MANGROV : Modular and Adaptable Notation for Generalized Representation Of Vehicles

Simon Verley, Raphaël Perret

MANGROV: MODULAR AND ADAPTABLE NOTATION FOR GENERALIZED REPRESENTATION OF VEHICLES

Simon Verley
ONERA
Salon de Provence, France

Raphaël Perret
ONERA
Salon de Provence, France

ABSTRACT

In this paper, we propose a new standard to organise and use data representing any kind of vehicle and their components. It defines a way to store any kind of data for this representation, and aims to be unambiguous, generalist, modular, extensible and live as well as static. The structure defines relationship between components, with a possible multiparentality definition such as mechanical and energetic relationship. It also offers a way to describe missions and related results. All of these is demonstrated by rules definition and example of use. This standard is open to the community, and amendements and propositions are more than welcome.

EMERGENCE OF THE NEED FOR A NEW DATA STRUCTURE

In the context of a collaboration building an eVTOL configuration, with numerous different fields and tools having various nomenclatures, the need for a common view on data and its interpretation arose.

We first turned to existing data representations, mainly from rotary wing flight mechanic's code, but found several downsides in their systems for our intended uses.

HOST, presented in (Ref. 1), has a rigid data structure that gives little context to the data. FlightLab (Ref. 2) uses an interesting tree structure, but the data structure is rigid and non evolving. CPACS (Ref. 3) propose an interesting and thorough structure, but this structure stops existing at run time. OpenVSP presented in (Ref. 4) and (Ref. 5) has a very good structure, with living data accessible through an API, for multi domain applications (although not all domains are represented), but limits the fidelity of its data to its models, and doesn't allow to store new or unexpected data. None of them match the need we had to embed all expert fields around the table (mechanics, aerodynamics, acoustics, systems, energetics...) with the capability to keep all these data together in a living object for code to code coupling.

Most of the data structure that can be found provides context for the data, with more or less rigidity in its structure. However, none provide dynamic living data structure once the data is read from the file. At most, living data can be access through an API.

From the incompatibility of our needs with the preexisting tools emerged a list of requirements. Those requirements can be summarised as follows by order of importance:

1. The structure should provide context: every data should be interpreted easily depending on the context provided by the structure. There should be no doubt if we are talking about the rotor's diameter or the left beam's diameter, about the required power of a single rotor or of the full aircraft, about the blade speed in its relative frame or the speed of the aircraft in Earth frame.
2. The structure should be general: any field of application should be able to store its required data along other field, and any kind of aircraft, with any kind of elements, should be representable.
3. The structure should be extensible: unforeseen usage of the structure should be allowed to exist without deteriorating it, and without the need to redefine part of the structure.
4. The structure should be dynamic: the structure should not cease to exist at run time, and should follow the evolution of the data it contains. This allows a common access of the data by different codes at the same time, as well as simplifying the save and load steps.
5. The structure should be modular: we should be able to move an element from an aircraft to another.

There was however an interesting data structure that provided some of our requirements: CGNS (Ref. 6) ! (Ref. 7). It was in fact the only one, to the best of our knowledge, to provide

both the context, the dynamic and a part of the general (via the node definition) requirements to the extent intended to our use.

In the following sections, we will strive to respond to the above list of requirements in order to build a norm for this new data structure.

DEVELOPING A NORM FROM THE EXISTING CGNS NORM

The above requirements for a suitable data structure oriented our searches towards the existing CFD General Notation Standard (CGNS). We will build on this existing standard which fulfils some of our requirements, and expend its capacities to our full list of requirements.

This section will present the CGNS shortly and proceed to the development of its application to flight mechanics. A good introduction of CGNS has been made by Poinot and Rumsey (Ref. 6).

What is CGNS

CGNS is an open standard for CFD data (CFD General Notation System). It aims to be a database which stores meshes and associated fields as well as boundary conditions, farfield conditions and all others physical data needed to fully define the simulation. This database is structured by the CGNS Standard Interface Data Structures (SIDS). (Ref. 8) indicates: "The major design goal of the SIDS is a comprehensive and unambiguous description of the 'intellectual content' of information that must be passed from code to code in a multizone Navier-Stokes analysis system." It is an extendable database oriented for CFD analysis and provides a standard format for data based on a hierarchy, defining a context to every data, which is structured in a tree, depending on its path. Thanks to this context, any code that interprets CGNS can be set up using the same database to run the same simulation. The SIDS also defines the way the results have to be stored in the database.

All the required data are stored into a multi-level tree. Each level corresponds to a function which can be assimilated to a filter. This filter accumulation defines a comprehensive context, as well as a path to access and identify the corresponding data.

Each data at every level is defined by a node containing its name, its value, its label and references to its children. The name is a unique ID of the node in the context of its parent (all children should have different name). The value is an array that stores the data. The label is one of the labels defined by the SIDS storing the kind of node it is.

On top of the commonly defined data, CGNS also allows the use of UserDefinedData, data that aren't supposed to be parsed by every code since they are not defined by SIDS.

CGNS trees are convenient and effective. Their hierarchical definition and data naming has led it to be a widely adopted standard in CFD.

Available requirements in CGNS

As one can understand from the above section, CGNS presents several of our requirements. First of all, its tree structure provides a clear and unambiguous context to the data. It is extendable, and we will use this extensibility in order to adapt it to our needs. Finally, it also provides a dynamic structure at run time, that will provide data to all codes and store results.

In order to build on CGNS, we use the same definition of a node as defined in (Ref. 9). We have reproduced this definition below for the reader's comfort. A node is composed of different parts:

Node Identifier. The Node ID is a floating point number assigned by the system when the database is opened or created. Applications may record the ID and use it to return directly to the corresponding node when required. The Node ID is valid only while the database is open; subsequent openings of the same database may be expected to yield different IDs.

Name. The Name field holds a character string chosen by the user or specified by the SIDS to identify the particular instance of the data being recorded. It is unique at the node level (homonym brothers are forbidden).

Label. The Label, also a character string, is specified by the CGNS mapping conventions and identifies the kind of data being recorded. For example, a node with label Zone_t may record (at and below it) information on the zone with Name "UnderWing." No node may have more than one child with the same name, but the CGNS mapping conventions commonly specify many children with the same label. For some nodes, the mapping conventions specify that the name field has significance for the meaning of the data (e.g., EnthalpyStagnation). Although the user may specify another name, these "paper" conventions serve the transfer of data between users and between applications. These names and their meanings are established by the SIDS.

Data Type, Dimension, Dimension Values, Data. Nodes may or may not contain data. For those that do, CGNS specifies a single array whose type (integer, etc.), dimension, and size are recorded in the Data Type, Dimension, and Dimension Value fields, respectively. The mapping conventions specify some nodes that serve to establish the tree structure and point to further data below but contain no data themselves. For these nodes, the Data Type is MT, and the other fields are empty. A link to another node within the current or an external CGNS database is indicated by a Data Type of LK

Child Table. The Child Table contains a list of the node's children. It is maintained by the database manager as children are created and deleted.

In the remainder of this article, a node will be represented with its children as in the figure 1. The part of the node containing the data will be referred to as the Value of the node.

Figure 1: Example of a node with 2 children

Missing requirements

Although CGNS provides some of our requirements, others are not readily available. CGNS has been developed for a specific use which adds some constraints.

The first requirements to be hindered by those constraints is the general requirement (req. #2). CFD has little interest in many of the aspects required to model properly the flight mechanics of an aircraft. Therefore, a lot of fields are not considered or are ill defined.

A second aspect, not as developed as wanted, is the modular requirements (req. #5). Indeed, CGNS provides some modularity, but there is no reason for us to be as restrictive as it is. An interesting part of the modularity to represent an aircraft, is the capacity to place part of the aircraft inside other parts. This is prevented by the "one level, one function" rule of CGNS – "mesh grid", aka Zone_t, will always be at root level + 2.

The last requirement is linked to the tools that CGNS provides. They do not encompass unforeseen uses of the tree. Although the norm may evolve, there are no general set of rules to extend it on one's own (req. #3).

The next section will seek to improve those points, while keeping the requirements already provided.

THE STANDARD DEFINITION

This section presents a first version of the norm, providing specialised nodes for the purpose of representing aircraft and storing simulation results.

The main part of the work is to define new Label such as Vehicle_t, Component_t, Field_t and established a set of organisational rules to store data, that will be called grammar. Furthermore, we propose a common vocabulary for data naming that will help compatibility. This vocabulary is mainly optional and data can be named freely for most nodes.

An example of vehicle

Lets start by defining a dummy vehicle using an approach fitting our requirements.

The first step is to define a Node that should define the vehicle. This node needs a Name to be accessible easily. This name will also be used to create a context for all nodes held by it (req. #1). The node also needs to define what it is made for, a Vehicle_t in this case. This information is defined as its Label. The kind of vehicle defined by it is contained by the Value of the node.
At this point, we have defined a context (a Vehicle_t as XWings) named T-65.

Figure 2: A vehicle node

This vehicle will be defined by different kind of data which can have different range of meaning, called Scope and labelled Scope_t. A Scope_t node specifies the context for these data. The Local, Global or Inherited values of a Scope_t are defined as follow :

Local Define a context for data that are valid only in this Component_t or Vehicle_t.

Inherited Define a context for data that come from others Component_t linked to the local one.

Global Define a context for data that are valid in this Component_t or Vehicle_t and all of its linked Component_t.

For example, to run a time simulation, we will define the vehicle's maximal take off weight in the Local scope because it does not depend on any possible Component_t. In an other hand, the total mass of the Vehicle_t will be set in the Global scope as it corresponds to the sum of all masses present in the Local scopes of each Component_t. However, in the case of a design simulation, where the maximal take off weight is an unknown, a node containing its computed value as a consequence of the design process will be placed in the Global Scope_t.

The mass is a well-defined characteristic of a Vehicle_t, with almost no ambiguity. But a length, a surface, a diameter could be more ambiguous and lead to a confusion on the meaning of the data. As an example, the surface of the wing can be a wet surface, a projected surface or simply a surface value used for an aerodynamics solver but with a different value. This specialisation of data meaning can be achieved by modifying the Name of each data but leads to a large number of names to describe a Vehicle_t. Moreover, if each team working on the same Vehicle_t uses its own naming convention without taking care of others, data naming will sooner or later overlap from one team to another and so, the Vehicle_t will have a false representation for one or more teams. Thus, it has been decided to use another level to create specialised context with a new Label named Field_t. The Field_t node specifies the field to which the data belongs through its value, which could be among Mechanics, Aerodynamics, Energetics... This way, a Surface node can be set in as much Field_t as needed and be valid only in this context.

We now have the possibility to hold values for data in a DataArray_t with a context that defines its meaning. Those Label are complementary to the CGNS labels', to the exception of the DataArray_t label, which corresponds to the last level of the tree, and is defined to hold the data while providing minimal information to it through its name.

Path

All data stored in the proposed tree can be designated by a unique identifier, called a path, and built like a GNU-POSIX

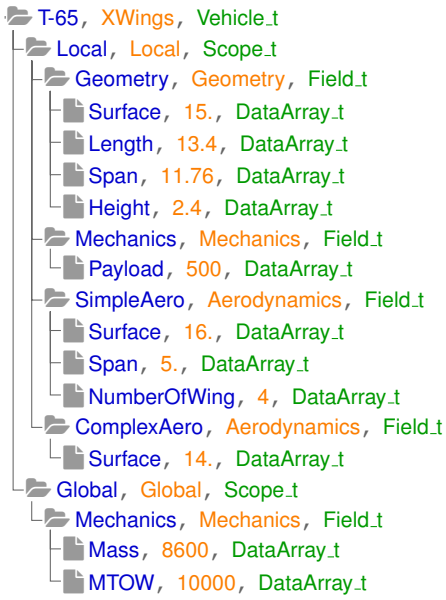


Figure 3: A vehicle tree: a more complete example

path using the node's name for each level. This prohibits two nodes on the same level to have the same name. For example, the path for the local mass node of the T-65 is:

/T-65/Global/Mechanics/Mass

This path, by referring to all parent nodes of the data, allows to create context for data (here, the mass). At each level, the local context is defined by the couple of (Label, Value). The full context of the data is the combination of all local context. The previous path thus designate the following combination of contexts:

/T-65 ⇒ (Vehicle_t, XWings)
/Global ⇒ (Scope_t, Global)
/Mechanics ⇒ (Field_t, Mechanics)

DataArray_t nodes stores **Data** as the couple (Name, Value). This means that the **Data** needs to be interpreted its context, as the one explicited just before. In the previous example, the **Data** is the Mass with a value of 8600 kg in the context of [(Vehicle_t, XWings),(Scope_t, Global),(Field_t, Mechanics)]

Component description

It is possible to use a simplified and complete description of the vehicle only in a Vehicle_t node, as shown in Fig. 3. This implies that software used can handle it.
Nonetheless, it can be convenient to describe this vehicle using Component_t nodes, such as Wing, Engine, Fuselage, etc., to use more specific descriptions and solvers. To do that, we will store the Component_t nodes as Vehicle_t node's children. This is illustrated in Fig. 4, where the parts describing the aircraft are stored as children of the vehicle, on the same level as Scope_t nodes of the Vehicle_t node.

In the case of the X-Wing, we could define at least 3 types of Component_t: Fuselage, XWing, Engine. These Component_t

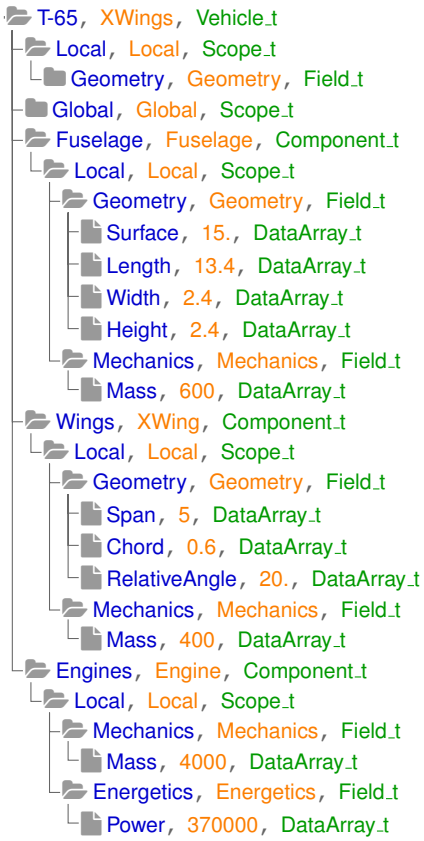


Figure 4: A vehicle tree: a more complete example

nodes create a new context in order to store specific data for each one of them.

Components position and frames definition

The placement of the elements are done through the creation of frames attached to the component to be placed by using Frame_t nodes stored at the root of the component. The value of the frame node is either the name of the parent component, or may be Absolute or Relative.

By convention the reference frame is named ReferenceFrame. The position of the frame with respect to its parent is made explicit through the OriginLocation node for the position, and through the RotationaAngle for the rotations. Rotations may also be specified through RotationMatrix or Quaternion nodes. If multiple nodes with the same purpose (RotationAngle, RotationMatrix and Quaternion) are present in the same context, it is the responsibility of the one modifying one of them to keep them coherent all together.

Fig. 5 shows the T-65 configuration with a ReferenceFrame for all Component_t. Placement of these Frame_t are done relatively one to another as indicated by their node's value.

All placements of elements inside the components are made with respect to the reference frame by default. For example, the GravityCenterLocation is at the reference frame origin if its value is (0.0,0.0,0.0). It is however possible to create other frames inside a component, but they should always be defined with respect to the reference frame. Those frames can be

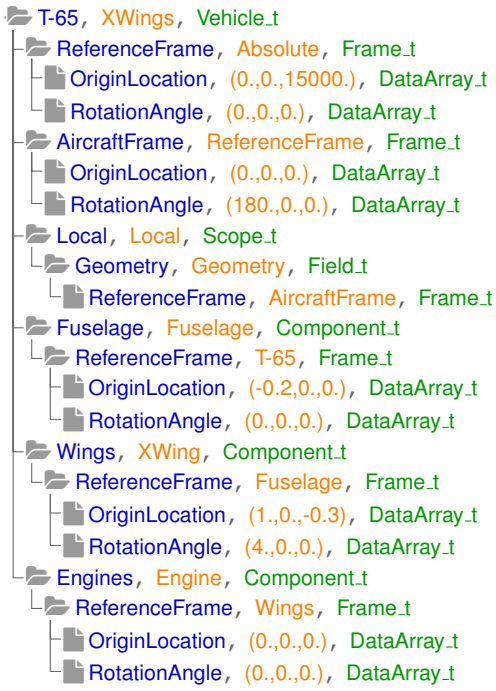


Figure 5: Example of various components placements

placed in a lower level in the tree hierarchy, in which case they give a new frame context to the data on their level and further down the tree. To put it simply, the frame in which a data is defined is the closest ReferenceFrame to the data in term of tree depth.

Modularity makes things easier

As we can see in the previous section, the placement of a Component_t can be done by referring to another Component_t which makes positioning easier than positioning all the Component_t in the Vehicle_t frame.

Nevertheless, if we want to fit with our modular requirement (req. #5), it is useful not to refer to another Component_t node by its name but with a logical approach. To allow such mechanism, we will build the vehicle like a LEGO® by adding Component_t nodes onto other Component_t nodes. Thus, Absolute refers to a placement with respect to the reference frame of the Vehicle_t, while Relative refers to the parent (as ascendant) of the component in the tree. At the Vehicle_t children level, Absolute and Relative have the same meaning.

In order to keep an equivalence between the modular version of Figure 6 and the flat version of Figure 5, we add the constrain that two Component_t nodes may not share the same name, even if the Component_t nodes are not on the same level. In other words, Component_t node name have to be unique in a Vehicle_t node.

This feature highlights that a Vehicle_t node is built and behaves exactly as a Component_t node. However, a Vehicle_t node allows to define an absolute ReferenceFrame for all Component_t nodes in the Vehicle_t context.

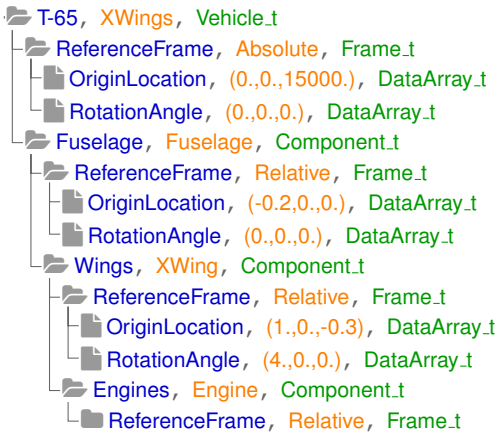


Figure 6: Example of relative components positioning

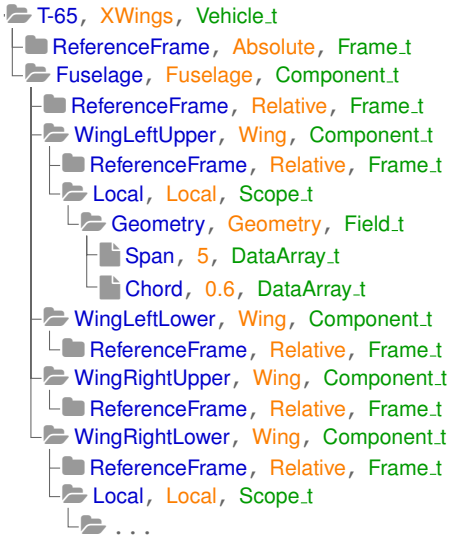


Figure 7: Example of a generalist description

The generalist approach

As long as a XWing can be handled by your code as a Component_t, the previous definition is fine. But to be more generalist, we can describe each wings as a unique Wing which has more chance to be handled by any codes.

As shown in Fig. 7, each wing can be described locally with its own values, different positions and descriptions. Using this kind of description makes the definition more widely compatible and its comprehension by anyone more explicit. In any case, some specific DataArray_t can be added to any Field_t node to store code-dependent data that will be interpreted only by your code.

Data can also be added outside of a Field_t to store data that do not fit the standard. We use for this purpose the UserDefinedData_t node from CGNS. The goal of this data is to be able to use the tree structure to hold data that don't need to be interpreted except if you know what they are made for.

Create a reference to another node

In order to reuse the data already stored in the structure, a new Label has been created on top of CGNS. A Reference_t node

allows to create a symbolic relation between data in the structure. The relationship is defined by two pieces of information.

The Value of the Reference_t node is a reference to the targeted node. It can contain a single or a space-separated list of either names or paths referring other nodes in the Vehicle_t.

The Name of the node defines its purpose through a keyword defining the way the linked data should be interpreted. The suffix Name is added to this keyword in order to constitute the node's name. As shown in Fig. 8, the name of the Reference_t node is GeometryName which means that data referred by the path in the Value should act as a Geometry node.

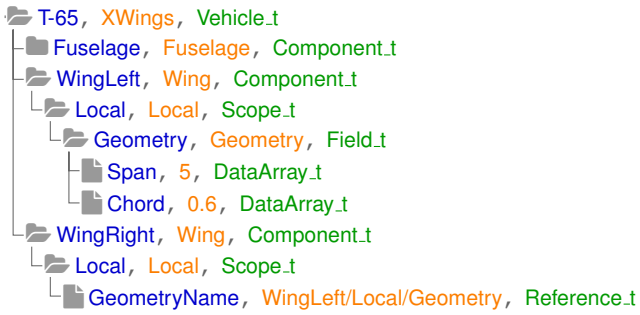


Figure 8: Example of a referenced description

The GeometryName node has for value a path WingLeftLower/Local/Geometry. These paths are always relative to a Component_t so a reference to a Component_t will only have the name of the component as its Value. This behaviour is driven by the modularity requirement #5 in order to be able to move Component_t nodes in the tree structure without modifying values. This also implies that all Component_t of a Vehicle_t need to have unique names even if they don't have the same parent in the tree (this rule was already presented in section).

Those references can more generally be used in order to reference data of a sub part of a component inside the component. For example, a BladeName can be found in a Rotor Component_t in order to specify some information about the blade.

The Reference_t nodes can be used in a variety of situations, in order to reference any kind of nodes. Many use cases are shown in following sections.

Sharing definitions

One of the label added to the CGNS is the Template_t. It resembles the idea of the Family already present in CGNS. The aim of a template is to define a part of the tree only once in order to duplicate it in several places. This enables to share the descriptions of elements (req. #5).
A good example of this is in the definition of components that are used more than once. Indeed, instead of repeating the definition of the components every time, one can simply define it once in a Template_t, and reference this template with a Reference_t named TemplateName. This would look like what is presented in Figure 9.

Figure 9: An example of template

Similarly to the object oriented concept, the template defines something similar to a class with default values, while elements referencing the template act as instances of this class, possibly with modified values.

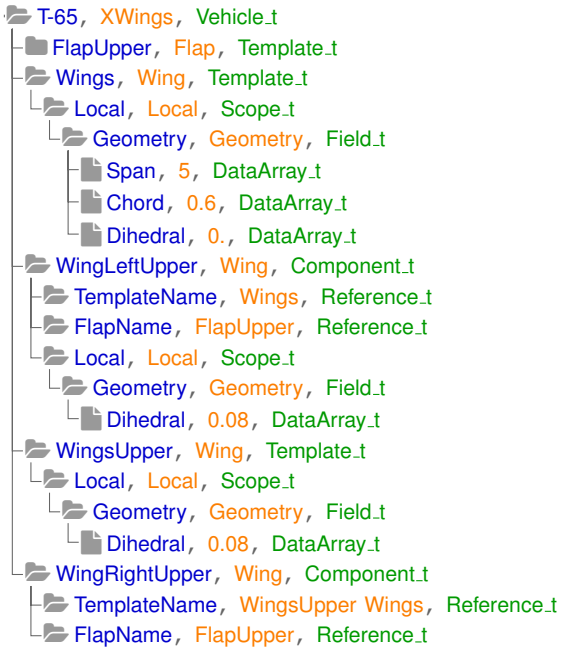


Figure 10: An example of multiple templates

There is a rule of priority of the data if an element locally redefines a node that is already defined in a template. It is possible to reference multiple templates, in which case the templates have priority in the order in which the templates are listed.

For example in Figure 10, WingLeftUpper will not take the Dihedral value of the Wings Template_t node since a local node exists in an equivalent relative path (Local/Geometry/Dihedral). On the other hand, WingRightUpper will have its Dihedral value from the priority order resolution of its TemplateName referring list (the first item has the highest priority). This resolution leads to the value of the WingsUpper/Local/Geometry/Dihedral node. In this example, both WingLeftUpper and WingRightUpper will have the same dihedral angle since both Dihedral nodes have the same value: 0.08 radians.

It is also possible to refer to other nodes than Template_t nodes through TemplateName. One can for example use another Component_t node to act as a Template_t by referring to this component name in the TemplateName Reference_t node's Value. This will use all definitions stored in the Component_t node as default value. Nevertheless, a Template_t node is restricted to mimic the definition of a Component_t

node. Thus, if this template holds a Component_t or Template_t node as children in tree, they are not embedded by the Template_t mechanism, and will not be accessible through the TemplateName. In Fig. 11, WingRightUpper uses WingLeftUpper as a Template_t. In this example, the length of the EngineLeftUpper is not set directly or by a template but by the reference feature describe in the previous section.

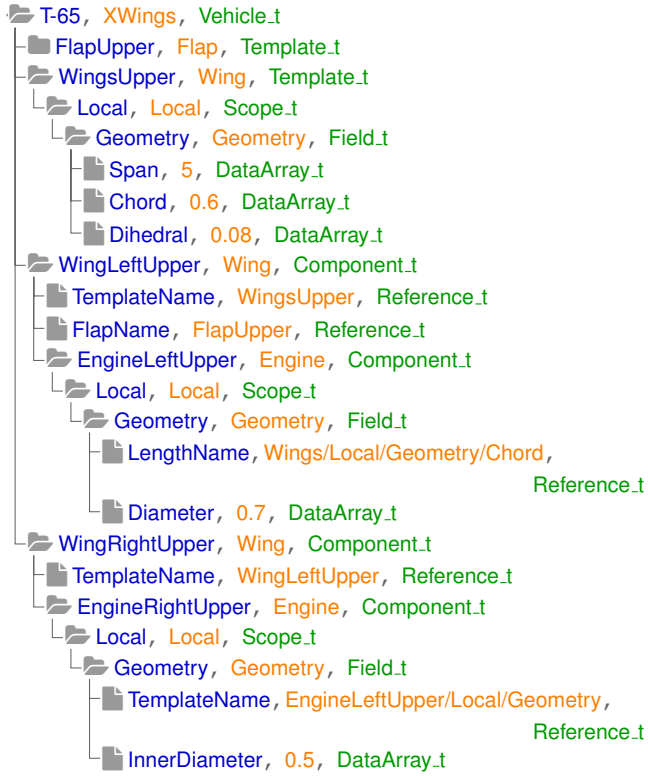


Figure 11: An example of templating a component

It is also possible to refer to a Field_t node of another Component_t node in a Field_t node. The main difference with the GeometryName Reference_t node of Fig. 8, which acts as a Geometry field, is that the Reference_t node is acting as a Template_t, saying that it only provides default value which can be locally specified as shown in Fig. 11. Basically, the templates provide default values, to the difference of other Reference_t nodes, which provide additional information depending on the name of this Reference_t node.

Due to the nature of the template mechanism, there is no direct path from a Component_t referring a Template_t node to another node containing the default value of the template. For example, the Span value of the WingRightUpper in Fig. 11 cannot be accessed by /T-65/WingRightUpper/Local/Geometry/Span since this node does not exists in the tree. It has to be accessed by /T-65/WingsUpper/Local/Geometry/Span which can be resolved using the value of /T-65/WingRightUpper/Local/Geometry/TemplateName.

It is also possible to reference another template inside a template, in which case the referenced template has less priority. It can be seen as a distance of +1 from the local template. The template used in the Template_t node will behave in the exact same way as if it was referred from a Component_t node. A Template_t node can have children they will behave like in any Component_t, but one should keep in mind that a Template_t node does not act as a Component_t node. As said previously, referring to a Template_t node does not include any Component_t or Template_t node it can have as children. As a consequence, Template_t nodes are constrained by the same naming rule of Component_t node: they all must have a unique name in a Vehicle_t node.

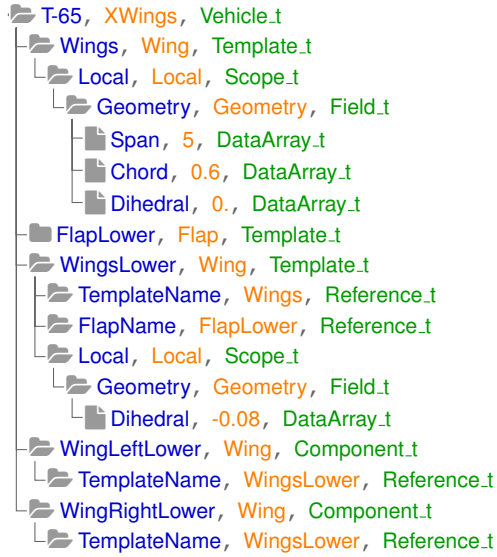


Figure 12: Example of template in template

Linking components

Up to this part, all the Component_t nodes have been considered linked to other through the definition of the ReferenceFrame. This create a parenting relationship between nodes of the tree. This relationship is, by default, a mechanical link and the implicit behaviour is a rigid link between them.

Here, we will introduce a way to link Component_t nodes in a more general way, but also to create other kind of links, such as an energetic link for source and loads, or an aerodynamic interaction between a rotor and a wing. In other words, we are allowing multiple parenting tree in the structure.

To explicit such relations, a Link_t node is used. Placed inside a field, which defines the function of the connection, it references another element through its value, similarly to a Reference_t node, and holds information defining the link with its children nodes. Link_t nodes may contain a DataArray_t node named LinkType commonly having a specific value defining usual types of links, such as Rigid. This LinkType can be more general, in which case it contains the name of a measure explicitly defining the link. The simplest Link_t node defines that there is a link from the Component_t node containing the link to the referenced Component_t, and that the Component_t requires the link information to be resolved. For example, a helicopter fuselage will contain an aerodynamic link referencing a rotor in order to account for its

downwash. Fig. 13 shows different Link_t nodes, with different level of precision. The kinematic Link_t node in the Wings template redefines explicitly the rigid link between the fuselage and the wings, while the Link_t node in the Energetics Field_t node of Engine1 defines its dependence on the fuel tank. In this last example, the type of link is explicitly defined as a MassFlowRate type of link.

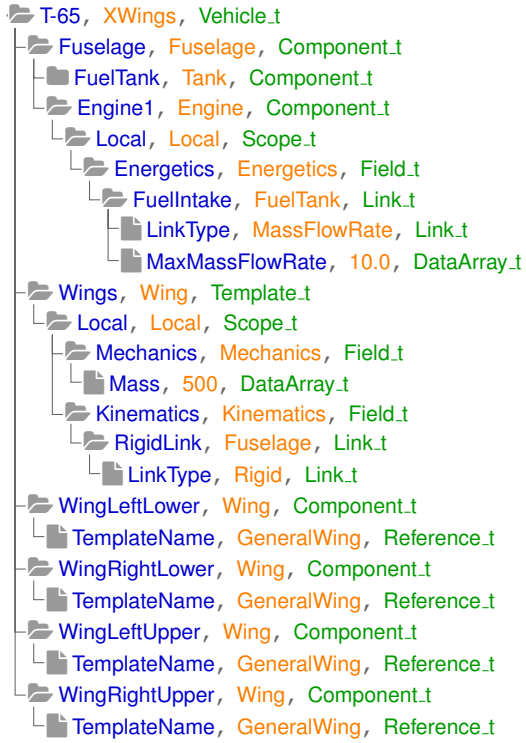


Figure 13: An example of Link_t nodes

It is to be noted that a kinematic link from component A to component B will imply a kinematic link from component B to component A. We leave to the user the choice to explicit this kinematic link which by default is also Rigid.

The Link_t nodes create networks, on top of the one defined by the structure of the tree, allowing to expand it, but also to introduce nodes with multiple parents, and thus graphs behaviours.

Storing a coherent set of data

Quite often, data only makes sense in relation with other data. For example, the definition of a polar will require the association of the angle of attack with the lift coefficient. To ease this kind of association, we use a DataContainer_t node. DataContainer_t allow to explicit the independant variables through their value, and define a set of coherent data. In Fig. 14, AngleOfAttack, SideSlipAngle, MachNumber and ReynoldsNumber are independant variables, which give the domain over which DragCoefficient and LiftCoefficient are defined. The data are dependent of four variables, and their sizes is thus the product of the independent variables sizes, respectively N_i, N_j, N_k, N_l. Data are thus ravelled and we use the "C-style" order (row-major). In the present case, accessing to a specific dependent value is done using the following algorithm where index starts from 0:

$$
\begin{aligned}
\alpha &= \text{AngleOfAttack}[i] \\
\beta &= \text{SideSlipAngle}[j] \\
\text{Mach} &= \text{MachNumber}[k] \\
\text{Re} &= \text{ReynoldsNumber}[l] \\
C_l(\alpha,\beta,\text{Mach},\text{Re}) &= \text{LiftCoefficient} \\
&\quad [i + N_i * j + N_i * N_j * k + N_i * N_j * N_k * l] \\
C_d(\alpha,\beta,\text{Mach},\text{Re}) &= \text{DragCoefficient} \\
&\quad [i + N_i * j + N_i * N_j * k + N_i * N_j * N_k * l]
\end{aligned}
$$

DataContainer_t nodes can be used for all kind of purpose, from geometry definition to polars, and are situated on the same level as the DataArray_t, in order to benefit from the context definition.

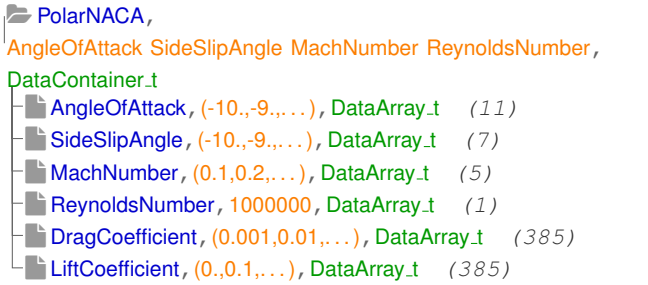


Figure 14: An example of a DataContainer_t node, where the size of the data has been specified.

Adding an environment

Nodes with the label PhysicalEnvironment_t allow to specify the data for the environment in which the vehicle manoeuvres and the various model required to represent it. Those include typically an atmospheric model, a gravity model and a weather model. Those nodes are placed at the root of the tree, on the same level as Vehicle_t nodes, to ease their access by the vehicles.

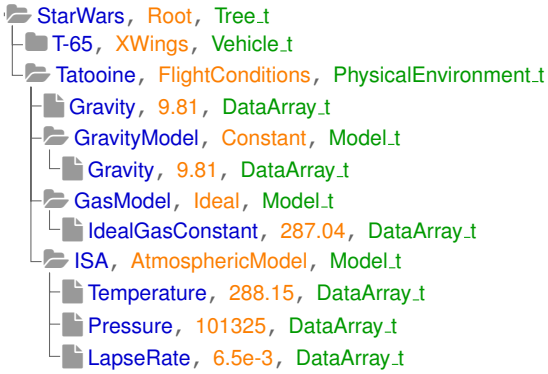


Figure 15: An example of PhysicalEnvironment_t nodes

An example of such a tree is presented on Figure 15: on the same level as the T-65, the physical environment is described.

This introduces the use of the root of the tree, which will be, by definition, the level 0.

All information stored in a PhysicalEnvironment_t node are self-sufficient. Such nodes can be easily reused from one simulation to another.

Defining mission to be run

Mission_t nodes are containers divided into Phase_t nodes, allowing to set values to run simulations on the defined vehicles or components. A given mission can also be subdivided into Mission_t nodes that will contain Phase_t nodes.

The Phase_t nodes define the kind of simulation that are to be run. In a general way, running a simulation can be defined either by a set of hard or soft constraints, with or without an objective, to be realised with a set of modifiable parameters, in a way similar to an optimisation problem. Finding an equilibrium can be defined as a constraint on accelerations.

Three kind of phases are defined here, but they are not an exhaustive list of the phases values that could be defined:

- Static phases allow to run equilibrium simulation where the aim is to reach imposed states. The solution is time-independent. They are mainly used for trims. An example is presented in Figure 16, with the Trim phase, which is to be run before the rest of the mission as an initialisation.
- Dynamic phases are used for dynamic simulation. They define time varying data to be followed, such as objectives or control values. In Figure 16, two phases are defined leading the aircraft to hover conditions. Those phases only prescribe the objectives to be reached, and not the actual values.
- Design phases are used to design an aircraft or a component. They define a list of objectives and controls to an optimisation process.

In most cases, there is no reference to the content of a Vehicle_t in phases. But for some reasons, mainly to define specific objectives such as failures, one can refer to the name of a Component_t node or to the path of a value hold by a Vehicle_t involved in the mission. This is more the case in Design phases where objectives are values of vehicle or component definition.

In all cases, the methods used to resolve the missions as well as numerical parameters are left to the codes. Nonetheless, it is possible and could be useful to use the tree to store such information in UserDefinedData_t nodes. These information are thus dedicated to certain software so it needs to be clear to all other codes they are not to be interpreted. The name of these nodes can be a clue to achieve this precaution.

Figure 16: An example of Mission_t and Phase_t nodes

Storing results

Missions can be read and interpreted by any codes, and their results are to be stored in Result_t nodes. The Result_t nodes refer to a specific Phase_t with a PhaseName node of type Reference_t which contains the phase path as its value. They are then placed in their concerned fields in order to benefit from the full context of the tree. They can be created in any component, so that any vehicles or components may save results of any missions.

Their behaviour is similar to the one of a DataContainer_t node in the sense that their value refers to the independent variables, while the DataArray_t nodes they contain should form a coherent set of data.

A Result_t node is owned by a specific code and should not be modified by another code. It is the sole prerogative of the code who created it to update it. If two codes work in the same field for the same phase, they should use two different Result_t nodes.

In Figure 17, examples of Result_t nodes are shown. The trim phase having no independent variables, their value is simply set to None. The results of the dynamic phase are for their part dependent on the time of the simulation. The apparent redundancy of storing several time DataArray_t nodes might seem contradictory, but each Result_t node needs its own independe nt variable which may or not be the same. Please note that in Figure 17, the arrays have been flatten for lack of a better representation, but n-dimensional arrays are a possibility.

Of course, some results of some codes may modify directly values of the trees, others than in the Result_t nodes. This simply requires to distinguish between the current state of the vehicle during the mission and the stored history of the mission. For example, a time simulation might both modify the position of an aircraft while storing it in a Result_t node.

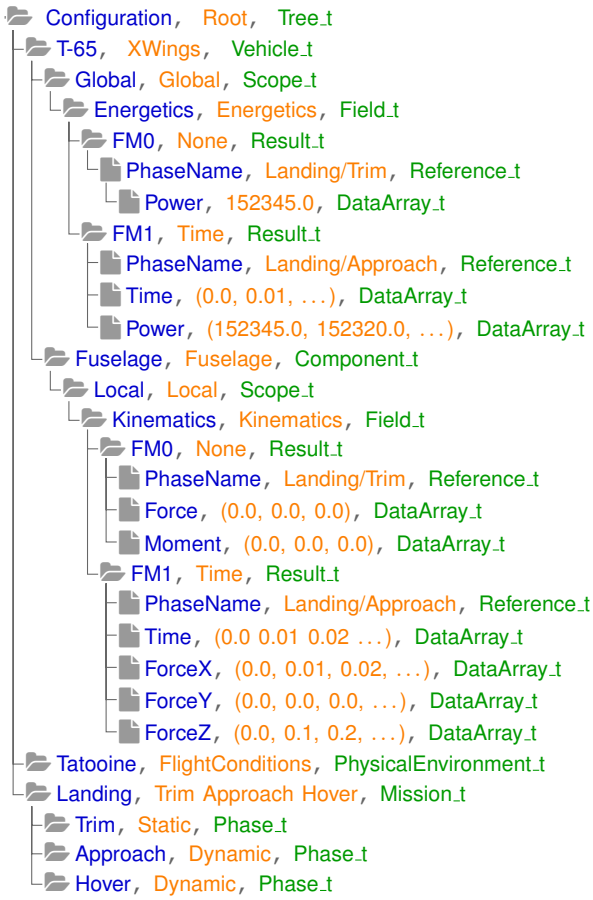


Figure 17: An example of Analysis_t and Result_t nodes (the results presented here may not be accurate)

NAMING

A number of basis have been laid out in the previous sections, in order to ease the reader's comprehension. Here we will define in a more detailed way the different labels, as well as a number of reserved or preferred names.

As this norm is open and intends to encapsulate all kind of fields, it is not exhaustive and proposition from users are welcomed.

Labels

The following grammar is built on nested node tree. Each node functionality is defined by its Label, so these labels need to be commonly defined and are reserved names.

Labels are defined hereafter :

Tree_t is the first node of the full tree (its root). It is unique in the tree as it defines the only node not hold in a node's children list.

Vehicle_t defines a full vehicle, such as an aircraft.

Component_t defines a component of a vehicle, such as a wing, an engine... Components are designed to facilitate the assembly and the modelling part but they are not mandatory. Their value defines the kind of Component_t represented by the node, and are most commonly among Wing, Propeller, Fuselage, Engine, ...

Frame_t defines a local frame, which can be relative or absolute. By convention, all positioning data refer to the closest Frame_t node named ReferenceFrame.

Scope_t defines the scope in which data has to be understood, such as local to the component, global to the system or inherited from external component. The type of scope is defined by its value, which are among Local, Inherited and Global.

Field_t defines the kind of field data are associated to, like mechanics or aerodynamics fields, defined by its value. Common values can be Aerodynamics, for data linked to the aerodynamic properties of a Component_t, Mechanics for structure and material properties, Energetics for data related to the generation and consumption of power, Kinematics to store velocities, ...

DataArray_t store data itself. The name of this node defines which measure is stored and the node value store the value of this measure. Most of the common measures are defined by CGNS SIDS and this norm comes to complete a part of it. The more the measures are commonly named, the more the tree can be interpreted widely.

DataContainer_t store a set of correlated data.

Reference_t defines a reference to another node. Their name are built following the strict rule defined in section and hold the node identifier they are referring to as their value.

Template_t defines a node containing default values of a shared definition for several other nodes, that will be able to refer to this definition through a Reference_t node. This allows to define only once a Component_t, such as a rotor, and use this definition several times.

Link_t defines a unidirectional link between two Component_t node in a given Field_t.

Mission_t defines a mission to simulate. Missions are independent of each others and there can be as many as needed. They are referring to one or more PhysicalEnvironment_t node. Their value contains the ordered list of the Phase_t they are constituted of.

Phase_t defines a phase (sub-mission) to simulate. They can be assimilated to mission task element.

PhysicalEnvironment_t defines the physical environment in which the vehicle is manoeuvring. Several PhysicalEnvironment_t nodes can be stored in the Tree_t for different missions or even different phases in a mission.

Model_t defines modelling data for world interactions, such as gravity, atmospheric, gas, wind. They are not mandatory if the model is a constant itself (such as Gravity which could be set as a DataArray_t directly as a PhysicalEnvironment_t child.

GRAMMAR

The grammar defines rules that need to be satisfy to be able to have a common comprehension of the data. It also allows a common system of storing so anyone can retrieve data in a fixed way. These rules are independent from the implementation language of the norm.

This grammar defines two absolute levels in the tree: Level 0, for the Tree_t node, and Level 1, for the Vehicle_t, Mission_t and PhysicalEnvironement_t nodes. Deeper levels are defined relatively to their parents but they have a minimum level ranking up to level 5. This is one of the main difference with CGNS, which does not allow this relative behaviour.

A number of rules concerns the naming of the nodes in the tree: the names of the nodes should not contain any spaces character and should follow a PascalCase (UpperCamelCase) convention. Two nodes at the same level may not share the same name, and two Component_t may not share the same name. Template_t nodes have a role similar to Component_t nodes, and therefore may not share a common name with Component_t or other Template_t nodes.

Other rules are concerned with the use of the tree, and ways to properly include data in it.

For those manipulating the tree, it should always be considered coherent, and maintaining this coherence is up to the one manipulating the tree.

The order of the node in the tree is not an information. If this order matters, it should be stored as the value of a node (see for example the orders of the phases in a mission).

Due to the living nature of the structure, the tree in fact stores references to the list of its children and a reference to the data value. This means that the reference should not be broken, in order to avoid other codes to lose the reference on the value.

On top of those grammatical rules, a number of meta rules have been followed to constitute the present definitions. We lay down those rules in this paper in order to facilitate the extension of this standard by a user in a coherent fashion.

Similarly to CGNS, the first meta rule is that the structure aims at minimising the duplication of data in order to avoid inconsistencies.

A second meta rule imposes that a node should provide enough information through its Label and Value to know what to expect, without having to read its children.

Except for DataArray_t and Reference_t nodes, the name of the nodes does not provide any kind of information, only an identifier. There is also some reserved names for which the name holds information (as TemplateName).

EXAMPLE

To highlight the benefits of this data structure, we use the example of a mass performance convergence loop ran on an eVTOL aircraft. A Python version of the standard has been implemented. This implementation allows to fully answer the dynamic requirement: the tree is a Python object that lives and follows the evolution of the data along the simulation process. In order to perform this convergence loop, a flight mechanics code, which provided Fig. 18a, and a mass code providing Fig. 18b were setup in order to interpret the tree. A third code, providing a free wake model, represented on Fig. 18c, allowed to refit the lower fidelity model of the flight mechanics code. Once the interpretation of the tree is setup, the coupling of the codes is done with a Python script: the mass code will compute the mass, centering and inertia of the aircraft, and fill all the weight nodes in both the Local and Global scopes of the components. The flight mechanics code will then run the performance computations with the updated values of the aircraft's global mass and inertia that can be found in the Global Scope_t and Mechanics Field_t. This update is "free" because of the dynamic nature of the tree. When the mass code writes the global mass in the tree, it is directly the value held by the flight mechanics code which is modified, since both codes use the same tree. In fact, in Python, both codes share the memory blocks of the tree. Then the power requirements of the motors and battery are updated, and the mass code is rerun, with the updated values from the flight mechanics code.

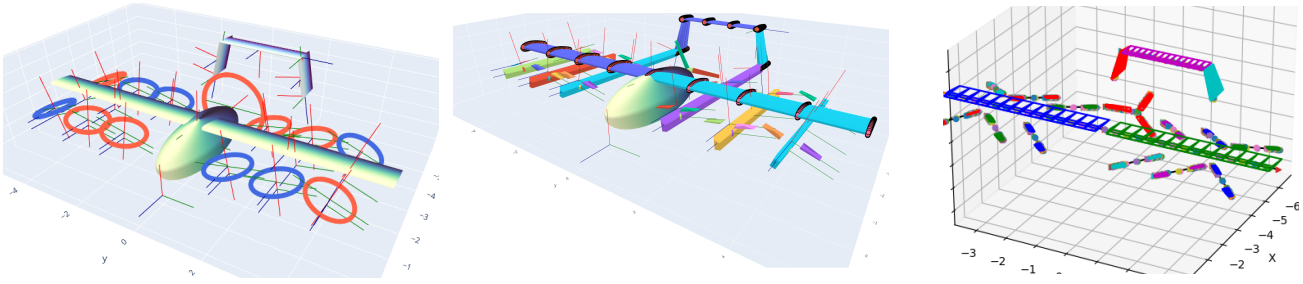
This drastically reduces the work required to interface different codes, to the price of a a single interfacing with the tree. Once this interfacing is done, all later couplings are simplified.

In this case, the flight mechanics code, and the mass code do not use the same set of data to define their components. For example, the rotors are highly simplified in the view of the flight mechanics code, while the mass code has a more refine view of their geometry. However, the free wake code requires the same level of fidelity of the mass code in the geometry description, with additional information of the aerodynamic profiles, while the mass code requires material and structural information. This shows that several levels of fidelity may coexist in the tree, and that data can be shared between codes sharing the same level of fidelity.

WHAT IS NOT INCLUDED IS THE NORM – OFFTOPIC

The norm has been built in order to store data, and to fit our requirements. Some uses have however been excluded. Among those uses, there are two main exclusions.

The first one concerns the resolution methods. Although the tree may contain sets of parameters for specific methods, the explicit methods, or their internal workings are out of scope for this data structure. However, it is always possible to store useful information for these methods in UserDefinedData_t nodes.



(a) View in DynaPyVTOL flight mechanics code

(b) View in a mass assessment code

(c) View in AMB free wake model code

Figure 18: Views from different codes of the same configuration for which a single tree has been built.

A second exclusion concerns the sequence or the procedure organising different codes. For example, during an multi disciplinary optimisation process, resolution codes are called in sequence. The tree will provide storage space for the inputs and outputs of each code, but the sequence in which they are called is not stored in the tree. At this time, there no way to store algorithms or data flow.

PERSPECTIVE

Conventions for shared definitions

The norm is accompanied with a set of conventions. Most notably those conventions concerns the orientation of the usual components, to define a default behaviour. The z axis of a rotor is oriented in the opposite direction of its thrust, and other similar rules. We will not go into the details of all the orientation convention considered here.

There are also conventions for the signs of quantities. For example, for the power, it should be positive for consumption and negative for production.

There are in fact many more conventions that we define implicitly, or that we have not yet stumbled upon. Therefore, the norm currently lacks a good way to define the implicit convention, as well as a way to define a new convention. In fact, the best way to define convention would be to describe and store this convention directly in the tree or at least, in an interpretive way for codes, not only on paper.

Units

As of now, all data are stored in SI units and there are no means to do otherwise. A nice addition to the norm would therefore be a way to include other units.

CGNS propose a way to add this behaviour, but it feels too cumbersome for our uses. It may be the right solution, and it is not excluded to use it, but a more convenient system would be welcome.

Vehicle links

Currently, we have not introduced properly a way to model any kind of links between two vehicles. Those links could be useful for in flight refuelling, ship deck landing, aerodynamic interaction between a helicopter wake and drones, spaceship on rocket, etc.

Dealing with equations

At a certain point, it would also be great to have a way to store equation instead of numerical value. The use of Reference_t node pointing to DataArray_t node of other component or field is a start but that does not really define equations. A new Equation_t node type could be useful but any proposition are very welcomed.

Parallel and distributed tree

Such tree can also use heavy computation to solve problems. In that case, softwares are usually parallelised on multiple CPU or even computers. The tree needs thus to be reproduced and synced on every representation of it. This kind of approach has been widely done in CFD softwares (citeelsA/CWIPY) with CGNS trees. We are looking to use methods and tools already released from CFD community to solve this.

CONCLUSION

In this paper, we have presented the development of a data structure fitting a set of requirements, that we propose to use for simulations of vehicles. Based on the existing CGNS, this structure allows to contextualised the stored data. It is general by developing enough context to fit all required fields. It is extensible, as some room is left to other uses. It is dynamic by remaining up to date during the full extend of the simulation. And finally, a modular behaviour has been introduced.

We would like this work to be widely used by the community, and we are open to share and amend it. Some tools have been developed to implement this standard in Python, and we will share openly soon.

ACKNOWLEDGEMENTS

The authors would like to thank Luis Bernardos for his advice on the development of the standard and Jean-Paul Reddinger for his help making the acronym.

REFERENCES

1. Bernard Benoit, Konstantin Kampa, W von Grunhagen, Pierre-Marie Basset, and Bernard Gimonet. Host, a general helicopter simulation tool for germany and france. In *Annual Forum Proceedings-American Helicopter Society*, volume 56, pages 1110–1131. AMERICAN HELICOPTER SOCIETY, INC, 2000.

2. RW Du Val and C He. Validation of the flightlab virtual engineering toolset. *The Aeronautical Journal*, 122(1250):519–555, 2018.

3. Marko Alder, Erwin Moerland, Jonas Jepsen, and Björn Nagel. Recent advances in establishing a common language for aircraft design with cpacs. 2020.

4. Robert A McDonald. Advanced modeling in openvsp. In *16th AIAA Aviation Technology, Integration, and Operations Conference*, page 3282, 2016.

5. Robert A McDonald and James R Gloudemans. Open vehicle sketch pad: An open source parametric geometry and analysis tool for conceptual aircraft design. In *AIAA SciTech 2022 Forum*, page 0004, 2022.

6. Marc Poinot and Christopher L. Rumsey. *Seven keys for practical understanding and use of CGNS*.

7. Diane Poirier, Steven Allmaras, Douglas McCarthy, Matthew Smith, and Francis Enomoto. The cgns system. In *29th AIAA, Fluid Dynamics Conference*, 1998.

8. CGNS Steering Committee. Standard interface data structures – design philosophy of standard interface data structures, 2016.

9. CGNS Steering Committee. Overview and entry-level document – elements and documentation, 2011.

Author contact: Simon Verley: simon.verley@onera.fr

CLASSIFICATION & COPYRIGHT