CGNS++ Design Document

Manuel Kessler* Udo Tremel†

15th February 2002

Abstract

We propose a new and improved interface for the CGNS library to be used with the C++ programming language. We explain some of the decisions made in the design process and later on while implementing.

1 Motivation

The design of a new wrapper to an existing library for a new language is a delicate task. On one hand, there is often already a way to access the functionality of the library (as is the case with C++, which is able to use the C interface of the CGNS midlevel library), and therefore many people may use the library already and are accustomed to some idiosyncracies and idioms for clever usage. For those people a more compatible interface is preferable. On the other hand, a new language provides new features to help the programmer get her job done, so it is a waste of time and effort not to use advanced features which support good programming style and aid in debugging.

We are convinced that the existing C and Fortran interfaces are well designed and the library is nearly bug free. For the foreseeable future they probably will be used from most people using CGNS, and therefore they definitely should be maintained and enhanced as necessary. But some people use already other languages for application development, most notably C++ (and possibly some others, like Python). Of course it is possible — and people have done so — to use the existing C interface in a C++ application, but this approach has some drawbacks. In fact, one is forced to use only a small subset of the language in a considerable part of the whole application. But why should we then bother using C++ in the first place.

Therefore we decided to design and implement a completely new object oriented interface to CGNS. It seemed as a waste of time to repeat only the C interface with C++ syntax and some small adaptations. People uncomfortable with a new interface and familiar with the existing C one may of course use it as they like. Others can use the new interface and

*University of Stuttgart, IAG
†EADS, Military Services

benefit from the features provided: more safety against accidental misuse (e. g. mix–up of identification numbers), less programming effort to achieve the desired result, and a completely object oriented interface more suitable in an object oriented application development.

Probably the most important bonus from using an advanced programming language like C++ is the automatic handling of many recurring tasks, like initialisation. For example, in the existing midlevel library you can easily try to access some zone data before actually initialising the zone itself. Of course, you will get a runtime error if you do it, but then several hundred expensive CPU hours on a supercomputer may be wasted (even worse, if you do not check error values everywhere, such a bug can slip through unnoticed for a long time). In our current approach, if you have an object of type Zone, it is always properly initialised, either by reading from a file or by creating it recently. Without a Zone object it is syntactically impossible to get at the data below, so if we try, a compilation error occurs. And it is enough to pass only one object to a function or subroutine doing some I/O, not several plain integer identification numbers, possibly of varying count.

Each object is responsible for a consistent structure and data in its subnodes. As far as possible, this consistency is enforced at compile time or link time (of course, some checks have to be postponed until runtime). Thanks to overloading we can make available many different flavours of the same function, with identical semantics, but different interfaces. For example, in the class DataArray there is a function readData, which is responsible for the actual data access. There exist many variants, to read into a bunch of integers, floats or doubles, with several striding capabilities, in several dimensions. They all read the data from the disk file, so it is sensible to name them identical, but depending on the data structure of the application, the actual read has to be done differently. So overloading reduces the learning time (only few function names and their semantics have to be remembered) while allowing a high flexibility (many different variants implement the semantic interface).

Unfortunately, as we had to learn from earlier experience, it is very difficult to implement such an interface as a simple wrapper on top of the existing midlevel library. The most problematic point is that is is not object oriented at all. While the SIDS specification may be easily used to conceptually map the nodes to objects, this is not the case with the midlevel library. There is no such thing as a single object identity for any of the nodes. For every SIDS node one needs a bunch of numbers, for example a simple solution field is accessed using the file, base, zone, solution and field numbers. Even worse, for nodes which can be attached to several parents, e. g. the DataClass, one has to use the cg_goto function with a string representation of the path down to the desired node. It is nearly impossible to handle this in a general, safe and consistent way. One can see this in the existing implementation, where much functionality like validity checking is duplicated again and again. Because of this, we opted in the process of implementing for a different approach, namely a new library parallel to the existing one, implemented in C++ and on top of (a simple OO wrapper on top of) the ADF library. As far as we can see right now, where about half of the midlevel library functionality is implemented already, we did not need more time or effort than a conservative estimate of the alternative approaches would give.

We point out that the proposed library is functionally equivalent to the midlevel library, with syntax and semantics appropriately adapted for the C++ language. It is not a high level tool to manipulate CGNS databases in general (like arbitrarily moving and copying nodes etc.). Such tools may be built with the aid of our library, of course, the same way they could be built with the midlevel library.

2 Language

There are several basic design aspects to consider at the very beginning, since they influence the whole process. One is the above mentioned decision to build really a new interface, not a cosmetic variant of an existing one. Another one is the programming language to use. While C++ was intended, still several possibilities exist. The language is relatively (compared to C or even Fortran) new, and some older compilers lack important features, like templates, namespaces and exception handling or the standard library specified for the language, or more esoteric ones, like Koenig lookup or exported templates. However, there is now the ISO 14882 standard since the end of 1997, and the features most people use sensibly have settled since 1994 or earlier. So decent compilers should not have problems, and older ones should be dying out slowly.

Now a very feature–rich programming language is at our disposal, but we have to decide what to use and what not. In CFD performance is one of the most important aspect of an application, so if a feature does hinder performance too much, we should not use it. But as often, this is a difficult engineering decision, since compiler (and hardware) technology evolves rapidly. We have to draw a line somewhere and fix the ideas (in no particular order):

- Templates are being used in the implementation, but don't appear in the interface. The latter is a simple consequence, since there seems to be no good use for it. The usage of templates in the implementation helps with development, because many similar tasks can concentrated in a single place and used in many different contexts, where they have not to be duplicated again and again (with the corresponding maintenance effort). Template technology is now quite mature, and even newer features like typename and traits are known for a long time now. However, we don't use template template parameters (they are not needed anyway).

- Namespaces are used as well. Technically they are not really necessary, but there is hardly any compiler with no basic support for it. We don't think anybody is taken out by this decision alone.

- Exceptions are a clever way of error handling. There is much discussion whether they should be used or not, especially in the area of high performance computing. We decided to do so, but in a way not to rule out alternatives (by concentrating the exception machinery in a single place). There is more to be said about this topic later on.

- The standard library is used, wherever appropriate, and in the standard way. Standard library support is still evolving on many systems, but the parts we use — mainly the auto_ptr template and some containers — should be available everywhere. If experience proves that this is not the case, the relevant parts may be changed or replaced without too much hassle.

- For names we use the standard string class in the interface. The string class helps a lot in the implementation, because we do not have to bother with memory management for character arrays. This is even more important since we use exceptions. And if we use it in the implementation, why not in the interface? For string literals (or char * in general) there exists an implicit conversion constructor to std::string, so this is not a problem in user code for parameters. For return values the user can easily call the c_str() member function to get a const char *, if necessary. String support in compilers usually is quite good, at least for the plain char variant.

- We do not use virtual inheritance. It may have been useful to simplify some parts of the implementation, but we found other ways to express our design without having to resort to multiple inheritance and the then needed virtual inheritance. Thereby we reduce the conceptual complexity of our interface a little bit. Our alternatives are also slightly better for performance, but the difference is probably hardly measurable. Finally, compiler support is sometimes flaky, namely on the development platform.

Now we come back to error handling. Basically, there are two ways used in most applications to handle error situations: simply abort (after dropping an appropriate message), which is a sensible way to go for batch applications, or try to recover up to the point where the error may be resolved, and the action restarted. The latter is more useful (or even indispensable) for long running interactive applications. We support both styles. If you know and do nothing about error handling, and an error occurs in the library, which results in an exception being raised, the standard procedure is to finally abort the application. Fine. If you enclose your main() function (respectively the body) with a try block, you can catch exceptions raised by the library (and others) and at least write a message with some more specific information. This is better, since it adds less than ten lines of code to the application in total and is much more informative than a core dump. However, it is possible, of course, to enclose any call to the library with a try block, and act appropriately if an exception is catched. That way a total control over each and every call is possible, but it is also possible to guard many related calls with a single try block. Therefore the granularity of error handling and the corresponding actions is totally left to the user.

The existing midlevel library gives an error flag back. While it is possible to check every call as well, all this must be made by hand, but with exceptions the compiler takes care of tidying up if an exception occurs. Another argument is performance. It is possible to compile (nearly all) code using exceptions without any runtime overhead, if no exceptions are thrown, and many compilers do so already. If an exception is actually thrown, the (hopefully highly optimised) runtime system takes care of the necessary actions. If error codes are used instead, the status has to be checked explicitly by the user after each and

every call. So exceptions give us the power of fine–grained error checking, but have the flexibility of a less detailed handling, and do not hinder performance.

For systems where exceptions are not available, it is possible to disable them at compile time and to revert to an error handling by aborting (after dropping a message, of course). While in this case no fine–grained handling is possible, this situation probably happens rarely. After all, on nearly all workstations, which may be the kind of systems used for interactive applications, decent compilers exist, at least the GNU one.

The last statement holds as well for the other features required. On all major workstations decent C++ compilers exist, and if not the GNU compiler is available in one version or the other. So there is no need to cripple a new interface intended to be used for some time to come with workarounds or suboptimal solutions because of obsolete tools.

3 Semantics

Before implementation can start the library as a whole has to be designed with some basic semantic properties used consistently. The intended use of the library gives valuable insight into a properly designed interface.

3.1 Object Mapping

The first decision is the mapping between the SIDS specifications and actual C++ objects. It seems obvious that every SIDS node corresponds to a C++ class object, but it is not the best way to choose a direct 1:1 mapping between a C++ class object and each ADF node. To be more specific, many ADF nodes only hold a single value, and some of them are required if their parents are present. For example, the Descriptor_t subnode does only hold a string of text, or the ZoneType_t subnode holds an enumeration value (string) and is required for each zone. It is not necessary to handle them the same way as the hierarchical nodes, and more sensible to provide access to them in the particular parent node. Our design therefore assigns C++ classes only to nodes where child nodes are allowed or required by the SIDS, but simple subnodes representing only single (or — as e. g. Rind_t — a small fixed number of) values are handled in the parent node classes.

Wherever possible, enumerations are used for the plain value nodes instead of plain strings, which have to be parsed in the application anyway. Instead, the library handles the parsing (possibly more efficient, certainly more reliable) and user code is simplified. The only exceptions are user–defined data arrays, which can be named arbitrarily (as long as no name collision happens).

3.2 Object Semantics

Probably the most difficult job is to define semantics for the classes. Basically it is possible to give them value or reference semantics. For example, if we copy an object with value semantics (like the builtin types, as int or double), a new distinct object is created which

can be used completely detached from the original one. If we change one object the other one is not affected. On the contrary, if we copy a reference, we create only a new handle to the underlying existing object. If we change it through one reference, the change is visible immediately through the copied one. If we want to have a "real" duplicate (if this makes sense at all) we have to call a special function to do the job, which returns a new handle to the just copied underlying object.

The discussion is language specific. In C, for example, objects (the builtin types, structs etc.) have value semantics, and you have to explicitly use a pointer if you need a handle. In many other languages, like Java or interpreted ones like Perl or Python, all object are basically references to some underlying data. In the latter case the reference counting mechanism and some kind of garbage collection are handled automatically by the system. In contrast, in C++ the programmer has the choice, since the basic operations construction, copying, assignment and destruction are accessible to her and may be given arbitrary semantics. As a drawback, if we need reference counting, we have to do it ourself.

Since objects definitely need some semantics, we have to think about it, even if we simply choose to disable copying and assignment explicitly. Otherwise the compiler would create default copying and assignment functions, which may not behave as desired. So what are the arguments for one scheme or the other.

Looking at the existing midlevel library, the library is responsible for the data handling, not the application, and the user is concerned only with a handle to the data, namely the "bunch of index numbers" (fileNo, baseNo, zoneNo, ...) mentioned earlier. This corresponds closely to the notion of "the data is in the file". The midlevel library caches part of this data, mostly the structural content, but this is not visible from the interface. It could as well read any data as requested. It sounds sensible to mimic this approach.

Another approach is by thinking about copy or assignment semantics. What does it mean to make a copy of e. g. a zone? Should all the data in the existing zone duplicated? If yes, we should at least give the new zone a distinct name. Under which base should it be placed? The same one as the existing zone? None? Then most of the data, at least most of the large arrays, are useless anyway since two zones in one base hardly use exactly the same coordinates, for example. Assignment is even worse. Should the existing structure and data be disbanded or merged with the assigned one? If there are conflicts, which one takes precedence? What happens if we assign an unstructured zone to a structured one? These operations — if needed at all — definitely are up to the application, and not to the proposed library.

Clearly copying and assignment are at least problematic to define sensibly with value semantics. On the other hand, if we use reference semantics, copying and assignment are trivial, since they only create new handles to existing objects (or reassign handles to other existing objects). The library should provide only functionality to access the data, and is not a tool to manipulate it (besides the writing capabilities for the SIDS nodes).

In conclusion, reference semantics for the user visible interface are the right way to implement the database design specified by the SIDS. This decision is similar as for the existing midlevel library, defines clear semantics for copying and assignment and makes for safe use.

Nevertheless, we are not entirely finished with this topic yet. While the semantics are clear now, some syntactic and minor semantic issues are still to clarify. Should we use the builtin C++ references to represent the semantics or should we create handle classes, which hide the mechanics of dereferencing the real data? In user code, the only difference is a simple & sign present in the declaration of an object or not. So this seems to be a minor point. But there are some consequences to consider. If we use plain references, the real classes containing the real data have to be specified in the header files, together with the access functions. From a data abstraction viewpoint this is unnecessary clutter in the user (programmer) visible interface. Furthermore, it requires a complete recompilation of the user application whenever some internals of the library are changed. This is usually not a problem for an in–house application, but if a third–party library is available only as object code compiled with an outdated version of the CGNS++ library things become difficult. Another point, which is a safety issue: if nodes are to be deleted, the problem of dangling references occurs. This may lead to subtle and hard to find errors.

On the other hand, if we create handle classes, the interface is kept to the minimum and the real objects containing real data can be hidden easily. The user code cannot see them and does not have to, since any access goes through the handle class. Since the binary representation of the handle class remains stable for a much longer time than a single version library itself, a linking step to a new version is enough to upgrade. If we finally use reference counting for the real objects, no dangling references can occur. If we access a node which has been deleted before, an exception is thrown. Such a reliable action is definitely preferable to the erratic possibilities of undefined behaviour, namely using old data, overwriting accidentally other useful data (or even code) or just aborting.

To sum it up, all SIDS nodes possibly containing children are modelled by C++ handle classes. Copying them just duplicates the handle, which is a simple operation and therefore very useful even for function parameters or return values. Existing handles to deleted nodes show defined behaviour, namely throwing an exception if used. We decided to explicitely disallow assignment, since in contrast to copying we cannot imagine a sensible use for it. If somebody convinces us otherwise, implementation is next to trivial.

3.3 Child Handling

Another issue is the handling of the different children of a class with different cardinalities. For optional singular children we choose to expose a hasXXX function, which returns the availability of the child in question, and a getXXX function to return a handle to it. In the latter case, if the child requested does not exist, an exception is thrown. There are not many sensible alternatives to this approach. If we take DataClass as an example, this would translate into a hasDataClass() function and a getDataClass() function.

Things are slightly different for children with an arbitrary cardinality. We decided to provide a function that returns the number of children of this kind, getNumXXX, and an access by name, getXXX(name) to get a handle to the specified child. Again, an exception is thrown if not available. But that is not sufficient. Often the names are unknown or irrelevant, we just need a way to handle every child of some kind. For example, zone names

are usually quite arbitrary in a solver application, which just has to solve the equations in all zones. Therefore we added a way to iterate over all children of the requested kind, by providing an iterator to the first one (beginXXX) and past the last one (endXXX). They can be used in the spirit of the standard template library of C++ (STL) to handle for example all zones of a single base. Technically, they have forward_iterator semantics. We do not consider it necessary to enrich the iterator semantics with more functionality, simply iterating over all children should be sufficient.

Like access, deletion (deleteXXX) is possible either by name or by an iterator. Of course, after deletion the iterator becomes invalid. However, other iterators remain valid if children are deleted or newly created.

Child creation does not pose many problems. If there are required data values or child nodes, they have to be specified at the creation call (writeXXX), and the ADF structure is built accordingly. If there exists already a child with the name specified (or dictated by the SIDS), this child is cleaned from any previous subnodes and data and the new data is applied. The existing midlevel library handles this case in the same way.

Since linking capabilities are currently in the implementation stage for the midlevel library, we have to include them as well. They will be addressed as soon as the functionality offered by the midlevel library is fixed. We do not anticipate any problems in supporting linking. The same holds for user defined data arrays, which are to be implemented as well. In general, the addition of new child nodes is as simple as generating two short files for the functionality of the child nodes (mainly by copying and slightly adapting an existing file), adding the declaration of the node in the main header file and a macro invocation in each of the possible parent nodes.

3.4 Hierarchical Access

The SIDS document specifies for some entities scope and precedence rules within the database hierarchy. These include FlowEquationSet, ReferenceState, DataClass and DimensionalUnits. Basically it is stated, that if they do not exist at the current level (which may be, for example, a DataArray node representing CoordinateX), we have to look in the parent node, and if there is none as well, proceed to the next higher level. This is cumbersome to remember and handle in practice. Although the use of globally applicable data is probably most common, more complex scenarios with local overriding are explicitly allowed as well. Therefore it is quite convenient to have a function which returns for example the DataClass applicable to a given node, wherever this DataClass may be encountered the way up in the hierarchy. This is implemented in a way such that each node possibly having such flexible nodes has a pointer to its parent node, where it can ask the information if necessary. If there is no applicable node at all present in the database, an exception is thrown. In this case, the description of the data is incomplete anyway and the interpretation of such a database is up to the application (including the refusal of it or falling back to a default description).

3.5 Data Access

For the actual access to the large data arrays complete functionality is provided. For the storage of the data in a specific application there are plenty of possibilities, including sophisticated container libraries. It would be impossible to support them all. In order to do so anyway, the emphasis is on writing and reading to and from raw memory. Fortunately, nearly all usable libraries provide a way to get at a raw memory pointer, where the data is stored. If data access with raw memory pointers is fast and easy to use, the actual I/O calls are pretty painless. The only (currently anticipated) exception is support for one dimensional standard C++ vectors or containers with equivalent interfaces. There are simple wrapper templates which handle the conversion to and from the raw data pointers, which make the interface slightly more comfortable.

First there are simple methods to read the complete data into a given memory area. If necessary (and sensible), these provide data conversion between different data type sizes, like I4 $\leftrightarrow$ I8 or R4 $\leftrightarrow$ R8. Additionally, instead of writing into raw memory, it is possible to use an output iterator as destination, which enables the use of higher level containers for the data. Next, it is possible to read arbitrary slices, hyperplanes and subsections from the ADF file into memory, which is especially important when doing multigrid or domain decomposition. The strides and positions can be specified for the file and memory independently. This enables the application to handle for example the different array conventions in C++ and Fortran already at the I/O time, without having to transpose large data arrays in memory later on. Finally, we want to extend the data access capabilities to read the content in arbitrary dimensional units, which enables unit conversion on the fly. It will be possible to read the data in e. g. SI units, even if it is stored in the ADF file in pounds/feet/seconds/Rankine/degrees. The necessary factors are built into the library, which takes care of the exponents and automagically applies the correct conversion.

The writing capabilities mirror exactly the reading capabilities. The only exception is unit conversion on the fly. If there is enough demand, this could be added easily.

3.6 Database Handling

The CGNS++ library handles access to the underlying ADF database through a thin OO layer on top of the existing ADF library. This is not strictly necessary, but it simplifies many issues with memory management and gathers common operations through a simple interface. The OO layer is implemented with the same design decisions as written above including the implementation language used and handle semantics for the node class. Currently this layer is not strictly an ADF only layer but includes some functionality to help the CGNS++ implementation. With other database systems in mind it might be preferable to restrict the layer to the ADF functionality and add an additional one providing CGNS specific services. This could be done with only trivial changes to the CGNS++ library itself.

Since binary compatibility is an important point in using CGNS, other database layers are currently not an option. However, if there should be demand, for example to handle

parallel access, this OO layer could be set on top of any database management system, which provides the necessary services. These include the representation of the data as a directed graph, the notion of a name and (some kind of) a label for each node, and the possibility to store binary data of different types and dimensions in each node.

One possibility could be HDF5, which maps quite nicely to the required database structure. It provides parallel I/O and support for threaded applications. The future will show whether it provides all the features needed by the community, whether an enhanced ADF library will be more appropriate or something entirely different. We anticipate the porting effort to HDF5 (or a similar database layer) to be less than a week. This porting might be combined with the separation of the ADF++ layer itself and the CGNS utility services mentioned above.

4 Syntax

The syntax of the library is in this early stage not very important. The names of classes and methods are still subject to change, as they reflect personal coding styles and taste. Our current suggestion is a little bit based on the conventions in the SIDS and to a lesser extent in the existing midlevel library.

- Classes which represent SIDS nodes are named as in the SIDS, but without the trailing _t. Examples include Zone, GridCoordinates, FlowSolution. A notable exception is Base, which is named CGNSBase_t in the SIDS specification. Since the library lives in the CGNS namespace, the CGNS seemed to be redundant. The same naming scheme applies to the nodes represented as simple enumerations or trivial structs, like DataClass, GridLocation or IndexRange.

- Enumeration types are named in the same way as classes. In contrast, the enumeration values are written in capitals, using underscores for word separation, for example I_FACE_CENTER, TETRA_4 or NORMALIZED_BY_UNKNOWN_DIMENSIONAL. This is partly inconsistent with the SIDS, where nearly all enumeration values are in mixed case, but even there for example the ElementType_t values are named like ours. As we regard the enumeration values as quite different from types, we consider a different naming scheme as helpful.

- The iterator types necessary to access multiple child nodes are contained within the parent class and named as the child class with an _iterator_t suffix. For example, a Base contains a typedef (to the actual opaque iterator type) Zone_iterator_t, and zone contains a FlowSolution_iterator_t. The iterator type has the usual forward iterator semantics and supports copying, assignment, comparison, advancing and dereferencing.

- Access methods for optional children are named as follows. For a child of type XXX there is a method named hasXXX, which returns whether such a child node

exists, a method getXXX which returns a handle to the corresponding child, a method writeXXX, which possibly takes some required values to construct the child node and returns the new child, and finally deleteXXX to get rid of it. If we try to call getXXX or deleteXXX while there is no such thing, an exception is thrown. Nodes having possibly subnodes with hierarchical scope additionally have a method hasLocalXXX, to ask if the information is local to this node or looked up further up the hierarchy.

- For children with arbitrary cardinality there are several methods as well. If the type of the children is again XXX, we have getNumXXX, which returns the number of children of this type, and getXXX, which takes a name and returns the child with the given name of the correct type. Iteration is supported through the methods beginXXX and endXXX, which return an iterator to the first and one past the last one. To create children there is a function writeXXX, which takes the name and possibly required parameters and again returns the newly created child. deleteXXX again gets rid of the named child, where instead of a name an iterator is accepted as well to designate the one to be deleted.

- Modifications of the data in some of the nodes are currently not supported. Of course this decision is debatable. The reason is that many such in–place modifications render child node data (or even the child structure) invalid. For example, if the dimensions of a base are changed, nearly all the data below, especially the large arrays, are immediately invalid. There is no obvious cause why such a change should be allowed, and what consequences for the structure below it should have. If we need such a completely different base, we have to create a new one and fill it with the necessary details. However, for some smaller nodes, possibly with only trivial subnodes independent of their parent data, the possibility of change may be more desirable. More input from users on this topic would be welcome.

These are the user visible choices in the naming of types, values and functions. A discussion of this scheme is welcome, especially if supported by sound arguments.

5 Implementation

The initial implementation covers the major part of the SIDS document. At the moment, Base, Zone, FlowSolution and GridCoordinates are fully functional and the list expands continuously.

Many nodes share a major part of their properties. For example, nearly half of it may contain DimensionalUnits and DataClass subnodes, and nearly all may contain Descriptor subnodes and an Ordinal. To avoid duplication of all this commonality, inheritance is used heavily. However, this should be regarded as an implementation detail only. The only inheritance intentionally supported is the derivation of all hierarchical node classes from the base Node class, which provides basic functionality as Descriptor and Ordinal subnodes. Strictly speaking, Ordinal subnodes are not allowed to be attached to any node, but we

decided not to complicate the design by specifically disabling that feature for some nodes. Probably it does no harm if there is an Ordinal subnode even if it does not belong there.

Some intermediate classes provide functionality shared by several nodes. Dimensioned, for example, handles DimensionalUnits and DataClass subnodes, Stated additionally ReferenceState subnodes and Arrayed is responsible for DataArray subnodes. As said before, these are implementation details and are subject to change without notice. There does even exist an idiom which disables any direct access to the intermediate classes, but it is still an open question whether this trick is sanctioned by the C++ standard and therefore we cannot portably use it.

Most of the child nodes are either optional or even have arbitrary cardinality. To ease maintenance we create the function interfaces required for access to child nodes (like hasXXX, getXXX, getNumXXX, deleteXXX and so on) by using preprocessor macros. In the same way we use macros to provide implementations for these functions as far as possible. However, some functions require special behaviour and are therefore written by hand. Although preprocessor macros are usually regarded as an inferior mechanism, in this case they act as some kind of code generator and help to reduce the development and even more the debugging effort required. As development proceeds these macros are refined as required and the improved functionality is immediately available for all classes using the macros. Since there are 30 or so classes, each having very similar functionality (access to several child nodes of some types and possibly some data values), a change in the implementation would be quite tedious to apply to each one separately without those macros. Additionally there is alway the risk not to do it properly in one case or the other or simply forget one. As an additional bonus the source code is much more compact and simpler to read and understand.

6 Documentation

The documentation for the proposed library is already in development. A great help with this important part of the software is a tool to extract (most of) the documentation from the source code itself. This way documentation and implementation are kept close together, which aids in the tedious task of keeping them consistent, accurate and up-to-date. We use doxygen for this purpose, which is probably the most evolved tool for this job. It is able to create documentation in various formats, including HTML (for online viewing), RTF (Word), LaTeX(for printing), troff (man pages) and even XML (for post-processing). Support for this tool is excellent and the current version is quite stable and reliable. There is some room for improvement for our special requirements, but we hope to get them included in the near future. Currently the reference manual consists of 135 pages showing all accessible types, functions, classes and methods of the library and includes an index for simple retrieval.

Despite the importance of a reference manual, a user's guide and especially examples are sometimes even more helpful, particularly for beginners. In order to provide them some material, we include even at this early stage some examples, which show typical

application uses of the library. They are selected according to the examples set out in the existing User's Guide to CGNS and could be incorporated there easily. Of course they are included in the test suite, which is highly automated to support the development of the library. The test suite evolves together with the library to be sure to test all the functionality available.

7 Status and Outlook

As said before several times, the described functionality is not fully available yet. Nevertheless, the work already done expands daily and we expect a full implementation of all the SIDS by the end of this year. An experimental version with the most important features will be made available on the day before the telecon, together with some examples to show the use of the library as well as the simplified programming with a truly OO interface.

We have access to Linux, IRIX, Solaris, HP/UX, IBM/AIX, Cray T3E and NEC SX. As soon as the needed features are implemented, porting to these platforms will begin. As for now, a pretty decent C++ compiler is required. As development platform we use GNU C++ 3.0.2 on Linux (IA32), but the compilers from Comeau and KAI should be able to handle the code as well. In the porting process the requirements on the compiler probably will be relaxed. We will keep you informed about the progress.

A lively discussion about the aspects described in this document is most welcome. We kindly ask for input from experts in library design and C++, and application programmers, as well, as they are the intended users. Although it will most certainly prove impossible to satisfy everybody's needs and ideas, we can hopefully reach a sensible consensus.