pyCGNS.VAL/Manual

Release 4.2.0

Marc Poinot

October 24, 2012

CONTENTS

The **VALidator** tool is a CGNS/Python tree checker. It parses a CGNS/Python tree and applies three kind of verifications. The first is the structural check, the second is the CGNS/SIDS check and the last is the user defined check.

The **CGNS.VAL** tool has a command line interface, it can be run in a user's shell window and returns a set of diagnostics on the standard output. The **CGNS.VAL** tool has an embedded check tool which actually makes calls to the *VALidator* API. Then any user can use this *VALidator* API to make his own embedded check tool.

CHAPTER

ONE

QUICK START

The **CGNS.VAL** tool takes a CGNS/HDF5 file as argument and checks its contents, with respect to tree structure and CGNS/SIDS recommandations. The simple command line is:

```
CGNS.VAL naca012.hdf
```

Which loads the file, runs the checks and returns a list of diagnostics:

```
----------------------------------------------------------------------------
/Base#1/domain21/blockName
[S004:E] DataType [MT] not allowed for this node
[S007:E] Node [ElementRange] of type [IndexRange_t] is mandatory
[S191:E] Bad node value shape
[U110:E] Cannot handle such ElementType [None]

----------------------------------------------------------------------------
/Base#1/domain24
[U104:W] No ReferenceState found at Zone level

----------------------------------------------------------------------------

### CGNS/Python tree *NOT* Compliant
```

Each diagnostic entry starts with the node path followed by the list of warnings and errors detected at this node level.

CHAPTER

TWO

REFERENCE

2.1 Command line

The **CGNS.VAL** tool uses the *VALidator* API. The **CGNS.NAV** tool uses this API as well, and in this case the diagnostics output are displayed into a graphical window in a hierarchical form.

The **CGNS.VAL** command line usage is the following:

```
CGNS.VAL [options] file.hdf

-p <path>    : Start check at this node
-f           : Flat mode, do not recurse on tree
-u <key>     : Check user requirements identified by <key>
-k           : Gives list of known user requirements keys
-l           : List all known diagnostics
-r <idlist> : remove the list of ids ( -r U012:U023:U001 )
-m           : Output by message id instead of path
-h           : help
-v           : verbose (trace)
```

The usual command line such as:

```
CGNS.VAL -r U103:U104 -v -u elsA naca012.hdf
```

runs a check on the **elsA** user defined grammar, excluding the diagnostics *U103* and *U104*.

You want to known the complete diagnostics list, you run:

```
CGNS.VAL -lu elsA
```

and you get the messages for the *U103* and *U104* diagnostics:

```
[U103:W] No ReferenceState found at Base level
[U104:W] No ReferenceState found at Zone level
```

you have ignored in the previous command.

The -k option returns the list of known user grammars.

2.2 Diagnostics

There are two levels of diagnostics, the *Error* and the *Warning*. The check process continues even if it has and *Error*. A diagnostics has a key, *S003* for example, which is composed of a single letter and a unique number for each letter. The *G* series stand for *generic*, the *S* for *SIDS* and *U* for *User*.

The checking process always starts with *G* checks, then *S* checks, then *U* checks if it has been requested.

2.2.1 Generic diagnostics

The first checks are the generic checks. **VAL** performs very low level controls on the node structure, node name and *CGNSLibraryVersion*. This latter is an exception, it is the only *SIDS*-related data to be verified. As a matter of fact, *CGNSLibraryVersion* is more an implementation node than a *SIDS* node.

2.2.2 SIDS diagnostics

The core checks are *SIDS* checks. All the **CGNS/SIDS** document requirements are inspected, default values are reported.

2.2.3 User defined diagnostics

A user defined grammar is identified by a *key*.

2.2.4 Diagnostics list

The following table gives existing diagnostics including the *elsA* user defined set of checks:

```
[G001:E] CGNSLibraryVersion [%s] is too old for current check level
[G002:E] CGNSLibraryVersion is incorrect
[G003:E] Name [%s] is not valid
[G004:E] Name [%s] is a duplicated child name
[G005:E] PANIC: Cannot find node with path [%s]
[G006:E] PANIC: Node data is not numpy.ndarray or None (child of [%s])
[G007:E] PANIC: Node children is not a list (child of [%s])
[G008:E] PANIC: Node name is not a string (child of [%s])
[G009:E] PANIC: Node is not a list of 4 objects (child of [%s])
[G010:E] PANIC: Node is empty list or None (child of [%s])
[S001:E] Unknown SIDS type [%s]
[S002:E] SIDS type [%s] not allowed as child of [%s]
[S003:E] SIDS type [%s] not allowed for this node
[S004:E] DataType [%s] not allowed for this node
[S005:E] Node [%s] of type [%s] is not allowed as child
[S006:E] Node [%s] of type [%s] is allowed only once as child
[S007:E] Node [%s] of type [%s] is mandatory
[S101:E] Unknown ZoneType value
[S102:E] Unknown SimulationType value
[S103:E] Unknown GridLocation value
[S104:E] Unknown GridConnectivityType value
[S105:E] Unknown DataClass value
[S106:E] Unknown BCDataType value
[S107:E] Unknown RigidMotionType value
[S108:E] Unknown BCType value
[S109:E] Unknown ElementType value
[S110:E] Unknown MassUnit value
[S111:E] Unknown TimeUnit value
[S112:E] Unknown LengthUnit value
[S113:E] Unknown TemperatureUnit value
[S114:E] Unknown AngleUnit value
[S115:E] Unknown ElectricCurrentUnit value
[S116:E] Unknown SubstanceAmountUnit value
[S117:E] Unknown LuminousIntensityUnit value
[S151:W] Default GridLocation is set to Vertex
[S152:W] Default GridConnectivityType is set to Overset
[S191:E] Bad node value shape
[S201:E] Inconsistent PhysicalDimension/CellDimension
[S202:E] Bad value for CellDimension
[S203:E] Bad value for PhysicalDimension
```

```
[S204:E] Bad Transform values
[S205:E] Bad ElementSizeBoundary value
[S301:E] Reference to unknown family [%s]
[S302:E] Reference to unknown additional family [%s]
[U101:W] No Zone in this Base
[U102:W] No GridCoordinates in this Zone
[U103:W] No ReferenceState found at Base level
[U104:W] No ReferenceState found at Zone level
[U105:E] At least one structured Zone is required in the Base
[U106:E] Transform is not right-handed (direct)
[U107:W] No FlowSolution# found for output definition
[U108:W] No FlowSolution#Init found for fields initialisation
[U109:E] Cannot handle such GridLocation [%s]
[U110:E] Cannot handle such ElementType [%s]
```

Note: The list has been generated with the command *CGNS.VAL -lu elsA*.

2.3 Extending with user defined rules

The extension requires a top file with the name *CGNS_VAL_USER_<key>.py* with the *CGNS_VAL_USER_Checks* declaration. This file has to be importable, which means the file should be in one of the *PYTHONPATH* directories.

Writing an extension requires good Python skills. The user has to define a new class, with *CGNS.VAL.grammars.SIDS.SIDSbase* as one of its base class, a set of per-node check methods and a set of associated diagnostics.

2.3.1 Diagnostics

A diagnostic is composed of a key, a level and a message:

```
('U101',CGM.CHECK_WARN,'No Zone in this Base')
```

The *CGM.CHECK_WARN* enumerate comes from the *CGNS.VAL.parse.messages* module, in that example imported as *CGM*.

All diagnostics are set into the class as a dictionnary, using the method *addMessages* of the *log* object of the class. This is performed in the *__init__* method of the user class:

```
def __init__(self,log):
  CGS.SIDSbase.__init__(self,log)
  self.log.addMessages(USER_MESSAGES)
```

Do not forget the base class initialisation. The *log* argument is managed by the **CGNS.VAL** internals, you should not take care about it.

2.3.2 Per-node check methods

The **CGNS.VAL** internals are in charge of parsing the tree. Each time a node is entered, the corresponding check function is called. The functions args are always the same: the *path* of the node in which you are entering, the *node* itself as a **CGNS/Python** list, the *parent* node as a **CGNS/Python** list, the complete **CGNS/Python** *tree* and the *log* object.

The function should have the name of the entered node type. For example, the *Zone_t* is called each time the parser enters into a *Zone_t* node:

```
def Zone_t(self,pth,node,parent,tree,log):
  rs=self.sids.Zone_t(self,pth,node,parent,tree,log)
  if (CGK.GridCoordinates_s not in CGU.childNames(node)):
    rs=log.push(pth,NOGRIDZONE)
  if (not CGU.hasChildNodeOfType(node,CGK.ReferenceState_ts)):
    rs=log.push(pth,NOZREFSTATE)
  return rs
```

The first step is to call the *Zone_t* for the *SIDS* checker. This is strongly recommanded, but not mandatory... The *rs* variable contains the current status for this node: that can be *CHECK_GOOD*,'CHECK_FAIL' or *CHECK_WARN*. This status should be the return of your check function.

We check values of the node and in case of problem, we *push* the diagnostic into the log. The *path* of the node is pushed as well.

Now each check is a test on several values of the current node, the use of *CGNS.PAT.cgnsutils*, *CGNS.PAT.cgnstypes* and *CGNS.PAT.cgnskeywords* would help.

You should not parse the tree by yourself, unless you have to check consistency with other node values. The example below shows a control on the mandatory *ElementRange_s* node of an *Elements_t* node (The *context* object is detailled in the next section).:

```
def Elements_t(self,pth,node,parent,tree,log):
  rs=CGM.CHECK_OK
  if (CGU.getShape(node)!=(2,)):
    rs=log.push(pth,BADVALUESHAPE)
  else:
    et=node[1][0]
    eb=node[1][1]
    self.context[CGK.ElementType_s]=et
    self.context[CGK.ElementSizeBoundary_s]=eb
    if (et not in range(0,len(CGK.ElementType)+1)):
      rs=log.push(pth,UKELEMTYPE)
    if (eb==0): bad_eb=False
    elif (eb<0): bad_eb=True
    else:
      bad_eb=True
      ecnode=CGU.getNodeByPath(tree,pth+'/'+CGK.ElementRange_s)
      if (    (ecnode is not None)
          and (CGU.getShape(node)==(2,))
          and (CGU.getValueDataType(ecnode)==CGK.I4)
          and (ecnode[1][1]>eb)): bad_eb=False
    if (bad_eb):
      rs=log.push(pth,BADELEMSZBND)
  return rs
```

2.3.3 Context

The parser has a *context* for global and local data. It is a dictionnary with *SIDS* names as keys, the values are overwritten during the parse but a value is always correct for a given sub-tree. For example to set the base dimension attributes:

```
cd=node[1][0]
pd=node[1][1]
self.context[CGK.CellDimension_s]=cd
self.context[CGK.PhysicalDimension_s]=pd
```

And to get them later on in another node function:

```
if (not CGS.transformIsDirect(tr,self.context[CGK.CellDimension_s])):
  rs=log.push(pth,NOTRHTRANSFORM)
```

In this test, *CGS* stands for *CGNS.APP.sids.utils*.

CHAPTER

THREE

GLOSSARY

cgns.org The official CGNS web site, by extension any document on this web site has an *official* taste...

CGNS The specific purpose of the CFD General Notation System (CGNS) project is to provide a standard for recording and recovering computer data associated with the numerical solution of the equations of fluid dynamics. See also the *How to?*.

CGNS/SIDS The Standard Interface Data Structure is the specification of the data model. This public document describes the syntax and the semantics of all tree-structured data required or proposed for a CFD simulation.

CGNS/MLL The Mid-Level Library is an example implementation of *CGNS/SIDS* on top of *CGNS/ADF* and *CGNS/HDF5* mappings. This library has a C and a Fortran API.

CGNS/ADF The Advanced Data Format *CGNS/SIDS* implementation. A binary storage format and its companion library, developped by *Boeing*.

CGNS/HDF5 The Hierarchical Data Format *CGNS/SIDS* implementation. A binary storage format and its companion library (see below).

CGNS/Python The Python programming language *CGNS/SIDS* implementation.

CHLone A *CGNS/HDF5* compliant implementation. The CHLone library is available on SourceForge.

HDF5 A powerful storage system for large data. The HDF5 library should be seen as a middleware system with a lot of powerful features related to efficient, portable and trustable storage mean.

python An object oriented interpreted programming language.

cython A compiler tool that translate Python/Numpy into C code for performance purpose.

numpy The numerical library for Python. *Numpy* is used to store the data in Python arrays which have a direct memory mapping to actual C or Fortran memory.

VTK A visualization toolkit used to display 3D objects ni *CGNS.NAV*.

PySide The Python interface for the Qt toolkit. PySide

Qt A powerful graphical toolkit available under GPL v3, LGPL v2 and a commercial license. The current use of Qt is under LGPL v2 in pyCGNS.

3.1 VAL Index

- *genindex*