Seven keys for practical understanding and use of CGNS

Marc Poinot[*]
SAFRAN
marc.poinot@safrangroup.com

Christopher L. Rumsey[†]
NASA Langley Research Center
c.l.rumsey@nasa.gov

We present key features of the CGNS standard, focusing on its two main elements, the data model (CGNS/SIDS) and its implementations (CGNS/HDF5 and CGNS/Python). The data model is detailed to emphasize how the topological user oriented information, such as families, are separated from the actual meshing that could be split or modified during the CFD workflow, and how this topological information is traced during the meshing process. We also explain why the same information can be described in multiple ways and how to handle such alternatives in an application. Two implementations, using HDF5 and Python, are illustrated in several use examples, both for archival and interoperability purposes. The CPEX extension formalized process is explained to show how to add new features to the standard in a consensual way; we present some of the next extensions to come. Finally we conclude by showing how powerful a consensual public approach like CGNS can be, as opposed to a stand-alone private one. All throughout the paper, we demonstrate how the use of CGNS could be of great benefit for both the meshing and CFD solver communities.

I. Introduction

The *CFD General Notation System*[1][2] (CGNS) is a public standard for the CFD community. It has had more than 20 years of feedback from users throughout the world in industry, universities, and government research labs. The CGNS name is now well known in the CFD community, but it is often only associated with the CGNS library (CGNS/MLL) used by CFD tools. In this paper, we detail some of the less well-known features of the CGNS standard, particularly those that can help with facets other than simply the storage of large data on a disk. Occasionally remarks are made in public forums about CGNS that show a misconception of the standard, and, probably the most important point, a lack of knowledge of how users can interact with CGNS in order to extend it to their needs. The following list represents some common comments concerning CGNS:

- CGNS is a library
- CGNS cannot fit my own data structure
- There are too many ways to describe the same feature in CGNS
- CGNS cannot handle parallel processing
- CGNS is only for archival data
- CGNS files are too big
- The recommended implementation is HDF5 but only ADF is available
- There are few useful tools for CGNS manipulation
- I already have HDF5 and I do not need CGNS
- CGNS inefficiently stores time data and statistics

In this paper, we address these remarks. However, rather than giving answers to each comment at the start, we think one should first look to the whole scope of CGNS and follow a didactic outline by introducing some general concepts that form the underlying basis of the CGNS standard. In particular, we have identified seven key points that will help the reader to better embrace the CGNS scope and to find practical uses for the CGNS components in his or her work.

[*]SAFRAN, Modelisation and Simulation Dept., 78772 Magny-les-Hameaux, France.
[†]NASA Langley Research Center, Computational AeroSciences Branch, Mail Stop 128, Hampton, VA 23681, U.S.A.

I.A. CGNS components

Behind the CGNS acronym resides more than a library. The CGNS term itself can be a reference to the group of people who gather to define an interoperable CFD system, a reference to a specification document, a library or even a process such as the extension proposal process. The exact use of the correct acronym helps in the understanding of the introductory remarks we noted before. For example, saying *CGNS is not parallel* is not the same as saying *CGNS/MLL is not parallel* or *CGNS/SIDS is not parallel.* Table 1 lists the most common acronyms we find in the CGNS scope; we use them in the next sections of this paper.

Table 1. Usual CGNS acronyms.

Acronym	Traduction	Entity
CGNS	CFD General Notation System	Identifier
CGNS/SC	Steering Committee	Group of people
CGNS/SIDS	Standard Interface Data Structure	Textual document
CGNS/CPEX	CGNS Proposal for EXtension	Procedure and textual document
CGNS/FMM	File Mapping Manual	Textual document
CGNS/HDF5	HDF5 implementation	Textual document
CGNS/Python	Python implementation	Textual document
CGNS/MLL	Mid-Level Library	Software library

The most important acronym is the *SIDS*. This is the actual standard specification of the CGNS data model. The *steering committee (SC)* is the group of people committed to the *CGNS charter*; they decide together how to extend the data model, how to maintain the library, and how to spread the standard. The *extension process (CPEX)* is a loosely formalized suite of steps that include rationales, documentation reviews, votes and finally implementations. Then we go into actual implementations. The guide describing the way the *SIDS* is mapped to any support is the *File Mapping Manual.* It can be applied to files as well as to any other software system that can be represented by a tree. Two main implementations are supported today, the *HDF5* and the *Python* implementations. Finally, the *Mid-level library* is an example high level interface compliant to the *CGNS/HDF5* mapping with both C and Fortran Application Programming Interfaces (APIs).

I.B. Two example configurations

We use the *AIAA High Lift Workshop* and *Drag Prediction Workshop* configurations to support our examples throughout this paper. The Computer Aided Design (CAD) support, the meshes and many other materials are available from NASA websites.[a]

The first configuration is the *High Lift Common Research Model* (HL-CRM) from HiLiftPW-3.[3] (The associated CGNS examples for HL-CRM are only meshes; these are generated by many different tools by the workshop committee and participants, but most of them do not illustrate the expressiveness of CGNS. For this purpose, we added a more complete CGNS representation for our second configuration below.) The original CAD file for the HL-CRM wing-body configuration includes geometric entities. These are the actual elements the user wants to refer to, such as WING, BODY, SLAT, FLAP, etc. They are identified in the geometry by labels. The mesh tools **should** keep them in the CGNS file. The traceability of these labels during all the use of the configuration is critical; the user does not want to keep track of the translation of these entities in meshes. The particular example file used in this paper can be found on the NASA High Lift Workshop web site as a *committee grid* in:

```
B1-HLCRM_UnstrTet_PW/FullGap/CGNS/Woeber_Pointwise_HLCRM_FullGap_Tets_Coarse.cgns
```

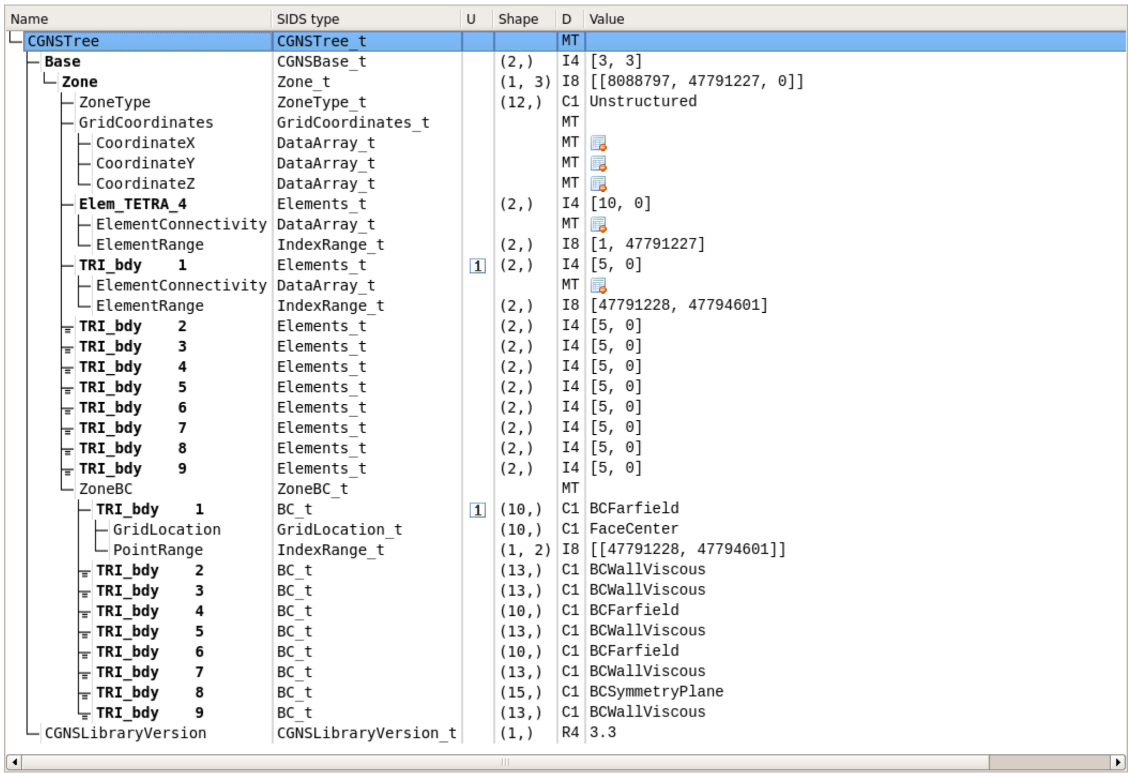
This file has the *CGNS/ADF* file format. It requires the `cgnsconvert` tool from the *CGNS/MLL* distribution to transform it to a *CGNS/HDF5* file.

The second configuration, the Common Research Model (CRM) from DPW-5,[4] shows a more complete CGNS representation. There are four files, linked to each other by HDF5 links, so that if the top tree is opened and parsed, then each subtree can be entered in a transparent way. This linking mechanism is

[a] `https://hiliftpw.larc.nasa.gov` and `https://aiaa-dpw.larc.nasa.gov`, cited 9/26/2017.

described in more detail in section III.I. This second CGNS example includes all the information required for a CFD computation; it contains the grids, the initial fields and the results. Some of the data structure refers to well-defined CGNS patterns from the SIDS while others are CGNS compliant user-defined structures. The files for this example are shipped with the *pyCGNS* Python package and can be found on the *pyCGNS* website.[b]

Figures 1 and 2 show screenshots of the HL-CRM and CRM CGNS trees, respectively. These views are partial views of the full CGNS configuration. For both examples, each node in the tree has a name, a SIDS type, a value with its shape (dimensions), its data type and the actual value. Note that in the examples, some information has been lost, in particular the end-user information related to the logical CAD parts of the configuration.



Name	SIDS type	U	Shape	D	Value
CGNSTree	CGNSTree_t			MT	
Base	CGNSBase_t		(2,)	I4	[3, 3]
Zone	Zone_t		(1, 3)	I8	[[8088797, 47791227, 0]]
ZoneType	ZoneType_t		(12,)	C1	Unstructured
GridCoordinates	GridCoordinates_t			MT	
CoordinateX	DataArray_t			MT	▦
CoordinateY	DataArray_t			MT	▦
CoordinateZ	DataArray_t			MT	▦
Elem_TETRA_4	Elements_t		(2,)	I4	[10, 0]
ElementConnectivity	DataArray_t			MT	▦
ElementRange	IndexRange_t		(2,)	I8	[1, 47791227]
TRI_bdy 1	Elements_t	1	(2,)	I4	[5, 0]
ElementConnectivity	DataArray_t			MT	▦
ElementRange	IndexRange_t		(2,)	I8	[47791228, 47794601]
TRI_bdy 2	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 3	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 4	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 5	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 6	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 7	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 8	Elements_t		(2,)	I4	[5, 0]
TRI_bdy 9	Elements_t		(2,)	I4	[5, 0]
ZoneBC	ZoneBC_t			MT	
TRI_bdy 1	BC_t	1	(10,)	C1	BCFarfield
GridLocation	GridLocation_t		(10,)	C1	FaceCenter
PointRange	IndexRange_t		(1, 2)	I8	[[47791228, 47794601]]
TRI_bdy 2	BC_t		(13,)	C1	BCWallViscous
TRI_bdy 3	BC_t		(13,)	C1	BCWallViscous
TRI_bdy 4	BC_t		(10,)	C1	BCFarfield
TRI_bdy 5	BC_t		(13,)	C1	BCWallViscous
TRI_bdy 6	BC_t		(10,)	C1	BCFarfield
TRI_bdy 7	BC_t		(13,)	C1	BCWallViscous
TRI_bdy 8	BC_t		(15,)	C1	BCSymmetryPlane
TRI_bdy 9	BC_t		(13,)	C1	BCWallViscous
CGNSLibraryVersion	CGNSLibraryVersion_t		(1,)	R4	3.3

Figure 1. A CGNS partial tree of the HiLiftPW-3 HL-CRM configuration; limited to the grid, boundary conditions, and mesh connectivities.

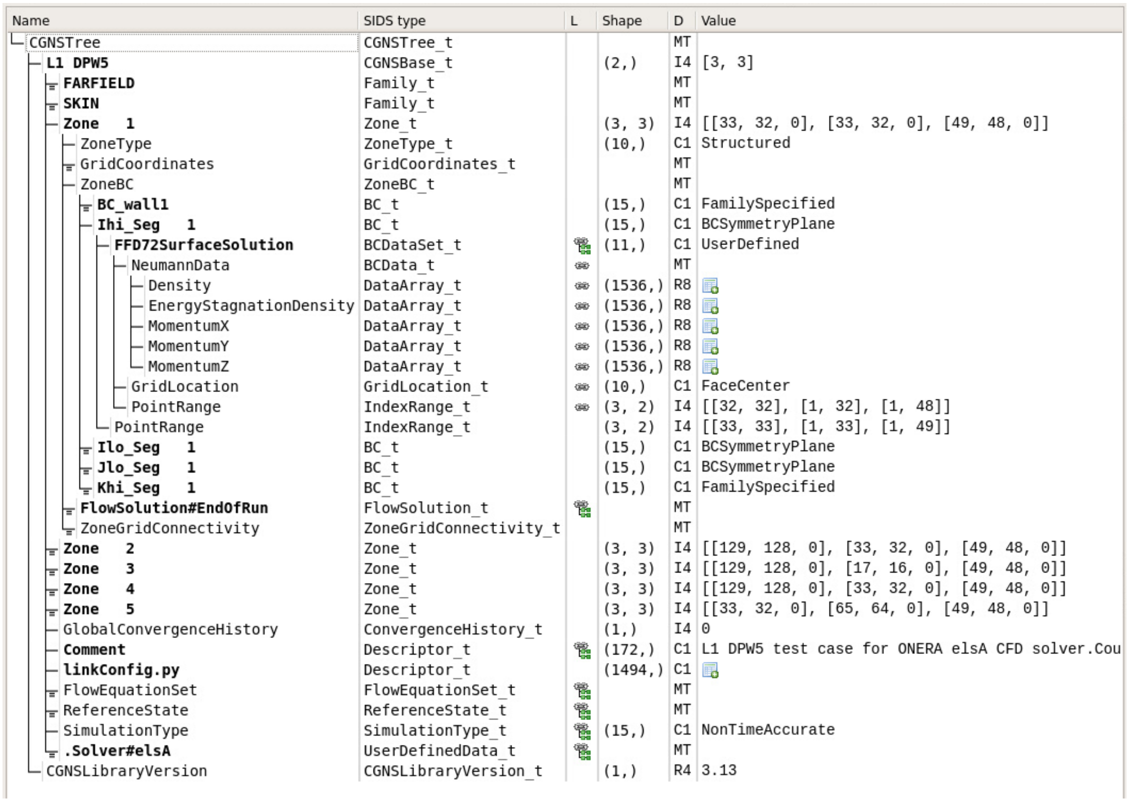
II. The CGNS fundamental points

We have identified seven key points that we believe should be pushed as part of CGNS advocacy. We hope that these points will help give the reader greater fundamental knowledge about what CGNS really is, and also encourage its use in a larger context than as just another file format.

II.A. First key: Abstract model

A *data model* is an abstraction of a specialized set of objects and their relationships. It is dedicated to an application scope, which is *CFD* in our case. We describe in an abstract way the meshes, the boundary conditions, the equations we solve and many other CFD objects. This model helps people to understand exactly what is contained in the file. It forms a common basis in which you are detailing the objects you are dealing with in your activities. There is a need for such an accepted convention or vocabulary for all members of a team, or for partners in a project, or for a customer/contractor relationship. You have to set the context, the terms and be sure all the meaning you put in your words are understood the same way by all the people with whom you are communicating. You can always define your own convention for describing

[b]`https://github.com/pyCGNS/pyCGNS/tree/master/demo/DPW5`, cited 10/02/2017.



```
Name                                   SIDS type                 L  Shape     D   Value
└ CGNSTree                             CGNSTree_t                             MT
  ├ L1 DPW5                            CGNSBase_t                   (2,)      I4  [3, 3]
  │ ├ FARFIELD                         Family_t                               MT
  │ ├ SKIN                             Family_t                               MT
  │ ├ Zone   1                         Zone_t                       (3, 3)    I4  [[33, 32, 0], [33, 32, 0], [49, 48, 0]]
  │ │ ├ ZoneType                       ZoneType_t                   (10,)     C1  Structured
  │ │ ├ GridCoordinates                GridCoordinates_t                      MT
  │ │ ├ ZoneBC                         ZoneBC_t                               MT
  │ │ │ ├ BC_wall1                     BC_t                         (15,)     C1  FamilySpecified
  │ │ │ ├ Ihi_Seg   1                  BC_t                         (15,)     C1  BCSymmetryPlane
  │ │ │ │ ├ FFD72SurfaceSolution       BCDataSet_t                  (11,)     C1  UserDefined
  │ │ │ │ │ ├ NeumannData              BCData_t                               MT
  │ │ │ │ │ │ ├ Density                DataArray_t                  (1536,)   R8
  │ │ │ │ │ │ ├ EnergyStagnationDensity DataArray_t                 (1536,)   R8
  │ │ │ │ │ │ ├ MomentumX              DataArray_t                  (1536,)   R8
  │ │ │ │ │ │ ├ MomentumY              DataArray_t                  (1536,)   R8
  │ │ │ │ │ │ └ MomentumZ              DataArray_t                  (1536,)   R8
  │ │ │ │ │ ├ GridLocation             GridLocation_t               (10,)     C1  FaceCenter
  │ │ │ │ │ └ PointRange               IndexRange_t                 (3, 2)    I4  [[32, 32], [1, 32], [1, 48]]
  │ │ │ │ └ PointRange                 IndexRange_t                 (3, 2)    I4  [[33, 33], [1, 33], [1, 49]]
  │ │ │ ├ Ilo_Seg   1                  BC_t                         (15,)     C1  BCSymmetryPlane
  │ │ │ ├ Jlo_Seg   1                  BC_t                         (15,)     C1  BCSymmetryPlane
  │ │ │ └ Khi_Seg   1                  BC_t                         (15,)     C1  FamilySpecified
  │ │ ├ FlowSolution#EndOfRun          FlowSolution_t                         MT
  │ │ └ ZoneGridConnectivity           ZoneGridConnectivity_t                 MT
  │ ├ Zone   2                         Zone_t                       (3, 3)    I4  [[129, 128, 0], [33, 32, 0], [49, 48, 0]]
  │ ├ Zone   3                         Zone_t                       (3, 3)    I4  [[129, 128, 0], [17, 16, 0], [49, 48, 0]]
  │ ├ Zone   4                         Zone_t                       (3, 3)    I4  [[129, 128, 0], [33, 32, 0], [49, 48, 0]]
  │ ├ Zone   5                         Zone_t                       (3, 3)    I4  [[33, 32, 0], [65, 64, 0], [49, 48, 0]]
  │ ├ GlobalConvergenceHistory         ConvergenceHistory_t         (1,)      I4  0
  │ ├ Comment                          Descriptor_t                 (172,)    C1  L1 DPW5 test case for ONERA elsA CFD solver.Cou
  │ ├ linkConfig.py                    Descriptor_t                 (1494,)   C1
  │ ├ FlowEquationSet                  FlowEquationSet_t                      MT
  │ ├ ReferenceState                   ReferenceState_t                       MT
  │ ├ SimulationType                   SimulationType_t             (15,)     C1  NonTimeAccurate
  │ └ .Solver#elsA                     UserDefinedData_t                      MT
  └ CGNSLibraryVersion                 CGNSLibraryVersion_t         (1,)      R4  3.13
```

Figure 2. CGNS partial tree of the DPW-5 CRM configuration; includes grid, boundary conditions, mesh connectivities, and complete results from a CFD computation (courtesy ONERA).

something, but it is convenient to find one already usable and even better if this convention is a public standard.

The abstract model helps define the difference between the concepts you are dealing with and the way you actually implement them. The code you have developed, or the code you are using, is the translation of these concepts in terms of computer programming language, and you almost always have to make choices that cause you to lose some information.

II.A.1. Recipe: define a configuration instead of a mesh

The HL-CRM configuration comes with two options: the gapped flap and the partially-sealed flap. Actually, this is almost exactly the same vehicle in both cases. So why not have the same configuration with an included option? If you consider the same configuration in both cases, the point would be to identify what really is the difference between these two. As a matter of fact, it is a CAD variant that leads to two different mesh sets.

The HL-CRM mesh has the volume elements and the boundary condition descriptions. If you refer to the NASA website you will find a significant amount of additional information, much of which is important for the mesh generation itself or for the understanding of the configuration. While it may not be a good idea to put everything in your CGNS data model, it makes sense to put data you would have to refer to on a sheet of paper on your desk, while you are working on your computer. Let us give some examples: we extracted some requirements from the NASA website and we translated it into a possible addition to the abstract model of our configuration. This information is outlined in Table 2.

All of this information may not be strictly necessary for your code, but it may be relevant for the whole process. The data model is the place to define all the information that is required or even helpful to reproduce the computation. There is **no useless data** in this case, because this data helps to understand or reproduce the simulation.

Table 2. Possible additional non-mesh information useful for the HL-CRM configuration.

Web site remark	Abstract model addition
Note that there are two (2) sets of distinct three-dimensional geometry categories being used with HiLiftPW-3.	Add elements shared by both geometries, with a high level description not depending on meshes.
All grids are to be constructed using these geometry files.	Add traceability references to the geometry files used to generate the mesh.
Wing semispan = 1156.75 in.	All numerical values can be described in CGNS; this is the place to archive long-term information related to your computation.
HL-CRM grids should be supplied in inches.	Many variants can be defined; the relevant context of data, such as units, should be present.

II.B. Second key: Interoperability

One of the first goals of CGNS when it was initiated in the 1990s was to store simulation input and output for archival purposes. The requirement was to be able to reread or reuse the data later, and avoid losing computation time (cost) and data. This archival capability has led to a beneficial side effect: one can exchange data with others and thus achieve an interoperability goal. The exchange was originally considered to be file-based, but people realized that one could exchange information about a configuration without any file. For example, just saying: 'I have a `WING_UPPER_SURFACE` family in my configuration, if you find all the boundary limits with this label you would find the associated `Pressure` in the `BCDatasets`'. This is abstract interoperability. No file format is involved here.

In the CGNS data model, the support for interoperability at the application level is the *family*. A *family* is a logical set of objects in your configuration. The set is an arbitrary gathering of elements such as mesh blocks, boundary limits or any application objects. CAD tools often do provide their users with a hierarchical structure of the CAD entities; these are often translated as *families*. The user can define his own families, and CGNS allows as many families as desired. This helps with the support of high level application concepts.

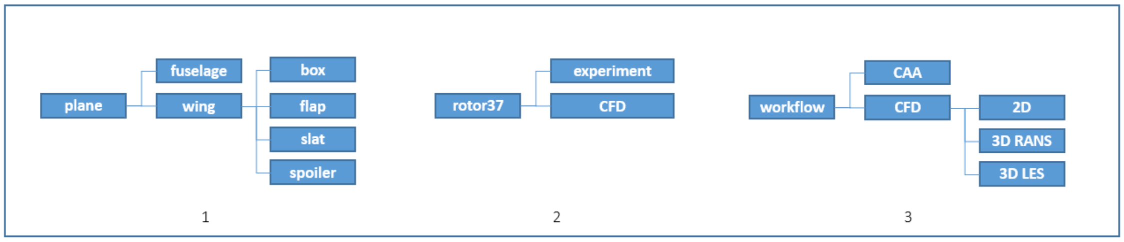


Figure 3. Three examples of hierarchical families: (1) CAD families, (2) dataset families and (3) workflow families

Families can be arranged together in a hierarchical way. For example, like a CAD hierarchy, one can define a *plane* family as a parent family for the *wing* and a *fuselage* families, as shown in Fig. 3. Two other examples are also given in the figure. Families can also be referenced through different CGNS bases. This allows the reference to common configurations having different dimensions (1D, 2D or 3D) and insures consistency between different methodologies or processes in a simulation workflow. A single object of a CGNS model, such as a zone (a mesh block) can be associated by a tag to several families. There is no limitation to the number of families that can be used; in fact, the use of families for all high level information you want to pass all along your CFD workflow is encouraged.

II.B.1. Recipe: CAD families in every CGNS dataset

In the example of the HL-CRM, the use of CAD families would be extremely useful. Unstructured meshes often produce a single zone, at least for the volume. In the HL-CRM, we have a single volume and a set of surface meshes that exactly match the boundary limits. One could then easily add one family per boundary condition and associate them to another family. The description would become less dependent on the mesh generator, and all CGNS generated files could use the **same** basis for the boundary conditions (BCs).

The families would even be better with a high-level tag for the user, such as *wing* or *slat*. Table 3 gives some examples of a possible HL-CRM CGNS tree path, the related BC and the proposed family. In this case, all BC types would become *FamilySpecified*, and a *FamilyName* added with the name referring to the family with BC information.

Table 3. Family defined boundary limits.

Path	BC	Family
`/Base/Zone/ZoneBC/TRI_bdy 1`	BCFarfield	FARFIELD
`/Base/Zone/ZoneBC/TRI_bdy 2`	BCWallViscous	WING
`/Base/Zone/ZoneBC/TRI_bdy 7`	BCSymmetryPlane	SYMMETRY

The table also shows that mesh generator identifiers are often counters, so that the user has to maintain his own table on paper in order to find out which number corresponds to which part. Actually, the mesh generator already has this information, and it is not a big effort to add the information in the generated CGNS file. This generic information, understandable by the users, is a large part of the interoperability effort at the end-user level.

II.C. Third key: Functional coverage

The *CGNS/SIDS* covers a large part of the CFD workflow needs, including the usual input and output CFD concepts, constants, fields, and all the physical and numerical parameters. One could always find a part of his own data not currently covered by *CGNS/SIDS*, but likely more than 90% of the volume of data that a typical CFD workflow manipulates is already defined by *CGNS/SIDS* and ready to be used. CGNS can describe the data you handle in your application, and there is no need to add another data model or data format, such as an EXtensible Markup Language (XML) dialect, to store some sparse information.

The *CGNS/SIDS* has dedicated structures to store:

- Several bases, i.e., several configurations that do not necessarily have the same physical and topological dimensions (a surface is a 2D topological entity described in 3D physical coordinates)
- Base scoped families, with hierarchy of families to define CAD information or any other user information
- Physical parameters, numerical parameters, equations solved and other simulation contexts
- Multiple structured or unstructured mesh types and boundary conditions
- Solution or initialization fields, discrete data, surface data, node, cell, or arbitrary element localized data with units
- Motions of meshes, symmetries, periodicities
- Any other user defined data

It is important to understand that even the `UserDefinedData` structure, which allows the user to create his own subtree of data, tagged with families and having most of the available node types as contents, is *CGNS/SIDS* compliant. This means that user data is defined in a compliant frame that allows any other application to read/write it as far as the application knows the meaning of the data. The description is fully CGNS, and the user saves a lot of time and coding effort to use this mechanism instead of defining his own data model or storage system elsewhere.

The CGNS data structure is parallel aware. As an example, the zone to zone connectivities are duplicated so that you can read one zone per processor without requiring the parsing of the opposite (donor) zone. A breadth-first parse of the CGNS tree makes it possible to first discover the overall structure of the simulation, the main physical parameters, the equations to be solved, the groups of boundary conditions, and the types and dimensions of meshes. At a depth level of 3, you already have enough information for a minimal split of the data on a High Performance Computing (HPC) host.

The *CGNS/SIDS* functional coverage is enough for most of today's CFD applications. The question is *do we reinvent something new to handle new CFD concepts or other simulation fields related to standard CFD, like Fluid-Structure Interaction (FSI) or Computational Aeroacoustics (CAA)?* The answer we give is clearly **no**. We do not see any limitation in the data model or in the existing simulations that would make new extensions impossible. But, we do know that rebuilding a new standard from scratch, with all the aspects we have detailed in this paper, is a huge amount of work.

II.C.1. Recipe: A stand-alone simulation context

Comparing the HL-CRM and the CRM CGNS trees (as shown in earlier), we see that the HL-CRM only includes the meshes and the BCs while the CRM has a complete computation context: all the data used to perform the simulation is provided in the tree. The simulation becomes completely reproducible, as all the input data is stored in this single data container. Even some Python source files are embedded, which allows the user to archive a part of the process to produce data as well as simply store the data itself.

We provide two examples of extra data used to make sure the whole simulation context is in the tree. First, the solver parameters, from ONERA's elsA solver,[5] are stored in the tree as separate attributes instead of a file of parameters (the `.Solver#elsA` node of `UserDefinedData_t` type). This helps the CGNS tools to display and modify them in an easier way. It is also possible to use diff/merge applications to find the difference between two computations having almost the same configuration except for one or two parameters. Second, the `FlowSolution#EndOfRun` node is a flow solution node describing the expected output of the solver. In that case, the elsA solver knows it has to fill in these fields, with respect to the solver's `GridLocation` requirements. This flow solution is a pattern. It is not a foreseen application of the CGNS standard, but the way it is described, it is very easy to understand, and it makes sense. The elsA solver used the CGNS standard as a standard means to represent and store computation information.

II.D. Fourth key: High performance implementation

Most CFD computations today require an HPC system. These systems are quite sensitive to compiler options, library tuning, and system resource usage. It can take a full time expert to make a simulation run efficiently on such a computer. Years ago, the *CGNS/SC* decided to change the low layer interface to the file system from *ADF* to *HDF5*.[6] This middleware is now widely spread on HPC systems and is the *de facto* standard for large simulation data storage. The HDF5 data structure is a tree. Because the *CGNS/FMM* explains how a simple CGNS node can be mapped in order to obtain a complete tree, it was easy to release the *CGNS/HDF5* mapping. The HDF5 group, in charge of the HDF5 life cycle, is a member of the *CGNS/SC* and insures a tight maintenance of CGNS tools on top of their middleware.

The HDF5 middleware has many advantages. When dealing with very large amounts of data, you do not want to duplicate memory zones. All the software memory mapping should be dedicated to the actual use in the application. But when you have a memory mapping that differs from one application to another, memory duplication may be required unless you have a smart translation mechanism to decode/recode from the original memory layout to the destination memory layout.

Fortunately, with HDF5 such a smart translation mechanism is available for memory-to-memory translation. The primary need in a translation is to get a description of the layout you have on both sides, so that you know how to decode the origin and recode the destination. It is better to have a more or less formal way to describe this layout, and HDF5 has the beginning of such a description with the *hyperslab*.

A translation between different memory mappings is often required in numerical simulations. The most common cases we can find are:

- Translating interlaced data to noninterlaced. Some algorithms require a $x_1 ... x_n, y_1 ... y_n, z_1 ... z_n$ layout (so-called noninterlaced) and some other want interlaced with $x_1, y_1, z_1 ... x_n, y_n, z_n$ layout. The translation of such data layout for a particular algorithm can lead to very large performance gains or losses.
- Taking into account the addition or removal of rind data. In these cases the rind information implies a memory shift of the data without rind compared to the data with rind.
- Reading/writing only a part of the data (e.g., a sub-block or a plane). Numerical simulations may deal with several dimensions. For example, you can compute a 3D volume solution to get the 2D results desired on a surface.
- Organizing data as chunks with a size multiple of a system constant. In the case of a very low-level optimization, at the processor level, you may have to arrange data in a way that fits into the processor vector size to increase its performance.

Regardless of the specific case, different memory mappings in the application software used in a CFD workflow require translation. You generally cannot succeed in using the same memory mapping, because it is often tightly bound to the algorithms. Therefore, we propose that it simply be described. Once described, using the HDF5 hyperslab notation, it is easy to select the application with the correct memory mapping or to add an *on-the-fly* memory translator.

II.D.1. Recipe: A smart memory mapping

The DPW-5 CGNS files include a structured multiblock mesh. The CFD solver requires some user-defined *rind*, which has to be set at run-time. The actual reads of the coordinates or field arrays have to fill in an array with a particular memory layout with a shift of cells due to this rind. The reader has to set the *offset* as the first index where the source data starts, the *count* is the number of elements in a *"row"* in 2D or *"plane"* in 3D between two sets of rind cells, the *stride* is the number of rind cells between two rows, and the *block* is the number of rows. See Fig. 4.

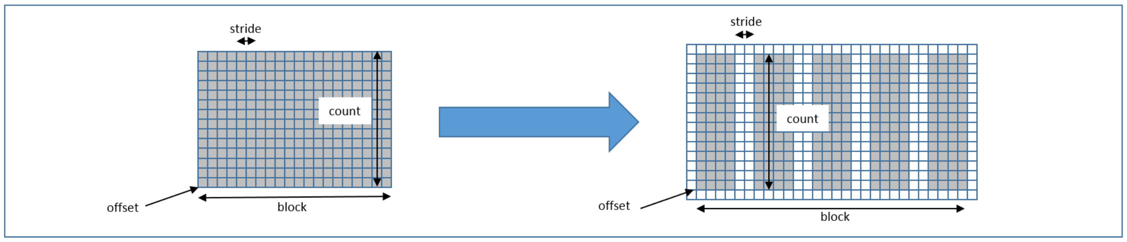


Figure 4. The memory layout can be changed during a disk/disk, disk/memory, memory/memory transfer, using the HDF5 hyperslab mechanism.

II.E. Fifth key: Versatile implementation

Now we have very efficient computation in terms of data volume and speed. What about versatility? What about adding a new preprocessing or post-processing step in an existing simulation? The *CGNS/Python* mapping is dedicated to such a purpose: it provides a Python view of the CGNS tree. The *Python* programming language is now the reference for modules integration. It has demonstrated its power in many different scientific areas, in particular with the *numpy* module, which provides a smart mapping of data arrays. Arrays can come from *Fortran*, *C/C++* or *Python* itself and be shared without any copy. The *CGNS/Python* mapping uses this capability, so that when you are working on your *CGNS/Python* tree you are actually reading and writing the same memory zone as your C or Fortran code. Of course, this requires some memory layout controls, in particular if you are mixing memory coming from C array ordering and Fortran array ordering.

Finally, the *CGNS/Python* mapping has been defined to avoid any required library coming from CGNS. One can implement a *CGNS/Python* interface independently: only plain Python objects are used, and no proprietary package. As the *CGNS/Python* tree contains only plain Python objects, such as lists, strings and numpy arrays, it has a default serialization. This is the required mechanism when you want to send and receive data across the network. All the objects parsed into memory are gathered into a contiguous chunk of memory in order to be sent, and then to be received on the other side. Sending a complete CGNS tree through the network needs only a few lines.

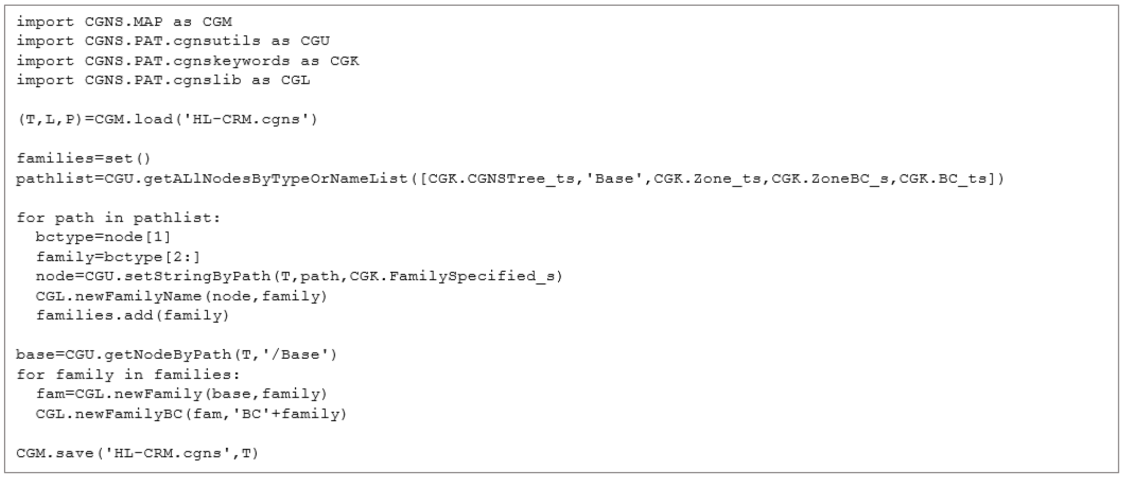
II.E.1. Recipe: A simple post-processing of the tree

Once the HL-CRM is generated by the mesh tool, it is possible to use some Python scripts to add the extra information needed. For example, say we want to change all the boundary conditions types by family-defined BCs. The example in Fig. 5 shows such a post-processing script.

II.F. Sixth key: Extensibility

The *CGNS/CPEX* allows anyone to extend the standard to suit their needs. The standard is alive, it is not something set in stone, and it **should** evolve to meet all user operational requirements. The standard update is a consensual process. Because users already have many existing CGNS applications and data files, the *CGNS/SIDS* data model, the *CGNS/HDF5* and the *CGNS/Python* implementations cannot be changed without some degree of caution.

The points that must be respected are:



```
import CGNS.MAP as CGM
import CGNS.PAT.cgnsutils as CGU
import CGNS.PAT.cgnskeywords as CGK
import CGNS.PAT.cgnslib as CGL

(T,L,P)=CGM.load('HL-CRM.cgns')

families=set()
pathlist=CGU.getALlNodesByTypeOrNameList([CGK.CGNSTree_ts,'Base',CGK.Zone_ts,CGK.ZoneBC_s,CGK.BC_ts])

for path in pathlist:
  bctype=node[1]
  family=bctype[2:]
  node=CGU.setStringByPath(T,path,CGK.FamilySpecified_s)
  CGL.newFamilyName(node,family)
  families.add(family)

base=CGU.getNodeByPath(T,'/Base')
for family in families:
  fam=CGL.newFamily(base,family)
  CGL.newFamilyBC(fam,'BC'+family)

CGM.save('HL-CRM.cgns',T)
```

Figure 5. A compact example of Python code, which reads the HL-CRM CGNS file, changes the BCs to family specified BCs and creates the corresponding families.

- The extension should not break existing files or applications
- It should not redefine an already existing concept/entity of the standard, unless there is a good reason for doing so
- The *CGNS/SC* must be given all elements required for a CPEX submission

A "champion" is required to to push any desired extension. CGNS is maintained by volunteers, and nobody would carry your extension. A good approach is to gather a group of people interested in your extension, then develop and push through the CPEX all together. Depending on how deep your extension process is, you generally have to provide at least these elements:

1. The rationale explaining the scope and the goal of the extension/modification
2. The modifications or additions required in the official documentation, such as *CGNS/SIDS*
3. If the extension implies a new API, it should be provided by the proposer (providing the code and tests is even better)

The process is completed by many email or teleconference exchanges and discussion to reach the final vote. In order to avoid an excessively lengthy process once the CPEX elements are available, the *CGNS/SC* often sets a deadline. Some of the most recent CGNS/CPEX elements either recently approved, in process or foreseen are:

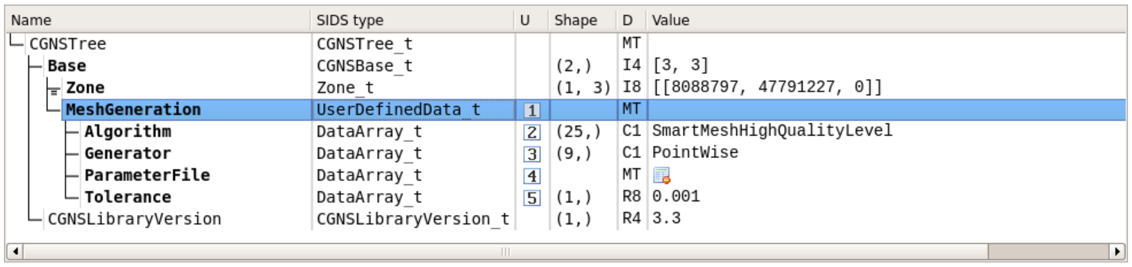
- New NGON elements array layout for HPC performance (accepted)
- Extension for arbitrary high order meshes (in process)
- Arbitrary reference frame (planned)
- Enhanced chemical data (planned)

II.F.1. Recipe: A mesh generator characterization

Here we reimagine the real scenario in which the *High Lift Workshop* proposes a CAD configuration to be meshed by participants. There is a clear relationship between the CAD – the same for all participants – and the meshes, which are all different. It would be useful to track the way each mesh is generated, so that one could archive both the CAD and the characterization of the meshing process, instead of merely each mesh itself.

To proceed with such an extension, each participant could add his own `UserDefined` structure, as an addition to the *CGNS/SIDS*, describing his own understanding of how to best characterize the mesh generation. An extension has no predefined form. One could add the data to be defined, as shown show in Fig. 6. This example shows a new new node (marked `1`) as a root of a subtree containing enumerate attribute (marked `2`), a free string (marked `3`), a full binary or text file[c] (marked `4`) and a single real (marked `5`). Then, all participant proposals could be gathered, factorized and formalized to initiate a *CGNS/CPEX*.

[c]In this screenshot the data type is `MT`. This means the actual value is *empty* but this is a side effect of the CGNS viewer



Name	SIDS type	U	Shape	D	Value
CGNSTree	CGNSTree_t			MT	
Base	CGNSBase_t		(2,)	I4	[3, 3]
Zone	Zone_t		(1, 3)	I8	[[8088797, 47791227, 0]]
MeshGeneration	UserDefinedData_t	1		MT	
Algorithm	DataArray_t	2	(25,)	C1	SmartMeshHighQualityLevel
Generator	DataArray_t	3	(9,)	C1	PointWise
ParameterFile	DataArray_t	4		MT	
Tolerance	DataArray_t	5	(1,)	R8	0.001
CGNSLibraryVersion	CGNSLibraryVersion_t		(1,)	R4	3.3

Figure 6. A prototype CGNS extension in the HL-CRM tree.

II.G. Seventh key: Open systems

The definition of an *Open System* is *'a system in which entities interoperate only through public interfaces.'* The entry points with which one can interoperate in the CFD workflow are each step of any given simulation. A step is a code call, for example a mesh generator, a CFD solver run, a post-processing task, etc. Between each step, the data are exchanged at two levels. See Fig. 7. The low level is the implementation: it is a file or a memory address in most cases. The high level is the concept exchanged. The high level request *'give me the pressure on the wing upper surface'* has one or more corresponding low level requests such as *'fill in this array of float arrays.'* Between these two level, we do not care how the process is performed.

From the CFD workflow end-user point of view, one needs to address high level concepts, and those are defined using the *CGNS/SIDS*. From the application point of view, actual data needs to be exchanged or manipulated in memory or on the disk. These points are covered by *CGNS/HDF5* and *CGNS/Python*.

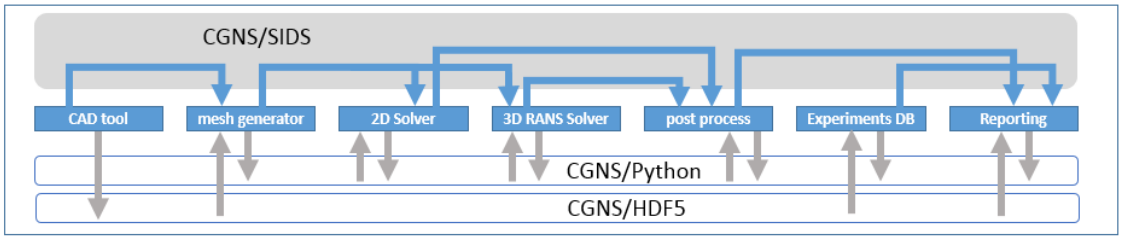


Figure 7. The Open System interfaces: the high level is CGNS/SIDS, low levels are CGNS/HDF5 and CGNS/Python. All simulation workflow codes are interacting only through these interfaces. The high level is concepts and data model interoperability. The low level is disk and memory.

CGNS is a true *Open System*, using public interfaces, both at a high and a low level. The high level data model is a consensual and public description of CFD data. The two low level implementations, HDF5 and Python, are using well known public releases of nonproprietary software. The dependencies of these implementations are minimal. For example, in the case of HDF5, the whole open source middleware can be installed on any HPC system if it is not already present. With the Python module, there is no dependency other than the Python interpreter itself and the numpy package.

Anyone can write his own CGNS compliant application, driver, translator, or whatever tool is desired, provided that the high level interface is compliant with *CGNS/SIDS* and the low level to one of *CGNS/HDF5* or *CGNS/Python*. If an application is able to read/write a file using the public HDF5 library, and if the result of these operations follow the requirements defined in the *CGNS/HDF5* mapping document, then the HDF5 driver can read the nodes required, handle proprietary memory layout, check the directory where to find a link destination, or any other application-specific need.

II.G.1. Full black-box workflow

Now we have a common high level interface and a common disk or memory implementation. We can connect applications together without the actual applications on your host. We can perform an integration and even some parts of tests, without the actual codes. We can apply this approach to the high-lift mesh generation

`cg_look` which has a threshold on data arrays to avoid loading very large arrays. The small icon with the red sign warns the user of such a side-effect.

and its associated computations. Each step is a black-box, which handles high level concepts and low level implementation. A typical CFD workflow can be:

1. mesh generation
2. computation parameters added
3. check configuration
4. HPC distribution
5. CFD solver
6. run-time convergence monitoring
7. post-processor

Each step has its own input and output subset of *CGNS/SIDS*. The CGNS tree goes from one step to another, but even if the tree is transient between two or more steps, it should be compliant, so that at any step a black-box can be exchanged with another having the same input and output subset of *CGNS/SIDS*.

The *Open System* approach helps you to dissociate what you want to do from how you do it. Although it is quite difficult to achieve such an abstraction goal in CFD, it is a fundamental guideline that we have for CGNS.

III. Frequent comments and misconceptions about CGNS

We have just detailed seven general key points that form the basis of the CGNS standard and justify CGNS usage. Now we return to the specific comments and misconceptions about CGNS mentioned in the introduction, along with some others, and we answer them in an informal order.

III.A. CGNS is a library

False. CGNS is a set of documents and one library, the *CGNS/MLL*, which is a sample implementation of *CGNS/HDF5*. The *CGNS/HDF5* mapping is a public specification, but anyone can write their own HDF5 reader/writer for their own purpose. For example, the *CHLone*[d] library is a another implementation of the HDF5 mapping. There are many other tools using the HDF5 API that can access *CGNS/HDF5* files without using the MLL. Writing such a reader is not difficult. For example, you can use any of the HDF5 Python packages available on the web, and in several lines you have your own dedicated parser.

III.B. CGNS is not fitted to my needs in CFD

False. Do you want to create your own data model? Then you are going against open systems. Let us push the process to its end: you want to create your own data model. You decide to map it directly to your code implementation, i.e., the data structure you use in *C++* or *Fortran*. In this case, you know that you will not be able to exchange data with any code other than yours. But then you change your mind and try to write down a pivot format. During this process you would have to identify all the objects you are dealing with, describe them, and you may compare it to other existing pivot formats. Of course, you have to develop all the tools one would expect to have with your format. At the end, the process you are following is exactly the same as the one that was followed by the CGNS standard, except that your format only applies to your own applications; it has not been used for other applications.

III.C. XML is far more standardized and it would be better for CFD

False. Most people do use *XML* to describe their own data model. Some models, such as XDMF[e], do provide two separate representations for data: one for the large data and another for the so-called *metadata*. The metadata definition is not clear. It is an arbitrary classification setting whether information is relative to the computation or to the way you perform the computation. For example, a pressure field would be data while the number of processors would be a metadata. We think both are relevant in a single computation context. Probably the pressure data is not of the same scope as the number of processes, but both have to be described, stored and retrieved in some way. Then if the data model is able to address both definitions,

[d] `http://chlone.sourceforge.net/`, cited 10/02/2017
[e] `http://www.xdmf.org/index.php/XDMF_Model_and_Format`, cited 10/02/2017

which is the case for *CGNS/SIDS*, we believe that the use of a second data representation, such as XML, is useless.

In all cases, the use of XML requires a support grammar. This grammar has to be sound, documented, and some tools have to understand it, like editors, translators or any other data manipulation services (diff, merge, etc.). The process of defining a suitable grammar, accepted by a large community, is the same process already used in CGNS.

III.D. The extension process is too time-consuming

True. The current process includes reviews and comments. You can go faster by defining your own data model and by avoiding the comments. In the end, the time you gain by quickly implementing your own model will be lost when you would have to:

- write new translators from/to your own model
- have to maintain this model alone instead of delegating to a long-standing committee
- realize you did not take into account a scope of application covered by others but not by you

The extension process is time-consuming, but, as with any consensual process, once adopted it provides a stable reference that can be trusted. Another point is that once an extension is adopted, there is a greater likelihood of its use by many other people or organizations who see it not as a proprietary effort, but as a publicly available model.

III.E. There are too many ways to define the same data

True. In some cases, you can find more than one way to define the same thing in CGNS. If you understand *CGNS/SIDS* as a specification language, you can define the same information in different ways. For example, the `ReferenceState` node can be located once at the `Base` or it can be spread with as many `ReferenceState` nodes as you have `Zones`. Or the use of `Elements_t` is not always defined. For example in the HL-CRM, there is one element entry per boundary condition, while other mesh generators would produce one element entry per element type. Each tool has its own patterns, one for input and one for output. However, if you parse the tree, it is possible to understand the data you find during this parse. The standard is there to help you.

Let us take the opposite point of view. Imagine you have a nonpermissive standard that forces you to use exactly one single way to store your elements. Then, if this does not fit with your application, you would have to lose information (the parts you cannot put in the data model) or else create a separate file with this extra information. Now another application could not use this data unless you write it down with comments. If you multiply this process over many applications, you end up with as many translators and data models as applications.

Having multiple ways to define some types of data in CGNS can make translators tedious, but at least with CGNS the data model is already defined, and it is a public one. The first step to follow to address this issue in CGNS is to describe the way you are defining your tree and to add all the contextual data you can.

III.F. CGNS is not parallel-aware

False. The standard has been designed for multiblock meshes, in a way that allows a complete subtree of the complete configuration to be used as a stand-alone structure. This means first the global data is found in the parents of the subtree, and second the information shared by specific subtrees, such as a mesh to mesh connectivity, is duplicated in order to achieve this stand-alone capability.

III.G. An abstract model for HPC is not important

False. You always have an abstract model. It can be in your mind or in your code, but it exists. The process of describing it helps you and other people interacting with you to understand the way you represent data. A common remark is that HPC requires very finely tuned data structures. The performance of the system is so tightly bound to the hardware that any abstraction becomes a disadvantage. Then an abstract model may ask some out-of HPC context constraint that would break some application efficiency. You can always add your own data array. As a matter of fact, if the data structure you want is tightly bound to your run time platform and your memory layout, it would benefit from standardization.

III.H. CGNS is only for archival data

False. The standard defines a data model that acts as a specification support and an implementation on HDF5, on disk, to insure long term archiving and reuse. But, as this data model is public and described in an implementation independent way, you can use it for any other interoperability purpose as well. You can define the data you have, the data you want, the way the data changes during your simulation workflow steps, and all definitions as textual information, like a document specification. The two implementations *CGNS/HDF5* and *CGNS/Python* also act as interoperable support, but they have different scopes. The *CGNS/HDF5* is mainly disk based: you store/load your data to/from the file system. This is archival, even if your archival duration only lasts seconds or minutes. The *CGNS/Python* is for memory use, it is not reasonable to store large data as Python data. However, it is appropriate for all memory-based exchanges, such as process to process shared memory or network transfers.

III.I. CGNS files are too big

True. CFD data is big. Because of this, the reuse of common files, such as meshes, is important. If you have a 10Gb mesh, you do not want to duplicate it each time you change a `Mach` or an `AngleOfAttack` value. The *links* feature helps you to reuse data, as described in Fig. 8. In this example, a top file refers to the *linked-to* file containing the 10Gb mesh. With *links*, it is up to the user to draw a policy for sharing files, transferring them from one HPC cluster to a workstation, or any other good practice one should have for efficient teamwork and resource management.

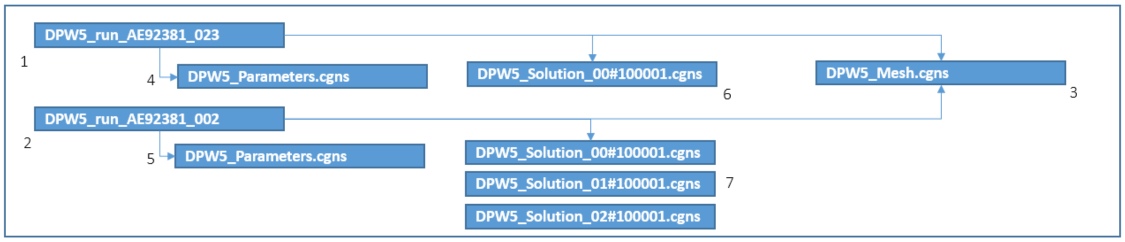

DPW5_run_AE92381_023
1
4
DPW5_Parameters.cgns
DPW5_Solution_00#100001.cgns
6
DPW5_Mesh.cgns
3
DPW5_run_AE92381_002
2
5
DPW5_Parameters.cgns
DPW5_Solution_00#100001.cgns
DPW5_Solution_01#100001.cgns
7
DPW5_Solution_02#100001.cgns

Figure 8. Links are the mechanism for sharing. In the DPW-5 computation, two simulations (1) and (2) are using the same mesh (3). They have a separate set of parameters (4) and (5) and set of solutions (6) and (7). If you open any part of this file, the links are automatically followed and the user sees a single tree.

III.J. CGNS official mapping is ADF

False. The *ADF* file format was the legacy format for CGNS files. It was a smart and compact format, well suited to the CGNS implementation requirements at the time. However, it was not HPC-aware. The file access was direct through the system without any middleware to handle the specific HPC file systems or libraries. So the *CGNS/SC* decided to move the reference implementation on disk on top of the HDF5 middleware. This HDF5 library is usually installed on HPC systems and is often tuned to fit with the host hardware. HDF5 is not shipped with the standard library, while *ADF* is. However, HDF5 is free and easy to install, and *CGNS/MLL* includes tools that make it easy to interoperate between HDF5 and *ADF*.

III.K. CGNS has few useful tools

True. The *CGNS/MLL* comes with a minimum set of tools, and these are not always enough for advanced use in a complete CFD workflow. As a standard, CGNS expects third party tools' editors to provide up-to-date versions of their products, or open source initiatives to provide users with generic all-purpose tools. The *CGNS/Python* helps users to quickly write their own purpose tools, but you still have to make the effort. The following is a list of common basic needs associated with CGNS, for which current tools already exist:

- A *CGNS/SIDS* compliant node/tree editor (at least two available: `cgnsview` and pyCGNS `cg_look`)
- A validation tool to check a tree against *CGNS/SIDS* or against a user defined set of rules (`cgnscheck`)
- A diff tool and a merge tool (a diff is available in pyCGNS `cg_look`, the merge is available as a Python function in the same package)
- A smart 1D, 2D, 3D viewer (`cgnsview` is a simple viewer; most VTK-based tools only read a subset of CGNS, they do not handle families or BC datasets)

- Translators to/from other commonly-used formats (some are shipped with *CGNS/MLL*)

HDF5 compliant tools also are available, which can read/write/browse a *CGNS/HDF5* file but in a raw manner. A tool that would be useful, but currently does not exist, would handle links, such as creation, suppression, and editing.

III.L. CGNS is redundant with HDF5

False. The HDF5 data model and library defines an anonymous node type. If you want to know what information you are parsing or if you want to tell somebody else about this information, you have to write it down on paper. This "paper" is the specialization you are using on top of the HDF5 generic model; it is your own data model.

III.M. CGNS is not smart at defining data that depends on time or statistics

True. As currently defined, time dependent data in CGNS makes a lot of references to other nodes of the same subtree. Just like families, these relationships are adding another way of parsing the data structure. However, unlike families, this method creates a suite with a large number of snapshots of a CGNS tree. The `TimeIterativeData` offers an array to store these references instead of sparse families. The current version of this table stores the references as textual paths to the nodes, such as grids, boundary limits, connectivities or other CGNS elements. A better way to store such information would be to use links instead of strings. A string needs to be analyzed by an application, while the link is managed in a transparent way. This issue could be eventually remedied or improved via the CPEX process.

III.N. CGNS is only for CFD

True today. The standard has been created for CFD. However, there is no equivalent in other domains of the simulation. Many proprietary software has their own data model and implementation, but there is no example of a public data model for structural mechanics, acoustics, thermal analysis, combustion and many other physical domains related to a multiphysics simulation. We believe CGNS is the best candidate for an extension to these domains, not because of the data model it has today, but because of the existing background and material one could use to define these extensions.

IV. Conclusion

The CGNS standard is often associated solely with its implementation library. This reduces its scope to a simple library used to read/write files on disks. Actually, CGNS is more a data specification; it defines a common data model which is a basis for data description in documents and in software application interfaces.

The question is now: where to start with CGNS? The answer clearly is: the *CGNS/SIDS*, the data model. You can build small software prototypes using *CGNS/Python*, and once you know which data you want to define, manipulate, or store with which volume and which parallel distribution, you can use *CGNS/HDF5*.

We believe you have now enough information to apply CGNS to your own needs, and eventually participate in its improvement by joining the *CGNS/SC*.

References

[1]Poirier D., Allmaras S. R., McCarthy D. R., Smith M. F., and Enomoto F. Y., "The CGNS System," AIAA Paper 98-3007, June 1998.

[2]Rumsey C. L., Wedan B., Hauser T., and Poinot M., "Recent Updates to the CFD General Notation System (CGNS)," AIAA Paper 2012-1264, January 2012.

[3]Rumsey C. L. , Slotnick J. P., and Sclafani, A. J., "Overview and Summary of the Third AIAA High Lift Prediction Workshop," AIAA Paper to appear, January 2018.

[4]Hue D., "Fifth Drag Prediction Workshop: Computational Fluid Dynamics Studies Carried Out at ONERA," *Journal of Aircraft*, Vol. 51, No. 4, 2014, pp. 1295–1310, doi:10.2514/1.C032054.

[5]Cambier L., Heib S., and Plot S., "The Onera elsA CFD Software : Input from Research and Feedback from Industry," *Mechanics & Industry*, Vol. 14, No. 3, 2013, pp. 159–174, doi:10.1051/meca/2013056.

[6]Poinot M., "Five Good Reasons to Use the Hierarchical Data Format," *Computing in Science & Engineering*, Vol. 12, No. 5, 2010, pp. 84–90.