ) const [inline]

This function provides access to the *name* attribute of a *griddedTableDef*. The *name* attribute is optional. If the gridded table has no name attribute or has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is returned by reference.

#### 4.14.3.7 const Provenance& getProvenance ( ) const [inline]

This function provides access to the Provenance instance associated with a GriddedTableDef instance. There may be zero or one of these elements for each gridded table in a valid dataset, defined either directly or by reference.

Returns

The Provenance instance is returned by reference.

See also

Provenance

#### 4.14.3.8 const dstoute::aStringList& getStringData ( ) const [inline]

This function provides access to a vector of alphanumeric data stored in a GriddedTableDef instance. This vector contains the data strings in the same sequence as they were presented in the *dataTable* of the corresponding XML dataset.

Returns

The list of strings containing alphanumeric content of the GriddedTableDef instance is returned by reference.

#### 4.14.3.9 Uncertainty& getUncertainty ( ) [inline]

This function provides access to the Uncertainty instance associated with a GriddedTableDef instance. There may be zero or one *uncertainty* element for each *griddedTableDef* in a valid dataset. For *griddedTableDefs* without *uncertainty*, for *griddedTables*, and for *dependentVarPts*, the corresponding GriddedTableDef instance includes an empty Uncertainty instance.

Returns

The Uncertainty instance is returned by reference.

See also

Uncertainty

#### 4.14.3.10 const dstoute::aString& getUnits ( ) const [inline]

This function provides access to the *units* attribute of the *griddedTableDef* represented by this GriddedTableDef instance. A gridded table's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by AD APS [7] in accordance with SI and other systems. If the instance has not been initialised from a DOM, or if no *units* attribute is present, an empty string is returned.

Returns

The *units* string is returned by reference.

#### 4.14.3.11 const bool& hasProvenance ( ) const [inline]

This function indicates whether a *griddedTableDef* element of a DAVE-ML dataset includes either *provenance* or *provenanceRef* element.

Returns

A boolean variable, 'true' if the *griddedTableDef* includes a provenance, defined either directly or by reference.

See also

Provenance

#### 4.14.3.12 const bool& hasUncertainty ( ) const [inline]

This function indicates whether a *griddedTableDef* element of a DAVE-ML dataset includes an *uncertainty* child element. A variable described by a *griddedTableDef* without an *uncertainty* element may still have uncertainty, if it is dependent on other variables or tables with defined uncertainty.

Returns

A boolean variable, 'true' if a griddedTableDef definition includes an *uncertainty* child element.

See also

Uncertainty

#### 4.14.3.13 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

An uninitialised instance of GriddedTableDef is filled with data from a particular *griddedTableDef* element within a DOM by this function. If another *griddedTableDef* element pointer is supplied to an instance that has already been initialised, data corruption will occur and the entire

Janus instance will become unusable. This function can also be used with the deprecated *griddedTable* element. For backwards compatibility, Janus converts a *griddedTable* to the equivalent *griddedTableDef* within this function. Where a *griddedTableDef* or *griddedTable* lacks a *gtID* attribute, this function will generate a random *gtID* string for indexing within the Janus class.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *griddedTableDef* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

#### 4.14.3.14 bool isDataTableEmpty ( ) const [inline]

This function indicates whether the numeric table associated with the *griddedTableDef* element of a DAVE-ML dataset contains data or is empty.

Returns

A boolean variable, 'true' if the numeric table is empty.

#### 4.14.3.15 bool isStringDataTableEmpty ( ) const [inline]

This function indicates whether the alphanumeric table associated with the *griddedTableDef* element of a DAVE-ML dataset contains data or is empty.

Returns

A boolean variable, 'true' if the alphanumeric table is empty.

#### 4.14.3.16 void setBreakpointRefs ( const std::vector< size_t > *breakpointRef* ) [inline]

This function permits a vector of *breakpointRef's* to be manually set for the *griddedTableDef* element of this GriddedTableDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *breakpointRef* | a vector if breakpointRef indices (size_t) representing the position of the breakpoint identifier in the breakpoint list managed by the Janus instance. |

#### 4.14.3.17 void setDescription ( const dstoute::aString & *description* ) [inline]

This function permits the optional *description* of the *griddedTableDef* element to be reset for this GriddedTableDef instance. A *griddedTableDef's description* child element consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means pretty formatting of the XML source will also appear in the returned description.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *description* | a string representation of the description. |

#### 4.14.3.18 void setGtID ( const dstoute::aString & *gtID* ) [inline]

This function permits the *gtID* index attribute of the *griddedTableDef* element to be reset for this GriddedTableDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *gtID* | a string detailing the *gtID* index attribute. |

#### 4.14.3.19 void setJanus ( Janus * *janus* ) [inline]

This function permits the pointer to the base Janus class to be set manually. This function is used internally within a Janus instance by the Function class when it is instantiating a locally defined gridded table.

Parameters

| | |
|---|---|
| *janus* | a pointer to the base Janus instance |

#### 4.14.3.20 void setName ( const dstoute::aString & *name* ) [inline]

This function permits the *name* attribute of the *griddedTableDef* element to be reset for this GriddedTableDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set

before being written to an output XML based file.

Parameters

| | |
|---|---|
| *name* | a string detailing the *name* attribute. |

#### 4.14.3.21 void setTableData ( const std::vector< double > *dataPoints* ) [inline]

This function permits a vector of data points to be manually set for the *griddedTableDef* element of this GriddedTableDef instance. The data points are interpreted as the numeric data table associated with a gridded table.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *dataPoints* | a vector if numeric data values representing the data table for the *griddedTableDef*. |

#### 4.14.3.22 void setUnits ( const dstoute::aString & *units* ) [inline]

This function permits the *units* attribute of the *griddedTableDef* element to be reset for this GriddedTableDef instance.

If the instance has not been initialised from a DOM then this function permits it to be set before being written to an output XML based file.

Parameters

| | |
|---|---|
| *units* | a string detailing the *units* attribute. |

The documentation for this class was generated from the following files:

- GriddedTableDef.h
- GriddedTableDef.cpp

## 4.15 InDependentVarDef Class Reference

`#include <InDependentVarDef.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement, const bool &asPts=true)
- const std::vector< double > & getData () const
- ExtrapolateMethod getExtrapolationMethod () const
- InterpolateMethod getInterpolationMethod () const
- const double & getMax () const
- const double & getMin () const
- const dstoute::aString & getName () const
- const dstoute::aString & getSign () const
- const dstoute::aString & getUnits () const
- int getVariableReference () const
- const dstoute::aString & getVarID () const
- InDependentVarDef ()
- InDependentVarDef (const DomFunctions::XmlNode &elementDefinition)
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, const bool &isIndependentVarDef=true)
- void setVariableReference (int varRef)

## 4.15.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to the In-Dependent variable definitions contained in a DOM that complies with the DAVE-ML DTD.

A breakpointDef is where gridded table breakpoints are given. Since these are separate from function data, they may be reused.

bpVals is a set of breakpoints; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table.

## 4.15.2 Constructor & Destructor Documentation

### 4.15.2.1 InDependentVarDef ( )

The empty constructor can be used to instance the BreakpointDef class without supplying the DOM *breakpointDef* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *breakpointDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.15.2.2 InDependentVarDef ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *breakpointDef* element within a DOM, instantiates the BreakpointDef class and fills it with alphanumeric data from the DOM. The string content of the *bpVals* element is converted to a double precision numeric vector within the instance.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *independentVarElement* component within the DOM. |

### 4.15.3 Member Function Documentation

#### 4.15.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement*, const bool & *asPts* = *true* )

This function is used to export the *InDependentVarDef* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |
| *asPts* | a boolean indicating whether data is exported as points or the entry is exported as a reference to predefined data. |

#### 4.15.3.2 const std::vector< double>& getData ( ) const [inline]

This function provides access to the array of data values stored within this instance of the InDependentVarDef class.

Returns

a vector of numeric values, representing the data for this InDependentVarDef instance, is returned by reference.

#### 4.15.3.3 ExtrapolateMethod getExtrapolationMethod ( ) const [inline]

This function provides access to the *extrapolate* attribute of a *independentVarPts*. The data for the independentVarPts is stored as breakpoints, and hence uses the *breakpointDef* construct. The *extrapolate* attribute is optional. If the independentVarPts has no extrapolate attribute or has not been initialised from a DOM, a 'neither' string is returned.

Returns

An expolation enum containing the extrappolation technique applicable to the independent variable selected.

#### 4.15.3.4 InterpolateMethod getInterpolationMethod ( ) const [inline]

This function provides access to the *interpolate* attribute of a *independentVarPts*. The data for the independentVarPts is stored as breakpoints, and hence uses the *breakpointDef* construct. The *interpolate* attribute is optional. If the independentVarPts has no interpolate attribute or has not been initialised from a DOM, a 'linear' string is returned.

Returns

An interpolation enum containing the interpolation technique applicable to the independent variable selected.

#### 4.15.3.5 const double& getMax ( ) const [inline]

This function provides access to the *max* attribute of a *independentVarPts* element. This is used to bound the interpolation or extrapolation of breakpoint data when evaluating a Function element.

Returns

The *max* bound condition is returned by reference.

#### 4.15.3.6 const double& getMin ( ) const [inline]

This function provides access to the *min* attribute of a *independentVarPts* element. This is used to bound the interpolation or extrapolation of breakpoint data when evaluating a Function element.

Returns

The *min* bound condition is returned by reference.

#### 4.15.3.7 const dstoute::aString& getName ( ) const [inline]

This function provides access to the *name* attribute of the *breakpointDef* element represented by this BreakpointDef instance. The *name* attribute is optional. If the instance has not been initialised from a DOM, or if no *name* attribute is present, an empty string is returned.

Returns

The *name* string is passed by reference.

#### 4.15.3.8 const dstoute::aString& getSign ( ) const [inline]

This function provides access to the *sign* attribute of a *independentVarPts*. The data for the independentVarPts is stored as breakpoints, and hence uses the *breakpointDef* construct. The *sign* attribute is optional. If the independentVarPts has no sign attribute or has not been initialised from a DOM, an empty string is returned.

Returns

The *sign* string is returned by reference.

#### 4.15.3.9 const dstoute::aString& getUnits ( ) const [inline]

This function provides access to the *units* attribute of a *InDependentVarDef* represented by this InDependentVarDef instance. A breakpoint array's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by AD **FS!** (**FS!**) [7] in accordance with SI and other systems. The *units* attribute is optional. If the instance has not been initialised from a DOM, or if no *units* attribute is present, an empty string is returned.

Returns

The *units* string is passed by reference.

#### 4.15.3.10 int getVariableReference ( ) const [inline]

This function provides access to the variableDef reference for this instance of the InDependentVarDef class. This is the index of the variableDef entry within the list of variableDef elements managed by the base Janus instance.

Returns

the index of the variableDef associated with this instance of the InDependentVarDef class.

#### 4.15.3.11 const dstoute::aString& getVarID ( ) const [inline]

This function provides access to the *varID* attribute of a *InDependentVarDef*. This attribute is used for indexing variableDefs within an XML dataset. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *varID* string is passed by reference.

#### 4.15.3.12 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, const bool & *isIndependentVarDef* = *true* )

An uninitialised instance of InDependentVarDef is filled with data from a particular *InDependentVarDef* element within a DOM by this function. The string content of the *bpVals* element is converted to a double precision numeric vector within the instance. If another *InDependentVarDef* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *breakpointDef* component within the DOM. |
| *isIndependentVarDef* | is a boolean indicating whether the definition represents an independent (TRUE) or a dependent variable (FALSE). |

#### 4.15.3.13 void setVariableReference ( int *varRef* ) [inline]

This function is used to set the index of the variableDef associated with this instance of the InDependentVarDef class. This is the index of the variableDef entry within the list of variableDef elements managed by the base Janus instance. This function is called when instantiating gridded and ungridded table elements.

Parameters

| | |
|---|---|
| *varRef* | a index of the variableDef associated with this instance of the InDependentVarDef class |

The documentation for this class was generated from the following files:

- InDependentVarDef.h
- InDependentVarDef.cpp

## 4.16 InternalValues Class Reference

`#include <InternalValues.h>`

Inherits SignalList.

### Public Member Functions

- InternalValues ()
- InternalValues (const DomFunctions::XmlNode &internalValuesElement)

## 4.16.1 Detailed Description

An InternalValues instance functions as a container for the Signal class, and provides the functions that allow a calling StaticShot instance to access the *signal* elements that define the internal values for a check case. A *internalValues* element must contain *signal* elements that include *varID* (*signalID* is deprecated) elements.

The InternalValues class is only used within the *janus* namespace, and should only be referenced indirectly through the StaticShot class.

Typical usage:

```
Janus test( xmlFileName );
const CheckData& checkData = test.getCheckData();
size_t nss = checkData.getStaticShotCount( );
for ( size_t j = 0 ; j < nss ; j++ ) {
  const StaticShot& staticShot = checkData.getStaticShot( j );
  const InternalValues& internalValues = staticShot.getInternalValues();
  size_t nciv = internalValues.getSignalCount();
  cout << " staticShot[" << j << "] : " << endl
       << "      Name                          = "
       << staticShot.getName( ) << endl
       << "      Number of check internal values = " << nciv << endl;
  for ( size_t k = 0 ; k < nciv ; k++ ) {
    cout << "  internalValues[" << k << "] : " << endl
         << "    signalVarID                 = "
         << internalValues.getVarID( k ) << endl
         << "    signalValue                 = "
         << internalValues.getValue( k ) << endl
         << "    signalTol                   = "
         << internalValues.getTolerance( k )
         << endl << endl;
  }
}
```

## 4.16.2 Constructor & Destructor Documentation

### 4.16.2.1 InternalValues ( ) [inline]

The empty constructor can be used to instance the InternalValues class without supplying the DOM *internalValues* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *internalValues* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

#### 4.16.2.2 InternalValues ( const DomFunctions::XmlNode & *internalValuesElement* ) [inline]

The constructor, when called with an argument pointing to a *internalValues* element within a DOM, instantiates the InternalValues class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *internalValuesElement* | is an address of an *internalValues* component within the DOM. |

The documentation for this class was generated from the following file:

- InternalValues.h

## 4.17 Janus Class Reference

`#include <Janus.h>`

Inherits XmlElementDefinition.

### Public Types

### Public Member Functions

- virtual void clear ()
- void displayCheckDataSummary (const CheckData &checkData)
- virtual size_t exportToBuffer (std::ostringstream &documentBuffer)
- virtual size_t exportToBuffer (unsigned char *&documentBuffer)
- virtual size_t exportToFile (const dstoute::aFileString &dataFileName)
- VariableDef * findVariableDef (const dstoute::aString &varID)
- BreakpointDefList & getBreakpointDef ()
- const CheckData & getCheckData (const bool &evaluate=true)
- DomFunctions::XmlNode getDomDocument () const
- const FileHeader & getFileHeader () const
- const FunctionList & getFunction () const
- Function & getFunction (size_t index)
- GriddedTableDefList & getGriddedTableDef ()
- const char * getJanusVersion (VersionType versionType=HEX) const
- PropertyDefList & getPropertyDef ()
- PropertyDef & getPropertyDef (size_t index)

- PropertyDef & getPropertyDef (const dstoute::aString &ptyID)
- UngriddedTableDefList & getUngriddedTableDef ()
- VariableDefList & getVariableDef ()
- VariableDef & getVariableDef (size_t index)
- VariableDef & getVariableDef (const dstoute::aString &varID)
- int getVariableIndex (const dstoute::aString &varID) const
- const dstoute::aFileString & getXmlFileName () const
- void initiateDocumentObjectModel (const dstoute::aString &documentType="DAVEfunc")
- bool isJanusInitialised () const
- Janus ()
- Janus (const dstoute::aFileString &documentName, const dstoute::aFileString &keyFileName="")
- Janus (unsigned char *documentBuffer, size_t documentBufferSize)
- Janus (const Janus &rhs)
- Janus & operator= (const Janus &rhs)
- virtual void setXmlFileBuffer (unsigned char *documentBuffer, const size_t &documentBufferSize)
- virtual void setXmlFileName (const dstoute::aFileString &documentName, const dstoute::aFileString &keyFileName="")
- virtual ~Janus ()

### Protected Member Functions

- virtual void exportToDocumentObjectModel (const dstoute::aString &documentType="DAVEfunc")
- std::vector< size_t > getAllAncestors (size_t ix)
- std::vector< size_t > getAllDescendents (size_t index)
- std::vector< size_t > getIndependentAncestors (size_t ix)
- double getLinearInterpolation (Function &function, const std::vector< double > &dataTable)
- Uncertainty::UncertaintyPdf getPdfFromAntecedents (size_t index)
- double getPolyInterpolation (Function &function, const std::vector< double > &dataTable)
- double getUngriddedInterpolation (Function &function, const std::vector< double > &dataColumn)
- virtual void initialiseDependencies ()
- void isJanusInitialised (bool isInitialised)
- virtual void parseDOM ()
- void releaseJanusDomParser ()

- const Provenance & retrieveProvenanceReference (const dstoute::aString &parentID, size_t provIndex)
- bool writeDocumentObjectModel (const dstoute::aFileString &dataFileName) const
- bool writeDocumentObjectModel (std::ostringstream &sstr) const

### 4.17.1 Detailed Description

Janus is an XML dataset interface class. A Janus instance holds in its allocated memory the DOM corresponding to a DAVE-ML compliant XML dataset source file, and data structures derived from that DOM. It also provides the functions that allow a calling program to access the DOM and related data structures, including means of evaluating output variable values that are dependent on supplied input variable values. The DOM is normally only accessed during initialisation, which transfers its contents to more easily accessible structures within the class.

The documentation for this class was generated from the following files:

- Janus.h
- GetDescriptors.cpp
- Janus.cpp
- VariableDefLuaScript.cpp

## 4.18 MathMLDataClass Class Reference

`#include <MathMLDataClass.h>`

### 4.18.1 Detailed Description

A MathMLDataClass instance holds in its allocated memory alphanumeric data derived from *MathML* elements of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The data may include tags defining the *MathML* element and its attributes, a list of children associated with the *MathML* element, and call-backs to functions to evaluate the element.

The MathMLDataClass class is only used within the Janus.

The documentation for this class was generated from the following files:

- MathMLDataClass.h
- MathMLDataClass.cpp

## 4.19 Modification Class Reference

`#include <Modification.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const AuthorList & getAuthor () const
- const Author & getAuthor (const size_t &index) const
- size_t getAuthorCount () const
- const dstoute::aString & getDate () const
- const dstoute::aString & getDescription () const
- size_t getExtraDocCount () const
- const dstoute::aString & getExtraDocRefID (const size_t &index) const
- const dstoute::aString & getModID () const
- const dstoute::aString & getRefID () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- Modification ()
- Modification (const DomFunctions::XmlNode &elementDefinition)

#### 4.19.1 Detailed Description

A Modification instance holds in its allocated memory alphanumeric data derived from a *modificationRecord* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes the author and content of a modification to a dataset. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Modification class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader class.

A typical usage is:

```
Janus test( xmlFileName );
FileHeader header = test.getFileHeader();
int nMod = header.getModificationCount();
cout << " Number of modification records   : " << nMod << "\n\n";

for ( int i = 0 ; i < nMod ; i++ ) {
  Modification modification = header.getModification( i );
  cout << " Modification Record " << i << " : \n"
       << "   modID               : "
       << modification.getModID( ) << "\n"
       << "   date                : "
       << modification.getDate( ) << "\n"
       << "   refID               : "
       << modification.getRefID( ) << "\n\n";
  int nModAuthors = modification.getAuthorCount( );
  for ( int j = 0 ; j < nModAuthors ; j ++ ) {
    Author author = modification.getAuthor( j );
    cout << " Author " << j << " : Name                 : "
```

```
        << author.getName( ) << "\n"
        << "          Organisation            : "
        << author.getOrg( ) << "\n";
  }
  cout << "   description         : "
       << modification.getDescription( )
       << "\n" << "\n";

  int nExdoc = modification.getExtraDocCount( );
  cout << " Number of extra documents related to modification : "
       << nExdoc << "\n\n";

  for ( int j = 0 ; j < nExdoc ; j++ ) {
    cout << "  Extra document " << j << " refID : "
         << modification.getExtraDocRefID( j ) << "\n";
  }
  cout << "\n";
}
```

### 4.19.2 Constructor & Destructor Documentation

#### 4.19.2.1 Modification ( )

The empty constructor can be used to instance the Modification class without supplying the DOM *modificationRecord* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *modificationRecord* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.19.2.2 Modification ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *modificationRecord* element within a DOM, instantiates the Modification class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *modificationRecord* component within the DOM. |

### 4.19.3 Member Function Documentation

#### 4.19.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *Function* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.19.3.2 const AuthorList& getAuthor ( ) const [inline]

This function returns a reference to the is of Author instances within a Modification instance.

Returns

The list of Author instances is returned by reference.

See also

Author

#### 4.19.3.3 const Author& getAuthor ( const size_t & *index* ) const [inline]

This function returns a reference to the selected Author instance within a Modification instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getAuthorCount() - 1 ), and selects the required Author instance. An attempt to access a non-existent *author* will throw a standard out_of_range exception. |

Returns

The requested Author instance is returned by reference.

See also

Author

#### 4.19.3.4 size_t getAuthorCount ( ) const [inline]

This function returns the number of authors listed in a Modification. If the instance has not been populated from a DOM element, zero is returned.

Returns

An integer number, one or more in a populated instance.

See also

Author

#### 4.19.3.5 const dstoute::aString& getDate ( ) const [inline]

This function returns the *date* attribute of a *modificationRecord*. The format of the dataset string is determined by the XML dataset builder, but DAVE-ML recommends the ISO 8601 form ("2004-01-02" to refer to 2 January 2004). If the Modification has not been populated from the DOM element, the function returns an empty string.

Returns

The *date* string is returned by reference.

#### 4.19.3.6 const dstoute::aString& getDescription ( ) const [inline]

This function returns the *description* from a *modificationRecord*, if one has been supplied in the XML dataset. The description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description string. If the *modificationRecord* contains no *description*, the returned string is blank.

Returns

The *description* string is returned by reference.

#### 4.19.3.7 size_t getExtraDocCount ( ) const [inline]

This function returns the number of *extraDocRef* elements listed in a *modificationRecord*. A Modification can have no, one or multiple *extraDocRef* components. If the instance has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

#### 4.19.3.8 const dstoute::aString& getExtraDocRefID ( const size_t & *index* ) const [inline]

This function returns the *refID* of a selected *extraDocRef* component from a *modificationRecord*. A Modification can have no, one or multiple *extraDocRef* components. The *refID* provides cross-referencing to *reference* definitions elsewhere in the FileHeader.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getExtraDocCount() - 1 ), and selects the required *extraDocRef* record. An attempt to access a non-existent *extraDocRef* will throw a standard out_of_range exception. |

Returns

The requested *refID* string is returned by reference.

See also

Reference

#### 4.19.3.9 const dstoute::aString& getModID ( ) const [inline]

A *modID* is a single letter used to identify all modified data associated with a modification record. It allows *modificationRecord* elements to be referenced by elements other than their immediate parent. This function returns the *modID* of the Modification instance.

Returns

The *modID* string is returned by reference.

#### 4.19.3.10 const dstoute::aString& getRefID ( ) const [inline]

The *refID* attribute is an optional document reference for a *modificationRecord*. It may be used for cross-referencing a list of references contained in the *fileHeader*. This function returns the *refID* of a *modificationRecord*, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *refID* string is returned by reference.

See also

Reference
FileHeader

#### 4.19.3.11 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of Modification is filled with data from a particular *modificationRecord* element within a DOM by this function. If another *modificationRecord* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *modificationRecord* component within the DOM. |

The documentation for this class was generated from the following files:

- Modification.h
- Modification.cpp

## 4.20 ParseMathML Class Reference

`#include <ParseMathML.h>`

### 4.20.1 Detailed Description

This class contains functions for parsing mathematics procedures defined using the MathML syntax. Data detailing each MathML operation is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

The ParseMathML class is only used within the Janus.

The documentation for this class was generated from the following files:

- ParseMathML.h
- ParseMathML.cpp

## 4.21 Provenance Class Reference

`#include <Provenance.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement, const bool &isReference=false)
- const AuthorList & getAuthor () const
- const Author & getAuthor (const size_t &index) const
- size_t getAuthorCount () const
- const dstoute::aString & getCreationDate () const
- const dstoute::aString & getDescription () const
- size_t getDocumentRefCount () const
- const dstoute::aStringList & getDocumentRefID () const
- const dstoute::aString & getDocumentRefID (const size_t &index) const
- const dstoute::aStringList & getModificationModID () const
- const dstoute::aString & getModificationModID (const size_t &index) const
- size_t getModificationRefCount () const
- const dstoute::aString & getProvID () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- Provenance ()
- Provenance (const DomFunctions::XmlNode &elementDefinition)

#### 4.21.1 Detailed Description

A Provenance instance holds in its allocated memory alphanumeric data derived from a *provenance* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Provenances may apply to a complete dataset or to individual components within a dataset. Not all provenances will contain all possible *provenance* components. The Provenance instance also provides the functions that allow a calling Janus instance to access these data elements.

The Provenance class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader, VariableDef, GriddedTableDef, UngriddedTableDef, Function or CheckData classes.

A typical usage is:

```
Janus test( xmlFileName );
FileHeader header = test.getFileHeader();
int nProv = header.getProvenanceCount();
cout << " Number of header provenance elements   : " << nProv << "\n\n";

for ( int i = 0 ; i < nProv ; i++ ) {
  Provenance provenance = header.getProvenance( i );
  cout << " Header Provenance " << i << "   : \n"
       << "    provID                : "
       << provenance.getProvID() << "\n"
       << "    creationDate          : "
       << provenance.getCreationDate() << "\n";
```

```
  for ( int j = 0 ; j < provenance.getAuthorCount() ; j++ ) {
    cout << "     author " << j << "  : "
         << provenance.getAuthor( j ).getName() << "\n";
  }
  cout << "    description           : \n"
       << provenance.getDescription() << "\n\n";
}

for ( int i = 0 ; i < test.getNumberOfFunctions() ; i++ ) {
  cout << " Function \" "
       << test.getFunctionName( i ) << "\" \n";
  if ( true == test.getFunction().at( i ).hasProvenance() ) {
    Provenance provenance = test.getFunction().at( i ).getProvenance();
    cout << " Provenance \n"
         << "     provID            : "
         << provenance.getProvID() << "\n"
         << "     creation date     : "
         << provenance.getCreationDate() << "\n";
    for ( int j = 0 ; j < provenance.getAuthorCount() ; j++ ) {
      cout << "     author " << j << "  : "
           << provenance.getAuthor( j ).getName() << "\n";
    }
    cout << "    description           : \n"
         << provenance.getDescription() << "\n\n";
  }
}
```

### 4.21.2 Constructor & Destructor Documentation

#### 4.21.2.1 Provenance ( )

The empty constructor can be used to instance the Provenance class without supplying the DOM *provenence* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *provenance* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.21.2.2 Provenance ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *provenance* element within a DOM, instantiates the Provenance class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *provenance* component within the DOM. |

### 4.21.3 Member Function Documentation

#### 4.21.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement*, const bool & *isReference* = `false` )

This function is used to export the *provenance* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |
| *isReference* | a boolean flag indicating the provenance element should be treated as a reference. |

#### 4.21.3.2 const AuthorList& getAuthor ( ) const `[inline]`

This function returns a reference to the ist of Author instances within a Provenance instance.

Returns

The list of Author instances is returned by reference.

See also

Author

#### 4.21.3.3 const Author& getAuthor ( const size_t & *index* ) const `[inline]`

This function returns a reference to the selected Author instance within a Provenance instance.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getAuthorCount() - 1 ), and selects the required Author instance. An attempt to access a non-existent *author* will throw a standard out_of_range exception. |

Returns

The requested Author instance is returned by reference.

See also

Author

#### 4.21.3.4 size_t getAuthorCount ( ) const [inline]

This function returns the number of authors listed in a Provenance. If the instance has not been populated from a DOM element, zero is returned.

Returns

An integer number, one or more in a populated instance.

See also

Author

#### 4.21.3.5 const dstoute::aString& getCreationDate ( ) const [inline]

This function returns the *creationDate* attribute of a *provenance* element (*fileCreationDate* is a deprecated alternative). The format of the dataset string is determined by the XML dataset builder, but DAVE-ML recommends the ISO 8601 form ("2004-01-02" to refer to 2 January 2004). If the Provenance has not been populated from a DOM element , the function returns an empty string.

Returns

The *creationDate* string is returned by reference.

#### 4.21.3.6 const dstoute::aString& getDescription ( ) const [inline]

This function returns the *description* from a *provenance* element, if one has been supplied in the XML dataset. The description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description string. If the *provenance* contains no *description*, the returned string is blank.

Returns

The *description* string is returned by reference.

#### 4.21.3.7 size_t getDocumentRefCount ( ) const [inline]

This function returns the number of document references listed in a a *provenance* element. A *provenance* can include no, one or multiple *documentRef* components. If the Provenance has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

#### 4.21.3.8 const dstoute::aStringList& getDocumentRefID ( ) const [inline]

This function returns the list of *refID's* from the *documentRef* child element of a *provenance* element (*docID* is a deprecated alternative). The *refID's* allows *reference* elements elsewhere in the DOM to be referenced by elements other than their immediate parent.

Returns

The list of *refID* strings from the selected *documentRef* is returned by reference.

See also

Reference

#### 4.21.3.9 const dstoute::aString& getDocumentRefID ( const size_t & *index* ) const [inline]

This function returns the selected *refID* from the *documentRef* child element of a *provenance* element (*docID* is a deprecated alternative). The *refID* allows *reference* elements elsewhere in the DOM to be referenced by elements other than their immediate parent.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getDocumentRefCount() - 1 ), and selects the required *documentRef* from which the *refID* string is to be returned. An attempt to access a non-existent *documentRef* will throw a standard out_of_range exception. |

Returns

The *refID* string from the selected *documentRef* is returned by reference.

See also

Reference

#### 4.21.3.10 const dstoute::aStringList& getModificationModID ( ) const [inline]

This function returns the list of *modID's* from the *modificationRef* child element of a *provenance* element. The *modID* allows *modificationRecord* elements elsewhere in the DOM to be referenced by elements other than their immediate parent.

Returns

The list of *modID* strings from the selected *modificationRef* is returned by reference.

See also

Modification

#### 4.21.3.11 const dstoute::aString& getModificationModID ( const size_t & *index* ) const [inline]

This function returns the selected *modID* from the *modificationRef* child element of a *provenance* element. The *modID* allows *modificationRecord* elements elsewhere in the DOM to be referenced by elements other than their immediate parent.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getModificationRefCount() - 1 ), and selects the required *modificationRef* from which the *modID* string is to be returned. An attempt to access a non-existent *modificationRef* will throw a standard out_of_range exception. |

Returns

The *modID* string from the selected *modificationRef* is returned by reference.

See also

Modification

#### 4.21.3.12 size_t getModificationRefCount ( ) const [inline]

This function returns the number of modification references listed in a *provenance* element. A *provenance* can include no, one or multiple *modificationRef* components. If the Provenance has not been populated from a DOM element, zero is returned.

Returns

An integer number, zero or more in a populated instance.

#### 4.21.3.13 const dstoute::aString& getProvID ( ) const [inline]

The *provID* allows *provenance* elements to be referenced by elements other than their immediate parent. It is an optional attribute. This function returns the *provID* of the referenced *provenance* element, if one has been supplied in the XML dataset. If not, it returns an empty string.

Returns

The *provID* string is returned by reference.

#### 4.21.3.14 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of Provenance is filled with data from a particular *provenance* element within a DOM by this function. If another *provenance* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *provenance* component within the DOM. |

The documentation for this class was generated from the following files:

- Provenance.h
- Provenance.cpp

## 4.22 Reference Class Reference

`#include <Reference.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const dstoute::aString & getAccession () const
- const dstoute::aString & getAuthor () const
- const dstoute::aString & getClassification () const
- const dstoute::aString & getDate () const
- const dstoute::aString & getDescription () const
- const dstoute::aString & getHref () const
- const dstoute::aString & getRefID () const
- const dstoute::aString & getTitle () const
- const dstoute::aString & getXLink () const
- const dstoute::aString & getXLinkType () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)

- Reference ()
- Reference (const DomFunctions::XmlNode &elementDefinition)

### 4.22.1 Detailed Description

A Reference instance holds in its allocated memory alphanumeric data derived from a *reference* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes an external document relevant to the dataset. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Reference class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader class. A typical usage is:

```
Janus test( xmlFileName );
FileHeader header = test.getFileHeader();
int nRef = header.getReferenceCount();
cout << " Number of reference records      : " << nRef << "\n\n";

for ( int i = 0 ; i < nRef ; i++ ) {
  Reference reference = header.getReference( i );
  cout << " Reference " << i << "    : \n"
       << "   xmlns:xlink    : "
       << reference.getXLink( ) << "\n"
       << "   xlink:type     : "
       << reference.XLinkType( ) << "\n"
       << "   refID          : "
       << reference.getRefID( ) << "\n"
       << "   author         : "
       << reference.getAuthor( ) << "\n"
       << "   title          : "
       << reference.getTitle( ) << "\n"
       << "   date           : "
       << reference.getDate( ) << "\n"
       << "   classification : "
       << reference.getClassification( ) << "\n"
       << "   accession   : "
       << reference.getAccession( ) << "\n"
       << "   xlink:href     : "
       << reference.getHref( ) << "\n"
       << "   description    : "
       << reference.getDescription( ) << "\n"
       << "\n";
}
```

### 4.22.2 Constructor & Destructor Documentation

#### 4.22.2.1 Reference ( )

The empty constructor can be used to instance the Reference class without supplying the DOM *reference* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *reference* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.22.2.2 Reference ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *reference* element within a DOM, instantiates the Reference class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *reference* component within the DOM. |

### 4.22.3 Member Function Documentation

#### 4.22.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *author* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.22.3.2 const dstoute::aString& getAccession ( ) const [inline]

This function returns the *accession* attribute of a *reference* element. The *accession* attribute is a string containing the accession number (ISBN or organisation report number) of the referenced document. This is an optional attribute. If the Reference instance does not contain a *accession* attribute, or has not been initialised from a DOM, an empty string is returned.

Returns

The *accession* string is returned by reference.

#### 4.22.3.3 const dstoute::aString& getAuthor ( ) const [inline]

This function returns the *author* attribute of a *reference* element. The *author* attribute is a string containing the name of the author of the referenced document. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *author* string is returned by reference.

#### 4.22.3.4 const dstoute::aString& getClassification ( ) const [inline]

This function returns the *classification* attribute of a *reference* element. The *classification* attribute is a string containing the security classification of the referenced document. This is an optional attribute. If the Reference instance does not contain a *classification* attribute, or has not been initialised from a DOM, an empty string is returned.

Returns

The *classification* string is returned by reference.

#### 4.22.3.5 const dstoute::aString& getDate ( ) const [inline]

This function returns the *date* attribute of a *reference* element. The *date* attribute is a string containing the publication date of the referenced document. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *date* string is returned by reference.

#### 4.22.3.6 const dstoute::aString& getDescription ( ) const [inline]

This function returns the *description* child element of a *reference* instance. The *description* child element is a (possibly lengthy) string containing information regarding the referenced document, whose format within the XML dataset will be preserved by this function. It is an optional attribute. If the Reference instance does not contain a *description* attribute, or has not been initialised from a DOM, an empty string is returned.

Returns

The *description* string is returned by reference.

#### 4.22.3.7 const dstoute::aString& getHref ( ) const [inline]

This function returns the *xlink:href* attribute of a *reference* element. The *xlink:href* attribute is a string containing a URL of an on-line copy of the referenced document. This is an optional attribute. If the Reference instance does not contain a *xlink:href* attribute, or has not been initialised from a DOM, an empty string is returned.

Returns

The *xlink:href* string is returned by reference.

#### 4.22.3.8 const dstoute::aString& getRefID ( ) const [inline]

This function returns the *refID* associated with a Reference instance. The *refID* allows *reference* elements to be cited by elements throughout the DOM, by elements other than their immediate parent, *fileHeader*. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *refID* string is returned by reference.

#### 4.22.3.9 const dstoute::aString& getTitle ( ) const [inline]

This function returns the *title* attribute of a *reference* element. The *title* attribute is a string containing the title of the referenced document. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *title* string is returned by reference.

#### 4.22.3.10 const dstoute::aString& getXLink ( ) const [inline]

This function returns the *xmlns:xlink* associated with a Reference instance. If the instance has not been initialised from a DOM, the string is is set to "http://www.w3.org/19999/xlink", and returned.

Returns

The *xmlns:xlink* string is returned by reference.

#### 4.22.3.11 const dstoute::aString& getXLinkType ( ) const [inline]

This function returns the *xlink:type* associated with a Reference instance. If the instance has not been initialised from a DOM, the string is is set to "simple", and returned.

Returns

The *xlink:type* string is returned by reference.

#### 4.22.3.12 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of Reference is filled with data from a particular *reference* element within a DOM by this function. If another *reference* element pointer is supplied to an instance

that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *reference* component within the DOM. |

The documentation for this class was generated from the following files:

- Reference.h
- Reference.cpp

## 4.23 Signal Class Reference

`#include <Signal.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const dstoute::aString & getName () const
- const double & getTolerance () const
- const dstoute::aString & getUnits () const
- const double & getValue () const
- const dstoute::aString & getVarID () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, const SignalTypeEnum &signalType)
- Signal ()
- Signal (const DomFunctions::XmlNode &elementDefinition, const SignalTypeEnum &signalType)

### 4.23.1 Detailed Description

A Signal instance holds in its allocated memory alphanumeric data derived from a *signal* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance may describe inputs, internal values of a computation, or outputs. The class also provides the functions that allow a calling StaticShot instance to access these data elements. A *signal* must have *signalName* and *signalUnits* if it is a child of *checkInputs* or *checkOutputs*. Alternatively, if it is a child of *internalValues*, it must have a *varID* (*signalID* is deprecated). This class accepts whichever of these children it finds in the XML dataset, and leaves applicability to its parents to sort out.

The Signal class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot, CheckInputs, InternalValues and CheckOutputs classes.

Typical usage:

```
Janus test( xmlFileName );
CheckData checkData = test.getCheckData();
size_t nss = checkData.getStaticShotCount();
for ( size_t j = 0 ; j < nss ; j++ ) {
  StaticShot staticShot = checkData.getStaticShot( j );
  CheckOutputs checkOutputs = staticShot.getCheckOutputs();
  size_t ncout = checkOutputs.getSignalCount();
  cout << " staticShot[" << j << "] : " << endl
       << "      Name                      = "
       << staticShot.getName( ) << endl
       << "      Number of check outputs   = " << ncout << endl;
  for ( size_t k = 0 ; k < ncout ; k++ ) {
    cout << "  checkOutputs[" << k << "] : " << endl
         << "    signalName                = "
         << checkOutputs.getName( k ) << endl
         << "    signalUnits               = "
         << checkOutputs.getUnits( k ) << endl
         << "    signalValue               = "
         << checkOutputs.getValue( k ) << endl
         << "    signalTol                 = "
         << checkOutputs.getTolerance( k ) << endl
         << endl;
  }
}
```

### 4.23.2 Constructor & Destructor Documentation

#### 4.23.2.1 Signal ( )

The empty constructor can be used to instance the Signal class without supplying the DOM *signal* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *signal* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.23.2.2 Signal ( const DomFunctions::XmlNode & *elementDefinition*, const SignalTypeEnum & *signalType* )

The constructor, when called with an argument pointing to a *signal* element within a DOM, instantiates the Signal class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *signal* component within the DOM. |

Parameters

| | |
|---|---|
| *signalType* | is a enumeration identifying the signal as either an input, and output, or an internal value. |

### 4.23.3 Member Function Documentation

#### 4.23.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *Signal* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.23.3.2 const dstoute::aString& getName ( ) const [inline]

This function returns the content of the signal's *signalName* child element. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *signalName* content string is returned by reference.

#### 4.23.3.3 const double& getTolerance ( ) const [inline]

This function returns the content of a signal's *tol* child element, if the *signal* is part of either an *internalValues* or a *checkOutputs* element. If the Signal has not been populated from a DOM, NaN will be returned. If the *signal* is part of a *checkInputs* element, or a tolerance is not specified for the *signal* within the XML dataset, this function will return zero.

Returns

A double precision variable containing the tolerance on the *signal* value is returned.

#### 4.23.3.4 const dstoute::aString& getUnits ( ) const [inline]

This function returns the content of the signal's *signalUnits* child element. The *signalUnits* content is a string of arbitrary length, but normally short, and complying with the format requirements chosen by AD APS [7] in accordance with SI and other systems. If the Signal has not been initialised from a DOM, an empty string is returned.

Returns

The *signalUnits* content string is returned by reference.

#### 4.23.3.5 const double& getValue ( ) const [inline]

This function returns the content of the signal's *signalValue* child element. It represent the numeric value that a particular variable from the XML dataset should return for the check case that forms the parent of this signal. If the Signal has not been populated from a DOM element, NaN is returned.

Returns

A double precision variable containing the *signal* value is returned.

#### 4.23.3.6 const dstoute::aString& getVarID ( ) const [inline]

This function returns the content of the signal's *varID* child element. The *varID* is a unique (per list of check case elements), short string not including whitespace that indicates the VariableDef the signal corresponds with, and is used for signal indexing. If the *signal* element owns a (deprecated alternative) *signalID* child element, that will be returned by this function. If the Signal has not been initialised from a DOM, an empty string is returned.

Returns

The *varID* or *signalID* content string is returned by reference.

#### 4.23.3.7 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, const SignalTypeEnum & *signalType* )

An uninitialised instance of Signal is filled with data from a particular *signal* element within a DOM by this function. If another *signal* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *signal* component within the DOM. |
| *signalType* | is a enumeration identifying the signal as either an input, and output, or an internal value. |

The documentation for this class was generated from the following files:

- Signal.h
- Signal.cpp

## 4.24 SignalList Class Reference

`#include <SignalList.h>`

Inherits XmlElementDefinition.

Inherited by CheckInputs, CheckOutputs, and InternalValues.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- int getIndex (const dstoute::aString &name) const
- const dstoute::aString & getName (const size_t &index) const
- const Signals & getSignal () const
- size_t getSignalCount () const
- const double & getTolerance (const size_t &index) const
- const dstoute::aString & getUnits (const size_t &index) const
- const double & getValue (const size_t &index) const
- const dstoute::aString & getVarID (const size_t &index) const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, const SignalTypeEnum &signalType)
- SignalList ()
- SignalList (const DomFunctions::XmlNode &elementDefinition, const SignalTypeEnum &signalType)

### 4.24.1 Detailed Description

A SignalList instance functions as a container for the Signal class, and provides the functions that allow a calling StaticShot instance to access the *signal* elements that define either the input or output values for a check case. A *signalList* element contains a list of *signal* elements, that include *signalName*, *signalUnits*, *signalValue* elements. An optional *tol* element may be included.

The SignalList class is only used within the janus namespace, and is inherited by the CheckInputs and CheckOutputs classes. It should only be referenced indirectly through the StaticShot class.

Typical usage:

```
Janus test( xmlFileName );
CheckData checkData = test.getCheckData();
size_t nss = checkData.getStaticShotCount( );
for ( size_t j = 0 ; j < nss ; j++ ) {
  StaticShot staticShot = checkData.getStaticShot( j );
  CheckOutputs checkOutputs = staticShot.getCheckOutputs();
  size_t ncout = checkOutputs.getSignalCount();
  cout << " staticShot[" << j << "] : " << endl
```

```
        << "      Name                        = "
        << staticShot.getName( ) << endl
        << "      Number of check outputs     = " << ncout << endl;
  for ( size_t k = 0 ; k < ncout ; k++ ) {
    cout << "  checkOutputs[" << k << "] : " << endl
         << "    signalName                  = "
         << checkOutputs.getName( k ) << endl
         << "    signalUnits                 = "
         << checkOutputs.getUnits( k ) << endl
         << "    signalValue                 = "
         << checkOutputs.getValue( k ) << endl
         << "          tol                   = "
         << checkOutputs.getTolerance( k ) << endl
         << endl;
  }
}
```

### 4.24.2 Constructor & Destructor Documentation

#### 4.24.2.1 SignalList ( )

The empty constructor can be used to instance the SignalList class without supplying the DOM *signal* elements from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *signal* elements list before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseElement

#### 4.24.2.2 SignalList ( const DomFunctions::XmlNode & *elementDefinition*, const SignalTypeEnum & *signalType* )

The constructor, when called with an argument pointing to *signal* elements within a DOM, instantiates the SignalList class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a list of *signal* components within the DOM. |
| *signalType* | is a enumeration identifying the signal as either an input, and output, or an internal value. |

### 4.24.3 Member Function Documentation

#### 4.24.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *SignalList* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.24.3.2 int getIndex ( const dstoute::aString & *name* ) const

This function returns the index number within the SignalList instance that corresponds with a specified signal *name*.

Parameters

| | |
|---|---|
| *name* | is a string containing *name* of the signal of interest. |

Returns

An integer value for the index corresponding to the specified *name* is returned. If the specified name does not appear in any signal within the SignalList instance, -1 is returned.

See also

Signal

#### 4.24.3.3 const dstoute::aString& getName ( const size_t & *index* ) const [inline]

This function returns the *signalName* from a Signal associated with the referenced SignalList instance. If the Signal has not been initialised from a DOM, an empty string is returned.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getSignalCount() - 1 ), and selects the required Signal component. Attempting to access a Signal outside the available range will throw a standard out_of_range exception. |

Returns

The selected *signalName* string is returned by reference.

See also

Signal

#### 4.24.3.4 const Signals& getSignal ( ) const [inline]

This function provides access to the signal definitions instances that have been defined for the singnalList instance. An empty vector will be returned if the Signal instance has not been populated from a DOM. In all other cases, the vector will contain at least one signal instance.

Returns

An vector of signal definitions instances.

See also

Signal

#### 4.24.3.5 size_t getSignalCount ( ) const [inline]

This function provides the number of signals making up the referenced SignalList instance. If the instance has not been populated from a DOM element, zero is returned. For a full check case, this function will return the number of output variables, explicit or implicit, in the XML dataset.

Returns

An integer number, one or more in a populated instance.

See also

Signal

#### 4.24.3.6 const double& getTolerance ( const size_t & *index* ) const [inline]

This function returns the *tol* component from a Signal associated with the referenced SignalList instance. If the Signal has not been populated from a DOM, NaN will be returned. If a tolerance is not specified for the *signal* within the XML dataset, this function will return zero.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getSignalCount() - 1 ), and selects the required Signal component. Attempting to access a Signal outside the available range will throw a standard out_of_range exception. |

Returns

A double precision variable containing the tolerance on the requested signal value is returned.

See also

Signal

#### 4.24.3.7 const dstoute::aString& getUnits ( const size_t & *index* ) const [inline]

This function returns the *signalUnits* from a Signal associated with the referenced SignalList instance. If the Signal has not been initialised from a DOM, an empty string is returned.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getSignalCount() - 1 ), and selects the required Signal component. Attempting to access a Signal outside the available range will throw a standard out_of_range exception. |

Returns

The selected *signalUnits* string is returned by reference.

See also

Signal

#### 4.24.3.8 const double& getValue ( const size_t & *index* ) const [inline]

This function returns the *signalValue* from a Signal associated with the referenced SignalList instance. If the Signal has not been populated from a DOM element, NaN is returned.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getSignalCount() - 1 ), and selects the required Signal component. Attempting to access a Signal outside the available range will throw a standard out_of_range exception. |

Returns

A double precision variable containing the requested signal value is returned.

See also

Signal

#### 4.24.3.9 const dstoute::aString& getVarID ( const size_t & *index* ) const [inline]

This function returns the *varID* from a Signal associated with the referenced SignalList instance. If the Signal has not been initialised from a DOM, an empty string is returned.

Parameters

| | |
|---|---|
| *index* | has a range from zero to ( getSignalCount() - 1 ), and selects the required Signal component. Attempting to access a Signal outside the available range will throw a standard out_of_range exception. |

Returns

The selected *varID* string is returned by reference.

See also

Signal

#### 4.24.3.10 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition,* const SignalTypeEnum & *signalType* )

An uninitialised instance of SignalList is filled with data from a particular list of *signal* elements within a DOM by this function. If another list of *signal* elements pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a list of *signal* components within the DOM. |
| *signalType* | is a enumeration identifying the signal as either an input, and output, or an internal value. |

The documentation for this class was generated from the following files:

- SignalList.h
- SignalList.cpp

## 4.25 SolveMathML Class Reference

`#include <SolveMathML.h>`

### 4.25.1 Detailed Description

This class contains functions for solving mathematics procedures defined using the MathML syntax. Data detailing each MathML operation is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

The SolveMathML class is only used within the Janus.

The documentation for this class was generated from the following files:

- SolveMathML.h
- SolveMathML.cpp

## 4.26 StaticShot Class Reference

`#include <StaticShot.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const CheckInputs & getCheckInputs () const
- const CheckOutputs & getCheckOutputs () const
- const dstoute::aString & getDescription () const
- const InternalValues & getInternalValues () const
- const dstoute::aString & getInvalidVariable (const size_t &index) const
- size_t getInvalidVariableCount () const
- const dstoute::aString & getName () const
- const Provenance & getProvenance () const
- const dstoute::aString & getRefID () const
- bool hasInternalValues () const
- const bool & hasProvenance () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition)
- StaticShot ()
- StaticShot (const DomFunctions::XmlNode &elementDefinition)
- void verifyStaticShot (Janus *janus)

### 4.26.1 Detailed Description

XML datset content verification - static input / output correlation. A StaticShot instance holds in its allocated memory alphanumeric data derived from a *staticShot* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes the inputs and outputs, and possibly internal values, of a DAVE-ML model at a particular instant of time. The class also provides the functions that allow a calling Janus instance to access these data elements.

The StaticShot class is only used within the janus namespace, and should only be referenced indirectly through the CheckData class.

Units used in the *staticShot* element need not be identical to those used in the remainder of the dataset.

Typical usage:

```
Janus test( xmlFileName );
CheckData checkData = test.getCheckData();
size_t nss = checkData.getStaticShotCount( );
for ( size_t j = 0 ; j < nss ; j++ ) {
  StaticShot staticShot = checkData.getStaticShot( j );
  size_t nInvalid = staticShot.getInvalidVariableCount();
  if ( 0 < nInvalid ) {
    for ( size_t k = 0 ; k < nInvalid ; k++ ) {
      string failVarID = staticShot.getInvalidVariable(  k );
      cout << "  Problem at varID : " << failVarID << endl;
    }
  }
  else {
    cout << "  No problems from static shot " << j << " ... " << endl;
  }
}
```

### 4.26.2 Constructor & Destructor Documentation

#### 4.26.2.1 StaticShot ( )

The empty constructor can be used to instance the StaticShot class without supplying the DOM *staticShot* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *staticShot* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.26.2.2 StaticShot ( const DomFunctions::XmlNode & *elementDefinition* )

The constructor, when called with an argument pointing to a *staticShot* element within a DOM, instantiates the StaticShot class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *staticShot* component within the DOM. |

### 4.26.3 Member Function Documentation

#### 4.26.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *StaticShot* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.26.3.2 const CheckInputs& getCheckInputs ( ) const [inline]

This function provides access to the *checkInputs* element contained in a DAVE-ML *staticShot* element. There must be one of these elements for each static shot in a valid dataset.

Returns

The CheckInputs instance is returned by reference.

#### 4.26.3.3 const CheckOutputs& getCheckOutputs ( ) const [inline]

This function provides access to the *checkOutputs* element contained in a DAVE-ML *staticShot* element. There must be one of these elements for each static shot in a valid dataset.

Returns

The CheckOutputs instance is returned by reference.

#### 4.26.3.4 const dstoute::aString& getDescription ( ) const [inline]

This function returns the *description* of the referenced *staticShot* element, if one has been supplied in the XML dataset. If not, or if the StaticShot has not been initialised from a DOM, it returns an empty string.

Returns

The *description* string is returned by reference.

#### 4.26.3.5 const InternalValues& getInternalValues ( ) const [inline]

This function provides access to the *internalValues* element contained in a DAVE-ML *staticShot* element. There may be zero or one of these elements for each static shot in a valid dataset.

Returns

The InternalValues instance is returned by reference.

#### 4.26.3.6 const aString & getInvalidVariable ( const size_t & *index* ) const

This function uses the results computed by setStaticShotVerification() to indicate which internal or output values, computed in accordance with a DAVE-ML compliant dataset, are incompatible with the values contained in this *staticShot* element. It is used in conjunction with getInvalidVariableCount and other CheckData functions.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to ( getInvalidVariableCount() - 1 ), and selects the invalid variable whose identity is required. |

Returns

A string is returned by reference, containing either the *name* or *varID* attribute, as applicable, of the selected invalid variable.

#### 4.26.3.7 size_t getInvalidVariableCount ( ) const

This function uses the results computed by setStaticShotVerification() to indicate how many internal or output values computed in accordance with a DAVE-ML compliant dataset are incompatible with the values contained in this *staticShot* element.

Returns

An positive integer value is returned, being the total number of internal or output *variableDefs* in the dataset whose values are not compatible, within the specified tolerences, with the specified *staticShot* values.

See also

getInvalidVariable
setStaticShotVerification

#### 4.26.3.8 const dstoute::aString& getName ( ) const [inline]

This function returns the *name* attribute of a *staticShot*. If the static shot has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is returned by reference.

#### 4.26.3.9 const Provenance& getProvenance ( ) const [inline]

This function provides access to the Provenance instance associated with a StaticShot instance. There may be zero or one of these elements for each static shot in a valid dataset, defined either directly or by reference.

Returns

The Provenence instance is returned by reference.

See also

Provenance

#### 4.26.3.10 const dstoute::aString& getRefID ( ) const [inline]

The *refID* attribute is an optional document reference for a *staticShot*. This function returns the *refID* of a *staticShot* element, if one has been supplied in the XML dataset. If not, or if the StaticShot has not been initialised from a DOM, it returns an empty string.

Returns

The *refID* string is returned by reference.

#### 4.26.3.11 bool hasInternalValues ( ) const

This function indicates whether a *staticShot* element of a DAVE-ML dataset includes *internalValues*.

Returns

A boolean variable, 'true' if the *staticShot* includes internal values.

#### 4.26.3.12 const bool& hasProvenance ( ) const [inline]

This function indicates whether a *staticShot* element of a DAVE-ML dataset includes either *provenance* or *provenanceRef*.

Returns

A boolean variable, 'true' if the *staticShot* includes a provenance, defined either directly or by reference.

See also

Provenance

#### 4.26.3.13 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition* )

An uninitialised instance of StaticShot is filled with data from a particular *staticShot* element within a DOM by this function. If another *staticShot* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure, since optional elements may not be replaced.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *staticShot* component within the DOM. |

#### 4.26.3.14 void verifyStaticShot ( Janus * *janus* )

This function uses the contents of a *staticShot* element within a DAVE-ML compliant dataset to verify the functional relationships within the remainder of the dataset. It is called as part of the Janus instantiation process to flag problems in the dataset content. It should not normally be called directly by other classes. The results of the tests performed are placed in internal arrays for access by other Janus functions.

See also

getInvalidVariableCount
getInvalidVariable

The documentation for this class was generated from the following files:

- StaticShot.h
- StaticShot.cpp

## 4.27 Uncertainty Class Reference

`#include <Uncertainty.h>`

Inherits XmlElementDefinition.

### Public Types

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const BoundsList & getBounds () const
- const correlationPairList & getCorrelation () const
- const dstoute::aStringList & getCorrelationVarList () const

- const UncertaintyEffect & getEffect () const
- const size_t & getNumSigmas () const
- const UncertaintyPdf & getPdf () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- const bool & isSet () const
- void setBoundsSize (const int n)
- void setPdf (const UncertaintyPdf &pdf)
- Uncertainty ()
- Uncertainty (const DomFunctions::XmlNode &elementDefinition, Janus *janus)

### 4.27.1 Detailed Description

A Uncertainty instance holds in its allocated memory alphanumeric data derived from a *uncertainty* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The element is used in function and parameter definitions to describe statistical variance in the possible value of that function or parameter value. Only Gaussian (normal) or uniform distributions of continuous random variable distribution functions are supported. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Uncertainty class is only used within the janus namespace, and should only be referenced indirectly through the Janus class.

a possible usage is:

```
Janus test( xmlFileName );
int nv = test.getNumberOfVariables();
Uncertainty::UncertaintyPdf uncertaintyPdf;
for ( int i = 0 ; i < nv ; i++ ) {
  VariableDef& variableDef = test.getVariableDef( i );
  cout << " Variable        : "
       << test.getVariableID( i ) << endl
       << "        Value              = "
       << test.getVariableByIndex( i ) << endl
       << "        uncertaintyPdf     = ";
  Uncertainty::UncertaintyPdf& uncertaintyPdf =
    variableDef.getUncertainty().getPdf( );
  switch( uncertaintyPdf ) {
  case Uncertainty::UNIFORM_PDF:
    cout << "UNIFORM_PDF";
    break;
  case Uncertainty::NORMAL_PDF:
    cout << "NORMAL_PDF";
    break;
  case Uncertainty::UNKNOWN_PDF:
    cout << "UNKNOWN_PDF";
    break;
  case Uncertainty::ERROR_PDF:
    cout << "ERROR_PDF";
    break;
  default:
    break;
  }
}
```

### 4.27.2 Member Enumeration Documentation

#### 4.27.2.1 enum UncertaintyPdf

This enum defines the probability distribution functions that may be found in a DAVE-ML compliant dataset.

Enumerator

***NORMAL_PDF*** A normal or Gaussian probability distribution, defined in terms of its mean and standard deviation.

***UNIFORM_PDF*** A uniform or constant probability distribution, defined in terms of the bounds of the interval over which it applies.

***UNKNOWN_PDF*** A probability distribution that has not been specified in terms of the previous two allowable distributions.

***ERROR_PDF*** Error flag, generally associated with incompatible combinations of PDFs within the XML dataset.

### 4.27.3 Constructor & Destructor Documentation

#### 4.27.3.1 Uncertainty ( )

The empty constructor can be used to instance the Uncertainty class without supplying the DOM *uncertainty* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *uncertainty* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.27.3.2 Uncertainty ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

The constructor, when called with an argument pointing to a *uncertainty* element within a DOM, instantiates the Uncertainty class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *uncertainty* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to evaluate bounds with a functional dependence on the instance state. |

### 4.27.4 Member Function Documentation

#### 4.27.4.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *Uncertainty* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.27.4.2 const BoundsList& getBounds ( ) const [inline]

This function provides access to the *bounds* vector of the Uncertainty instance. A Gaussian PDF will have a vector length of one, and a uniform PDF will have a vector length of one or two, depending on the bounds symmetry or asymmetry about the variable value.

Returns

A list of double precision values, containing one or two elements, is returned by reference.

#### 4.27.4.3 const correlationPairList& getCorrelation ( ) const [inline]

This function applies only to Gaussian PDF uncertainties. It allows access to an array of indices and coefficients for variables whose Gaussian uncertainties are correlated with the uncertainty of the variable associated with this Uncertainty instance.

The correlation coefficients, which indicate the degree of non-randomness in the relationship between the variable associated with this Uncertainty instance and other variableDefs instances, are stored together with the index to the variableDef from the global list as a pair; that is, the variableDef index and the associated correlation coefficient are combined as a *correlationPair* having the form *pair<size_t, double>* with the first element being the index and the second the coefficient. This approach replaces the *correlation* and *correlatesWith* arrays that contained the same information separately.

Returns

The list of correlation pairs for this Uncertainty instance is returned as a reference to a vector of correlation pairs.

#### 4.27.4.4 const dstoute::aStringList& getCorrelationVarList ( ) const [inline]

This function applies only to Gaussian PDF uncertainties. It allows access to a list of the identifiers, *varIDs*, for variables that are correlated with the variable associated with this Uncertainty instance. It is used internally within the class when initialising the correlation

pairs - variable index and coefficient. This function permits external applications to retrieve this data for information.

Returns

The correlation varID list is returned by reference as a list of strings.

#### 4.27.4.5 const UncertaintyEffect& getEffect ( ) const [inline]

This function returns the *effect* of the referenced *uncertainty* element. It indicates how bounds should be interpreted (e.g. additive, multiplicative, percentage or absolute uncertainty).

Returns

An UncertaintyEffect enum. Where the Uncertainty instance has not been initialised, UNKNOWN_UNCERTAINTY is returned.

#### 4.27.4.6 const size_t& getNumSigmas ( ) const [inline]

This function applies only to Gaussian PDF uncertainties. It indicates how many standard deviations are represented by the corresponding bounds magnitude.

Returns

The *numSigmas* attribute of the *uncertainty* instance is returned as an integer.

#### 4.27.4.7 const UncertaintyPdf& getPdf ( ) const [inline]

This function indicates whether the referenced Uncertainty instance describes Gaussian, uniform or unknown uncertainty. In the case of a variable that has its uncertainty defined directly through an *uncertainty* child element (see hasUncertainty()), this function returns that definition's uncertainty type. For a *variableDef* that does not include an *uncertainty* child element, this function returns an uncertainty type determined by considering all variables and functions on which this variable depends.

Returns

An UncertaintyPdf enum. Where the Uncertainty instance has not been initialised, UNKNOWN_PDF is returned.

#### 4.27.4.8 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

An uninitialised instance of Uncertainty is filled with data from a particular *uncertainty* element within a DOM by this function. If another *uncertainty* element pointer is supplied to an instance that has already been initialised, the instance will be re-initialised with the new data. However, this is not a recommended procedure.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *uncertainty* component within the DOM. |

Parameters

| | |
|---|---|
| *janus* | is a pointer to the owning Janus instance, used within this class to evaluate bounds with a functional dependence on the instance state. |

#### 4.27.4.9 const bool& isSet ( ) const [inline]

This function is used to determine whether a PDF has been explicitly applied to a variable at the output stage. This is in contrast to a variable whose PDF is propagated through the computations from its independent variable values.

Returns

A boolean value, 'true' if the referenced variable has an explicitly-specified PDF.

#### 4.27.4.10 void setBoundsSize ( const int *n* ) [inline]

This function is used by Janus during initialisation, when setting up uncertainty dependencies. It should not be used by other classes or external programs.

Parameters

| | |
|---|---|
| *n* | sets the length of the vector required for a Bounds instance. Possible values are 1, for symmetric bounds, or 2, for asymmetric bounds. |

#### 4.27.4.11 void setPdf ( const UncertaintyPdf & *pdf* ) [inline]

This function is used by Janus during initialisation, when setting up uncertainty dependencies. It should not be used by other classes or external programs.

Parameters

| | |
|---|---|
| *pdf* | describes the type of probability associated with this Uncertainty instance, as derived from its antecedent variables during the Janus initialization process. |

The documentation for this class was generated from the following files:

- Uncertainty.h
- Uncertainty.cpp

## 4.28 UngriddedTableDef Class Reference

`#include <UngriddedTableDef.h>`

Inherits XmlElementDefinition.

### Public Member Functions

- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const std::vector< std::vector< double > > & getData () const
- const std::vector< int > & getDataPointModID () const
- const size_t & getDataTableColumnCount () const
- const std::vector< std::vector< size_t > > & getDelaunay () const
- const std::vector< double > & getDependentData (const size_t &dataColumn=0) const
- const dstoute::aString & getDescription () const
- const DomFunctions::XmlNode & getDOMElement ()
- const dstomath::DMatrix & getIndependentData () const
- const size_t & getIndependentVarCount () const
- const dstoute::aString & getName () const
- const Provenance & getProvenance () const
- Uncertainty & getUncertainty ()
- const dstoute::aString & getUnits () const
- const dstoute::aString & getUtID () const
- const bool & hasProvenance () const
- const bool & hasUncertainty () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)
- UngriddedTableDef ()
- UngriddedTableDef (const DomFunctions::XmlNode &elementDefinition, Janus *janus)

### 4.28.1 Detailed Description

A UngriddedTableDef instance holds in its allocated memory alphanumeric data derived from a *ungriddedTableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes points that are not in an orthogonal grid pattern; thus, the independent variable coordinates are specified for each dependent variable value. The table data point values are specified as comma-separated values in floating-point notation. Associated alphanumeric identification and cross-reference data are also included in the instance.

The UngriddedTableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

Janus exists to abstract data form and handling from a modelling process. Therefore, in normal computational usage, it is unnecessary and undesirable for a calling program to even be aware of the existence of this class. However, functions do exist to access UngriddedTableDef contents directly, which may be useful during dataset development. A possible usage might be:

```
Janus test( xmlFileName );
const vector<UngriddedTableDef>& ungriddedTableDef =
  test.getUngriddedTableDef();
for ( int i = 0 ; i < ungriddedTableDef.size() ; i++ ) {
  cout << " Ungridded table " << i << " :\n"
       << "   name        = " << ungriddedTableDef.at( i ).getName() << "\n"
       << "   gtID        = " << ungriddedTableDef.at( i ).getGtID() << "\n"
       << "   units       = " << ungriddedTableDef.at( i ).getUnits() << "\n"
       << "   description = " << ungriddedTableDef.at( i ).getDescription()
       << "\n";
}
```

### 4.28.2 Constructor & Destructor Documentation

#### 4.28.2.1 UngriddedTableDef ( )

The empty constructor can be used to instance the UngriddedTableDef class without supplying the DOM *ungriddedTableDef* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing a *ungriddedTableDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.28.2.2 UngriddedTableDef ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

The constructor, when called with an argument pointing to a *ungriddedTableDef* element within a DOM, instantiates the UngriddedTableDef class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *ungriddedTableDef* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

### 4.28.3 Member Function Documentation

#### 4.28.3.1 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *UngriddedTableDef* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.28.3.2 const std::vector< std::vector<double> >& getData ( ) const [inline]

This function provides access to the vector of data points associated with an UngriddedTableDef instance. Each element of the vector is itself a vector of double precision values representing the contents of a *dataPoint* element from the DOM.

Returns

A vector of *dataPoint* vectors is returned by reference.

#### 4.28.3.3 const std::vector<int>& getDataPointModID ( ) const [inline]

This function provides access to the list of modification record indices associated with each of the data point records defined for this *ungriddedTable* instance.

Returns

A vector of indices (int) listing the modification record indices of the ungridded table dataPoint records is returned by reference.

#### 4.28.3.4 const size_t& getDataTableColumnCount ( ) const [inline]

This function provides returns the count of columns making up the ungridded table.

Returns

The count (size_t) of data columns is returned by reference.

#### 4.28.3.5 const std::vector< std::vector<size_t> >& getDelaunay ( ) const [inline]

This function provides access to the matrix of delaunay simplex vertices associated with an UngriddedTableDef instance.

Returns

A vector of vectors of simplex vertex indices is returned by reference.

#### 4.28.3.6 const std::vector<double>& getDependentData ( const size_t & *dataColumn* = *0* ) const [inline]

This function provides access to the list of dependent data for a nominated dependent data column of this ungridded table.

Parameters

| | |
|---|---|
| *dataColumn* | the index of the dependent data column starting from 0. A default column entry of 0 is used if this parameter is not provided. |

Returns

A vector of data (double) listing the dependent values for the ungridded table dependent data column of interest is returned by reference.

#### 4.28.3.7 const dstoute::aString& getDescription ( ) const [inline]

This function provides access to the *description* child of the *ungriddedTableDef* element represented by this UngriddedTableDef instance. An *ungriddedTableDef's* optional *description* child element consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description. If no description is specified in the XML dataset, or the UngriddedTableDef has not been initialised from the DOM, an empty string is returned.

Returns

The *description* string is returned by reference.

#### 4.28.3.8 const DomFunctions::XmlNode& getDOMElement ( ) [inline]

This function provides access to the document object model element, as a pointer, associated with this instance of the UngriddedTableDef. It is used internally within Janus when instantiating a DAVE-ML compliant XML file that contains ungridded tables. It should not be used by external applications as the pointer will be invalidated once a file has been successfully instantiated, and therefore may cause the external application to fail.

Returns

A pointer (DomFunctions::XmlNode) to the DOM element associated with this instance of the UngriddedTableDef.

#### 4.28.3.9 const dstomath::DMatrix& getIndependentData ( ) const [inline]

This function provides access to the independent data for this ungridded table.

Returns

A matrix of data (double) listing the independent values for the ungridded table is returned by reference.

#### 4.28.3.10 const size_t& getIndependentVarCount ( ) const [inline]

This function provides access to the *independentVarCount* attribute of the *ungriddedTableDef* represented by this UngriddedTableDef instance. An ungridded table's *independentVarCount* attribute is a count of the number of independent data variables defined in the *dataPoint* elements. If the instance has not been initialised from a DOM, or if no *independentVarCount* attribute is present, a zero value is returned.

Returns

The *independentVarCount* value is returned by reference.

#### 4.28.3.11 const dstoute::aString& getName ( ) const [inline]

This function provides access to the *name* attribute of a *ungriddedTableDef*. The *name* attribute is optional. If the ungridded table has no name attribute or has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is returned by reference.

#### 4.28.3.12 const Provenance& getProvenance ( ) const [inline]

This function provides access to the Provenance instance associated with a UngriddedTableDef instance. There may be zero or one of these elements for each ungridded table in a valid dataset.

Returns

The Provenence class instance is returned by reference.

See also

Provenance

#### 4.28.3.13 Uncertainty& getUncertainty ( ) [inline]

This function provides access to the Uncertainty instance associated with a UngriddedTableDef instance. There may be zero or one of these elements for each *ungriddedTableDef* in a valid dataset. For *ungriddedTableDefs* without *uncertainty*, and for *ungriddedTables*, the corresponding UngriddedTableDef instance includes an empty Uncertainty instance.

Returns

The Uncertainty instance is returned by reference.

See also

Uncertainty

#### 4.28.3.14 const dstoute::aString& getUnits ( ) const [inline]

This function provides access to the *units* attribute of the *ungriddedTableDef* represented by this UngriddedTableDef instance. An ungridded table's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS [7] in accordance with SI and other systems. If the instance has not been initialised from a DOM, or if no *units* attribute is present, an empty string is returned.

Returns

The *units* string is returned by reference.

#### 4.28.3.15 const dstoute::aString& getUtID ( ) const [inline]

This function provides access to the *utID* attribute of an *ungriddedTableDef*. This attribute is used for indexing ungridded tables within an XML dataset. Where an *ungriddedTableDef* within the DOM does not contain a *utID* attribute, or where an *ungriddedTable* has been placed in the UngriddedTableDef structure, a *utID* string is generated and inserted in the DOM at initialisation time. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *utID* string is returned by reference.

#### 4.28.3.16 const bool& hasProvenance ( ) const [inline]

This function indicates whether an *ungriddedTableDef* element of a DAVE-ML dataset includes either *provenance* or *provenanceRef*.

Returns

A boolean variable, 'true' if the *ungriddedTableDef* includes a provenance, defined either directly or by reference.

See also

Provenance

#### 4.28.3.17 const bool& hasUncertainty ( ) const [inline]

This function indicates whether a *ungriddedTableDef* element of a DAVE-ML dataset includes an *uncertainty* child element. A variable described by a *ungriddedTableDef* without an *uncertainty* element may still have uncertainty, if it is dependent on other variables or tables with defined uncertainty.

Returns

A boolean variable, 'true' if a ungriddedTableDef definition includes an *uncertainty* child element.

See also

Uncertainty

#### 4.28.3.18 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

An uninitialised instance of UngriddedTableDef is filled with data from a particular *ungriddedTableDef* element within a DOM by this function. If another *ungriddedTableDef* element pointer is supplied to an instance that has already been initialised, data corruption will occur and the entire Janus instance will become unusable. This function can also be used with the deprecated *ungriddedTable* element. For backwards compatibility, Janus converts an *ungriddedTable* to the equivalent *ungriddedTableDef* within this function. Where an *ungriddedTableDef* or *ungriddedTable* lacks a *utID* attribute, this function will generate a random *utID* string for indexing within the Janus class.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of an *ungriddedTableDef* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

The documentation for this class was generated from the following files:

- UngriddedTableDef.h
- UngriddedTableDef.cpp

## 4.29 VariableDef Class Reference

`#include <VariableDef.h>`

Inherits XmlElementDefinition.

### Public Types

### Public Member Functions

- void addDependentVarRef (const int &ix)
- void exportDefinition (DomFunctions::XmlNode &documentElement)
- const dstoute::aString & getAlias () const
- size_t getAncestorCount () const

- const std::vector< size_t > & getAncestorsRef () const
- const dstoute::aString & getAxisSystem () const
- double getCorrelationCoefficient (const size_t &index)
- std::vector< size_t > & getDependentVarRef ()
- const dstoute::aString & getDescription () const
- const DimensionDef & getDimension () const
- const DomFunctions::XmlNode & getDOMElement ()
- int getFunctionRef () const
- size_t getIndependentVarCount () const
- const std::vector< size_t > & getIndependentVarRef () const
- const double & getInitialValue () const
- const dstomath::DMatrix & getMatrix ()
- const double & getMaxValue () const
- const double & getMinValue () const
- const Model & getModel () const
- const dstoute::aString & getName () const
- const double & getOutputScaleFactor () const
- const Provenance & getProvenance () const
- const dstoute::aString & getSign () const
- const dstoute::aString & getStringValue ()
- const dstoute::aString & getSymbol () const
- const VariableType & getType () const
- Uncertainty & getUncertainty ()
- double getUncertaintyValue (const size_t &numSigmas)
- double getUncertaintyValue (const bool &isUpperBound)
- const dstoute::aString & getUnits () const
- double getValue () const
- double getValueMetric () const
- double getValueSI () const
- VariableFlag getVariableFlag () const
- const dstoute::aString & getVarID () const
- const dstomath::DVector & getVector ()
- const bool & hasDescendantsRefs ()
- const bool & hasDimension () const
- const bool & hasProvenance () const
- const bool & hasUncertainty () const
- void initialiseDefinition (const DomFunctions::XmlNode &elementDefinition, Janus *janus)

- const bool & isControl () const
- const bool & isCurrent ()
- const bool & isDisturbance () const
- const bool & isInput () const
- bool isMatrix ()
- const bool & isOutput () const
- const bool & isState () const
- const bool & isStateDeriv () const
- const bool & isStdAIAA () const
- bool isValue ()
- bool isVector ()
- void setAncestorsRef (const std::vector< size_t > &ancestorsRef)
- void setDescendantsRef (const std::vector< size_t > &descendantsRef)
- void setForced (bool isForced)
- void setForceUseOfMatrixCode (bool useMatrixOps=true)
- void setFunctionRef (const int &functionRef)
- void setHasUncertainty (const bool &hasUncertaintyArg)
- void setMathMLDependencies ()
- void setNotCurrent ()
- void setOutputScaleFactor (const double &factor)
- void setType (const VariableType &variableType)
- void setValue (const double &x, bool isForced=false)
- void setValue (const dstomath::DVector &x, bool isForced=false)
- void setValue (const dstomath::DMatrix &x, bool isForced=false)
- void setValueMetric (const double &xSI)
- void setValueSI (const double &xSI)
- void setVarIndex (const int &index)
- VariableDef ()
- VariableDef (const DomFunctions::XmlNode &elementDefinition, Janus *janus)

### 4.29.1 Detailed Description

A VariableDef instance holds in its allocated memory alphanumeric data derived from a *variableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes descriptive, alphanumeric identification and cross-reference data, and may include a calculation process tree for variables computed through MathML.

The VariableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

To determine the characteristics of a dataset's variables, typical usage is:

```
Janus test( xmlFileName );
vector<VariableDef> variableDef = test.getVariableDef();
for ( size_t i = 0 ; i < variableDef.size() ; i++ ) {
  cout << "  Variable " << i << " : \n"
       << "   ID           : "
       << variableDef.at( i ).getVarID() << "\n"
       << "   Name         : "
       << variableDef.at( i ).getName() << "\n"
       << "   Description  : "
       << variableDef.at( i ).getDescription() << "\n"
       << "   Units        : "
       << variableDef.at( i ).getUnits() << "\n" << "   Type         : ";
  switch ( variableDef.at( i ).getType() ) {
  case VariableDef::ISINPUT:
    cout << "ISINPUT \n";
    break;
  case VariableDef::ISCONTROL:
    cout << "ISCONTROL \n";
    break;
  case VariableDef::ISDISTURBANCE:
    cout << "ISDISTURBANCE \n";
    break;
  case VariableDef::ISOUTPUT:
    cout << "ISOUTPUT \n";
    break;
  case VariableDef::FUNCTION:
    cout << "FUNCTION \n";
    break;
  case VariableDef::FUNCTION_INTERNAL:
    cout << "FUNCTION_INTERNAL \n";
    break;
  case VariableDef::FUNCTION_OUTPUT:
    cout << "FUNCTION_OUTPUT \n";
    break;
  case VariableDef::MATHML:
    cout << "MATHML \n";
    break;
  case VariableDef::MATHML_INTERNAL:
    cout << "MATHML_INTERNAL \n";
    break;
  case VariableDef::MATHML_OUTPUT:
    cout << "MATHML_OUTPUT \n";
    break;
  default:
    cout << "\n";
  }
  cout << "   Axis System  : "
       << variableDef.at( i ).getAxisSystem() << "\n"
       << "   Initial Value: "
       << variableDef.at( i ).getInitialValue() << "\n" << "\n";
}
```

### 4.29.2 Member Enumeration Documentation

#### 4.29.2.1 enum VariableFlag

This enum is deprecated, and is only used by deprecated functions. In new programs, the variable characteristics it describes should be determined by direct interrogation of the related VariableDef instance. It is used by calling programs to indicate the characteristics of a variable relative to its use in equations of motion. One of these enums may be accessed for each

VariableDef to describe the associated variable.

Enumerator

***ISSTATE*** This *variableDef* represents a state variable in a dynamic model, either the output of an integrator (for continuous models) or a discretely updated state (for discrete models).

***ISSTATEDERIV*** This *variableDef* represents a state derivative variable in a dynamic model, for continuous models only.

***ISSTDAIAA*** This *variableDef* name is in accordance with "Standard Simulation Variable Names", Annex 1 of "Standards for the Exchange of Simulation Data", a draft AIAA standard. The name should be recognizable exterior to this class by code complying with the standard.

***ISSTATE_STDAIAA*** This *variableDef* represents a state variable, and is named in accordance with "Standard Simulation Variable Names".

***ISSTATEDERIV_STDAIAA*** This *variableDef* represents a state derivative variable, and is named in accordance with "Standard Simulation Variable Names".

***ISERRORFLAG*** Used as a flag, indicates function or variable index out of range.

#### 4.29.2.2 enum VariableType

This enum lists the types of variables that may be included in a DAVE-ML compliant XML dataset, based on the manner in which the variable value is determined. Calling programs may use it to determine whether a variable is an input, or an output of various types. Each VariableDef instance contains one of these enums, set during instantiation, to indicate the source of the associated variable.

Enumerator

***FUNCTION*** This *variableDef* is referenced as a dependent variable by a *function* definition, and its value will therefore be determined by a function evaluation, using either gridded or ungridded data. It will be available as a Janus output.

***FUNCTION_INTERNAL*** This *variableDef* is referenced as a dependent variable by a *function* definition, and its value will therefore be determined by a function evaluation, using either gridded or ungridded data. It is also referenced as an independent variable by another variable evaluation, and is not available as a Janus output.

***FUNCTION_OUTPUT*** This *variableDef* is the result of a gridded or ungridded function evaluation, but is also used as an independent variable for another computation. It also has been explicitly defined as an output by its child node.

***MATHML*** This *variableDef* includes a *calculation* child element, and its value will therefore be determined by a MathML function evaluation. It will be available as a Janus output.

***MATHML_INTERNAL*** This *variableDef* includes a *calculation* child element, and its value will therefore be determined by a MathML function evaluation. It is also referenced as an independent variable by another computation, and is not available as a Janus output.

***MATHML_OUTPUT*** This *variableDef* is the result of a MathML computation, but is also used as an independent variable for another computation. It also has been

explicitly defined as an output by its child node.

***SCRIPT*** This *variableDef* includes a *calculation* child element, and its value will therefore be determined by a script function evaluation. It will be available as a Janus output.

***SCRIPT_INTERNAL*** This *variableDef* includes a *calculation* child element, and its value will therefore be determined by a script function evaluation. It is also referenced as an independent variable by another computation, and is not available as a Janus output.

***SCRIPT_OUTPUT*** This *variableDef* is the result of a script computation, but is also used as an independent variable for another computation. It also has been explicitly defined as an output by its child node.

***ARRAY*** This *variableDef* is defined as either a vector or a matrix.

***ARRAY_INTERNAL*** This *variableDef* is defined as either a vector or a matrix. It is available as an independent variable for another variable. It has not explicitly been defined as an output variable, and therefore, is not available as a Janus output.

***ARRAY_OUTPUT*** This *variableDef* is defined as either a vector or a matrix. It is available as an independent variable for another variable. It has explicitly been defined as an output variable, and therefore, is available as a Janus output.

***MODEL*** This *variableDef* is defined as a dynamic system model.

***MODEL_INTERNAL*** This *variableDef* is defined as a dynamics system model. It is available as an independent variable for another variable. It has not explicitly been defined as an output variable, and therefore, is not available as a Janus output.

***MODEL_OUTPUT*** This *variableDef* is defined as a dynamic system model. It is available as an independent variable for another variable. It has explicitly been defined as an output variable, and therefore, is available as a Janus output.

***ISINPUT*** This *variableDef* has none of the possible output attributes and should be treated as an input.

***ISCONTROL*** This *variableDef* has none of the possible output attributes and should be treated as a control.

***ISDISTURBANCE*** This *variableDef* has none of the possible output attributes and should be treated as a disturbance.

***ISOUTPUT*** This *variableDef* is explicitly defined as an output by its child node, **and** is not the product of either a tabulated function or MathML evaluation.

### 4.29.3 Constructor & Destructor Documentation

#### 4.29.3.1 VariableDef ( )

The empty constructor can be used to instance the VariableDef class without supplying the DOM *variableDef* element from which the instance is constructed, but in this state is not useful for any class functions. It is necessary to populate the class from a DOM containing an *variableDef* element before any further use of the instanced class.

This form of the constructor is principally for use within higher level instances, where memory needs to be allocated before the data to fill it is specified.

See also

initialiseDefinition

#### 4.29.3.2 VariableDef ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

The constructor, when called with an argument pointing to a *variableDef* element within a DOM, instantiates the VariableDef class and fills it with alphanumeric data from the DOM.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *variableDef* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

### 4.29.4 Member Function Documentation

#### 4.29.4.1 void addDependentVarRef ( const int & *ix* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during initialisation, setting up cross-references of immediately dependent variables. Use in other circumstances may result in data corruption.

Parameters

| | |
|---|---|
| *ix* | is a VariableDef index associated with a dependent variable within the Janus instance, computed by Janus during the final stages of initialisation for each VariableDef and passed to it for use during computation. |

#### 4.29.4.2 void exportDefinition ( DomFunctions::XmlNode & *documentElement* )

This function is used to export the *variableDef* data to a DAVE-ML compliant XML dataset file as defined by the DAVE-ML document type definition (DTD).

Parameters

| | |
|---|---|
| *documentElement* | an address to the parent DOM node/element. |

#### 4.29.4.3 const dstoute::aString& getAlias ( ) const [inline]

This function provides access to the optional *alias* attribute of the *variableDef* represented by this VariableDef instance. A variable's *alias* attribute is a string of arbitrary length, but normally short, providing an alternative name (possibly facility specific) for the signal represented by the variable. If no alias is specified in the XML dataset, or the VariableDef has not been initialised from the DOM, an empty string is returned.

Returns

The *alias* string is returned by reference.

#### 4.29.4.4 size_t getAncestorCount ( ) const [inline]

This function returns the number of input variables that ultimately contribute to computation of the value of this variable. If the instance has not been populated from a DOM, zero is returned. In all other cases, there must be zero or more independent variables.

Returns

An integer number, zero or more in a populated instance.

#### 4.29.4.5 const std::vector<size_t>& getAncestorsRef ( ) const [inline]

This function provides access to the indices within the parent Janus instance of those input variables that ultimately contribute to computation of the value of this variable. These are the variables that must be set using setValue() before a valid result can be computed using getValue.

Returns

The vector of ultimate independent variable indices is returned by reference.

#### 4.29.4.6 const dstoute::aString& getAxisSystem ( ) const [inline]

This function provides access to the optional *axisSystem* attribute of the *variableDef* represented by this VariableDef instance. A variable's *axisSystem* attribute is a string of arbitrary length, but normally short, and complying with certain format requirements chosen by DST AD APS . Typical values include "Body" or "Intermediate". If no axis system is specified in the XML dataset, or the VariableDef has not been initialised from the DOM, an empty string is returned.

Returns

The *axisSystem* string is returned by reference.

#### 4.29.4.7 double getCorrelationCoefficient ( const size_t & *index* )

This function provides access to the level of correlation between the Gaussian uncertainty of the variable associated with this VariableDef and the Gaussian uncertainty of any other variable in the XML dataset.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (Janus::getNumberOfVariables() - 1), and selects the other variable to be addressed from the VariableDef vector within the parent Janus instance. |

Returns

The correlation coefficient relating the two variables' uncertainties is returned as a double. Where correlation has not been specified in the XML dataset, the coefficient is returned as zero.

#### 4.29.4.8 std::vector<size_t>& getDependentVarRef ( ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance, providing cross-referencing to those VariableDefs whose output values depend directly on the value within this VariableDef. It may be useful to external programs during XML dataset development.

Returns

A vector of integer indices is returned. If the calling VariableDef has no dependents, the vector will be of zero length.

#### 4.29.4.9 const dstoute::aString& getDescription ( ) const [inline]

This function provides access to the optional *description* of the *variableDef* element represented by this VariableDef instance. A *variableDef's description* child element consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description. If no description is specified in the XML dataset, or the VariableDef has not been initialised from the DOM, an empty string is returned.

Returns

The *description* string is returned by reference.

#### 4.29.4.10 const DimensionDef& getDimension ( ) const [inline]

This function provides access to the DimensionDef instance associated with a VariableDef instance. There may be zero or one of these elements for each variable definition in a valid dataset, either provided directly or cross-referenced through a *dimensionRef* element.

Returns

The DimensionDef instance is returned by reference.

See also

DimensionDef

#### 4.29.4.11 const DomFunctions::XmlNode& getDOMElement ( ) [inline]

This function provides access to the *DOMElement* node associated with the instance of the *variableDef* component. The function is used internally within Janus while initialising a DAVE-ML compliant \ XML dataset source file.

Returns

The *DOMElement* node in a DOM for the *variableDef* component is returned as a pointer.

#### 4.29.4.12 int getFunctionRef ( ) const [inline]

This function allows a calling program to determine the *function*, if any, which a *variableDef's* value is based. It should not be required during normal computations, but is used internally within the Janus instance and may be used by other programs during XML dataset development.

Returns

The index to Function instance, within the top-level Janus instance, upon which this VariableDef instance depends is returned. If the VariableDef is not based on a tabular function, -1 is returned.

#### 4.29.4.13 size_t getIndependentVarCount ( ) const [inline]

This function returns the number of independent variables that directly contribute to computation of the value of this variable. If the instance has not been populated from a DOM, zero is returned. In all other cases, there must be zero or more independent variables.

Returns

An integer number, zero or more in a populated instance.

#### 4.29.4.14 const std::vector<size_t>& getIndependentVarRef ( ) const [inline]

This function provides access to the indices within the parent Janus instance of those independent variables that directly contribute to computation of the value of this variable.

Returns

The vector of directly contributing independent variable indices is returned by reference.

#### 4.29.4.15 const double& getInitialValue ( ) const [inline]

This function provides access to the optional *initialValue* attribute of the *variableDef* represented by this VariableDef instance. A variable's *initialValue* attribute specifies an initial value for the signal represented by the variable. It is also used to store the value of a signal that remains constant. If the initial value is not specified in the XML dataset, or the VariableDef has not been initialised from the DOM, the a NaN value is returned.

Returns

A double precision floating point number is returned.

#### 4.29.4.16 const DMatrix & getMatrix ( )

This function fulfils the basic purpose of the Janus class. It is used during run-time to evaluate the variable associated with this VariableDef. It returns a data matrix based on the current state of the Janus instance, irrespective of variable type. It provides the major functionality of the Janus library. As well as returning the requested data values, it sets contributing values within the parent Janus instance and flags them as valid.

Returns

A double precision matrix (DMatrix) containing the values of the variable after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.17 const double& getMaxValue ( ) const [inline]

This function provides access to the optional *maxValue* attribute of the *variableDef* represented by this VariableDef instance. A variable's *maxValue* attribute provides a maximum boundary value that may be returned when evaluating the *variableDef*.

Returns

A double precision floating point number is returned.

#### 4.29.4.18 const double& getMinValue ( ) const [inline]

This function provides access to the optional *minValue* attribute of the *variableDef* represented by this VariableDef instance. A variable's *minValue* attribute provides a minimum boundary value that may be returned when evaluating the *variableDef*.

Returns

A double precision floating point number is returned.

#### 4.29.4.19 const Model& getModel ( ) const [inline]

This function provides access to the dynamic system model instance associated with a VariableDef instance. There may be zero or one of these elements for each variable definition in a valid dataset.

Returns

The Model instance is returned by reference.

See also

Model

#### 4.29.4.20 const dstoute::aString& getName ( ) const [inline]

This function provides access to the *name* attribute of the *variableDef* element represented by this VariableDef instance. A variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the ANSI/AIAA S-119-2011 standard [2] . If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *name* string is returned by reference.

#### 4.29.4.21 const double& getOutputScaleFactor ( ) const [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during computation, accessing the output scale factor within a VariableDef instance so that values that result from MathML computations can be appropriately scaled. The whole concept of output scale factors is fraught with problems, and they should not be used unless absolutely necessary.

Returns

The current multiplicative constant to be applied to this variable during computation of its value is returned as a double precision number.

#### 4.29.4.22 const Provenance& getProvenance ( ) const [inline]

This function provides access to the Provenance instance associated with a VariableDef instance. There may be zero or one of these elements for each variable definition in a valid dataset, either provided directly or cross-referenced through a *provenanceRef* element.

Returns

The Provenance instance is returned by reference.

See also

Provenance

#### 4.29.4.23 const dstoute::aString& getSign ( ) const [inline]

This function provides access to the optional *sign* attribute of the *variableDef* represented by this VariableDef instance. A variable's *sign* attribute is a string of arbitrary length, but normally short, describing the sign convention for the signal represented by the variable. Typical values include "TED +ve" or "LWD +ve". If no sign convention is specified in the XML dataset, or the VariableDef has not been initialised from the DOM, an empty string is returned.

Returns

The *sign* string is returned by reference.

#### 4.29.4.24 const aString & getStringValue ( )

As an extension to the normal behaviour of a DAVE-ML gridded table, support has been included for managing a table of strings in a similar manner to numeric tabular data. The strings are accessed in the same way as a numeric tabular function. The array of strings may be multi-dimensionsional, and its breakpoints in each dimension should be monotonic sequences of integers ( $1, 2, 3, \ldots n$ is a good choice), where the product of the breakpoint array lengths equals the number of strings. The independent variables must lie within the ranges of their corresponding breakpoints, and must be set to require "discrete" interpolation.

The strings can be delimited by any of: tab, newline, comma, semicolon. DO NOT start or end the strings with excess whitespace.

Janus detects a string table by looking for non-numeric characters, so a table consisting entirely of numeric data will never be detected as a string.

Returns

The required row of the string table, selected based on the current input state of the parent Janus instance, is returned by reference.

#### 4.29.4.25 const dstoute::aString& getSymbol ( ) const [inline]

This function provides access to the optional *symbol* attribute of the *variableDef* represented by this VariableDef instance. A variable's *symbol* attribute contains a Unicode representation of the symbol associated with a signal represented by the variable. A typical example might be $\alpha$ associated with angle of attack. If no symbol is specified in the XML dataset, or the VariableDef has not been initialised from the DOM, a blank Unicode character is returned.

Returns

The *symbol* Unicode character is returned by reference.

#### 4.29.4.26 const VariableType& getType ( ) const [inline]

A variable that is specified as an output, a function evaluation result, or a MathML function should not normally have its value set directly by the calling program. This function allows the caller to determine a variable's status in this regard.

Returns

The VariableType is returned on successful completion.

#### 4.29.4.27 Uncertainty& getUncertainty ( ) [inline]

This function provides access to the Uncertainty instance associated with a VariableDef instance. There may be zero or one *uncertainty* element for each *variableDef* in a valid dataset. For *variableDefs* without *uncertainty*, the corresponding VariableDef instance includes an empty Uncertainty instance.

Returns

The Uncertainty instance is returned by reference.

See also

Uncertainty

#### 4.29.4.28 double getUncertaintyValue ( const size_t & *numSigmas* ) [inline]

This function is used during run-time to evaluate the Gaussian uncertainty of the variable associated with this VariableDef. It returns a value based on the current state of the Janus instance. It supplements the major functionality of the Janus library. As well as returning the requested value, it sets contributing values within the parent Janus instance and flags them as valid. A variable with uniform uncertainty set, either directly or indirectly, will return a Gaussian uncertainty of zero. A variable with no uncertainty set, either directly or indirectly, will also return zero.

Returns

A double precision value containing the value of the Gaussian uncertainty after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.29 double getUncertaintyValue ( const bool & *isUpperBound* ) [inline]

This function is used during run-time to evaluate the uniform uncertainty of the variable associated with this VariableDef. It returns a value based on the current state of the Janus instance. It supplements the major functionality of the Janus library. As well as returning the

requested value, it sets contributing values within the parent Janus instance and flags them as valid. A variable with Gaussian uncertainty set, either directly or indirectly, will return a uniform bound of zero. A variable with no uncertainty set, either directly or indirectly, will also return zero.

Returns

A double precision value containing the value of the uniform uncertainty after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.30 const dstoute::aString& getUnits ( ) const [inline]

This function provides access to the *units* attribute of the *variableDef* represented by this VariableDef instance. A variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS [7] in accordance with SI and other systems. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *units* string is returned by reference.

#### 4.29.4.31 double getValue ( ) const

This function fulfils the basic purpose of the Janus class. It is used during run-time to evaluate the variable associated with this VariableDef. It returns a value based on the current state of the Janus instance, irrespective of variable type. It provides the major functionality of the Janus library. As well as returning the requested value, it sets contributing values within the parent Janus instance and flags them as valid.

Returns

A double precision value containing the value of the variable after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.32 double getValueMetric ( ) const

This function is an alternative to getValue. It is particularly useful for ensuring that all variables used by a calling program are in consistent units.

With getValueMetric(), fluid volumes will be returned in litres and not m3.

Returns

A double precision value containing the value of the variable expressed in Metric units after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.33 double getValueSI ( ) const

This function is an alternative to getValue. It is particularly useful for ensuring that all variables used by a calling program are in consistent units.

Returns

A double precision value containing the value of the variable expressed in SI units after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.34 VariableDef::VariableFlag getVariableFlag ( ) const

This function is deprecated. In new programs, the variable characteristics it describes should be determined by direct interrogation of the related VariableDef instance. The function allows the caller to determine a variable's status in respect of the flags specified in VariableFlag.

Returns

A copy of the VariableFlag enum.

See also

VariableFlag

#### 4.29.4.35 const dstoute::aString& getVarID ( ) const [inline]

This function provides access to the *varID* attribute of the *variableDef* element represented by this VariableDef instance. A variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", that uniquely defines the variable. It is used for indexing variables within an XML dataset, and provides underlying cross-references for most of the Janus library functionality. If the instance has not been initialised from a DOM, an empty string is returned.

Returns

The *varID* string is returned by reference.

#### 4.29.4.36 const DVector & getVector ( )

This function fulfils the basic purpose of the Janus class. It is used during run-time to evaluate the variable associated with this VariableDef. It returns a data matrix based on the current state of the Janus instance, irrespective of variable type. It provides the major functionality of the Janus library. As well as returning the requested data values, it sets contributing values within the parent Janus instance and flags them as valid.

Returns

A double precision vector (DVector) containing the values of the variable after all relevant computations based on the current input state of the parent Janus instance.

#### 4.29.4.37 const bool& hasDescendantsRefs ( ) [inline]

This function indicates if this VariableDef instance has descendants.

Returns

A boolean variable of, 'true' is returned if the *variableDef* has desendants.

#### 4.29.4.38 const bool& hasDimension ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset includes either *dimensionDef* or *dimensionRef*.

Returns

A boolean variable, 'true' if the *function* includes a dimensionDef, defined either directly or by reference.

See also

DimensionDef

#### 4.29.4.39 const bool& hasProvenance ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset includes either *provenance* or *provenanceRef*.

Returns

A boolean variable, 'true' if the *function* includes a provenance, defined either directly or by reference.

See also

Provenance

#### 4.29.4.40 const bool& hasUncertainty ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset includes an *uncertainty* child element. A variable described by a *variableDef* without an *uncertainty* element may still have uncertainty, if it is dependent on other variables or tables with defined uncertainty.

Returns

A boolean variable, 'true' if a variable definition includes an *uncertainty* child element.

See also

Uncertainty

#### 4.29.4.41 void initialiseDefinition ( const DomFunctions::XmlNode & *elementDefinition*, Janus * *janus* )

An uninitialised instance of VariableDef is filled with data from a particular *variableDef* element within a DOM by this function. If another *variableDef* element pointer is supplied to an instance that has already been initialised, data corruption may occur.

Parameters

| | |
|---|---|
| *elementDefinition* | is an address of a *variableDef* component within the DOM. |
| *janus* | is a pointer to the owning Janus instance, used within this class to set up cross-references depending on the instance state. |

#### 4.29.4.42 const bool& isControl ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been formally designated as a control for the represented model, using the *isControl* child element.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as a model control.

#### 4.29.4.43 const bool& isCurrent ( ) [inline]

This function indicates to the calling function is the variable has been evaluated and its value is current, or whether the variable needs to be re-evaluated.

#### 4.29.4.44 const bool& isDisturbance ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been formally designated as a disturbance to the represented model, using the *isDisturbance* child element.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as a model disturbance.

#### 4.29.4.45 const bool& isInput ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been formally designated as an input to the represented model, using the *isInput* child element.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as a model input.

#### 4.29.4.46 bool isMatrix ( )

Returns true if the VariableDef is a matrix or vector.

#### 4.29.4.47 const bool& isOutput ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been formally designated as an output of the represented model, using the *isOutput* child element.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as a model output.

#### 4.29.4.48 const bool& isState ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been formally designated as a state (i.e. the output of an integrator or a discrete time step computation) of the represented model, using the *isState* child element. Model states need not be formally designated as such.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as a model state.

#### 4.29.4.49 const bool& isStateDeriv ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been fomally designated as a state derivative of the represented model, using the *isStateDeriv* child element. State derivatives need not be formally designated as such.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as a model state derivative.

#### 4.29.4.50 const bool& isStdAIAA ( ) const [inline]

This function indicates whether a *variableDef* element of a DAVE-ML dataset has been formally designated, using the *isStdAIAA* child element, as complying with the AIAA standard naming convention "Standard Simulation Variable Names", Annex 1 of "Standards for the Exchange of Simulation Data". The name should be recognizable exterior to this class by

code complying with the standard. Variable names need be neither compliant with the AIAA convention nor formally designated as such.

Returns

A boolean variable, 'true' if a variable definition includes formal designation as compliant with the AIAA standard naming convention.

#### 4.29.4.51 bool isValue ( )

Returns true if the VariableDef is a single double value.

#### 4.29.4.52 bool isVector ( )

Returns true if the VariableDef is a vector.

#### 4.29.4.53 void setAncestorsRef ( const std::vector< size_t > & *ancestorsRef* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during initialisation, setting up cross-references of ultimately contributing variables. Use in other circumstances may result in data corruption.

Parameters

| | |
|---|---|
| *ancestorsRef* | is a vector of VariableDef indices within the Janus instance, computed by Janus during the final stages of initialisation and passed to each VariableDef for use during computation. |

#### 4.29.4.54 void setDescendantsRef ( const std::vector< size_t > & *descendantsRef* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during initialisation, setting up cross-references of ultimately dependent variables. Use in other circumstances may result in data corruption.

Parameters

| | |
|---|---|
| *descendantsRef* | is a vector of *variableDef* indices within the Janus instance, computed by Janus during the final stages of initialisation and passed to each VariableDef instance for use during computation. |

#### 4.29.4.55 void setForced ( bool *isForced* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance, maintaining consistency between variable condition flags as different input variables are set and different output variables are computed. Use in other circumstances may result in data corruption.

#### 4.29.4.56 void setForceUseOfMatrixCode ( bool *useMatrixOps* = *true* ) [inline]

This is used for speed test purposes only. DO NOT USE.

#### 4.29.4.57 void setFunctionRef ( const int & *functionRef* )

This function allows an external application to set the *functionDef* for the VariableDef instance. It is used internally within the Janus instance and may be used by other programs during XML dataset development.

Parameters

| | |
|---|---|
| *functionRef* | The index within the top-level Janus instance of a *function* instance upon which this VariableDef instance depends. |

#### 4.29.4.58 void setHasUncertainty ( const bool & *hasUncertaintyArg* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during initialisation, setting up a varaible's Uncertainty based on the XML dataset content. Use in other circumstances may result in data corruption.

Parameters

| | |
|---|---|
| *hasUncertaintyArg* | is a Boolean indication of the presence of an *uncertainty* element associated with a *variableDef* in an XML dataset. |

#### 4.29.4.59 void setMathMLDependencies ( )

The function should not be used by external programs. It is designed for use within a Janus instance during initialisation, setting up *MathML* cross-references to *variable* elements defined using the *'ci'* tag.

#### 4.29.4.60 void setNotCurrent ( ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance, maintaining consistency between variable condition flags as different input variables are set and different output variables are computed. Use in other circumstances may result in data corruption.

#### 4.29.4.61 void setOutputScaleFactor ( const double & *factor* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during computation, changing the output scale factor within a VariableDef instance so that values that result from MathML computations can be appropriately scaled.

Care should be taken when using this function as use by an external program could result in corruption of data.

Parameters

| | |
|---|---|
| *factor* | is a multiplicative constant to be applied to the computed value of this variable during the computation process. |

#### 4.29.4.62 void setType ( const VariableType & *variableType* ) [inline]

This function is provided for use with the Carna store modelling library, which requires the capability to reset a VariableDef type attribute.

Parameters

| | |
|---|---|
| *variableType* | the type enumeration defined for this variable definition |

#### 4.29.4.63 void setValue ( const double & *x*, bool *isForced* = `false` )

This function provides the means to set the current value of the variable associated with this VariableDef. It is the basic means permitting a calling program to pass an independent variable's state to a Janus instance prior to function evaluation. This function will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *x* | is the double precision scalar value, vector or matrix to which the current value of the indexed variable will be set. |
| *isForced* | a flag indicating whether to force the variable to be set to the value and thereby override other flags detailing the state of a variable |

#### 4.29.4.64 void setValue ( const dstomath::DVector & *x*, bool *isForced* = `false` )

This function provides the means to set the current value of the variable associated with this VariableDef. It is the basic means permitting a calling program to pass an independent variable's state to a Janus instance prior to function evaluation. This function will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *x* | is a vector of double precision values (DVector), to which this variable will be set. |
| *isForced* | a flag indicating whether to force the variable to be set to the value and thereby override other flags detailing the state of a variable |

#### 4.29.4.65 void setValue ( const dstomath::DMatrix & *x*, bool *isForced* = `false` )

This function provides the means to set the current value of the variable associated with this VariableDef. It is the basic means permitting a calling program to pass an independent variable's state to a Janus instance prior to function evaluation. This function will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *x* | is a matrix of double precision values (DMatrix), to which this variable will be set. |
| *isForced* | a flag indicating whether to force the variable to be set to the value and thereby override other flags detailing the state of a variable |

#### 4.29.4.66 void setValueMetric ( const double & *xSI* )

This function is an alternative to setValue. It will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

With setValueMetric(), fluid volumes will be in litres and not m3.

Parameters

| | |
|---|---|
| *xSI* | is the double precision value, expressed in SI units, to which the current value of the indexed variable will be set. |

#### 4.29.4.67 void setValueSI ( const double & *xSI* )

This function is an alternative to setValue. It will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *xSI* | is the double precision value, expressed in SI units, to which the current value of the indexed variable will be set. |

#### 4.29.4.68 void setVarIndex ( const int & *index* ) [inline]

This function should not be used by external programs. It is designed for use within a Janus instance during initialisation, setting up the self-reference for each variable. This is necessary for Gaussian correlation determination. Use in other circumstances will result in data corruption.

Parameters

| | |
|---|---|
| *index* | is the offset of this VariableDef instance within the Janus instance, set during initialisation and passed to each VariableDef instance for use during computation. |

The documentation for this class was generated from the following files:

- VariableDef.h
- VariableDef.cpp
- VariableDefExprTkParseMathML.cpp
- VariableDefExprTkScript.cpp
- VariableDefLuaScript.cpp

## 4.30 XmlElementDefinition Class Reference

`#include <XmlElementDefinition.h>`

Inherited by Array, Author, Bounds, BreakpointDef, CheckData, DimensionDef, FileHeader, Function, FunctionDefn, GriddedTableDef, InDependentVarDef, Janus, Model, Modification, PropertyDef, Provenance, Reference, Signal, SignalList, StatespaceFn, StaticShot, TransferFn, Uncertainty, UngriddedTableDef, and VariableDef.

### 4.30.1 Detailed Description

This file contains definitions of virtual functions that are used when instantiating a DAVE-ML compliant XML file using Janus. The *XmlElementDefinition* class is inherited by base element classes, such as *FileHeader* and *VariableDef*, which have specific versions of the virtual functions. These function calls are accessed internally within Janus through the *DomFunctions* class and permit abstraction of the process of interacting with the DOM. They do not provide a capability to external applications to interact with the XML encoded data file or the associated DOM.

The documentation for this class was generated from the following file:

- XmlElementDefinition.h

# 5 File Documentation

This section briefly documents each of the files that make up the Janus API library, including their dependencies on other external and Janus modules.

## 5.1 Array.cpp File Reference

```
#include <Ute/aString.h>
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "JanusConstants.h"
#include "Array.h"
```

### Namespaces

- janus

### 5.1.1 Detailed Description

An Array instance holds in its allocated memory alphanumeric data derived from an *array* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file.

It includes entries arranged as follows: Entries for a vector represent the row entries of that vector. Entries for a matrix are specified such that the column entries of the first row are listed followed by column entries for subsequent rows until the base matrix is complete. This sequence is repeated for higher order matrix dimensions until all entries of the matrix are specified.

The Array class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.2 Array.h File Reference

```
#include "XmlElementDefinition.h"
#include <Ute/aString.h>
```

### Classes

- class Array

### Namespaces

- janus

#### 5.2.1 Detailed Description

An Array instance holds in its allocated memory alphanumeric data derived from an *array* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file.

It includes entries arranged as follows: Entries for a vector represent the row entries of that vector. Entries for a matrix are specified such that the column entries of the first row are listed followed by column entries for subsequent rows until the base matrix is complete. This sequence is repeated for higher order matrix dimensions until all entries of the matrix are specified.

The Array class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.3 Author.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "Author.h"
```

### Namespaces

- janus

#### 5.3.1 Detailed Description

An Author instance holds in its allocated memory alphanumeric data derived from an *author* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance may describe an author of a complete dataset, or of a component of a dataset, or of a modification to a dataset. Author contact details may be expressed in either *address* or *contactInfo* forms. The *contactInfo* form is newer, more flexible and generally preferred. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Author class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader, Modification or Provenance classes.

## 5.4 Author.h File Reference

```
#include <Ute/aString.h>
#include "XmlElementDefinition.h"
```

### Classes

- class Author

### Namespaces

- janus

### 5.4.1 Detailed Description

An Author instance holds in its allocated memory alphanumeric data derived from an *author* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance may describe an author of a complete dataset, or of a component of a dataset, or of a modification to a dataset. Author contact details may be expressed in either *address* or *contactInfo* forms. The *contactInfo* form is newer, more flexible and generally preferred. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Author class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader, Modification or Provenance classes.

## 5.5 Bounds.cpp File Reference

```
#include <Ute/aMath.h>
#include <Ute/aString.h>
#include <Ute/aMessageStream.h>
#include "Janus.h"
#include "DomFunctions.h"
#include "Bounds.h"
#include "VariableDef.h"
#include "Function.h"
```

### Namespaces

- janus

### 5.5.1 Detailed Description

A Bounds instance holds in its allocated memory alphanumeric data derived from a *bounds* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The element contains some description of the statistical limits to the values the citing parameter element might take on. This can be in the form of a scalar value, a *variableDef* that provides a functional definition of the bound, a *variableRef* that refers to such a functional definition, or a private table whose elements correlate with those of a tabular function defining the citing parameter. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Bounds class is only used within the janus namespace, and should only be referenced indirectly through the Uncertainty class or through the variable functions within the Janus class.

## 5.6 Bounds.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
```

### Classes

- class Bounds

### Namespaces

- janus

### 5.6.1 Detailed Description

A Bounds instance holds in its allocated memory alphanumeric data derived from a *bounds* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The element contains some description of the statistical limits to the values the citing parameter element might take on. This can be in the form of a scalar value, a *variableDef* that provides a functional definition of the bound, a *variableRef* that refers to such a functional definition, or a private table whose elements correlate with those of a tabular function defining the citing parameter. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Bounds class is only used within the janus namespace, and should only be referenced indirectly through the Uncertainty class or through the variable functions within the Janus class.

## 5.7 BreakpointDef.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "JanusConstants.h"
#include "DomFunctions.h"
#include "BreakpointDef.h"
```

### Namespaces

- janus

### 5.7.1 Detailed Description

A BreakpointDef instance holds in its allocated memory alphanumeric data derived from a *breakpointDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes numeric break points for gridded tables, and associated alphanumeric identification data.

A *breakpointDef* is where gridded table breakpoints are defined; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table. These are separate from function data, and thus they may be reused. The *independentVarPts* element used within some DAVE-ML *functionDefn* elements is equivalent to a *breakpointDef* element, and is also represented as a BreakpointDef within Janus.

The BreakpointDef class is only used within the janus namespace, and should only be referenced through the Janus class.

Janus exists to handle data for a modelling process. Therefore, in normal computational usage it is unnecessary (and undesirable) for a calling program to be aware of the existence of this class. However, functions do exist to access BreakpointDef contents directly, which may be useful during dataset development.

## 5.8 BreakpointDef.h File Reference

```
#include <vector>
#include "XmlElementDefinition.h"
```

### Classes

- class BreakpointDef

### Namespaces

- janus

### 5.8.1 Detailed Description

A BreakpointDef instance holds in its allocated memory alphanumeric data derived from a *breakpointDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes numeric break points for gridded tables, and associated alphanumeric identification data.

A breakpointDef is where gridded table breakpoints are defined; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table. These are separate from function data, and thus they may be reused. The *independentVarPts* element used within some DAVE-ML *functionDefn* elements is equivalent to a *breakpointDef* element, and is also represented as a BreakpointDef within Janus.

The BreakpointDef class is only used within the janus namespace, and should only be referenced through the Janus class.

Janus exists to handle data for a modelling process. Therefore, in normal computational usage it is unnecessary (and undesirable) for a calling program to be aware of the existence of this class. However, functions do exist to access BreakpointDef contents directly, which may be useful during dataset development.

## 5.9 CheckData.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "CheckData.h"
```

### Namespaces

- janus

### 5.9.1 Detailed Description

Check data is used for XML dataset content verification. A CheckData instance holds in its allocated memory alphanumeric data derived from a *checkData* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It will include static check cases, trim

shots, and dynamic check case information. At present only static check cases are implemented, using *staticShot* children of the top-level *checkData* element. The functions within this class provide access to the raw check data, as well as actually performing whatever checks may be done on the dataset using the *checkData*.

The CheckData class is only used within the janus namespace, and should normally only be referenced through the Janus class.

## 5.10 CheckData.h File Reference

```
#include <vector>
#include "XmlElementDefinition.h"
#include "Provenance.h"
#include "StaticShot.h"
```

### Classes

- class CheckData

### Namespaces

- janus

### 5.10.1 Detailed Description

Check data is used for XML dataset content verification. A CheckData instance holds in its allocated memory alphanumeric data derived from a *checkData* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It will include static check cases, trim shots, and dynamic check case information. At present only static check cases are implemented, using *staticShot* children of the top-level *checkData* element. The functions within this class provide access to the raw check data, as well as actually performing whatever checks may be done on the dataset using the *checkData*.

The CheckData class is only used within the janus namespace, and should normally only be referenced through the Janus class.

## 5.11 CheckInputs.h File Reference

```
#include "SignalList.h"
```

### Classes

- class CheckInputs

### Namespaces

- janus

### 5.11.1 Detailed Description

A CheckInputs instance functions as a container for the Signal class through the use of the Signals class. It provides the functions that allow a calling StaticShot instance to access the *signal* elements that define the input values for a check case. A *checkInputs* element must contain *signal* elements that include *signalName* and *signalUnits* elements.

The CheckInputs class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot class.

## 5.12 CheckOutputs.h File Reference

```
#include "SignalList.h"
```

### Classes

- class CheckOutputs

### Namespaces

- janus

### 5.12.1 Detailed Description

A CheckOutputs instance functions as a container for the Signal class through the use of the Signals class. It provides the functions that allow a calling StaticShot instance to access the *signal* elements that define the output values for a check case. A *checkOutputs* element must contain *signal* elements that include *signalName* and *signalUnits* elements.

The CheckOutputs class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot class.

## 5.13 DimensionDef.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "DimensionDef.h"
```

### Namespaces

- janus

### 5.13.1 Detailed Description

A DimensionDef instance holds in its allocated memory alphanumeric data derived from a *dimensionDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes descriptive, alphanumeric identification and cross-reference data.

The DimensionDef class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.14 DimensionDef.h File Reference

```
#include <vector>
#include "XmlElementDefinition.h"
#include "DomFunctions.h"
```

### Classes

- class DimensionDef

### Namespaces

- janus

### 5.14.1 Detailed Description

A DimensionDef instance holds in its allocated memory alphanumeric data derived from a *dimensionDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes descriptive, alphanumeric identification and cross-reference data.

The DimensionDef class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.15 DomFunctions.h File Reference

```
#include <cstring>
#include <stdexcept>
#include <sstream>
#include <Ute/aString.h>
#include <Ute/aMessageStream.h>
#include "JanusConstants.h"
#include "DomTypes.h"
#include "XmlElementDefinition.h"
```

### 5.15.1 Detailed Description

This class contains common functions for interacting with a Document Object Model (DOM) containing data from a DAVE-ML compliant XML dataset source file.

## 5.16 DomTypes.h File Reference

```
#include <Ute/aList.h>
#include "pugixml.hpp"
```

### 5.16.1 Detailed Description

This class contains common types for interacting with a Document Object Model (DOM) containing data from a DAVE-ML compliant XML dataset source file.

## 5.17 ElementDefinitionEnum.h File Reference

### Namespaces

- janus

### 5.17.1 Detailed Description

This file contains enumeration parameters that are used when instantiating a DAVE-ML compliant XML dataset source file from a Document Object Model (DOM).

## 5.18 ExportMathML.cpp File Reference

```
#include <iostream>
#include <Ute/aString.h>
#include "ElementDefinitionEnum.h"
#include "ExportMathML.h"
#include "VariableDef.h"
#include "Janus.h"
```

### 5.18.1 Detailed Description

This class contains functions for exporting mathematics procedures defined using the MathML syntax to a DOM. Data detailing each MathML operation and is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

## 5.19 ExportMathML.h File Reference

```
#include <Ute/aMap.h>
#include "DomTypes.h"
```

### 5.19.1 Detailed Description

This class contains functions for exporting mathematics procedures defined using the MathML syntax to a DOM. Data detailing each MathML operation is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

## 5.20 FileHeader.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "FileHeader.h"
```

### Namespaces

- janus

#### 5.20.1 Detailed Description

A FileHeader instance holds in its allocated memory alphanumeric data derived from the *fileHeader* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. There is always one FileHeader instance for each Janus instance. It requires at least one author, a creation date and a version indicator; optional content are description, references and modification records. The class also provides the functions that allow a calling Janus instance to access these data elements.

The FileHeader class is only used within the janus namespace, and should only be referenced indirectly through the Janus class.

## 5.21 FileHeader.h File Reference

```
#include <vector>
#include "XmlElementDefinition.h"
#include "Author.h"
#include "Provenance.h"
#include "Modification.h"
#include "Reference.h"
```

### Classes

- class FileHeader

### Namespaces

- janus

#### 5.21.1 Detailed Description

A FileHeader instance holds in its allocated memory alphanumeric data derived from the *fileHeader* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. There is always one FileHeader instance for each Janus instance. It requires at least one author, a creation date and a version indicator; optional content are description, references and modification records. The class also provides the functions that allow a calling Janus instance to access these data elements.

The FileHeader class is only used within the janus namespace, and should only be referenced indirectly through the Janus class.

## 5.22 Function.cpp File Reference

```
#include "DomFunctions.h"
#include "Function.h"
#include "Janus.h"
#include <Ute/aMessageStream.h>
```

### Namespaces

- janus

### 5.22.1 Detailed Description

A Function instance holds in its allocated memory alphanumeric data derived from a *function* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Each function has optional description, optional provenance, and either a simple input/output values or references to more complete (possible multiple) input, output, and function data elements.

The Function class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.23 Function.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
#include "Provenance.h"
#include "GriddedTableDef.h"
#include "BreakpointDef.h"
#include "FunctionDefn.h"
#include "InDependentVarDef.h"
```

### Classes

- class Function

### Namespaces

- janus

#### 5.23.1 Detailed Description

A Function instance holds in its allocated memory alphanumeric data derived from a *function* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Each function has optional description, optional provenance, and either a simple input/output values or references to more complete (possible multiple) input, output, and function data elements.

The Function class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.24 FunctionDefn.cpp File Reference

```
#include <stdexcept>
#include <sstream>
#include <iostream>
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "FunctionDefn.h"
#include "Janus.h"
```

### Namespaces

- janus

#### 5.24.1 Detailed Description

A FunctionDefn instance holds in its allocated memory alphanumeric data derived from a *functionDefn* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Each function stores function data elements.

The FunctionDefn class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.25 FunctionDefn.h File Reference

```
#include <Ute/aList.h>
```

```
#include "XmlElementDefinition.h"
```

### Classes

- class FunctionDefn

### Namespaces

- janus

#### 5.25.1 Detailed Description

A FunctionDefn instance holds in its allocated memory alphanumeric data derived from a *functionDefn* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Each function stores function data elements.

The FunctionDefn class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.26 GetDescriptors.cpp File Reference

```
#include <Ute/aString.h>
#include "Janus.h"
```

### Namespaces

- janus

#### 5.26.1 Detailed Description

This code is used during interrogation of an instance the Janus class, and provides the calling program access to the descriptive elements contained in a DOM that complies with the DAVE-ML DTD.

In keeping with the data's descriptive nature, most returns from these functions are strings, although there are a few numerical values and an enum.

## 5.27 GriddedTableDef.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "Janus.h"
#include "JanusUtilities.h"
#include "DomFunctions.h"
#include "GriddedTableDef.h"
#include "BreakpointDef.h"
```

### Namespaces

- janus

### 5.27.1 Detailed Description

A GriddedTableDef instance holds in its allocated memory alphanumeric data derived from a *griddedTableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes points arranged in an orthogonal, multi-dimensional array, where the independent variable ranges are defined by separate breakpoint vectors. The table data point values are specified as comma-separated values in floating-point notation (0.93638E-06) in a single long sequence as if the table had been unravelled with the last-specified dimension changing most rapidly. Gridded tables in DAVE-ML and Janus are stored in row-major order, as in C/C++ (Fortran, Matlab and Octave use column-major order). Line breaks and comments in the XML are ignored. Associated alphanumeric identification and cross-reference data are also included in the instance.

NOTE: The *confidenceBound* entry of the *griddedTable* element is not supported, as it is expected to be deprecated in future version of the DAVE-ML syntax language document type definition.

## 5.28 GriddedTableDef.h File Reference

```
#include <Ute/aList.h>
#include <Ute/aString.h>
#include "XmlElementDefinition.h"
#include "Provenance.h"
#include "Uncertainty.h"
```

### Classes

- class GriddedTableDef

### Namespaces

- janus

### 5.28.1 Detailed Description

A GriddedTableDef instance holds in its allocated memory alphanumeric data derived from a *griddedTableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes points arranged in an orthogonal, multi-dimensional array, where the independent variable ranges are defined by separate breakpoint vectors. The table data point values are specified as comma-separated values in floating-point notation (0.93638E-06) in a single long sequence as if the table had been unravelled with the last-specified dimension changing most rapidly. Gridded tables in DAVE-ML and Janus are stored in row-major order, as in C/C++ (Fortran, Matlab and Octave use column-major order). Line breaks and comments in the XML are ignored. Associated alphanumeric identification and cross-reference data are also included in the instance.

NOTE: The *confidenceBound* entry of the *griddedTable* element is not supported, as it is expected to be deprecated in future version of the DAVE-ML syntax language document type definition.

## 5.29 InDependentVarDef.cpp File Reference

```
#include "JanusConstants.h"
#include "DomFunctions.h"
#include "InDependentVarDef.h"
#include <cfloat>
#include <Ute/aBiMap.h>
#include <Ute/aMessageStream.h>
#include <Ute/aMath.h>
```

### Namespaces

- janus

### 5.29.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to the In-Dependent variable definitions contained in a DOM that complies with the DAVE-ML DTD.

A breakpointDef is where gridded table breakpoints are given. Since these are separate from function data, they may be reused.

bpVals is a set of breakpoints; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table.

## 5.30 InDependentVarDef.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
```

### Classes

- class InDependentVarDef

### Namespaces

- janus

### 5.30.1 Detailed Description

This code is used during initialisation of the Janus class, and provides access to the InDependent variable definitions contained in a DOM that complies with the DAVE-ML DTD.

A breakpointDef is where gridded table breakpoints are given. Since these are separate from function data, they may be reused.

bpVals is a set of breakpoints; that is, a set of independent variable values associated with one dimension of a gridded table of data. An example would be the Mach or angle-of-attack values that define the coordinates of each data point in a two-dimensional coefficient value table.

## 5.31 InternalValues.h File Reference

```
#include "SignalList.h"
```

### Classes

- class InternalValues

### Namespaces

- janus

### 5.31.1 Detailed Description

An InternalValues instance functions as a container for the Signal class, and provides the functions that allow a calling StaticShot instance to access the *signal* elements that define the internal values for a check case. A *internalValues* element must contain *signal* elements that include *varID* (*signalID* is deprecated) elements.

The InternalValues class is only used within the *janus* namespace, and should only be referenced indirectly through the StaticShot class.

## 5.32 Janus.cpp File Reference

```
#include <iostream>
#include <cstdio>
#include <sstream>
#include <stdexcept>
#include "DomFunctions.h"
#include "Janus.h"
#include <Ute/aMessageStream.h>
#include <Ute/aString.h>
#include <Ute/aFile.h>
#include <Ute/aMath.h>
#include <Ute/aCrypt.h>
```

### Namespaces

- janus

### 5.32.1 Detailed Description

Janus performs XML initialisation the pugiXML parser loading the supplied XML file or data buffer a DOM structure. It holds the data structure and accesses it on request, doing interpolation or other computation as required for output. It cleans up on termination.

This header defines all the elements required to use the XML dataset for flight modelling, and should be included in any source code intended to activate an instance of the Janus class.

## 5.33 Janus.h File Reference

```
#include <stdexcept>
#include <vector>
#include <Ute/aString.h>
#include <Ute/aMessageStream.h>
#include "JanusConfig.h"
#include "JanusConstants.h"
#include "XmlElementDefinition.h"
#include "FileHeader.h"
#include "VariableDef.h"
#include "PropertyDef.h"
#include "BreakpointDef.h"
#include "GriddedTableDef.h"
#include "UngriddedTableDef.h"
#include "Function.h"
#include "CheckData.h"
#include "Author.h"
#include "Reference.h"
#include "Provenance.h"
#include "Modification.h"
#include "Uncertainty.h"
```

### Classes

- class Janus

### Namespaces

- janus

### 5.33.1 Detailed Description

Janus performs XML initialisation the pugiXML parser loading the supplied XML file or data buffer a DOM structure. It holds the data structure and accesses it on request, doing interpolation or other computation as required for output. It cleans up on termination.

This header defines all the elements required to use the XML dataset for flight modelling, and should be included in any source code intended to activate an instance of the Janus class.

## 5.34 JanusDeprecated.cpp File Reference

```
#include "Janus.h"
```

### Namespaces

- janus

### 5.34.1 Detailed Description

Janus performs XML initialisation the pugiXML parser loading the supplied XML file or data buffer a DOM structure. It holds the data structure and accesses it on request, doing interpolation or other computation as required for output. It cleans up on termination.

This header defines all the function definitions that were deprecated when Janus was restructured using C++ classes to manage elements defined in within the XML dataset.

These function should not be used in new programs. They are retained to retained legacy code support.

## 5.35 JanusDeprecated.h File Reference

### Enumerations

### Functions

- int applyOutputScaleFactorByIndex (size_t index, const double &factor)
- int applyOutputScaleFactorByVarID (const char *varID, const double &factor)
- double getCorrelationCoefficient (size_t index, size_t indx1, size_t indx2)
- double getCorrelationCoefficient (size_t indx1, size_t indx2)
- const char * getFunctionDefinitionName (size_t index) const
- const char * getFunctionDescription (size_t index) const
- const char * getFunctionName (size_t index) const
- dstoute::aString getHeaderName ()
- const char * getIndependentVariableAxisSystem (size_t indexf, size_t indexv) const
- double getIndependentVariableByIndex (const size_t &indexf, const size_t &indexv)
- const char * getIndependentVariableDescription (size_t indexf, size_t indexv) const
- janus::ExtrapolateMethod getIndependentVariableExtrapolation (size_t indexf, size_t indexv)
- const char * getIndependentVariableID (size_t indexf, size_t indexv) const

- int getIndependentVariableIndex (size_t indexf, const char *varID) const
- janus::InterpolateMethod getIndependentVariableInterpolation (const size_t &indexf, const size_t &indexv)
- const char * getIndependentVariableName (size_t indexf, size_t indexv) const
- int getIndependentVariableOrder (const size_t &indexf, const size_t &indexv) const
- const Provenance & getIndependentVariableProvenance (size_t indexf, size_t indexv) const
- VariableDef::VariableType getIndependentVariableType (const size_t &indexf, const size_t &indexv) const
- double getIndependentVariableUncertainty (const size_t &indexf, const size_t &indexv, const size_t &numSigmas)
- double getIndependentVariableUncertainty (const size_t &indexf, const size_t &indexv, const bool &isUpperBound)
- const char * getIndependentVariableUnits (size_t indexf, size_t indexv) const
- int getNumberOfFunctions () const
- int getNumberOfHeaderProvenances ()
- int getNumberOfIndependentVariables (size_t index) const
- int getNumberOfOutputs () const
- int getNumberOfProvenanceComponents (const char *parentID, size_t index, enum ProvenanceAttribute provenanceAttribute)
- int getNumberOfVariableCorrelations (size_t index)
- size_t getNumberOfVariables () const
- int getNumberOfXmlFileAuthorAddresses (size_t authorNumber)
- int getNumberOfXmlFileAuthorContacts (size_t authorNumber)
- int getNumberOfXmlFileAuthors ()
- int getNumberOfXmlFileModificationAuthorAddresses (size_t index, size_t author)
- int getNumberOfXmlFileModificationAuthorContacts (size_t index, size_t author)
- int getNumberOfXmlFileModificationAuthors (size_t index)
- double getOutputScaleFactorByIndex (size_t index) const
- double getOutputScaleFactorByVarID (const char *varID)
- double getOutputVariable (size_t index)
- const char * getOutputVariable (size_t index, int)
- const char * getOutputVariableAxisSystem (size_t index) const
- double getOutputVariableByVarID (const char *varID)
- const char * getOutputVariableDescription (size_t index) const
- const char * getOutputVariableID (size_t index) const
- int getOutputVariableIndex (const char *varID)
- const char * getOutputVariableName (size_t index) const

- const Provenance & getOutputVariableProvenance (size_t index) const
- VariableDef::VariableType getOutputVariableType (size_t index) const
- double getOutputVariableUncertainty (size_t index, size_t numSigmas)
- double getOutputVariableUncertainty (size_t index, bool isUpperBound)
- const char * getOutputVariableUnits (size_t index) const
- const char * getProvenance (const char *parentID, size_t index, enum ProvenanceAttribute provenanceAttribute, enum AuthorAttribute authorAttribute, size_t authorContactIndex, size_t componentIndex)
- const char * getVariableAxisSystem (size_t index) const
- double getVariableByIndex (size_t index)
- double getVariableByVarID (const char *varID)
- const char * getVariableDescription (size_t index) const
- VariableDef::VariableFlag getVariableFlag (size_t index) const
- const char * getVariableID (size_t index) const
- const char * getVariableName (size_t index) const
- VariableDef::VariableType getVariableType (size_t index) const
- double getVariableUncertainty (size_t index, size_t numSigmas)
- double getVariableUncertainty (size_t index, bool isUpperBound)
- const char * getVariableUnits (size_t index) const
- const char * getXmlFileAuthor (size_t authorNumber, AuthorAttribute authorAttribute, size_t addressNumber=0)
- const char * getXmlFileCreationDate ()
- const char * getXmlFileDescription ()
- const char * getXmlFileModification (size_t index, ModificationAttribute modificationAttribute, size_t authorNumber=0, size_t addressNumber=0)
- int getXmlFileModificationCount ()
- int getXmlFileModificationExtraDocCount (size_t index)
- const char * getXmlFileModificationExtraDocRefID (size_t index, size_t indxRef)
- const char * getXmlFileReference (size_t index, ReferenceAttribute referenceAttribute)
- int getXmlFileReferenceCount ()
- int getXmlFileReferenceIndex (const char *refID)
- const char * getXmlFileVersion ()
- int setIndependentVariableByIndex (size_t indexf, size_t indexv, const double &x)
- int setRsaKeyFileName (const char *fileName, const RsaKeyType keyType)
- int setVariableByID (const char *varID, const double &x)
- int setVariableByIndex (size_t index, const double &x)

### 5.35.1 Detailed Description

Janus performs XML initialisation the pugiXML parser loading the supplied XML file or data buffer a DOM structure. It holds the data structure and accesses it on request, doing interpolation or other computation as required for output. It cleans up on termination.

This header defines all the function definitions that were deprecated when Janus was restructured using C++ classes to manage elements defined in within the XML dataset.

These function should not be used in new programs. They are retained to maintain legacy code support.

## 5.36 LinearInterpolation.cpp File Reference

```
#include <Ute/aMath.h>
#include "InDependentVarDef.h"
#include "Janus.h"
```

### Namespaces

- janus

### 5.36.1 Detailed Description

This private function performs interpolations when all the degrees of freedom for a function are specified as linear or first order polynomial, or for the default condition when interpolationType is not specified.

Given 2^n uniformly gridded values of a function of $n$ variables, provided to the instance of the class by either `setVariableByIndex` or `setVariableByID`, this private function is called by `getOutputVariable` to perform a multi-linear interpolation between the values and returns the result. It maintains continuity of function across the grid, but not of derivatives of the function. NB if the fractions based on the grid direction variables are outside the range 0.0 -> 1.0 this function can perform an extrapolation, controlled by the 'extrapolate' attribute, with possibly dubious results depending on the shape of the represented function.

## 5.37 MathMLDataClass.cpp File Reference

```
#include <iostream>
#include "Janus.h"
#include "MathMLDataClass.h"
#include "VariableDef.h"
```

### 5.37.1 Detailed Description

This file defines the data structure used for interpreting MathML mathematics procedures.The data include a tag defining the MathML element, a list of children associated with the MathML element, and call-backs to functions to evaluate the element.

## 5.38 MathMLDataClass.h File Reference

```
#include <Ute/aList.h>
#include <Ute/aMatrix.h>
#include "SolveMathML.h"
```

### Classes

- struct MathMLData

### Namespaces

- janus

### Enumerations

### Variables

- const double EXPONENTIALE = 2.71828182845905

### 5.38.1 Detailed Description

This file defines the data structure used for interpreting MathML mathematics procedures.The data include a tag defining the MathML element, a list of children associated with the MathML element, and call-backs to functions to evaluate the element.

### 5.38.2 Enumeration Type Documentation

#### 5.38.2.1 enum MathRetType

The *MathRetType* enumeration is used to flag the type of argument returned from a mathematical operation. The type is based on the W3C MathML recommendations document. At present only *Real* and *Boolean* types are handled.

### 5.38.3 Variable Documentation

#### 5.38.3.1 const double EXPONENTIALE = 2.71828182845905

A MathMLDataClass instance holds in its allocated memory alphanumeric data derived from *MathML* elements of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The data may include tags defining the *MathML* element and its attributes, a list of children associated with the *MathML* element, and call-backs to functions to evaluate the element.

The MathMLDataClass class is only used within the Janus.

## 5.39 Modification.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "Modification.h"
```

### Namespaces

- janus

### 5.39.1 Detailed Description

A Modification instance holds in its allocated memory alphanumeric data derived from a *modificationRecord* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes the author and content of a modification to a dataset. A modificationRecord associates a single letter (such as modification "A") with modification author(s), address, and any optional external reference documents, in keeping with the AIAA draft standard. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Modification class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader class.

## 5.40 Modification.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
#include "Author.h"
```

### Classes

- class Modification

### Namespaces

- janus

#### 5.40.1 Detailed Description

A Modification instance holds in its allocated memory alphanumeric data derived from a *modificationRecord* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes the author and content of a modification to a dataset. A modificationRecord associates a single letter (such as modification "A") with modification author(s), address, and any optional external reference documents, in keeping with the AIAA draft standard. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Modification class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader class.

## 5.41 ParseMathML.cpp File Reference

```
#include <iostream>
#include <Ute/aMath.h>
#include <Ute/aMessageStream.h>
#include <Ute/aString.h>
#include "ElementDefinitionEnum.h"
#include "ParseMathML.h"
#include "VariableDef.h"
#include "Janus.h"
```

#### 5.41.1 Detailed Description

This class contains functions for parsing mathematics procedures defined using the MathML syntax. Data detailing each MathML operation and is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

## 5.42 ParseMathML.h File Reference

```
#include <Ute/aMap.h>
#include <Ute/aString.h>
#include "DomTypes.h"
```

### 5.42.1 Detailed Description

This class contains functions for parsing mathematics procedures defined using the MathML syntax. Data detailing each MathML operation is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

## 5.43 PolyInterpolation.cpp File Reference

```
#include <stdexcept>
#include <sstream>
#include <Ute/aMessageStream.h>
#include "InDependentVarDef.h"
#include "Janus.h"
```

### Namespaces

- janus

### 5.43.1 Detailed Description

This private function performs interpolations when *not* all the degrees of freedom for a function are specified as linear or first order polynomial.

If the interpolation order in the $i$th degree of freedom is $k_i$, then given $\Pi_1^n(k_i + 1)$ uniformly gridded values of a function of $n$ variables, provided to the instance of the class by either setVariableByIndex() or setVariableByID(), this private function is called by getOutputVariable() to perform a multi-dimensional polynomial interpolation between the values and returns the result. At present the maximum polynomial order is limited to 3. The interpolation maintains continuity of function across the grid, but not of derivatives of the function.

{**Note**}: this function can perform an extrapolation, which is controlled by the *extrapolate* attribute, but polynomial extrapolation is notoriously inaccurate and unstable and should not be relied on by users wanting to maintain modelling fidelity.

## 5.44 Provenance.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "Provenance.h"
```

### Namespaces

- janus

### 5.44.1 Detailed Description

A Provenance instance holds in its allocated memory alphanumeric data derived from a *provenance* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Provenances may apply to a complete dataset or to individual components within a dataset. Not all provenances will contain all possible *provenance* components. The Provenance instance also provides the functions that allow a calling Janus instance to access these data elements.

The Provenance class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader, VariableDef, GriddedTableDef, UngriddedTableDef, Function or CheckData classes.

## 5.45 Provenance.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
#include "Author.h"
```

### Classes

- class Provenance

### Namespaces

- janus

### 5.45.1 Detailed Description

A Provenance instance holds in its allocated memory alphanumeric data derived from a *provenance* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. Provenances may apply to a complete dataset or to individual components within a dataset. Not all provenances will contain all possible *provenance* components. The Provenance instance also provides the functions that allow a calling Janus instance to access these data elements.

The Provenance class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader, VariableDef, GriddedTableDef, UngriddedTableDef, Function or CheckData classes.

## 5.46 Reference.cpp File Reference

```
#include "DomFunctions.h"
#include "Reference.h"
```

### Namespaces

- janus

### 5.46.1 Detailed Description

A Reference instance holds in its allocated memory alphanumeric data derived from a *reference* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes an external document relevant to the dataset. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Reference class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader class.

## 5.47 Reference.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
```

### Classes

- class Reference

### Namespaces

- janus

### 5.47.1 Detailed Description

A Reference instance holds in its allocated memory alphanumeric data derived from a *reference* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes an external document relevant to the dataset. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Reference class is only used within the janus namespace, and should only be referenced indirectly through the FileHeader class.

## 5.48 Signal.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include <Ute/aMath.h>
#include "DomFunctions.h"
#include "Signal.h"
```

### Namespaces

- janus

### 5.48.1 Detailed Description

A Signal instance holds in its allocated memory alphanumeric data derived from a *signal* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance may describe inputs, internal values of a computation, or outputs. The class also provides the functions that allow a calling StaticShot instance to access these data elements. It is used to document the name, ID, value, tolerance, and units of measure for checkcases.

A *signal* must have *signalName* and *signalUnits* if it is a child of *checkInputs* or *checkOutputs*. Alternatively, if it is a child of *internalValues*, it must have a *varID* (*signalID* is deprecated). When used in a *checkOutputs* vector, the *tol* element must be present. Tolerance is specified as a maximum absolute difference between the expected and actual value. This class accepts whichever of these children it finds in the XML dataset, and leaves applicability to its parents to sort out.

The Signal class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot, CheckInputs, InternalValues and CheckOutputs classes.

## 5.49 Signal.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
```

### Classes

- class Signal

### Namespaces

- janus

### 5.49.1 Detailed Description

A Signal instance holds in its allocated memory alphanumeric data derived from a *signal* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance may describe inputs, internal values of a computation, or outputs. The class also provides the functions that allow a calling StaticShot instance to access these data elements. It is used to document the name, ID, value, tolerance, and units of measure for checkcases.

A *signal* must have *signalName* and *signalUnits* if it is a child of *checkInputs* or *checkOutputs*. Alternatively, if it is a child of *internalValues*, it must have a *varID* (*signalID* is deprecated). When used in a *checkOutputs* vector, the *tol* element must be present. Tolerance is specified as a maximum absolute difference between the expected and actual value. This class accepts whichever of these children it finds in the XML dataset, and leaves applicability to its parents to sort out.

The Signal class is only used within the janus namespace, and should only be referenced indirectly through the StaticShot, CheckInputs, InternalValues and CheckOutputs classes.

## 5.50 SignalList.cpp File Reference

```
#include <Ute/aMessageStream.h>
#include "DomFunctions.h"
#include "SignalList.h"
```

### Namespaces

- janus

### 5.50.1 Detailed Description

A SignalList instance functions as a container for the Signal class, and provides the functions that allow a calling StaticShot instance to access the *signal* elements that define either the input or output values for a check case. A *signalList* element contains a list of *signal* elements, that include *signalName*, *signalUnits*, *signalValue* elements. An optional *tol* element may be included.

The SignalList class is only used within the janus namespace, and is inherited by the CheckInputs and CheckOutputs classes. It should only be referenced indirectly through the StaticShot class.

## 5.51 SignalList.h File Reference

```
#include <vector>
#include "XmlElementDefinition.h"
#include "Signal.h"
```

### Classes

- class SignalList

### Namespaces

- janus

### 5.51.1 Detailed Description

A SignalList instance functions as a container for the Signal class, and provides the functions that allow a calling StaticShot instance to access the *signal* elements that define either the input or output values for a check case. A *signalList* element contains a list of *signal* elements, that include *signalName*, *signalUnits*, *signalValue* elements. An optional *tol* element may be included.

The SignalList class is only used within the janus namespace, and is inherited by the CheckInputs and CheckOutputs classes. It should only be referenced indirectly through the StaticShot class.

## 5.52 SolveMathML.cpp File Reference

```
#include <iostream>
```

```
#include "MathMLDataClass.h"
#include "SolveMathML.h"
#include "VariableDef.h"
#include <Ute/aMath.h>
#include <Ute/aMatrix.h>
```

### 5.52.1 Detailed Description

This class contains functions for solving mathematics procedures defined using the MathML syntax. Data detailing each MathML operation and is stored in a MathMLData structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

## 5.53 SolveMathML.h File Reference

```
#include <Ute/aMap.h>
#include <Ute/aString.h>
```

### 5.53.1 Detailed Description

This class contains functions for solving mathematics procedures defined using the MathML syntax. Data detailing each MathML operation is stored in a MathMLDataClass structure. This includes the sub-elements to which the operator is to be applied. Functions to process both scalar and matrix data are included.

## 5.54 StaticShot.cpp File Reference

```
#include <stdexcept>
#include <sstream>
#include <iostream>
#include <Ute/aMessageStream.h>
#include <Ute/aUnits.h>
#include "DomFunctions.h"
#include "StaticShot.h"
#include "Janus.h"
```

### Namespaces

- janus

### 5.54.1 Detailed Description

A StaticShot instance holds in its allocated memory alphanumeric data derived from a *staticShot* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes the inputs and outputs, and possibly internal values, of a DAVE-ML model at a particular instant in time. The class also provides the functions that allow a calling Janus instance to access these data elements.

The StaticShot class is only used within the janus namespace, and should only be referenced indirectly through the CheckData class.

## 5.55 StaticShot.h File Reference

```
#include <Ute/aList.h>
#include "XmlElementDefinition.h"
#include "Provenance.h"
#include "CheckInputs.h"
#include "InternalValues.h"
#include "CheckOutputs.h"
```

### Classes

- class StaticShot

### Namespaces

- janus

### 5.55.1 Detailed Description

A StaticShot instance holds in its allocated memory alphanumeric data derived from a *staticShot* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The instance describes the inputs and outputs, and possibly internal values, of a DAVE-ML model at a particular instant in time. The class also provides the functions that allow a calling Janus instance to access these data elements.

The StaticShot class is only used within the janus namespace, and should only be referenced indirectly through the CheckData class.

## 5.56 Uncertainty.cpp File Reference

```
#include <Ute/aBiMap.h>
```

```
#include <Ute/aMessageStream.h>
#include <Ute/aMath.h>
#include "Janus.h"
#include "DomFunctions.h"
#include "Uncertainty.h"
```

### Namespaces

- janus

### 5.56.1 Detailed Description

A Uncertainty instance holds in its allocated memory alphanumeric data derived from a *uncertainty* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The element is used in function and parameter definitions to describe statistical variance in the possible value of that function or parameter value. Only Gaussian (normal) or uniform distributions of continuous random variable distribution functions are supported. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Uncertainty class is only used within the janus namespace, and should only be referenced indirectly through the Janus class.

## 5.57 Uncertainty.h File Reference

```
#include <utility>
#include <Ute/aList.h>
#include <Ute/aString.h>
#include "XmlElementDefinition.h"
#include "Bounds.h"
```

### Classes

- class Uncertainty

### Namespaces

- janus

### 5.57.1 Detailed Description

A Uncertainty instance holds in its allocated memory alphanumeric data derived from a *uncertainty* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. The element is used in function and parameter definitions to describe statistical variance in the possible value of that function or parameter value. Only Gaussian (normal) or uniform distributions of continuous random variable distribution functions are supported. The class also provides the functions that allow a calling Janus instance to access these data elements.

The Uncertainty class is only used within the janus namespace, and should only be referenced indirectly through the Janus class.

## 5.58 UngriddedInterpolation.cpp File Reference

```
#include <utility>
#include <algorithm>
#include <Ute/aMath.h>
#include <Ute/aMatrix.h>
#include "InDependentVarDef.h"
#include "Janus.h"
```

### Namespaces

- janus

### 5.58.1 Detailed Description

This file contains private functions to perform linear interpolations on ungridded datasets. It is called by `getOutputVariable` to perform a multi-linear interpolation between the values and returns the result. It maintains continuity of function across the dataset, but not of derivatives of the function.

## 5.59 UngriddedTableDef.cpp File Reference

```
#include <cstdio>
#include <Ute/aMessageStream.h>
#include "Janus.h"
#include "DomFunctions.h"
#include "UngriddedTableDef.h"
#include "BreakpointDef.h"
#include <libqhull/libqhull.h>
#include <libqhull/mem.h>
#include <libqhull/qset.h>
#include <libqhull/geom.h>
#include <libqhull/merge.h>
#include <libqhull/poly.h>
#include <libqhull/io.h>
#include <libqhull/stat.h>
```

### Namespaces

- janus

### 5.59.1 Detailed Description

A UngriddedTableDef instance holds in its allocated memory alphanumeric data derived from a *ungriddedTableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes points that are not in an orthogonal grid pattern; thus, the independent variable coordinates are specified for each dependent variable value. The table data point values are specified as comma-separated values in floating-point notation. Associated alphanumeric identification and cross-reference data are also included in the instance.

The UngriddedTableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.60 UngriddedTableDef.h File Reference

```
#include <utility>
#include "Provenance.h"
#include "Uncertainty.h"
#include <Ute/aMatrix.h>
#include <Ute/aString.h>
```

### Classes

- class UngriddedTableDef

### Namespaces

- janus

### 5.60.1 Detailed Description

A UngriddedTableDef instance holds in its allocated memory alphanumeric data derived from a *ungriddedTableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes points that are not in an orthogonal grid pattern; thus, the independent variable coordinates are specified for each dependent variable value. The table data point values are specified as comma-separated values in floating-point notation. Associated alphanumeric identification and cross-reference data are also included in the instance.

The UngriddedTableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.61 VariableDef.cpp File Reference

```
#include <algorithm>
#include <tr1/functional>
#include <Ute/aMessageStream.h>
#include <Ute/aString.h>
#include <Ute/aMath.h>
#include "Janus.h"
#include "DomFunctions.h"
#include "VariableDef.h"
#include "ParseMathML.h"
#include "SolveMathML.h"
#include "ExportMathML.h"
```

### Namespaces

- janus

### 5.61.1 Detailed Description

A VariableDef instance holds in its allocated memory alphanumeric data derived from a *variableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes descriptive, alphanumeric identification and cross-reference data, and may include a calculation process tree for variables computed through MathML. The variable definition can include statistical information regarding the uncertainty of the values that it might take on, when measured after any calculation is performed. This class sets up a structure which manages the *variableDef* content.

The VariableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.62 VariableDef.h File Reference

```
#include <Ute/aMatrix.h>
#include <Ute/aUnits.h>
#include "XmlElementDefinition.h"
#include "DimensionDef.h"
#include "Array.h"
#include "Model.h"
#include "Provenance.h"
#include "Uncertainty.h"
#include "Function.h"
#include "MathMLDataClass.h"
```

### Classes

- class VariableDef

### Namespaces

- janus

### 5.62.1 Detailed Description

A VariableDef instance holds in its allocated memory alphanumeric data derived from a *variableDef* element of a DOM corresponding to a DAVE-ML compliant XML dataset source file. It includes descriptive, alphanumeric identification and cross-reference data, and may include a calculation process tree for variables computed through MathML. The variable definition can include statistical information regarding the uncertainty of the values that it might take

on, when measured after any calculation is performed. This class sets up a structure that manages the *variableDef* content.

The VariableDef class is only used within the janus namespace, and should only be referenced through the Janus class.

## 5.63 XmlElementDefinition.h File Reference

```
#include <Ute/aString.h>
#include "ElementDefinitionEnum.h"
#include "DomTypes.h"
```

### Classes

- class XmlElementDefinition

### Namespaces

- janus

### 5.63.1 Detailed Description

This file contains definitions of virtual functions that are used when instantiating a DAVE-ML compliant XML file using Janus. The *XmlElementDefinition* class is inherited by base element classes, such as *FileHeader* and *VariableDef*, which have specific versions of the virtual functions. These function calls are accessed internally within Janus through the *DomFunctions* class and permit abstraction of the process of interacting with the DOM. They do not provide a capability to external applications to interact with the XML encoded data file or the associated DOM.

# 6 Conclusion

This document has detailed the Janus application programming interface (API), which permits flight modelling and simulation applications to directly interface with aerospace vehicle datasets encoded using the DAVE-ML syntax. The process of instantiating Janus within an application has been presented, together with descriptions of the public functions that enable the application to interact with information stored within DAVE-ML style datasets.

The Janus API provides a capability to simplify the exchange of aerospace vehicle dynamic model data between simulation applications.

# 7 Acknowledgements

The authors wishes to acknowledge the contributions made by Michael Young, Michael Grant, Jonathan Dansie, Kylie Bedwell from DST Group; Dr Daniel Newman from Quantitative Aeronautics, and Robert Curtin from Advanced VTOL Technologies to the development of Janus.

# 8 Contact

Janus is made available subject to the conditions listed in the DSTO Open Source Licence Version 1.1, or later versions [10]. To request a copy of Janus enquiries can be sent to: janus@dst.defence.gov.au

# 9 References

[1] D. M. Newman, *Efficient Development and Use of Aircraft Flight Models – Survey of DSTO Usage*, Ball Solutions Group Report No. 1234.002, Melbourne, 2004.

[2] Anon., *Flight Dynamics Model Exchange Standard*, AIAA Modeling and Simulation Technical Committee, ANSI/AIAA S-119-2011, October 2007, 25 March 2011, http://daveml.org/AIAA_stds/index.html.

[3] E. B. Jackson & B. L. Hildreth, *Flight Dynamic Model Exchange using XML* AIAA 2002-4482, AIAA Modeling and Simulation Technologies Conference, Monterey, CA, August 2002.

[4] Anon., *Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) Reference*, AIAA Modeling and Simulation Technical Committee, Version 2.0.2, 12 July 2011, http://daveml.org/DTDs/index.html.

[5] D. M. Newman, *The Janus C++ Library An Interface Class for DAVE-ML Compliant XML-based Flight Model Datasets, Version 1*, Ball Solutions Group Report No. 31495.002, Melbourne, 18 July 2005.

[6] D. M. Newman, *The Janus C++ Library An Interface Class for DAVE-ML Compliant XML-based Flight Model Datasets, Version 1.10*, Quantitative Aeronautics Report No. 0708-232.001, Melbourne, 31 March 2010.

[7] G. Brian, *Flight Systems Units of Measure Guidelines*, viewed 12 December 2004.

[8] M. Abramowitz & I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, viewed 14 July 2008, http://www.math.sfu.ca/~cbm/aands/.

[9] G. Brian, *Vector and Matrix Variable Definitions in DAVE-ML*, DSTO-TN-1146, Defence Science and Technology Organisaton, December 2012.

[10] Anon., *DSTO OPEN SOURCE LICENSE, Version 1*, Defence Science and Technology Organisaton, August 2006.

THIS PAGE IS INTENTIONALLY BLANK

# Appendix A: Deprecated Functions

This appendix lists Janus functions that have been deprecated due to enhancements to the Janus API. The functions have been retained to provide backwards compatibility for applications that were developed using previous versions of the API. It is recommended that these functions should not be used when developing new applications, instead the various components should be accessed through the specific class interfaces.

## A.1 Janus - XML File Documentation

### Enumerations

### Functions

- dstoute::aString getHeaderName ()
- int getNumberOfHeaderProvenances ()
- int getNumberOfProvenanceComponents (const char *parentID, size_t index, enum ProvenanceAttribute provenanceAttribute)
- int getNumberOfXmlFileAuthorAddresses (size_t authorNumber)
- int getNumberOfXmlFileAuthorContacts (size_t authorNumber)
- int getNumberOfXmlFileAuthors ()
- int getNumberOfXmlFileModificationAuthorAddresses (size_t index, size_t author)
- int getNumberOfXmlFileModificationAuthorContacts (size_t index, size_t author)
- int getNumberOfXmlFileModificationAuthors (size_t index)
- const char * getProvenance (const char *parentID, size_t index, enum ProvenanceAttribute provenanceAttribute, enum AuthorAttribute authorAttribute, size_t authorContactIndex, size_t componentIndex)
- const char * getXmlFileAuthor (size_t authorNumber, AuthorAttribute authorAttribute, size_t addressNumber=0)
- const char * getXmlFileCreationDate ()
- const char * getXmlFileDescription ()
- const char * getXmlFileModification (size_t index, ModificationAttribute modificationAttribute, size_t authorNumber=0, size_t addressNumber=0)
- int getXmlFileModificationCount ()
- int getXmlFileModificationExtraDocCount (size_t index)
- const char * getXmlFileModificationExtraDocRefID (size_t index, size_t indxRef)
- const char * getXmlFileReference (size_t index, ReferenceAttribute referenceAttribute)
- int getXmlFileReferenceCount ()
- int getXmlFileReferenceIndex (const char *refID)

- const char * getXmlFileVersion ()

### A.1.1 Detailed Description

The documentation functions relate to the descriptive material contained in the XML dataset file header. This includes file authorship, modification records, and cross-references to source material. The functions provide access to this data for the calling program. Some of this reference material is optional, as defined by the DAVE-ML DTD, so many of these functions may return empty strings if requested data is not present.

Most of these functions are deprecated, retained for backwards compatibility. In future developments, the fileheader components should be accessed through the FileHeader class.

### A.1.2 Enumeration Type Documentation

#### A.1.2.1 enum AuthorAttribute

This enum is deprecated, and should not be used in new programs. Access data through the Author class instead. The enum serves as input to getXmlFileAuthor, and is used to indicate which of the XML dataset file's author attributes is required.

Enumerator

***NAME*** NAME. The author's name

***ORG*** ORG. The organisation directing the author's work on this dataset

***XNS*** XNS. Extensible Name Service (XNS) is an open XML-based protocol that specifies a way to establish and manage a universal addressing system. An XNS universal address serves as a permanent contact point for an individual or other legal entity, such as a business. This XNS entry provides contact details for the author or his organisation in reference to this dataset.

***EMAIL*** EMAIL. The e-mail address through which the author may be contacted in reference to this dataset

***ADDRESS*** ADDRESS. The postal mail address through which the author may be contacted in reference to this dataset

***CONTACTINFO_TYPE*** CONTACTINFO_TYPE. A specified type of contact information for the author

***CONTACTINFO_LOCATION*** CONTACTINFO_LOCATION. The location associated with a specified type of contact

***CONTACTINFO_CONTENT*** The content of a contact information element

#### A.1.2.2 enum ModificationAttribute

This enum is deprecated, and should not be used in new programs. Access data through the FileHeader class instead. It serves as input to getXmlFileModification, and is used to indicate which of the XML dataset file's *modificationRecord* attributes is required.

Not all datasets will contain all attributes. As defined in DAVEfunc.dtd, only the modificationID and the author base data are required.

Enumerator

***MODID*** The modificationRecord ID
***MOD_DATE*** The date of the modification
***MOD_REFID*** The reference ID on which the modification is based
***MOD_AUTHORNAME*** The name of the modification's author
***MOD_AUTHORORG*** The organisation directing the author's work on this modification of the dataset
***MOD_AUTHORXNS*** Extensible Name Service (XNS) is an open XML-based protocol that specifies a way to establish and manage a universal addressing system. An XNS universal address serves as a permanent contact point for an individual or other legal entity, such as a business. This XNS entry provides contact details for the author or his organisation in reference to this modification of the dataset.
***MOD_AUTHOREMAIL*** The e-mail address through which the author may be contacted in reference to this modification of the dataset
***MOD_AUTHORADDRESS*** The postal mail address through which the author may be contacted in reference to this modification of the dataset
***MOD_AUTHORCONTACT_TYPE*** A specified type of contact information for the modification author
***MOD_AUTHORCONTACT_LOCATION*** The location associated with a specified type of contact
***MOD_AUTHORCONTACT_CONTENT*** The content of a contact information element
***MOD_DESCRIPTION*** A description of the modification

#### A.1.2.3 enum ProvenanceAttribute

This enum is deprecated, and should not be used in new programs. Access data through the FileHeader class instead. It serves as input to getProvenance, and is used to indicate which *provenance* attribute or child element is required for a specified provenance node within the XML dataset. Not all provenences will contain all attributes or child elements. As defined in DAVEfunc.dtd, only the author base data and the creation date are required. However, the *provID* is strongly recommended, since it allows cross-referencing and thereby reduces repetition within the XML dataset.

Enumerator

***PROV_ID*** The provenance ID
***PROV_DATE*** The creation date of the data whose provenance is accessed
***PROV_AUTHOR*** Details of the data's author
***PROV_DOCUMENTREF*** The documentary data on which the XML data is based
***PROV_MODIFICATIONREF*** Cross-reference to the XML file modification by which the referenced data was created
***PROV_DESCRIPTION*** Description related to the provenence of the referenced data.

#### A.1.2.4 enum ReferenceAttribute

This enum is deprecated, and should not be used in new programs. Access data through the FileHeader class instead. It serves as input to getXmlFileReference, and is used to indicate which of the XML dataset file's *reference* attributes is required.

Not all datasets will contain all attributes. As defined in DAVEfunc.dtd, description, accession number and href are optional.

Enumerator

***REFID*** The reference ID
***AUTHOR*** The reference document's author name
***TITLE*** The reference document's title
***ACCESSION*** The reference document's library accession number
***DATE*** The date of publication of the reference document
***HREF*** The XLink address or identifier of the reference document
***DESCRIPTION*** A description of the reference document

### A.1.3 Function Documentation

#### A.1.3.1 dstoute::aString getHeaderName ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. The function returns the optional *name* attribute character string associated with the *fileHeader* element.

Returns

A string containing the file header name attribute. If the attribute is not present an empty string is returned.

#### A.1.3.2 int getNumberOfHeaderProvenances ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It returns the number of *provenance* elements contained in a DAVE-ML *fileHeader* element. It does not include provenance elements contained in other elements of the dataset, since there is at most one *provenance* or *provenanceRef* in each non-header element.

Returns

nProvenance is the integer count of provenance elements contained in the *fileHeader* element.

#### A.1.3.3 int getNumberOfProvenanceComponents ( const char * *parentID*, size_t *index*, enum ProvenanceAttribute *provenanceAttribute* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It returns the number of *provenance* child elements of specified type contained in a specified provenance element.

Parameters

| | |
|---|---|
| *parentID* | is a short string without whitespace, such as "MACH02", which uniquely defines the parent of the requested provenance. The allowable parentIDs are: "fileHeader", varID, gtID, utID, (function) "name", checkID. Where the parent is the *fileHeader*, there can be multiple provenance records and an index is required to uniquely specify the desired record. |
| *index* | is only required when the parentID is "fileHeader". It specifies the desired *fileHeader* provenance record. Its range is from 0 to (getNumberOfHeaderProvenances - 1). |
| *provenanceAttribute* | indicates which of the available *provenance* components is to be counted by this function call. |

Returns

nElements is the integer count of provenance elements of the specified type. Where the specified *parentID* has no attached provenance elements, zero is returned.

#### A.1.3.4 int getNumberOfXmlFileAuthorAddresses ( size_t *authorNumber* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the number of addresses listed for each primary author in the XML dataset file header.

Parameters

| | |
|---|---|
| *authorNumber* | indicates for which of one or more authors the number of addresses available in the XML dataset is required (see getNumberOfXmlFileAuthors). Indexing is C-style, starting at zero. |

Returns

The number of addresses for a primary author listed in the XML dataset. Possible values are 0 or more.

#### A.1.3.5 int getNumberOfXmlFileAuthorContacts ( size_t *authorNumber* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the number of *contactInfo* details listed for each primary author in the XML dataset file header.

Parameters

| | |
|---|---|
| *authorNumber* | indicates for which of one or more authors the number of *contactInfo* details available in the XML dataset is required (see getNumberOfXmlFileAuthors). Indexing is C-style, starting at zero. |

Returns

The number of addresses for a primary author listed in the XML dataset. Possible values are 0 or more.

#### A.1.3.6 int getNumberOfXmlFileAuthors ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the number of primary authors listed in the XML dataset file header.

Returns

The number of primary authors listed in the XML dataset. Possible values are 1 or more.

#### A.1.3.7 int getNumberOfXmlFileModificationAuthorAddresses ( size_t *index*, size_t *author* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the number of address entries for a particular author of a particular dataset modification as listed in the XML dataset file header.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed. |
| *author* | has a range from 0 to (getNumberOfXmlFileModificationAuthors() - 1), and selects the author of interest |

Returns

The number of addresses listed for a modification author in the XML dataset. Possible values are 0 or more.

#### A.1.3.8 int getNumberOfXmlFileModificationAuthorContacts ( size_t *index*, size_t *author* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the number of *contactInfo* entries for a particular author of a particular dataset modification as listed in the XML dataset file header.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed. |
| *author* | has a range from 0 to (getNumberOfXmlFileModificationAuthors() - 1), and selects the author of interest |

Returns

The number of *contactInfo* listed for a modification author in the XML dataset. Possible values are 0 or more.

#### A.1.3.9 int getNumberOfXmlFileModificationAuthors ( size_t *index* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the number of authors for a particular dataset modification as listed in the XML dataset file header.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed. |

Returns

The number of modification authors listed in the XML dataset. Possible values are 1 or more.

#### A.1.3.10 const char* getProvenance ( const char * *parentID*, size_t *index*, enum ProvenanceAttribute *provenanceAttribute*, enum AuthorAttribute *authorAttribute*, size_t *authorContactIndex*, size_t *componentIndex* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the components of a *provenance* element.

Not all provenences will contain all components. As defined in DAVEfunc.dtd, only the author base data and the creation date are required. Where a component does not exist in the XML dataset, an empty string will be returned. Where the required data is defined by reference, reference links will be followed to the basic data.

Parameters

| | |
|---|---|
| *parentID* | is a short string without whitespace, such as "MACH02", that uniquely defines the parent of the requested provenance. The allowable parentIDs are: "fileHeader", varID, gtID, utID, (function) name, checkID. Where the parent is the *fileHeader*, there can be multiple provenance records and an index is required to uniquely specify the desired record. |
| *index* | is only required when the parentID is "fileHeader". It specifies the desired *fileHeader* provenance record. Its range is from 0 to (getNumberOfHeaderProvenances - 1). |
| *provenanceAttribute* | indicates which of the available *provenance* components is requested by this function call. |
| *authorAttribute* | is only required when the *provenanceAttribute* is *PROV_AUTHOR*. It specifies the author data required in terms of the AuthorAttribute enum. |
| *authorContactIndex* | is only required when the *provenanceAttribute* is *PROV_AUTHOR*. It specifies which of multiple *address* or *contactInfo* entries for an author is requested by this function call. It uses C-style indexing, and defaults to zero. |
| *componentIndex* | specifies the desired component in cases where multiple components are allowable. The possible cases are *author*, *documentRef* and *modificationRef*. The parameter has a range from 0 to (getNumberOfProvenanceComponents - 1). |

Returns

A pointer to the requested component of the requested provenance element is returned. Where the requested component does not exist, or one of the specifiers is out of range, a pointer to an empty string is returned.

#### A.1.3.11 const char* getXmlFileAuthor ( size_t *authorNumber*, AuthorAttribute *authorAttribute*, size_t *addressNumber* = 0 )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the *author* element character strings contained in the XML dataset file header. Some attributes (described in AuthorAttribute) are optional.

Parameters

| | |
|---|---|
| *authorNumber* | indicates which of one or more authors an attribute is required (see getNumberOfXmlFileAuthors). Indexing is C-style, starting at zero. |

Parameters

| | |
|---|---|
| *authorAttribute* | indicates which of the available author attributes is required by this function call. |
| *addressNumber* | is only required if an address or contact info is required for the author (see getNumberOfXmlFileAuthorAddresses and getNumberOfXmlFileAuthorContacts). Indexing is C-style, starting at zero. |

Returns

A character pointer (char∗) to the requested attribute is returned. If an optional attribute is not present an empty string is returned.

#### A.1.3.12 const char∗ getXmlFileCreationDate ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the *fileCreationDate* character string contained in the XML dataset file header. The format of the dataset string is determined by the XML dataset builder, but DAVE-ML recommends the ISO 8601 form "2004-01-02" to refer to 2 January 2004.

Returns

a character pointer (char∗) to the XML file creation date string.

#### A.1.3.13 const char∗ getXmlFileDescription ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the *description* character string contained in the XML dataset file header. The description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description string. Since description of a file is optional, the returned string may be blank.

Returns

a character pointer (char∗) to the XML description string.

#### A.1.3.14 const char∗ getXmlFileModification ( size_t *index*, ModificationAttribute *modificationAttribute*, size_t *authorNumber* = *0*, size_t *addressNumber* = *0* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the *modificationRecord* attribute character strings contained in the XML dataset file header. Some attributes (described in ModificationAttribute) are optional.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed. |
| *modificationAttribute* | indicates which of the available *modificationRecord* attributes is required by this function call. |
| *authorNumber* | allows one of multiple authors of the modification to be selected, for those attributes that relate to a specific author. For items not related to a particular author, enter 0. |
| *addressNumber* | allows one of multiple addresses or *contactInfo* items related to a particular author to be selected, for those attributes that relate to a specific author. For items not related to a particular author address or contact info, enter 0. |

Returns

A character pointer (char∗) to the requested attribute is returned. If an optional attribute is not present, or a *index* out of range is requested, an empty string is returned.

#### A.1.3.15 int getXmlFileModificationCount ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It returns the number of *modificationRecord* records at the top level of the *fileHeader* component of the XML dataset.

Returns

the number of modification records in the XML dataset file header. Possible values are zero or more.

#### A.1.3.16 int getXmlFileModificationExtraDocCount ( size_t *index* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. As well as its basic *refID* cross-reference, each *modificationRecord* can have extra documents referenced. This function allows the calling program to determine how many, if any, extra document reference records are cross-referenced by each modification.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed. |

Returns

the number of extra documents referenced is returned. If a *index* out of range is requested, -1 is returned.

#### A.1.3.17 const char* getXmlFileModificationExtraDocRefID ( size_t *index*, size_t *indxRef* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. Where a *modificationRecord* references extra documents, this function allows the calling program to access the *refID* associated with each of those documents. The result of this function may be used in conjunction with getXmlFileReferenceIndex and getXmlFileReference to obtain the details of the associated reference material.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileModificationCount() - 1), and selects the modification record be addressed. |
| *indxRef* | has a range from 0 to (getXmlFileModificationExtraDocCount (index) - 1 ), and selects the extra document record to be addressed. |

Returns

A character pointer (char*) to the requested *refID* is returned. If a *index* or *indxRef* out of range is requested, an empty string is returned.

#### A.1.3.18 const char* getXmlFileReference ( size_t *index*, ReferenceAttribute *referenceAttribute* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the *reference* attribute character strings contained in the XML dataset file header. Some attributes (described in ReferenceAttribute) are optional.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getXmlFileReferenceCount() - 1), and selects the reference record to be addressed. |
| *referenceAttribute* | indicates which of the available reference attributes is required by this function call. |

Returns

A character pointer (char∗) to the requested attribute is returned. If an optional attribute is not present, or a *index* out of range is requested, an empty string is returned.

#### A.1.3.19 int getXmlFileReferenceCount ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It returns the number of *reference* records at the top level of the *fileHeader* component of the XML dataset.

Returns

the number of reference records in the XML dataset file header. Possible values are zero or more.

#### A.1.3.20 int getXmlFileReferenceIndex ( const char ∗ *refID* )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. The file header may contain a list of references. Each of these is associated with a unique reference ID that file modification and data provenance records use for cross-referencing. This function relates the reference ID to an *index* into the arrays of reference data, as used in Janus::getXMLFileReference.

Parameters

| | |
|---|---|
| *refID* | is a short, unique string used to refer to elements in the XML file header's list of references. |

Returns

an *index* in the range 0 to (getXmlFileReferenceCount() - 1) is returned. Where the file header contains no reference records, or *refID* does not match any reference records, -1 is returned.

#### A.1.3.21 const char∗ getXmlFileVersion ( )

This function is deprecated, and should not be used in new programs. Access this data through the FileHeader class instead. It provides access to the *fileVersion* character string contained in the XML dataset file header. The format of the version string is determined by the XML dataset builder. Since the file version is optional in the DAVE-ML DTD, the returned string may be blank.

Returns

a character pointer (char∗) to the XML file version string.

## A.2 Janus - XML Tabulated Functions

### Functions

- const char * getFunctionDefinitionName (size_t index) const
- const char * getFunctionDescription (size_t index) const
- const char * getFunctionName (size_t index) const
- int getNumberOfFunctions () const

### A.2.1 Detailed Description

These elements of the Janus class provide access to the *function* elements contained in a DOM that complies with the DAVE-ML DTD. Each *function* has optional description, optional provenance, and either a simple table of input/output values or references to more complete (possibly multiple) input, output, and function data elements. In general, calling programs should access function-based data through the *outputVariable* procedures rather than through these lower-level function access procedures.

All *function* and *dependentVariable* functions use an index based on the *function* Level 1 elements defined in the XML dataset (see Janus::getNumberOfFunctions). For example:

```
int nf = prop.getNumberOfFunctions();
cout << " Number of functions = " << nf << "\n\n";

for ( int i = 0 ; i < nf ; i++ ) {
  cout << "  Function " << i << " : \n"
       << "   Name                    : "
       << prop.getFunctionName( i ) << "\n";
}
```

The order of *functions* within the DOM is arbitrary and the calling program is responsible for determining which *index* addresses each *function*. The function *index* range is from zero to (getNumberOfFunctions() - 1).

These functions are deprecated, retained for backwards compatibility. In future program development, the *function* low-level components should be accessed through the Function class.

### A.2.2 Function Documentation

#### A.2.2.1 const char* getFunctionDefinitionName ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the Function class instead. A function definition's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard [2] if possible. Note that the *function definition* name is returned, not the *function*

name nor the variable ID associated with it. For functions defined in terms of a single list of variable values, there is no explicit function definition and an empty string is returned.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfFunctions() - 1), and selects the function to be addressed. |

Returns

A character pointer (char∗) to an XML *functionDefn* tag's *name* attribute string is returned. The *functionDefn* is a child node of the *function*, so the input *index* refers to the *function*. An *index* out of range will return a zero pointer.

See also

getFunctionName

#### A.2.2.2 const char∗ getFunctionDescription ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the Function class instead. A *function's* description consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the returned description string. Since description of a *function* is optional, the returned string may be blank.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfFunctions() - 1), and selects the *function* to be addressed. |

Returns

A character pointer (char∗) to an XML *function* tag's *description* child element contents string is returned. An *index* out of range will return a zero pointer.

#### A.2.2.3 const char∗ getFunctionName ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the Function class instead. A function's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the AIAA draft standard [2] if possible. Note that the *function* name is returned, not the function definition name nor the variable ID associated with it.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfFunctions() - 1), and selects the function to be addressed. |

Returns

A character pointer (char∗) to an XML *function* tag's *name* attribute string is returned. An *index* out of range will return a zero pointer.

See also

getFunctionDefinitionName

#### A.2.2.4 int getNumberOfFunctions ( ) const

This function is deprecated, and should not be used in new programs. Access this data through the Function class instead. The returned value includes all functions found in the DOM, and makes no distinction between function types. It may be used as the upper limit for an index to address functions and their associated dependent variables, although not all output variables are necessarily associated with functions (e.g. constants and MathML expressions).

Returns

Total number of all functions defined in the XML file and successfully loaded into the DOM.

See also

getFunction

## A.3 Janus - Output Variables Functions

### Functions

- int applyOutputScaleFactorByIndex (size_t index, const double &factor)
- int applyOutputScaleFactorByVarID (const char *varID, const double &factor)
- double getCorrelationCoefficient (size_t index, size_t indx1, size_t indx2)
- int getNumberOfOutputs () const
- double getOutputScaleFactorByIndex (size_t index) const
- double getOutputScaleFactorByVarID (const char *varID)
- double getOutputVariable (size_t index)
- const char * getOutputVariable (size_t index, int)
- const char * getOutputVariableAxisSystem (size_t index) const
- double getOutputVariableByVarID (const char *varID)
- const char * getOutputVariableDescription (size_t index) const
- const char * getOutputVariableID (size_t index) const
- int getOutputVariableIndex (const char *varID)
- const char * getOutputVariableName (size_t index) const
- const Provenance & getOutputVariableProvenance (size_t index) const
- VariableDef::VariableType getOutputVariableType (size_t index) const
- double getOutputVariableUncertainty (size_t index, size_t numSigmas)
- double getOutputVariableUncertainty (size_t index, bool isUpperBound)
- const char * getOutputVariableUnits (size_t index) const

### A.3.1 Detailed Description

Three types of output variables are defined by the DAVE-ML DTD. They are:

- Dependent variables resulting from *function* evaluation, but not forming an input to another calculation;
- Variables evaluated using MathML, but not forming an input to another calculation; and
- Variables explicitly defined as outputs using the *isOutput* attribute.

These functions provide the means to obtain the characteristics of variables that satisfy these criteria, and to obtain variable values based on the current state of all variables within the Janus instance. Normal usage of the Janus class should rely on these functions for output. They ensure that returned values are compatible with the current state of all inputs.

In general, since DAVE-ML Version 2.0RC3 and Janus Version 1.10 these functions are deprecated and the required data should be accessed directly through the VariableDef class instead.

### A.3.2 Function Documentation

#### A.3.2.1 int applyOutputScaleFactorByIndex ( size_t *index*, const double & *factor* )

This function is deprecated, and should not be used in new programs. If you must use output scale factors, set this data through the VariableDef class instead. This function should be used with extreme caution. The default scale factor for each output variable is unity. Each time this function is used, it multiplies by *factor* the current value of scale factor associated with the output variable referenced by *index*. The accumulated scale factor is applied to all subsequent computations used to determine a value for the output variable referenced by *index*. The use of this function is particularly discouraged for datasets where output from one function is defined as input to another function.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *factor* | is the new scale factor to be applied multiplicatively to the output variable's existing scale factor, and may be any double precision number, including zero. |

Returns

0 if scale factor is reset successfully.

See also

applyOutputScaleFactorByVarID
getOutputScaleFactorByIndex

#### A.3.2.2 int applyOutputScaleFactorByVarID ( const char * *varID*, const double & *factor* )

This function is deprecated, and should not be used in new programs. If you must use output scale factors, set this data through the VariableDef class instead. This function should be used with extreme caution. The default scale factor for each output variable is unity. Each time this function is used, it multiplies by *factor* the current value of scale factor associated with the output variable whose *varID* matches the input *varID*.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the output variable. If the *varID* input does not match any output variable ID within the DOM, a standard exception is thrown. |

Parameters

| | |
|---|---|
| *factor* | is the new scale factor to be applied multiplicatively to the output variable's existing scale factor, and may be any double precision number, including zero. |

Returns

0 if scale factor is reset successfully.

See also

applyOutputScaleFactorByIndex
getOutputScaleFactorByVarID

#### A.3.2.3 double getCorrelationCoefficient ( size_t *index*, size_t *indx1*, size_t *indx2* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function allows the caller to determine the amount of correlation between the uncertainty in two independent variables contributing to the same output. The direction of correlation and sequence of variables is irrelevant.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indx1* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the first independent variable to be addressed through the output variable that uses it. |
| *indx2* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the second independent variable to be addressed through the output variable that uses it. Because correlation is symmetric, order of the two independent variables is irrelevant. |

Returns

The correlation coefficient relating the two variables' uncertainties is returned as a double. Where correlation has not been specified in the XML dataset, the coefficient is returned as zero. If *index*, *indx1* or *indx2* is out of range, this function will return NaN.

#### A.3.2.4 int getNumberOfOutputs ( ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The returned value counts all outputs found in the DOM (explicitly defined outputs, and results of function evaluations or MathML expression evaluations that do not also form calculation inputs). It may be used as the upper limit for an index to outputs.

Returns

Total number of all output variables defined in the XML file, implicit and explict, and successfully loaded into the DOM.

#### A.3.2.5 double getOutputScaleFactorByIndex ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. If you must use output scale factors, access this data through the VariableDef class instead. The default scale factor for each output variable is unity. However, a reckless programmer can use applyOutputScaleFactorByIndex to change this value to any double precision number (including 0.0), so this function allows any such changes to be tracked, typically immediately prior to performing a computation of the output variable referenced by *index*.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

The current accumulated scale factor for output variable *index* is returned, which may be any double precision number, including zero.

#### A.3.2.6 double getOutputScaleFactorByVarID ( const char * *varID* )

This function is deprecated, and should not be used in new programs. If you must use output scale factors, access this data through the VariableDef class instead. The default scale factor for each output variable is unity. However, a reckless programmer can use applyOutputScaleFactorByVarID to change this value to any double precision number (including 0.0), so this function allows any such changes to be tracked, typically immediately prior to performing a computation of the output variable referenced by *varID*.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the output variable. |

Returns

The current accumulated scale factor for the output variable whose *varID* matches the input *varID* will be returned. It may be any double precision number, including zero. If the *varID* input does not match any output variable ID within the DOM, the return value is NaN.

### A.3.2.7 double getOutputVariable ( size_t *index* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This fulfils the basic purpose of the Janus class. It is used during run-time to evaluate output variables defined (explicitly or implicitly) within the XML source, based on independent variable values supplied to the instanced Janus class. The independent variable values must be set to required values, passing the aircraft state to the Janus instance using setIndependentVariableByIndex or the other value-input functions, before this function is used. One possible way of applying the Janus class to perform an output variable evaluation is:

```
int outputNumber = 0;
for ( int i = 0 ;
      i < prop.getNumberOfIndependentVariables( outputNumber ) ; i++ ) {
  cout << "\n  Enter value for "
       << prop.getIndependentVariableID( outputNumber, i )
       << " : ";
  double x;
  cin >> x;
  prop.setIndependentVariableByIndex( outputNumber, i, x );
}
double y = prop.getOutputVariable( outputNumber );
```

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A double precision value containing the value of the output after all relevant computations based on the current input state. An *index* out of range will return NaN.

See also

setIndependentVariableByIndex
setVariableByIndex
setVariableByID
getOutputVariableByVarID

### A.3.2.8 const char* getOutputVariable ( size_t *index*, int )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. As an extension to the normal behaviour of a DAVE-ML gridded table, support has been included for managing a table of strings in a similar manner to numeric tabular data. The strings are accessed in the same way as a numeric tabular function. The

array of strings may be multi-dimensionsional, and its breakpoints in each dimension should be monotonic sequences of integers ( $1, 2, 3, \ldots n$ is a good choice), where the product of the breakpoint array lengths equals the number of strings. The independent variables must lie within the ranges of their corresponding breakpoints, and must be set to require "discrete" interpolation.

The strings can be delimited by any of: tab, newline, comma, semicolon. DO NOT start or end the strings with excess whitespace.

Janus detects a string table by looking for non-numeric characters, so a table consisting entirely of numeric data will never be detected as a string. Note: The string table can only be interrogated by output variable index, not by varID or any of the other indices.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. The second input is a dummy to allow the function to be overloaded. |

Returns

a string pointer to the output variable based on the current input state.

#### A.3.2.9 const char* getOutputVariableAxisSystem ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An output variable's *axisSystem* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS . Typical values include "Body" or "Intermediate". Where no attribute is specified in the XML dataset, an empty string is returned.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *axisSystem* attribute string is returned. An *index* out of range will return a zero pointer.

#### A.3.2.10 double getOutputVariableByVarID ( const char * *varID* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This is an alternative approach to the basic purpose of the Janus class. It is used during run-time to evaluate output variables defined (explicitly or implicitly) within the XML source, based on independent variable values supplied to the instanced Janus class. The independent variable values must be set to required values, passing the aircraft

state to the Janus instance using setIndependentVariableByIndex or the other value-input functions, before this function is used.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the output variable. |

Returns

A double precision value containing the value of the output variable whose *varID* matches the input *varID* after all relevant computations based on the current input state. If the input does not match any variable ID within the DOM, a standard exception is thrown.

See also

setIndependentVariableByIndex
setVariableByIndex
setVariableByID
getOutputVariable

#### A.3.2.11 const char* getOutputVariableDescription ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An output variable's *description* consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the description. Since description of a variable is optional, the returned string may be blank.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *description* child element contents string is returned. An *index* out of range will return a zero pointer.

#### A.3.2.12 const char* getOutputVariableID ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An output variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", that uniquely defines the variable. It may be used for indexing. This function may be used by the calling program to determine the variable ID associated with each output variable location in the DOM.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A pointer to an XML *variableDef* tag's *varID* attribute string is returned.

#### A.3.2.13 int getOutputVariableIndex ( const char * *varID* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An output variable's *varID* attribute is uniquely related to the *variable* and may be used as an index. This function should be used by the calling program to establish the order of output variables within the DOM, since it is always more efficient to address an output variable by numeric index than by variable ID.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the output variable. |

Returns

An integer *index* in the range from 0 to (getNumberOfOutputs() - 1), corresponding to the output variable whose *varID* matches the supplied *varID*. If the input does not match any dependent variable ID within the DOM, the returned value is -1.

#### A.3.2.14 const char* getOutputVariableName ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An output variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the ANSI/AIAA S-119-2011 standard [2] .

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *name* attribute string is returned.

#### A.3.2.15 const Provenance& getOutputVariableProvenance ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function provides access to *provenance* elements contained in a DAVE-ML output *variableDef* element. There may be zero or one of these elements for each output variable in a valid dataset.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A pointer to the requested Provenence class instance is returned.

#### A.3.2.16 VariableDef::VariableType getOutputVariableType ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function allows the caller to determine from what source an output variable will be computed.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

The VariableType is returned on successful completion.

#### A.3.2.17 double getOutputVariableUncertainty ( size_t *index*, size_t *numSigmas* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function returns a number representing the uncertainty associated with a value obtained from getOutputVariable. The type of uncertainty is obtained from getOutputVariableUncertainty().getPdf(). This function returns as additive uncertainty the requested number of standard deviations. The uncertainty value may change depending on the values of input variables to the Janus class. A typical application of this group of functions might be:

```
int indx = 0;
double value = prop.getOutputVariable( indx );
UncertaintyPdf uncertaintyPdf =
   prop.getOutputVariableUncertainty().at( indx ).getPdf( );
if ( NORMAL_PDF == uncertaintyPdf ) {
   int numSigmas = 1;
   double uncertainty =
     prop.getOutputVariableUncertainty( indx, numSigmas );
}
```

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *numSigmas* | is an integer value greater than zero, specifying the number of standard deviations of a Gaussian PDF uncertainty that is required as output. |

Returns

A double precision value containing the value of the uncertainty after all relevant computations based on the current input state.

See also

getOutputVariable

### A.3.2.18 double getOutputVariableUncertainty ( size_t *index*, bool *isUpperBound* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function returns a number representing the uncertainty associated with a value obtained from getOutputVariable. The type of uncertainty is obtained from getOutputVariableUncertainty().getPdf(). This function returns the upper or lower absolute uncertainty bound. The uncertainty value may change depending on the values of input variables to the Janus class. A typical application of this group of functions might be:

```
int indx = 0;
double = prop.getOutputVariable( indx );
UncertaintyPdf uncertaintyPdf =
  prop.getOutputVariableUncertainty().at( indx ).getPdf( );
if ( UNIFORM_PDF == uncertaintyPdf ) {
  bool isUpperBound = true;
  double upperBound =
    prop.getOutputVariableUncertainty( indx, isUpperBound );
  isUpperBound = false;
  double lowerBound =
    prop.getOutputVariableUncertainty( indx, isUpperBound );
}
```

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *isUpperBound* | is a Boolean value that specifies whether the absolute upper or lower bound of a uniform pdf is required as output. |

Returns

A double precision value containing the value of the uncertainty after all relevant computations based on the current input state.

See also

getOutputVariable

#### A.3.2.19 const char* getOutputVariableUnits ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An output variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS [7] in accordance with SI or other systems.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *units* attribute string is returned.

## A.4 Janus - Variables of All Types

### Functions

- double getCorrelationCoefficient (size_t indx1, size_t indx2)
- int getNumberOfVariableCorrelations (size_t index)
- size_t getNumberOfVariables () const
- const char * getVariableAxisSystem (size_t index) const
- double getVariableByIndex (size_t index)
- double getVariableByVarID (const char *varID)
- const char * getVariableDescription (size_t index) const
- VariableDef::VariableFlag getVariableFlag (size_t index) const
- const char * getVariableID (size_t index) const
- const char * getVariableName (size_t index) const
- VariableDef::VariableType getVariableType (size_t index) const
- double getVariableUncertainty (size_t index, size_t numSigmas)
- double getVariableUncertainty (size_t index, bool isUpperBound)
- const char * getVariableUnits (size_t index) const
- int setVariableByID (const char *varID, const double &x)
- int setVariableByIndex (size_t index, const double &x)

### A.4.1 Detailed Description

All the *variableDef* functions use an *index* based on the *variableDef* elements at DTD Level 1 in the XML dataset. Each variable referenced by the *index* can be used by multiple functions, and can be dependent or independent. The order of variable definitions within the DOM is arbitrary and the calling program is responsible for determining which *index* to address. For example (also see getNumberOfVariables and getVariableID):

```
int nv = prop.getNumberOfVariables();
for ( int i = 0 ; i < nv ; i++ ) {
  cout << "  Variable " << i << " : \n"
  << "   ID           : "
  << prop.getVariableID( i ) << "\n";
}
```

The *index* also addresses a corresponding location in a static array of variable current values within the instance's data.

In general, since DAVE-ML Version 2.0RC3 and Janus Version 1.10 these functions are deprecated and the required data should be accessed directly through the VariableDef class instead.

### A.4.2 Function Documentation

#### A.4.2.1 double getCorrelationCoefficient ( size_t *indx1*, size_t *indx2* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The function allows the caller to determine the amount of correlation between the uncertainty in two variables. The direction of correlation and sequence of variables is irrelevant.

Parameters

| | |
|---|---|
| *indx1* | has a range from 0 to (getNumberOfVariables() - 1), and selects the first variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |
| *indx2* | has a range from 0 to (getNumberOfVariables() - 1), and selects the second variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

The correlation coefficient relating the two variables' uncertainties is returned as a double. Where correlation has not been specified in the XML dataset, the coefficient is returned as zero. If either *indx1* or *indx2* is out of range, this function will return NaN.

#### A.4.2.2 int getNumberOfVariableCorrelations ( size_t *index* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The function allows the caller to determine the number of variables whose Gaussian uncertainties are correlated with the uncertainty of the specified variable.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the specified variable from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

An integer indicating the number of correlations is returned. If *index* is out of range this function will return -1.

#### A.4.2.3 size_t getNumberOfVariables ( ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function returns the total number of variables in the DOM. It includes all variables, makes no distinction between variable types, and provides no indication of whether they are dependent, independent, constant, output, or unused.

Returns

Total number of all variables defined in the XML file and successfully loaded into the DOM.

#### A.4.2.4 const char* getVariableAxisSystem ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. A variable's *axisSystem* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS . Typical values include "Body" or "Intermediate". Where no attribute is specified in the XML dataset, an empty string is returned.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *axisSystem* attribute string is returned.

#### A.4.2.5 double getVariableByIndex ( size_t *index* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The function provides a means of determining the current values of all variables defined within a Janus instance, whether independent or otherwise. Each of these values corresponds to a *variableDef*. For example:

```
int nv = prop.getNumberOfVariables();
for ( int i = 0 ; i < nv ; i++ ) {
  cout << "  Variable " << i << " : \n"
       << "   Value        : "
       << prop.getVariableByIndex( i ) << "\n";
}
```

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable value to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. It addresses the corresponding location in a static array within the instance's data structures. |

Returns

A double precision value containing the current variable value.

See also

setVariableByIndex
getVariableByVarID

#### A.4.2.6 double getVariableByVarID ( const char * *varID* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function provides an alternative means of determining the current values of all variables defined within a Janus instance, whether independent or otherwise. Each of these values corresponds to a *variableDef*.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the variable of which it is an attribute. |

Returns

A double precision value containing the current variable value. If the input does not match any dependent variable ID within the DOM, a standard exception is thrown.

See also

setVariableByVarID
getVariableByIndex

#### A.4.2.7 const char* getVariableDescription ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. A variable's *description* consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the description. Since description of a variable is optional, the returned string may be blank.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *description* child element contents string is returned.

#### A.4.2.8 VariableDef::VariableFlag getVariableFlag ( size_t *index* ) const

This function is deprecated. In new programs, the variable characteristics it describes should be determined by direct interrogation of the related VariableDef instance. It allows the caller to

determine a variable's status in respect of the flags specified in VariableDef::VariableFlag.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

The VariableDef::VariableFlag is returned on successful completion.

#### A.4.2.9 const char* getVariableID ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. A variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", that uniquely defines the variable. It may be used for indexing of all variable definitions, without distinction between variable types, and without requiring to know whether they are dependent, independent, constant, output, or unused. This function provides the means to determine the identities of the variables defined at sequential locations within the DOM.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *varID* attribute string is returned.

#### A.4.2.10 const char* getVariableName ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. A variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the ANSI/AIAA S-119-2011 standard [2] .

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *name* attribute string is returned.

#### A.4.2.11 VariableDef::VariableType getVariableType ( size_t *index* ) const

This function is deprecated and should not be used in new programs. Access this data through the VariableDef class instead. A variable that is specified as an output, a function evaluation result, or a MathML function should not normally have its value set directly by the calling program. This function allows the caller to determine a variable's status in this regard.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

The VariableType is returned on successful completion.

#### A.4.2.12 double getVariableUncertainty ( size_t *index*, size_t *numSigmas* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The function returns a number representing the uncertainty associated with a current value of a variable contained within a Janus instance. The type of uncertainty is obtained from getVariableUncertainty().getPdf(). This function returns as additive uncertainty the requested number of standard deviations.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed. |
| *numSigmas* | is an integer value greater than zero, specifying the number of standard deviations of a Gaussian PDF uncertainty that is required as output. |

Returns

A double precision value containing the value of the uncertainty based on the current state of the instance.

#### A.4.2.13 double getVariableUncertainty ( size_t *index*, bool *isUpperBound* )

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The function returns a number representing the uncertainty associated with a current value of a variable contained within a Janus instance. The type of uncertainty is obtained from getVariableUncertainty().getPdf(). For a uniform PDf, this function returns the upper or lower absolute uncertainty bound.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed. |
| *isUpperBound* | is a Boolean value that specifies whether the upper or lower bound of a uniform pdf is required as output. |

Returns

A double precision value containing the value of the uncertainty based on the current state of the instance.

#### A.4.2.14 const char* getVariableUnits ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. A variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS [7] in accordance with SI and other systems.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *units* attribute string is returned.

#### A.4.2.15 int setVariableByID ( const char * *varID*, const double & *x* )

This function is deprecated, and should not be used in new programs. Set this data through the VariableDef class instead. This function provides an alternative means to set the current values

of independent variables defined within a Janus instance. Each of these values corresponds to a *variableDef*. For example:

```
int retVal = MachCoeff.setVariableByID( "Mach", 0.95);
if ( 0 == retVal ) {
  cout << "\n Mach value set ... \n";
}
```

This function will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the variable of which it is an attribute. |
| *x* | is the double precision value to which the current value of the indexed variable will be set. |

Returns

0 is returned on successful completion. If the input does not match any dependent variable ID within the DOM, a standard exception is thrown.

See also

VariableDef::setValue()
setVariableByIndex()
getVariableByVarID()

#### A.4.2.16 int setVariableByIndex ( size_t *index*, const double & *x* )

This function is deprecated, and should not be used in new programs. Set independent variable values through the VariableDef class instead. This function provides the means to set the current values of independent variables defined within a Janus instance. Each of these values corresponds to a *variableDef*. For example, setting all input variables before evaluation:

```
char fileName[] = "pika_aero.xml";
Janus aeroCoeff( fileName );
int nv = aeroCoeff.getNumberOfVariables();
for ( int i = 0 ; i < nv ; i++ ) {
  if ( ISINPUT == aeroCoeff.getVariableType( i ) ) {
    cout << " Variable name : "
         << aeroCoeff.getVariableName( i )
         << "\n Enter value : ";
    double x;
    cin >> x;
    int result = aeroCoeff.setVariableByIndex( i, x );
    if ( 0 == result ) {
      cout << "\n Variable "
           << aeroCoeff.getVariableName( i ) << " set ...\n";
    }
  }
}
int nf = aeroCoeff.getNumberOfOutputs();
for ( int i = 0 ; i < nf ; i++ ) {
  double y = aeroCoeff.getOutputVariable( i );
  cout << "\n Function " << i << " value = " << y << "\n";
}
```

This function will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfVariables() - 1), and selects the variable to be addressed from the list of *VariableDef* s at DTD Level 1 of the DOM. |
| *x* | is the double precision value to which the current value of the indexed variable will be set. |

Returns

0 is returned on successful completion.

See also

VariableDef::setValue()
setVariableByID()
getVariableByIndex()

## A.5 Janus - Independent Variables

### Functions

- const char * getIndependentVariableAxisSystem (size_t indexf, size_t indexv) const
- double getIndependentVariableByIndex (const size_t &indexf, const size_t &indexv)
- const char * getIndependentVariableDescription (size_t indexf, size_t indexv) const
- janus::ExtrapolateMethod getIndependentVariableExtrapolation (size_t indexf, size_t indexv)
- const char * getIndependentVariableID (size_t indexf, size_t indexv) const
- int getIndependentVariableIndex (size_t indexf, const char *varID) const
- janus::InterpolateMethod getIndependentVariableInterpolation (const size_t &indexf, const size_t &indexv)
- const char * getIndependentVariableName (size_t indexf, size_t indexv) const
- int getIndependentVariableOrder (const size_t &indexf, const size_t &indexv) const
- const Provenance & getIndependentVariableProvenance (size_t indexf, size_t indexv) const
- VariableDef::VariableType getIndependentVariableType (const size_t &indexf, const size_t &indexv) const
- double getIndependentVariableUncertainty (const size_t &indexf, const size_t &indexv, const size_t &numSigmas)
- double getIndependentVariableUncertainty (const size_t &indexf, const size_t &indexv, const bool &isUpperBound)
- const char * getIndependentVariableUnits (size_t indexf, size_t indexv) const
- int getNumberOfIndependentVariables (size_t index) const
- int setIndependentVariableByIndex (size_t indexf, size_t indexv, const double &x)

### A.5.1 Detailed Description

Independent variables are those that form the inputs to computation of an output variable value, and are defined relative to the output variable that requires them. Therefore independent variables are always referenced by Janus through output variables. The order of the *variableDefs* defining output variables at DTD Level 1 within the DOM is arbitrary and the calling program is responsible for determining which output variable to address. For output variables dependent on more than one input variable, the order of independent variable references by the output variable is also arbitrary and must be determined by the calling program. Note that a variable considered independent by one output may itself be the output of another computation, and one variable may be used to compute many outputs.

The independent variable procedures use a first index based on the output variable, and a second index based on the list of input variables required by it. The input variable, although

it may be used by multiple outputs, is thus referenced through a different index for each output variable.

In general, since DAVE-ML Version 2.0RC3 and Janus Version 1.10 these functions are deprecated and the required data should be accessed directly through the VariableDef class instead.

### A.5.2 Function Documentation

#### A.5.2.1 const char* getIndependentVariableAxisSystem ( size_t *indexf*, size_t *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

An independent variable's *axisSystem* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS . Typical values include "Body" or "Intermediate". Where no attribute is specified in the XML dataset, an empty string is returned.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *axisSystem* attribute string is returned.

#### A.5.2.2 double getIndependentVariableByIndex ( const size_t & *indexf*, const size_t & *indexv* )

This function is deprecated, and should not be used in new programs. Get independent variable values through the VariableDef class instead.

This function provides a means of determining the current values of all variables defined within a Janus instance that are required as input signals for an output signal evaluation. For example:

```
char fileName[] = "pika_aero.xml";
Janus aeroCoeff( fileName );
int nf = aeroCoeff.getNumberOfOutputs();
for ( int i = 0 ; i < nf ; i++ ) {
  cout << "  Output " << i << " : \n"
       << "   Name                    : "
       << aeroCoeff.getOutputName( i ) << "\n";
```

```
  iv = aeroCoeff.getNumberOfIndependentVariables( i );
  for ( int j = 0 ; j < iv ; j++ ) {
    cout << " Independent variable name : "
         << aeroCoeff.getIndependentVariableName( i, j )
         << "\n                      value : "
         << aeroCoeff.getIndependentVariableByIndex( i, j )
         << "\n";
  }
}
```

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A double precision value containing the current value value of the independent variable selected, for immediate use to compute the output variable selected. If the independent variable is undefined or inapplicable, or an index is out of range, either *indexf* for the output variable or *indexv* for its independent variables, this function will return a NaN.

#### A.5.2.3 const char* getIndependentVariableDescription ( size_t *indexf,* size_t *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

An independent variable's *description* consists of a string of arbitrary length, which can include tabs and new lines as well as alphanumeric data. This means text formatting embedded in the XML source will also appear in the description. Since description of a variable is optional, the returned string may be blank.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *description* child element contents string is returned.

#### A.5.2.4 janus::ExtrapolateMethod getIndependentVariableExtrapolation ( size_t *indexf*, size_t *indexv* )

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

The *extrapolate* attribute of an independent variable referenced by an output variable describes any allowable extrapolation in the independent variable's degree of freedom contributing to computation of the output. This function makes that characteristic available to the calling program.

The *extrapolate* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require extrapolation, and MathML computations can incorporate extrapolation within their defining computations). For more details see Function::getIndependentVarExtrapolate().

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

An Extrapolation enum containing the extrapolation constraint on the independent variable selected, when used to compute the output variable selected.

#### A.5.2.5 const char* getIndependentVariableID ( size_t *indexf*, size_t *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. An independent variable's *varID* attribute is normally a short string without whitespace, such as "MACH02", that uniquely defines the variable. It may be used for indexing. This function may be used by the calling program to determine the input variable ID associated with each independent variable required by each output variable defined in the DOM.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A pointer to an XML *variableDef* tag's *varID* attribute string is returned.

#### A.5.2.6 int getIndependentVariableIndex ( size_t *indexf*, const char * *varID* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

When an independent variable is referred to through an output signal that uses the variable, the variable's *varID* attribute is uniquely related to the output signal and may be used as an index. This function is used by the calling program to establish the order of independent variable references through the output variable, since it is always more efficient to address a variable by numeric index than by variable ID.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *varID* | is a short string without whitespace, such as "MACH02", that uniquely defines the independent variable from the list associated with the output variable that uses it. |

Returns

An integer *index* in the range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), corresponding to the independent variable whose *varID* matches the input *varID*. If the input does not match any variable ID within the DOM, or the output variable *indexf* is out of range, returned value is -1.

#### A.5.2.7 janus::InterpolateMethod getIndependentVariableInterpolation ( const size_t & *indexf*, const size_t & *indexv* )

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

The *interpolate* attribute of an independent variable referenced by an output variable describes the form of interpolation to be used in that independent variable's degree of freedom when computing the output variable's value based on interpolation between gridded data points.

This function makes that information available to the calling program.

The *interpolate* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require interpolation, and MathML computations can incorporate interpolation within their defining computations).

The *interpolate* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then the enum representing its value within the Janus instance defaults to LINEAR, which is identical to POLY of order 1.

Applications are free to ignore this attribute (e.g. to sacrifice accuracy for speed in real time computations). The current Janus implementation limits POLY to order 3, and does not allow CSPLINE or LEGENDRE interpolations. Setting *interpolate* to "discrete'' will limit values in that degree of freedom to discrete steps centred on the breakpoint or independent variable values, with exact midpoint values rounded in the positive direction. Setting *interpolate* to "floor'' will limit values in that degree of freedom to discrete steps, such that for an independent variable value within an interval between any two breakpoints, the dependent variable takes on the function value at the lower breakpoint. Setting *interpolate* to "ceiling'' will limit values in that degree of freedom to discrete steps, such that for an independent variable value within an interval between any two breakpoints, the dependent variable takes on the function value at the upper breakpoint.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. Attempting to access an output variable outside the available range or not based on a tabular function will throw a standard out_of_range exception. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. Attempting to access an independent variable outside the available range will throw a standard out_of_range exception. |

Returns

An Interpolation enum containing the form of interpolation for the independent variable selected, when used to compute the output variable selected.

#### A.5.2.8 const char* getIndependentVariableName ( size_t *indexf,* size_t *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

An independent variable's *name* attribute is a string of arbitrary length, but normally short. It is not used for indexing, and therefore need not be unique (although uniqueness may aid both programmer and user), but should comply with the ANSI/AIAA S-119-2011 standard [2] .

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A character pointer (char∗) to an XML *variableDef* tag's *name* attribute string is returned.

#### A.5.2.9 int getIndependentVariableOrder ( const size_t & *indexf,* const size_t & *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

The *interpolate* attribute of an independent variable referenced by an output variable describes the type and order of interpolation to be used in that variable's degree of freedom when when computing the output variable's value based on interpolation between gridded data points. This function makes the order available to the calling program, from static data within the Janus instance that is initialised with *interpolate* data from the XML dataset.

The *interpolate* attribute is only applicable to an output variable defined in terms of a tabular *function* (i.e. constants never require interpolation, and MathML computations can incorporate interpolation within their defining computations).

The *interpolate* attribute is optional for all degrees of freedom for any *function* within the XML dataset, and if it is not set for any particular degree of freedom then the order component of its representation within the Janus instance defaults to 1, representing linear interpolation under either linear or polynomial interpolation types

Applications are free to ignore this attribute (e.g. to sacrifice accuracy for speed in real time computations). The current Janus implementation allows linear interpolation of order 0 or 1 and polynomial interpolation of order 0 to 3 inclusive. An order of 0, derived from a "discrete'' *interpolate* attribute will limit values in that degree of freedom to discrete steps centred on the breakpoint values. An order of -1, derived from a "floor'' *interpolate* attribute will limit values in that degree of freedom to discrete steps, such that for an independent variable value within an interval between any two breakpoints, the dependent variable takes on the function value at the lower breakpoint. An order of -2, derived from a "ceiling'' *interpolate* attribute will limit values in that degree of freedom to discrete steps, such that for an independent variable value within an interval between any two breakpoints, the dependent variable takes on the function value at the upper breakpoint.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Parameters

| | |
|---|---|
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

The interpolation order is returned on successful completion. If the *interpolate* attribute is inapplicable, or an index is out of range, either *indexf* for the output variable or *indexv* for its independent variables, this function will return -10.

#### A.5.2.10 const Provenance& getIndependentVariableProvenance ( size_t *indexf*, size_t *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

This function provides access to *provenance* elements contained in a DAVE-ML independent *variableDef* element. There may be zero or one of these elements for each independent variable in a valid dataset.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A reference to the requested Provenence class instance is returned.

#### A.5.2.11 VariableDef::VariableType getIndependentVariableType ( const size_t & *indexf*, const size_t & *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead. An independent variable that is also specified as an output, a function evaluation result, or a MathML function should not normally have its value set directly by the calling program. This function allows the caller to determine a variable's status in this regard.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Parameters

| | |
|---|---|
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

The VariableType is returned on successful completion.

#### A.5.2.12 double getIndependentVariableUncertainty ( const size_t & *indexf,* const size_t & *indexv,* const size_t & *numSigmas* )

This function is deprecated, and should not be used in new programs. Get independent variable uncertainty values through the VariableDef class instead. This function provides a means of determining the current values of uncertainty bounds for all variables defined within a Janus instance that are required as input signals for an output signal evaluation. The type of uncertainty is obtained from getIndependentVariableUncertainty().getPdf(). This function returns as additive uncertainty the requested number of standard deviations.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |
| *numSigmas* | is an integer value greater than zero, specifying the number of standard deviations of a Gaussian PDF uncertainty that is required as output. |

Returns

A double precision value containing the value of the uncertainty based on the current state of the instance.

#### A.5.2.13 double getIndependentVariableUncertainty ( const size_t & *indexf,* const size_t & *indexv,* const bool & *isUpperBound* )

This function is deprecated, and should not be used in new programs. Get independent variable uncertainty values through the VariableDef class instead. This function provides a means of determining the current values of uncertainty bounds for all variables defined within a Janus instance that are required as input signals for an output signal evaluation. The type of uncertainty is obtained from getIndependentVariableUncertainty().getPdf(). This function returns the upper or lower absolute uncertainty bound.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |
| *isUpperBound* | is a Boolean value that specifies whether the upper or lower bound of a uniform pdf is required as output. |

Returns

A double precision value containing the value of the uncertainty based on the current state of the instance.

#### A.5.2.14 const char* getIndependentVariableUnits ( size_t *indexf*, size_t *indexv* ) const

This function is deprecated, and should not be used in new programs. Get independent variable data through the VariableDef class instead.

An independent variable's *units* attribute is a string of arbitrary length, but normally short, and complying with the format requirements chosen by DST AD APS [7] in accordance with SI and other systems.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |

Returns

A character pointer (char*) to an XML *variableDef* tag's *units* attribute string is returned.

#### A.5.2.15 int getNumberOfIndependentVariables ( size_t *index* ) const

This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. This function is deprecated, and should not be used in new programs. Access this data through the VariableDef class instead. The function returns the number of independent variables associated with any output variable. Note an independent variable can be of any of the types in the enum VariableDef::VariableType.

Parameters

| | |
|---|---|
| *index* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |

Returns

Total number of independent variables referenced by the output variable selected.

#### A.5.2.16 int setIndependentVariableByIndex ( size_t *indexf*, size_t *indexv*, const double & *x* )

This function is deprecated, and should not be used in new programs. Set independent variable values through the VariableDef class instead. This procedure is one of the means of setting an input variable value within the Janus instance's data structure. It will throw a standard exception if an attempt is made to modify the value of a variable that is specified in the XML dataset as being evaluated by computation.

Parameters

| | |
|---|---|
| *indexf* | has a range from 0 to (getNumberOfOutputs() - 1), and selects the output variable to be addressed. |
| *indexv* | has a range from 0 to (getNumberOfIndependentVariables ( indexf ) - 1), and selects the independent variable to be addressed through the output variable that uses it. |
| *x* | is the double precision value to which the current value of the indexed variable will be set. |

Returns

0 is returned on successful completion.

See also

VariableDef::setValue()

DISTRIBUTION LIST

The Janus C++ Library – An Interface Class for DAVE-ML Compliant XML-Based Flight Model Datasets

Geoff Brian and Shane D Hill

**Task Sponsor**

RL-APS

**S&T Program**

Chief of Aerospace Division
Research Leader – Aircraft Performance and Survivability
Group Leader – Aerodynamic and Aeroelasticity
Author(s): Geoff Brian and Shane D Hill

Michael Young
Michael Grant
Kylie Bedwell
Jonathan Dansie
Andrew Snowden

DST Group Research Library (Report Distribution Officer)

Defence Science Communications (ST Publications)

| DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA | | | 1. DLM/CAVEAT (OF DOCUMENT) |
|---|---|---|---|
| 2. TITLE<br>The Janus C++ Library – An Interface Class for DAVE-ML Compliant XML-Based Flight Model Datasets | | 3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED REPORTS THAT ARE LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION)<br>Document (U)<br>Title (U)<br>Abstract (U) | |
| 4. AUTHORS<br>Geoff Brian and Shane D Hill | | 5. CORPORATE AUTHOR<br>Defence Science and Technology Group<br>506 Lorimer St,<br>Fishermans Bend, Victoria 3207, Australia | |
| 6a. DST Group NUMBER<br>DST-Group–TN–1658 | 6b. AR NUMBER<br>016-923 | 6c. TYPE OF REPORT<br>Technical Note | 7. DOCUMENT DATE<br>July, 2017 |
| 8. Objective ID<br>AV9426462 | 9. TASK NUMBER | 10. TASK SPONSOR<br>RL-APS | |
| 13. DST Group Publications Repository<br>`http://dspace.dsto.defence.gov.au/dspace/` | | 14. RELEASE AUTHORITY<br>Chief, Aerospace Division | |

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

*Approved for Public Release*

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. RESEARCH LIBRARY THESAURUS

Flight simulation, Aerospace vehicles, Data Syntax, DAVE-ML, XML

19. ABSTRACT

The Dynamic Aerospace Vehicle Exchange Markup Language (DAVE-ML) is a syntactical language for exchanging flight vehicle dynamic model data. It has been developed in conjunction with the ANSI/AIAA S-119-2011 Flight Dynamics Model Exchange Standard prepared by the American Institute of Aeronautics and Astronautics (AIAA) Modeling and Simulation Technical Committee (MSTC). The purpose of DAVE-ML is to provide a framework to encode entire flight vehicle simulation data packages for exchange between simulation applications and the long-term archiving of model data. This document describes an application programming interface (API) to the DAVE-ML dataset structure that has been developed by the Defence Science and Technology (DST) Group. The API is known as 'Janus'.