Chapter 9

Input Output

In general most of the finite element applications have to communicate with pre and post processors ,except in some special cases in which the application generates its own input. This makes the input output (IO) an essential part of the application.

In this chapter, first different approaches in designing application's IO are discussed and a flexible and generic IO structure is presented. It follows a part dedicated to interpreter writing, which consists of small introduction to concepts and also brief explanation on the use of related tools and libraries. Next the use of Python as the interpreter is described and the reasons of using Python are explained. Finally a brief description of using boost python library is given.

9.1 Why an IO Module is Needed?

A typical finite element procedure consists of getting data from input sources, analyze it and send the result to an output. There are some applications which use the embedded IO structure. This means that they have their IO routines implemented in subroutines where an IO operation is needed. For example, element reads its properties directly from input file when they are needed and so on.

This approach is quite simple and easy to implement and in some cases eliminates some part of data storing overhead. However some cases exist in which an embedded IO approach introduces some difficulties in implementations or restrains the flexibility of the program.

In complex problems previous simple scheme changes to a repeating or multi input output scheme. These changes in strategy may introduce new IO statements, change some of them or invalidate some existing ones. This may result in many IO methods in different parts of the code which basically do the same things but with different objectives. Creating an IO module and use it in all of these statements help us to unify these efforts and simplify the maintenance of the code.

For a finite element program the possibility for connecting different programs is a great added value. The use of different pre and post processors or connecting different finite element applications are some typical examples. Connecting to each program means reading its output with given format, (in the case of pre processors) and generating their inputs also in their recognizable format (in the case of post-processors). It is obvious that each program may have an incompatible format with others. Using an embedded IO approach causes the program to be very rigid and difficult for extending to any new type of IO. In this approach a global revision of the code is needed to add any new interface and all the IO statements must be modified to include a new format. Placing

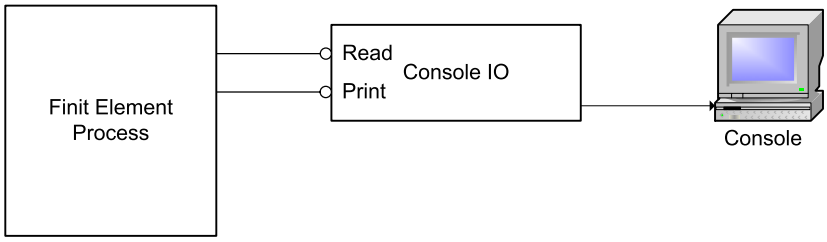

Figure 9.1: A Console IO interface

an IO module in the middle can solve this problem. Any request from inside or outside passed through IO, which is in charge of translate an outside format to inside one and vice versa. Now any new connection can be added by a new IO object.

Creating an IO module eases the team developing of an application. Without it each developer may put an IO statement in its part and cause conflicts in using files with others. In the same concept, easy file managing is another reason to use an IO module. IO module can open the file once, work with it and also close it at the end of work. If there is some problem with file opening it may report and try again. Finally handling multiple files is also simplified in this way.

As a conclusion it is useful to have a separate and robust IO module to handle the input and output tasks of the program for all different scheme with any format without problem. In this way flexibility guaranteed and extendibility can be achieved.

9.2 IO Features

Before starting with designing IO it is important to classify different aspects and features of IO for different programs. This helps to collect the IO features needed for a generic multi-disciplinary program.

9.2.1 IO Medium Type

People working in finite element analysis area are used to have files as input and output medium. This is correct for many cases but it is not always true. In general, an IO medium can be a file, some console stream, sockets for network communication or any other medium. Sometimes the difference in media comes from platforms and operating systems. For example opening a file and writing to it in Linux can be different from Windows. This make them virtually, more from implementation point of view, different media to interact.

This point of view to IO creates a set of open questions to be answered to before the designing phase. Does IO want to interact just with one type of medium or more? Is it supposed to be extendable in term of interacting with new media in future.

Our design can be depended on this questions. Working with just one type of media simplifies the IO interface and reduce implementation effort respect to the multiple media support. This difference can be large or small depending on the design, implementation and also to the type of supporting media and the way they are deferent. Let's make an example, considering a console IO as the first medium to interact. An interface to this medium just needs a `Read` and `Print` method to communicate. Assuming here that the console stream is always available. Figure 9.1 shows this simple interface.

Now let us add a file as a new medium. Here `Read` and `Write` methods provided for file accessing. Unfortunately a file is not always available to read or write as console is. We need to open it before any access and also close it at the end of procedure. This nature of file IO introduces two new methods to our interface, `Open` and `Close`. Figure 9.2 shows the interface for file IO.

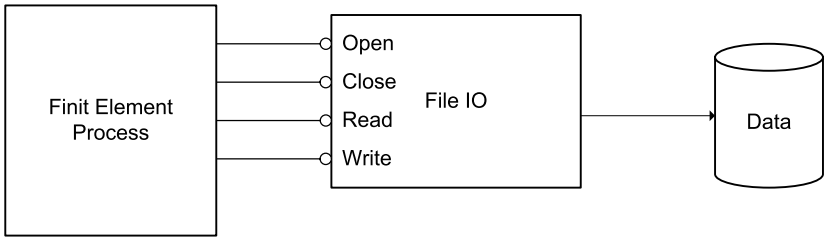

Figure 9.2: A file IO interface

Now, for having a multi-media IO a union of given interfaces is provided to interact with both media without problem.

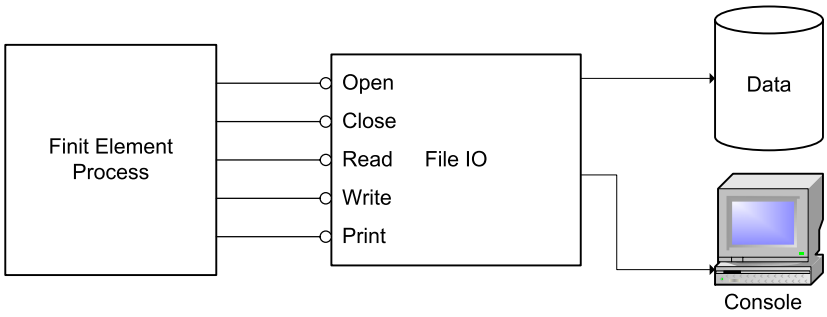

Figure 9.3: A multi-media interface

It is obvious that having the file interface, and unifying the `Print` and `Write` methods, no extra method is needed to handle the console IO. The intention of this example was to show the difference nature of each medium and not a serious design problem.

Handling each new medium may introduce the need for some new methods in interface, and making it extendable requires adding new layer to it. In other words, to do this another level of encapsulation is needed to separate different IO modules while keeping the established interface for all of them. Now, let's redesign our previous example and see how it can be organized to be extendible. Figure 9.4 shows the new structure.

Having above structure, any new medium can be supported using an IO interface via the new encapsulation level as shown in figure 9.5

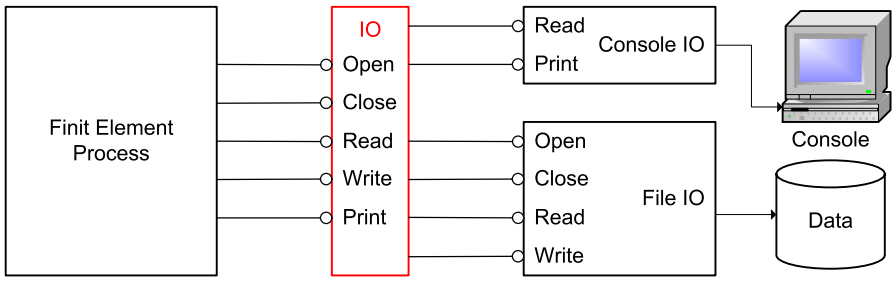

Figure 9.4: An Extendible IO interface

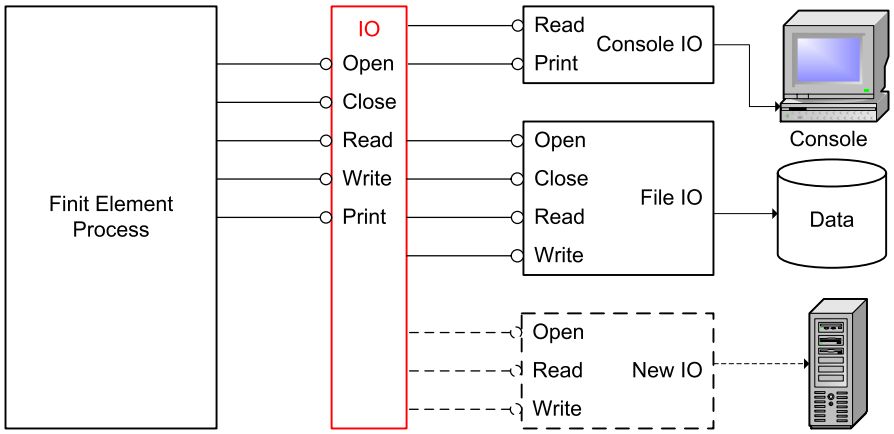

Figure 9.5: Extended IO

9.2.2 Single or Multi IO

A simple finite element procedure consists of getting data from an input source, analyze it and send the result to an output medium as shown in figure 9.6. A static structural analysis program is a good example of single IO.

This simple scheme also can be applied to some more complex problems in which there are various IO statements for certain points of program flow. Transient thermal problems, fluid dynamics and structural dynamics are typical examples of these type of solutions. In all of them program first reads the model data, then starts analyzing and meanwhile writes results for each time step. From the designing point of view this scheme is similar to the previous one as the IO statements are always the same. There are no changes due to the algorithm in reading or writing points.

In advanced problems there are some situations where IO statements change due to the nature of the algorithm. In other words the algorithm reads data or write them depending on the state of the problem. For example looking to some convergence criteria or displacement norms and not only in some certain points like the beginning of the analysis or the end of time steps. Optimization and interaction algorithms fall in this category. This situation implies more encapsulation of IO in time of designing. The reason is the extra flexibility needed to deal with these situations which

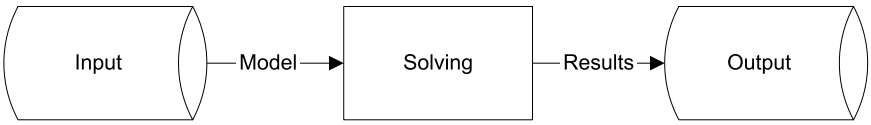

Figure 9.6: A single IO procedure

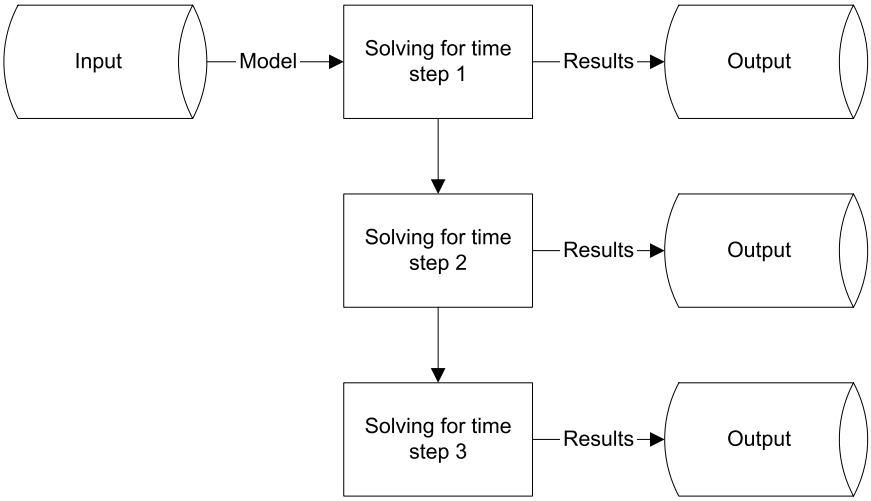

Figure 9.7: Repeating IO procedure

causes IO to be more independent.

9.2.3 Data Types and Concepts

The main task of IO is transferring data. A finite element program usually works with different types of data, like double, vector, matrix, etc. Also each data represents a concept like displacement, temperature, node's id and so on. Again there are decisions to be taken respect to data types and concepts to be supported. Which data types needed to be supported by IO? Is it needed to be extendible for new types? Does it supports new concepts?

These decisions are related to formats. Also they affect deeply the implementation of IO. From the designing point of view interfaces are affected by these decisions. Now, let us present some examples to explain these relations.

A simple thermal application is a good example of rigid IO. It has to read integers, doubles, vectors and matrices from input for concepts such as nodal information, elements information, properties like thermal conductivity and temperature and thermal flux as conditions and initial values. Output handles doubles and vectors needed to write temperature and fluxes. So in this case all the types are known and all the concepts are also defined as shown in table 9.1

It has to be mentioned that one can just handles doubles and treat all above types by their components as doubles and simplify more the IO. For the input part a column based format is well

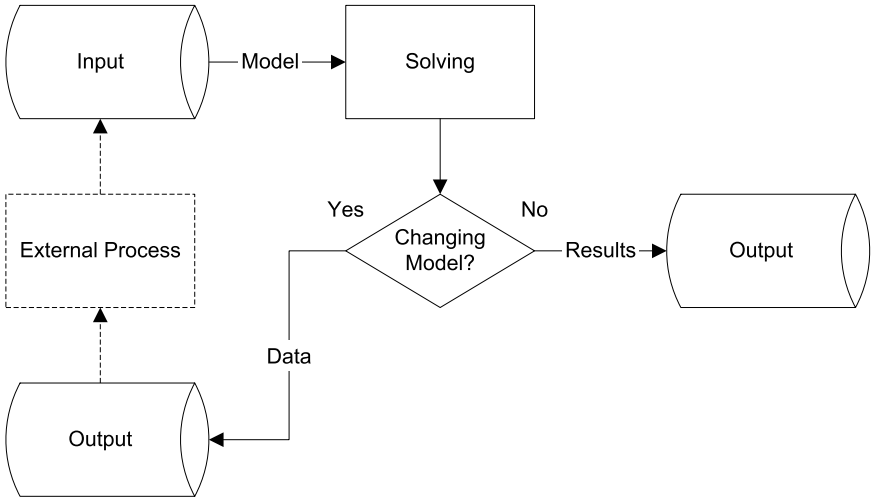

Figure 9.8: Complex IO procedure

Types	Concepts
int	Id, Connectivity
double	Temperature
vector	Thermal flow
Matrix	Thermal conductivity

Table 9.1: Thermal application types and concepts

suited and also makes implementation very simple.

```
# Format: NodeId X Y Z Temperature IsFixed
1 0.00 2.00 0.00 0.00 0
2 1.00 1.00 0.00 1.00 0
3 2.00 4.00 0.00 0.00 0
4 3.00 5.00 0.00 0.00 0
5 4.00 0.00 0.00 2.50 1
......
```

Implementation of IO for this example can be done easily and extremely efficiently by normal streaming or line by line scanning. Finally the interface can be defined very simply (although is very rigid) while all the types and concepts are known.

```
Read(NodesArray Nodes, ElementsArray Elements)

Write(vector Temperature, matrix ThermalFlows)
```

A more sophisticate interface can be designed to provide more control on IO sequences. for example

```
Read(NodesArray Nodes, ElementsArray Elements)
```

```
ReadNodes(NodesArray Nodes)

ReadElements(NodesArray Nodes, ElementsArray Elements)

Write(vector Temperature, matrix ThermalFlows)

WriteTemperature(vector Temperature)

WriteThermalFlow(matrix ThermalFlows)
```

Next example is a laplacian domain application. This refers to any problem govern by laplace equation like thermal, potential flow, low frequency electromagnetic, seepage problems or any combinations of them and solve it. In this example concepts are changing from one problem to other, or sometimes from one element to another, but the types of inputs are always the same. The input format gets more complex respect to previous example because concept definitions must be added to it. Here a tag mechanism helps to distinct the different concepts to be read.

```
# Format: NodeId X Y Z initialvalues IsFixed
1 0.00 2.00 0.00 VELOCITY_X 0.00 0
2 1.00 1.00 0.00 VELOCITY_X 1.00 1
3 2.00 4.00 0.00 TEMPERATURE 0.00 0
4 3.00 5.00 0.00 TEMPERATURE 0.00 0
5 4.00 0.00 0.00 TEMPERATURE 2.50 1
......
```

Flexibility in concepts introduces an extra cost in the implementation. A lookup table is needed to handle reading different variables via the names given and assign them internally. The lookup table to be used here can be a simple one with each variable name and their unique indices as shown in table 9.2.

Name	Index
TEMPERATURE	0
VELOCITY_X	1
VELOCITY_Y	2
THERMAL_FLOW	3
THERMAL_CONDUCTIVITY	4

Table 9.2: A sample part of lookup table

The interface design also must be changed due to these new features. First an initialize method is needed to pass the lookup table and setting the IO. Then `Write` method must be modified to get the variable to write as its argument. This is necessary because in this example the variable is not always Temperature and its Flow and the name of method cannot depend on it.

```
Initialize(Table LookupTable)

Read(NodesArray Nodes, ElementsArray Elements)

Write(string VariableName, double Value)
```

```
Write(string VariableName, vector Value)

Write(string VariableName, matrix Value)
```

Or the more complete version:

```
Initialize(Table LookupTable)

Read(NodesArray Nodes, ElementsArray Elements)

ReadNodes(NodesArray Nodes)

ReadElements(NodesArray Nodes, ElementsArray Elements)

Write(string VariableName, double Value)

Write(string VariableName, vector Value)

Write(string VariableName, matrix Value)
```

It can be seen that in this interface `Write` is overloaded for all different types handled by the output.

The final example is a generic IO module which the concepts and types both are extendible to new ones. In this case the type information and some basic operations must be included in the lookup table. This complicates even more the situation. To see this let us add a type extendibility to the previous example and use it for introducing a complex number to IO.

IO to handle new types needs to know how to perform some basic operations over them. For example, how to convert and string to them and viceversa, and also how to create them. So for each new types is necessary to give it some helper functions. These functions will help IO to manipulate the new types. These functions releasing IO from knowing anything about the new types. One can group all these helper functions in a TypeHandler class and pass it once to IO. Figure 9.9 shows this structure.

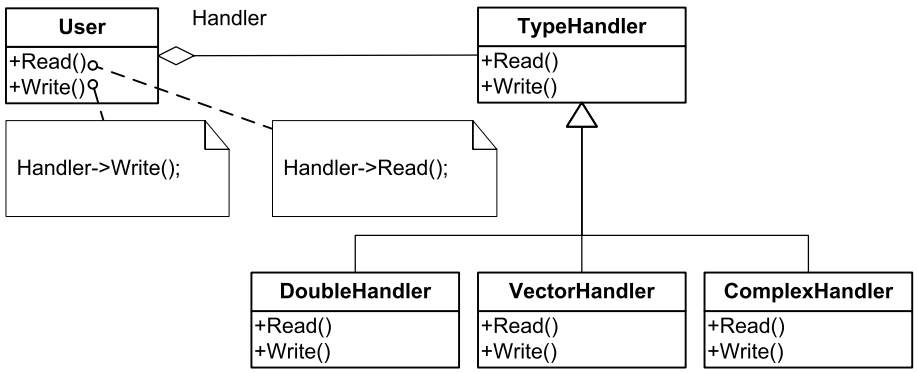

Figure 9.9: Using handlers to extend type supporting

The previous interface also changes due to this new mechanism. For example writing an interface can be changed to this new form:

```
Write(string VariableName, void* Value,
      TypeHandler& ThisTypeHandler)
```

Now, the `Write` method does not know about the type of the variable and just gets a void pointer and then handle it by a given type handler. This solution will work, but it is not type safe and leaves a possibility for serious errors. An example is to pass an integer value and a matrix type handler. This make `Write` method to write garbage and `Read` method may cause the system to crash by violating the memory. There is a more elegant way to handle this problem using templates. In this way the casting to void pointer is not necessary and the type of given data will be checked in compiling time.

```
template<class TTypeHandler>
void Write(string VariableName,
       typename TTypeHandler::DataType& Value,
       TTypeHandler const& ThisTypeHandler)
```

Going one step further, this TypeHandlers can be added to the lookup table for making this mechanism more automatic. In this manner, IO for each concept checks the lookup table as before and gets all the tools to handle this concept, even if is a new type of information.

It is important to mention that a variable base interface greatly helps in overcoming all these complexities and in designing a generic interface maintaining clarity and flexibility as explained in a later section.

9.2.4 Text and Binary Format

A text file is a sequence of characters stored as a file which depend on its content is human readable. Generally these files contains ASCII characters, where ASCII (American Standard Code for Information Interchange) is a character encoding based on English alphabet.

Writing strings to a text file is simple because there is no conversion needed. But for writing a number, first it has to be converted to a string format and then it can be written to the file. For example to write an integer variable which stores `1234567890` as its value, first it has to be converted from its binary form in memory which is `0100 1001 1001 0110 0000 0010 1101 0010` to its representative sting which is `"1234567890"` and then stored in file (Figure 9.10).

In time of reading again strings are read directly but numerical values must be converted from the string (Figure 9.11).

Binary format on the other hand, works with the binary value of each object and not with its representative string like a text file. In this manner a string or a number dump in the same way to the file. For example for the same above integer variable it is just enough to write the binary value 0100 1001 1001 0110 0000 0010 1101 0010 to the file directly (Figure 9.12).

Similarly for the reading just the sequence of bytes are read and put directly in the variable (Figure 9.13).

The advantage of the text format is its human readability. This makes it a good choice for small and academic applications because the aspect of input data can be verified and even changed manually and the output can be checked without any post-processor. For example to see if all results are zero. Disadvantage of this format is its latency time in read or write numbers due to the conversion time from and to its representative string. Another disadvantage is the large overhead in storing data size for numerical data. This comes from the fact that in most cases the representative string of a number occupies more memory than the number. Looking to our previous example, the integer `1234567890` occupies 4 bytes in memory like any other integer. Converting this to a string takes 10 bytes, one byte per character, without null at the end which is 150% overhead. However for small numbers, less than 4 digits, the representative string is smaller but

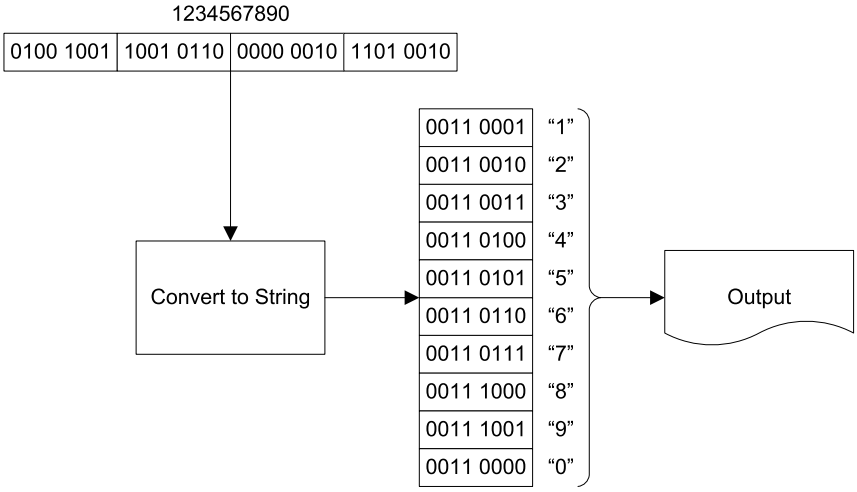

Figure 9.10: Writing a number to text format

in general finite element data usually consist of longer numerical data. These two problems make difficult the use of text format for very large problems in finite element applications.

The first advantage of using binary format is the direct read and write of data which increases the performance respect to text file format. Having no overhead in stored data respect to their real size is another advantage of using the binary format. These two advantages make this format a good choice for large problems data storing. Binary format is not human readable and using it as an input our output format complicates the debugging process and prevents users from manual checking of data for any obvious error, like having all coordinates equal to zero or so on. Finally, because there is no interest to open and read binary files, they can be compressed to occupy even less space than they normally need.

Using Ascii or binary format affect more the implementation details than its design. The interface and structure can be the same and the implementation details can be hiden via IO module. So using one or the other will not affect the implementation costs and the choice depends only on which is more suited to our needs.

9.2.5 Different Format Supporting

The previous section started with formatting concepts by describing two different category of formats, text and binary. In this section the concept is viewed from the format supporting point of view.

Format is the way data is represented and organized in a medium. Any variation in the data representation way or its organization creates a new format of data. As described in the previous section text and binary formats are different in the way they put their data. In the same manner, changing data organization in each one causes new format either binary or text.

Nowadays, a large number of data formats are defined and used by programs in different areas. Some of them are popular while many of them are just for specific programs. Also each different format has some advantages and disadvantages as seen before for the text and binary formats. All these aspects makes it necessary to take some decisions before starting the implementation. Using an existing format or creating a new one? Supporting an additional format or one is just enough?

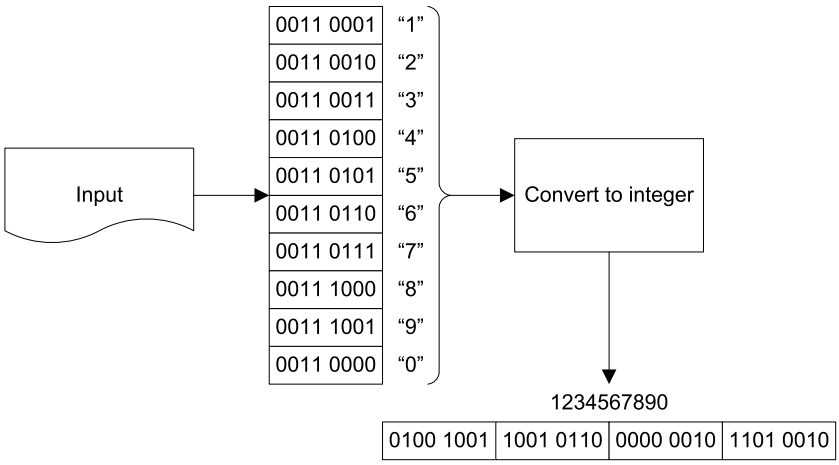

Figure 9.11: Reading a number from text format

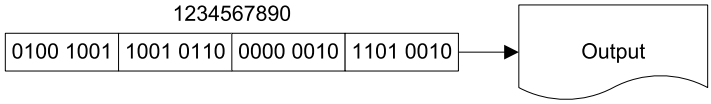

Figure 9.12: Writing a number as binary format

Can IO be extendible to support a new format in the future or not?

Most these decisions affect the design and implementation in the same way as for supporting media. The first choice is to support only one format. Choosing this option makes both design and implementation very simple. The interface for this case can be very simple without any argument or method to specify the format. Implementation also is simple while there are neither switch and case to handle a given format and there is not a duplicated method for different formats. Figure 9.14 shows an example of interface for an IO which supports only text formats.

It is important to mentioned that the simplicity of implementation mentioned above is respect to global design aspects and not the cost of handling each specific format which may cost a complete parser to be implemented.

To take advantage of more than one format, the previous design must be modified to handle each format separately. Interface must provide some way for users to set their format. A simple approach for small amount of formats to handle is to create separate methods. This leads a very clear interface and in some cases eases the implementation. This design from one side keeps each format support implementation separate and hence it increases the maintainability of the code. However it makes it more rigid and static. Figure 9.15 shows an example of this interface for a dual format supporting. The clarity and readability can be seen, while it is obvious that for supporting some more formats this approach is not suitable.

There is a more elegant approach which just passes a flag through interface to identify which format is being used. In this manner the interface is clear and adding a new format is easy while using it via hierarchy is not very helpful. Figure 9.16 shows the previous example with a new

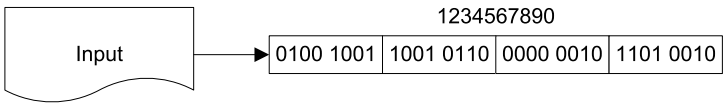

Figure 9.13: Reading a number from Binary format

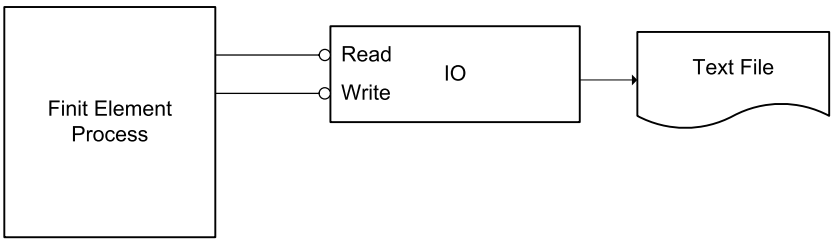

Figure 9.14: IO single format example

interface. Now the interface is simple and more dynamic than before.

Let us go now one step further and give flexibility to add new formats without changing previous ones. This can be done with the same methodology as before for the extendibility of media. A new layer of encapsulation takes place in the IO structure to unify the interface and the support for each format is encapsulated separately. This design provides a flexible structure which adding new format will not change any other part of the code and grantees the extendibility. This design imposes that all supported formats must use a unique interface. So to complete this structure a generic interface needs to handle all formats in the same way without critical restrictions. Reusing previous examples for applying the new design, results in a more flexible structure. Figure 9.17 shows this new structure.

Now supporting a new format implies adding another part without any modification of other the IO parts. Figure 9.18 shows the extended version.

9.2.6 Data IO and Process IO

It is usual for single purpose finite element applications to get data from the input, process it and write the results on an output. These applications always use their own implemented algorithm to solve the problem. They also give some options to the user to modify some aspects of algorithms via input data but not the algorithm itself. A typical input for these programs contains nodes, elements, conditions, properties and some algorithm parameter like convergence criteria, maximum number of iterations and so on. Some other codes also give the possibility to choose some parts of the algorithm by some given options, like changing the solver, static or dynamic strategies, etc.

For more complex problems and for multi-disciplinary programs the previous approach is too rigid to be applicable to all problems. For example for a thermo-mechanical application sometimes the mechanical solution is wanted not from the beginning, but from a certain time. So a mechanism is necessary to start solving the mechanical part from a certain time to avoid unnecessary calculation overhead. Another example is a fluid structure interaction problem, which may cause the program to change its global strategy depending on the type of structure and fluid interacting

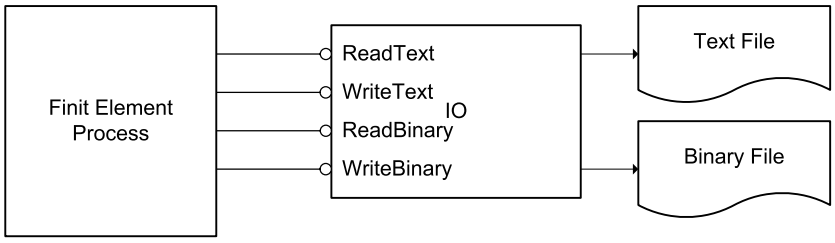

Figure 9.15: IO dual format example with specialized methods

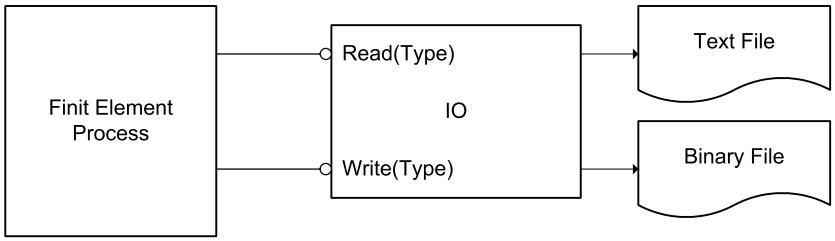

Figure 9.16: IO dual format example with passing type as argument

with each other. All these makes developers to implement their codes to get not only data from users but also the process necessary for the solution.

The first approach is to create a semi-language like input data which controls the order of execution of code blocks and also gives option to add processes in specific statements of algorithms. In this manner the user can define the global layout of the program flow by choosing when to read, write, solve, etc. via the input file. Also it can put some extra process in certain places like at the beginning of each time step and so on. Limiting the flexibility of modifying a program's algorithm imposed by this approach comes from two facts. First the lack of a good interpreter in the input part makes it impossible to introduce a whole algorithm. Second is the internal design of the code which is not well split and can limit the way one can put different algorithms together in order to create a new algorithm.

Usually codes written in fortran use previous approaches to handle different algorithms due to their limited facilities to write an interpreter.

Another approach is to implement a high level language interpreter as input in order to manage the global flow of the program. This gives a great flexibility to the code and new algorithms can be easily implemented. A working example of this approach is the Object Oriented Finite Elements method Led by Interactive Executor (OOFELIE) [77] developed by Cardona, *et al.* [25, 53, 54] which has its own command interpreter to deal with different algorithms in multi-disciplinary problems.The advantage is the flexibility in input language syntax. One can define a very minimal language which adopts well to its needs. This can give a clear and powerful input format for certain type of applications. The backward of this approach is the implementation cost. Writing a good interpreter is hard work and maintaining it is even worse. C++ programmer are more motivated to write their own interpreters because there are a lot of available libraries which can help them

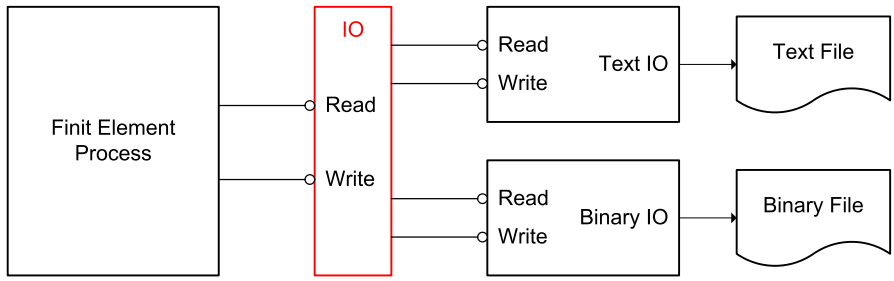

Figure 9.17: Multi format extendible IO example

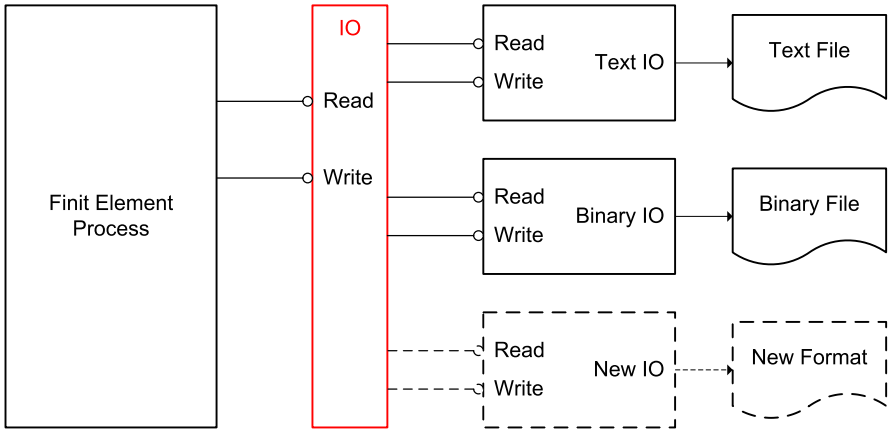

Figure 9.18: Extended IO to a new format

[67, 6, 2, 5]. Even though using libraries and compilers creating tools, implementing an interpreter for a finite element program can introduce a great overhead to its cost. Also many finite element programmer do not have compiler writing knowledge in their background and learning or hiring some specialist causes again extra cost for the program.

The last option is to use an already implemented interpreter and not to write a new one. The first advantage of this approach is omitting cost and time overhead needs to develop a new interpreter. Another good news is the freedom from knowing how to write a compiler. Also an existing language has its documentations like tutorials, reference manuals and so on. This reduces the program IO documentations just to documenting the extended part of the language and its specific commands. However this approach has its backwards too. First is the effort to connect the program to the interpreter can be significant depending on the programming language of the FE code and the interpreter chosen. Second disadvantage is the generic concept of popular languages which may prevent them from fitting well into finite element concepts. Another backward is their overhead in memory and performance. After all there are good news again. A good selection of language and its interpreter can reduce or effectively remove these disadvantages. There are many industrial applications using this approach, for example ABAQUS [10, 11] uses Python [89, 63] or

ANSYS [1] uses TCL/TK.

9.2.7 Robust Save and Load

Programs manipulating any kind of document usually have to provide some mechanism to save and load again their documents. In computer science an object which keep its state after the execution of program is referred as a *persistence* object. Also persistence is known as the program ability to store and retrieve its data. Finite element applications usually do not need to save their model in the same way they read it. On the contrary, they just read the model and write the results. Pre and post processors are responsible to store the model and processed results for future use. Still there are some situations that save and load features are needed for a finite element solver. For example to analyze big problems is sometimes useful to stop the process and resume it sometimes later due to some resources schedule. A typical reason is working with computers that are available at night. This makes necessary for the code to store its state before ending the process and retrieve it next time it is executed to be started from the last state.

There are different ways to implement persistency. Two typical approaches are object persistency and global or database persistency.

Object persistency follows the idea that each object knows how to store and retrieve itself using certain interface. This approach is also known as *serialization*. This comes from the fact that data is stored and retrieved serially in this mechanism. To implemented a serialization mechanism first an interface is introduced to all objects that provide a unified form of save and load their state. For example including a `Serialize` method for each of them. Then each object implements its manner of storing and retrieving its data. For objects consisting of simple data this can be done simply by storing the data sequentially and read it back also sequentially in the same order. Save and load for a complex object of some other object and also some simple data, consist of putting simple data as they are and calling `Serialize` method of all member objects sequentially and then load them again in the same order also by calling the members `Serialize` methods. The important point is calling the `Serialize` method of each component, which cause each part to serialize its data and call again `Serialize` of its components. In this way starting from top of the data pyramid and start to save or load, system will traverse to the bottom automatically and cover all data from top to bottom.

An example will this method. Let us implement a serialization mechanism for a simple `Mesh` class containing `Node`s and `Element`s and each `Element` has its `Properties` as shown in Figure 9.19.

The first step is to create the necessary interface by adding a `Serialize` method to all objects. Figure 9.20 shows this interface.

Now let us implement the `Serialize` methods. While the `Mesh` does not have any simple data by itself it just call the `Serialize` methods of all `Node`s and `Element`s and also some additional information for retrieving itself.

```
Serialize(File, State) {
  if(State == IsRead){
    File >> NumberOfNodes;
    File >> NumberOfElements;
    CreateNodesArray(NumberOfNodes);
    CreateElementsArray(NumberOfElements);
  }
  else{
    File << NumberOfNodes;
    File << NumberOfElements;
```

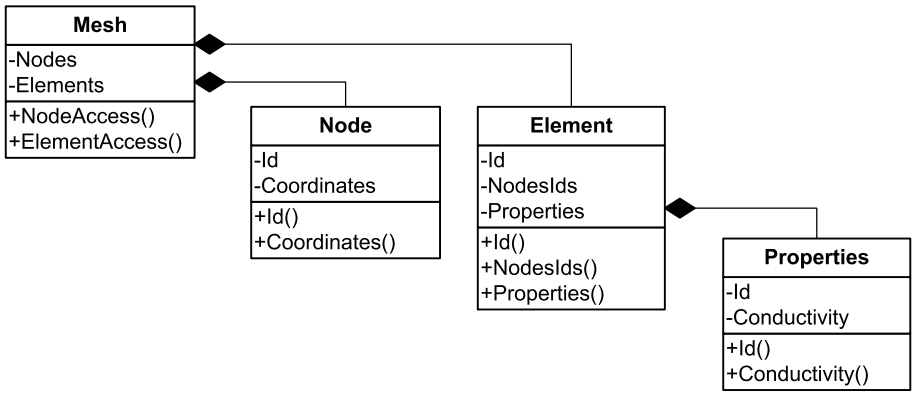

Figure 9.19: A simple `Mesh` class and its components

```
  }
  for(int i = 0 ; i < NumbreOfNodes ; i++)
    NodesArray[i].Serialize(File, State);
  for(int i = 0 ; i < NumbreOfElements ; i++)
    ElementsArray[i].Serialize(File, State);
}
```

`Node`'s `Serialize` method is very simple and just transfer its two simple data, `Id` and `Coordinates`.

```
Serialize(File, State) {
  if(State == IsRead){
    File >> Id;
    File >> Coordinates[0] >> Coordinates[1] >> Coordinates[2];
  }
  else{
```

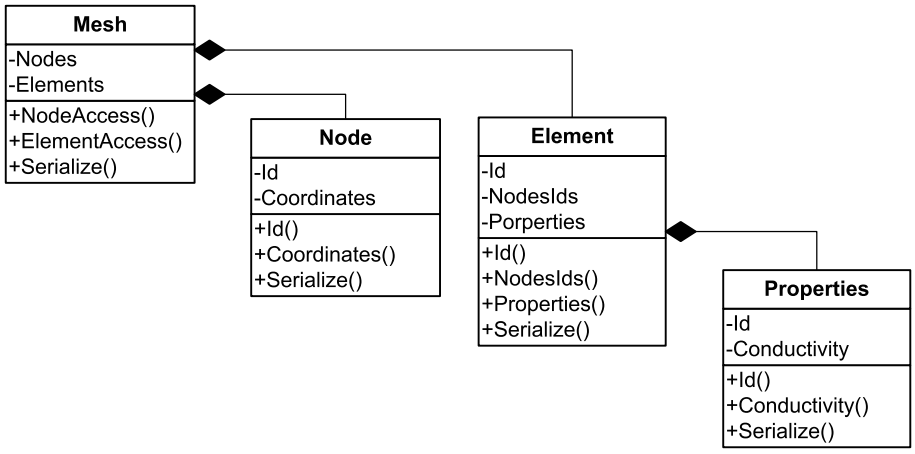

Figure 9.20: Adding the `Serialize` method to create necessary interface

```
    File << Id;
    File << Coordinates[0] << Coordinates[1] << Coordinates[2];
  }
}
```

But `Element` has simple data like its `Id` and also a component which is the `Properties`. To handle them the `Serialize` method will be something like:

```
Serialize(File, State) {
  if(State == IsRead){
    File >> Id;
    File >> NodesIds[0] >> NodesIds[1] >> NodesIds[2];
  }
  else{
    File << Id;
    File << NodesIds[0] << NodesIds[1] << NodesIds[2];
  }
   Properties.Serialize(File, State);
}
```

And finally the `Serialize` method of `Properties` finishes by adding its data to the file.

```
Serialize(File, State) {
  if(State == IsRead){
    File >> Id;
    File >> Conductivity;
  }
  else{
    File << Id;
    File << Conductivity;
  }
}
```

Figure 9.21 Shows the global scheme of the implemented example.

Now to save the `Mesh` we just need to call its `Serialize` method with a writing flag:

```
mesh.Serialize(File, IsWrite);
```

This causes all `Nodes` and `Elements` write themselves in the output file and also each `Element` causes its `Properties` to write itself too. Figure 9.22 shows the sequence of data written in the output file.

To retrieve data from the file its just necessary to call again the `Serialize` method of the `Mesh` but this time with the reading flag:

```
mesh.Serialize(File, IsRead);
```

This method takes first `Mesh` information and then begin to load `Nodes` and after that `Elements`, as shown in the above pseudo code. Looking to the written data's sequence shown in Figure 9.22, it can be easily verified keeping the same order for writing and reading, results in the same structure.

In this simple example the `Mesh` creates the object by knowing there are `Node` or `Element`. In practice there are many cases which the type of object is not known before reading. For example a real mesh can have various type of elements. So how can we create them before calling there `Serialize` method? There are two ways to solve this problem. The first way is using C++ *Run Time Type Identification (RTTI)* to store the object name and create an object using this name. The second way is to name each object manually and store it as a reference name in order to create

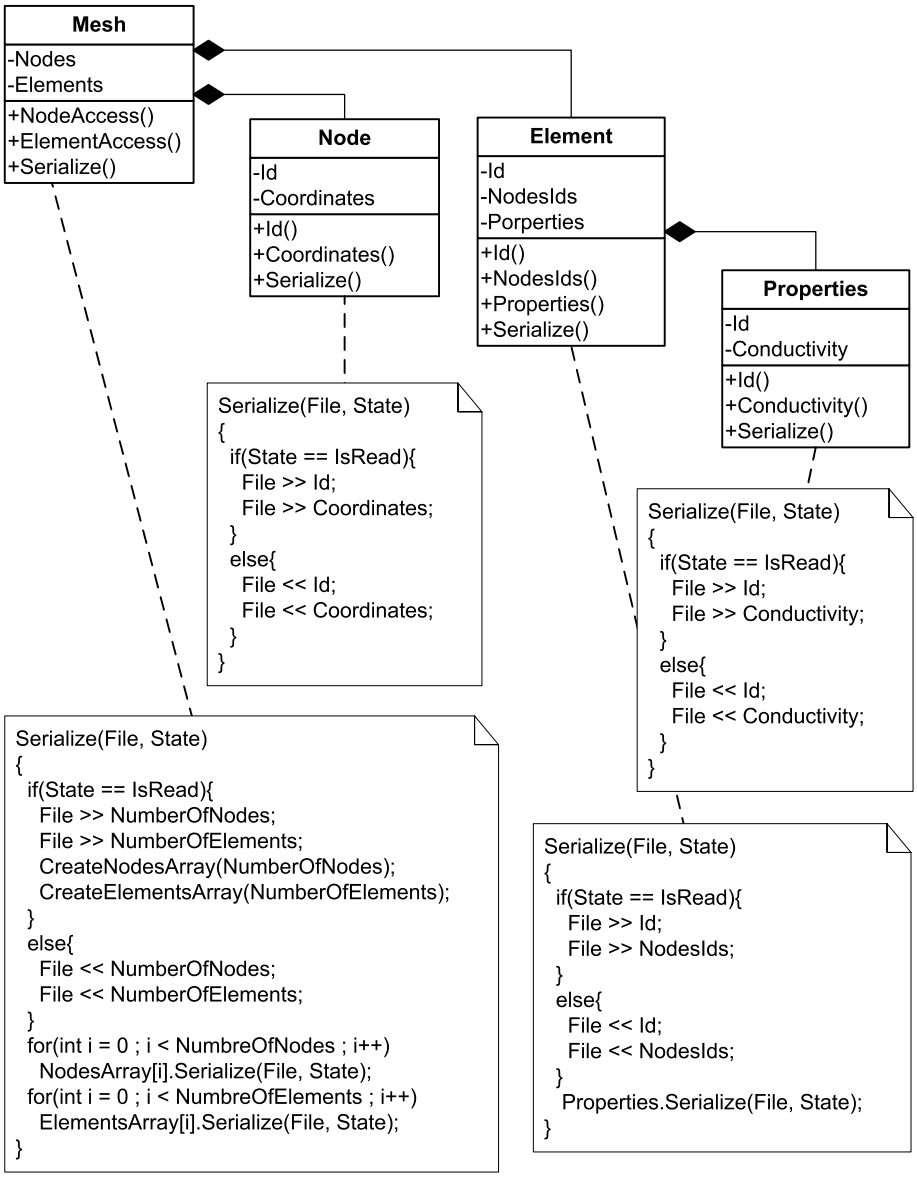

Figure 9.21: Pseudo implementation of `Serialize` methods.

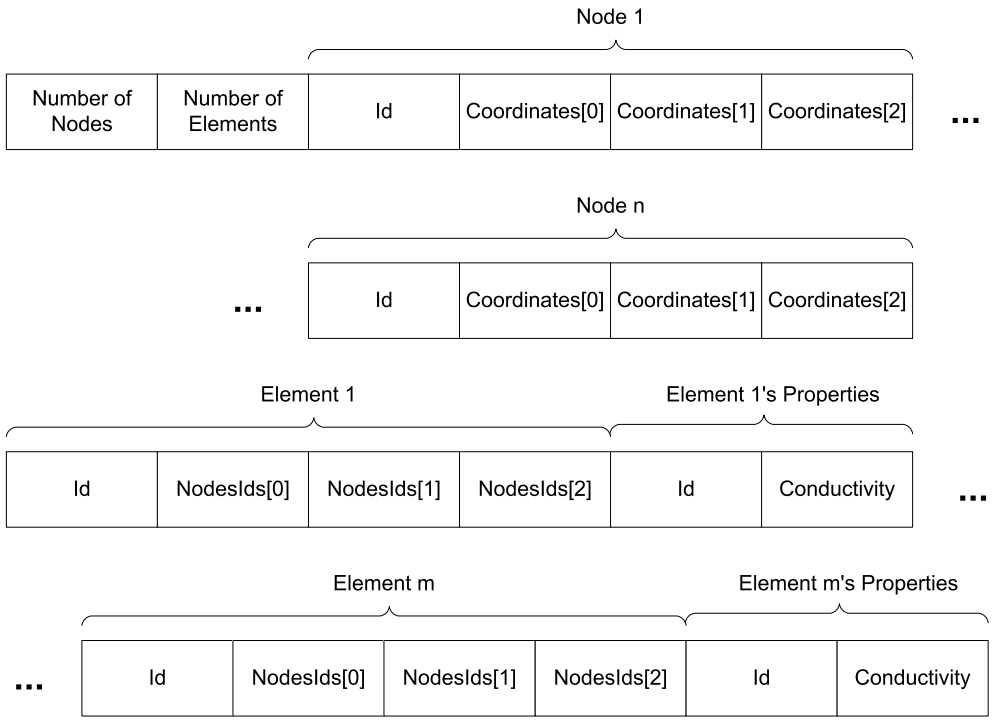

Figure 9.22: Data sequence in output file resulting from the `Mesh` serialization example

it again. The first method is very robust and easy to use. It is applicable to any type of classes and is very extendible. The second one is less robust because names are given manually and any coincidence may cause a problem but it is more tunable because different names can be given to different states of a class. Both approaches need a registry database which for any given object name indicates the registered way of create it.

Now let us take a look at pointer management and serializing pointers. Considering an element class which keeps pointers to its nodes instead of their indices. Applying our standard method will save the memory address of nodes stored in pointers. While the objects after restoring can be in other position of memory, this stored address is completely useless to recover the node. A more careful implementation can save a copy of the pointed node, but this results duplicated nodes and still it is not what we want. The solution is to give a unique index to each object and store and retrieve pointed objects by these indices. In our finite element case nodes and elements' indices fit well for this purpose.

Serialization is very much dependent on the internal structure of the program. Any time a component of objects changes the previous stored data will be invalidated. Using versions and retrieve data by their version solves this problem and it helps to use it in practice.

Serialization in general is very extendible and well suited as a generic solution in saving and loading tasks. There are libraries which provides a good implementation of serializing libraries and can be used in other codes to give persistency with minimal effort. XParam [9] and Boost Serialization library [4] are examples of these libraries. Microsoft Foundation Class (MFC) library also includes a serialization mechanism but using it introduces a platform dependency on the code and makes it not portable to other platforms.

Database persistency on the other hand, comes from the idea that a database must know how to save and load its data. This approach is less extendible but it works well for working with databases. Any new object which can be stored in a given database even by its component can be stored and restored via existing IO without modification. This is the great advantage of this approach.

9.2.8 Generic Multi-Disciplinary IO

After a short introduction to each different features in IO, now let us see which is necessary and which can be useful for a generic multi-disciplinary IO.

Supported Media Supported media can be considered as an optional feature. However in order to add portability to our code or to make it a usable library in some cases it can become necessary. Treating media like formats can give an effective solution to maintain the extendibility to new media without putting extra effort on it.

Single or Multi IO It is obvious that at the time of designing a generic IO there is no clue about how it will be used. This increases the amount of encapsulation in our design which is also the key point to make it usable in multi IO schemes. Also many multi-disciplinary algorithms are working with multi IO schemes which makes it an essential part of our design.

Data Types and Concepts Making IO extendible to new concepts is essential for a generic library and also for a multi-disciplinary one. Extendibility to new types is essential for generic library to guarantee the ease of using it.

Different Format Supporting Though supporting different formats is not a necessary feature, it is a great added value for any generic library. To develop a generic library having an extendible interface to new formats increases the use of the generated library because any new user can make it work with its already existing format. All these makes format extendibility a part of our IO features to be implemented.

Text or Binary Format Selecting one format depends on the use of the program and not on the multi-disciplinary nature of it. Deciding to have a extendible multi-format IO opens the way to implement text or binary formats when they are necessary.

Data or Process IO Process IO is one of the essential features for a multi-disciplinary IO. The variety of algorithms in solving different problems is source of constant changes in the code without process IO. Giving the user the ability to change the algorithms or introduce new algorithms without modifying the code dramatically increases the extendibility and reusability of the library.

Serialization As mentioned earlier, serialization can help to store the state of process to resume it later. This is a useful feature but in many cases is not necessary. The intention is to use an existing library (like Boost Serialization) without altering the IO design respect to it. After all the database approach is always available to add save and load features to the program.

9.3 Designing the IO

In previous section a quick review of differen IO features was presented. Also a brief description of their influence in design and implementation was given and finally a list of the selected features was prepared. The next step is to design an IO module considering all those features.

9.3.1 Multi Formats Support

Let us take the multi formats supporting feature to start the design process. The first goal is to make an extendible multi format IO. As mentioned before a proper design is to establish an interface and encapsulate each format support in a separate class. The strategy pattern is what we are looking for. As mentioned in section 3.4.1, this pattern defines a family of algorithms and make them interchangeable via encapsulating each one separately, which is what we are looking for. Applying this pattern to our problem results in the IO structure shown in Figure 9.23.

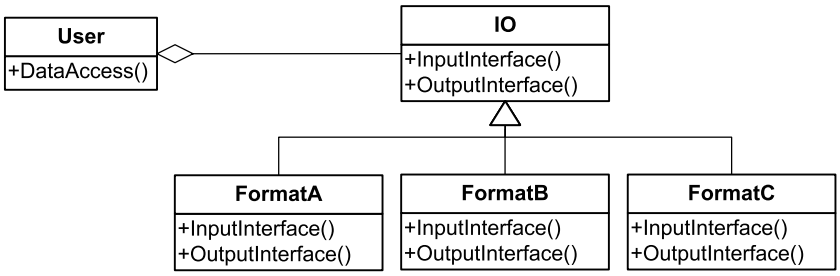

Figure 9.23: Multi format IO structure

9.3.2 Multi Media Support

Multi media support can be added as the second component for designing IO. The first approach is to assume that each medium can be used to store any format without dependency. In this case the bridge pattern can be used to decouple these two concepts and cut their dependencies. Introducing this pattern to the previous design results in the structure shown in Figure 9.24.

The other approach is treating media like formats and put them in the same structure. This approach simplifies very much the structure and its implementation. Beside this advantage it makes strong relation between formats and media and having the same format in two media may duplicate the code. Figure 9.25 shows this simple modification.

This seems to be too rigid but it can be improved to be more flexible with some minor changes. Also the independency of medium and format interfaces is guaranteed by the bridge pattern is not highly necessary in our case and comes at the cost of complicating the structure. In practice big variety of finite element formats are used to create files, and other types of media are used to work with other class of formats. So a simplified approach is to work with the second simpler structure and enrich it via templates or multiple hierarchy as shown in figure 9.26.

9.3.3 Multi IO support

This structure by itself provides multi IO support. In any places where an IO statement is needed, one can create any IO object and perform its IO without duplicating code. Also in time of implementation providing some controls can reduce conflicts. Only exception, which is related to the layering nature of Kratos, is that IO statements cannot be in `Node`, `Element` or `Conditions` because it violates the layer order as it requires to place upper layer objects in lower layer ones.

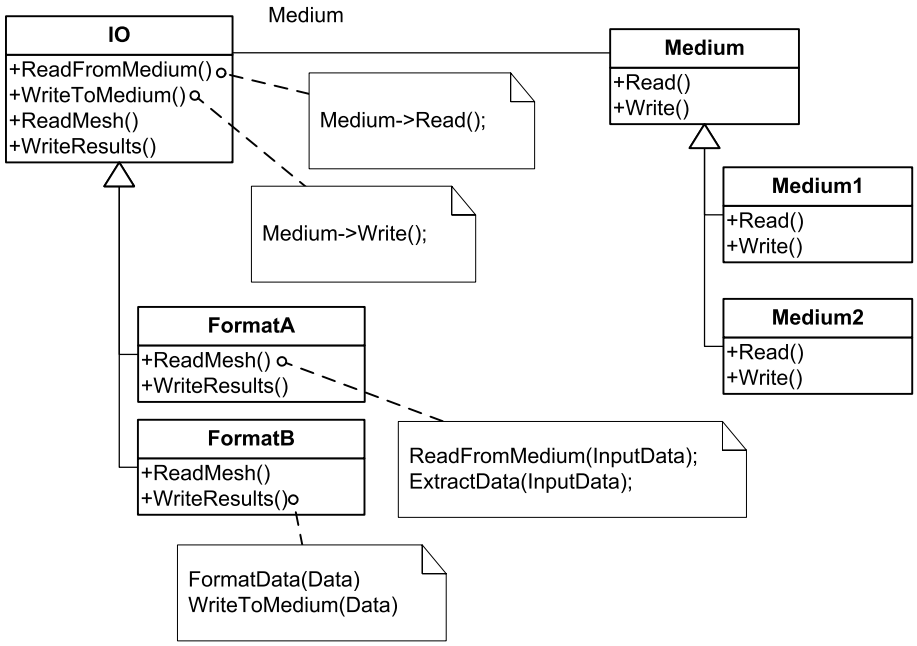

Figure 9.24: Multi format and multi media IO structure

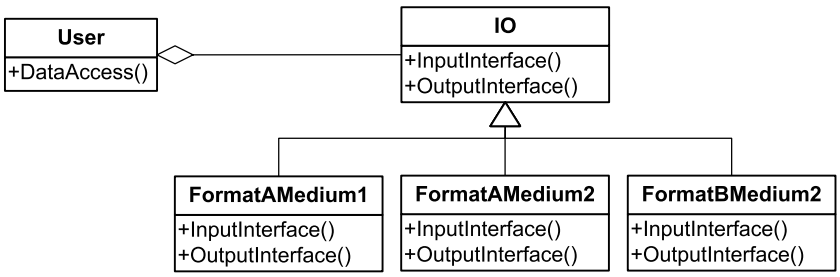

Figure 9.25: Multi format and Medium IO structure

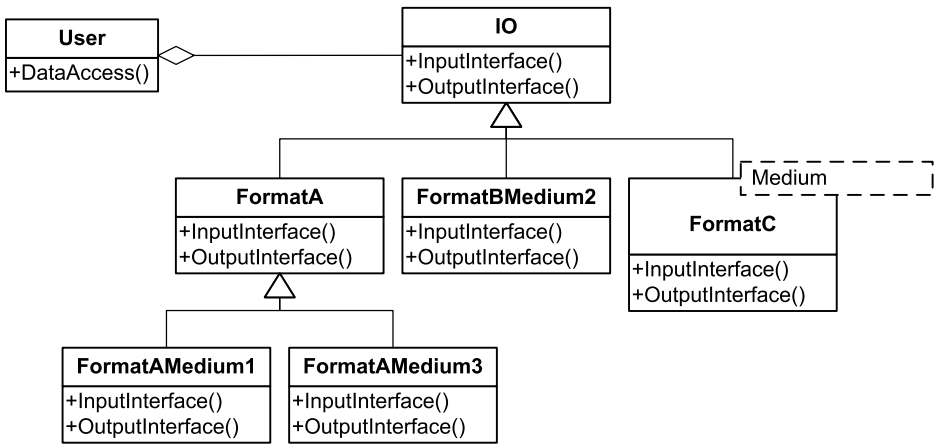

Figure 9.26: Extended Multi format and Medium IO structure

9.3.4 Multi Types and Concepts

The next feature to add is the extendibility to new types and concepts. As mentioned before this is an essential feature for a generic multi-disciplinary library. Here the variable base interface comes handy and it is used to generalize the IO.

The first step is providing IO extendibility to new concepts. As described earlier this can be done by introducing a lookup table which relates the concept names and their internal handlers. A simple list of `Variable`s is enough for IO to take as its lookup table. Each `Variable` knows its name and also its reference number. In time of reading IO reads a tag and searches in the list for the `Variable` whose name coincides with the tag. Then use the variable to store the tagged value in the data structure. For example, when IO reads a `"TEMPERATURE=418.651"` statement from input, takes the `"TEMPERATURE"` tag and searches in the list to find the `TEMPERATURE` variable. Having this variable is enough to use the variable base interface of the data structure to store the value in it. For writing results there is no need to search in the table. IO can use the variable to get its value in the database and use its name as the tag. Here is an example of `WriteNodalResults` method.

```
template<class TDataType>
 void WriteNodalResults(Variable<TDataType> const& rVariable,
                        NodesContainerType& rNodes,
                        std::size_t SolutionStepNumber)
 {
    for (NodesContainerType::iterator i_node = rNodes.begin();
         i_node != rNodes.end() ; ++i_node)
                Output << rVariable.Name()
                       << "="
                       << i_node->GetSolutionStepValue(rVariable,
                                    SolutionStepNumber) )
                       << std::endl;
 }
```

Now the IO is extendible to new concepts, but what about new `Element`s and `Condition`s? Each new `Element` or `Condition` also is a new type which implies that the IO does not know how

to create it. Again the idea of lookup tables comes handy. Here we need a table for each `Element` or `Condition` and their relative factory method. Prototype pattern helps to manage this situation in a generic way. Here we reuse each object as its prototype by adding a `Clone` method to it. Figure 9.27 shows the `Element` prototyping mechanism used by IO.

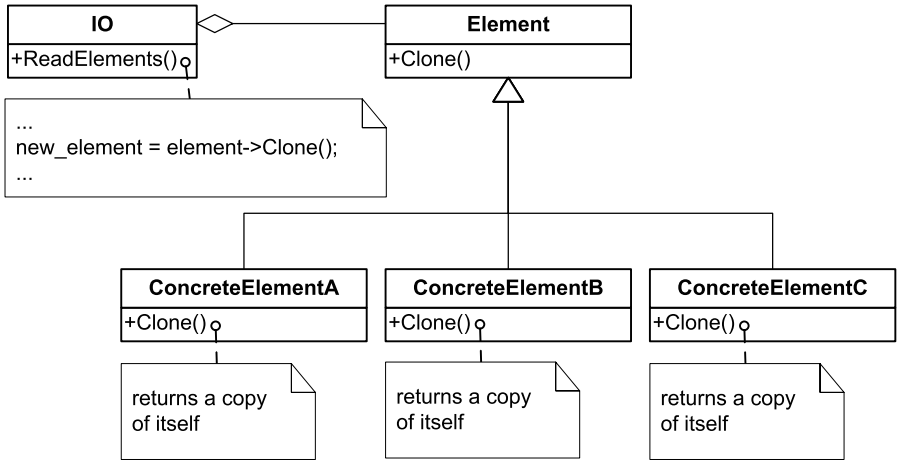

Figure 9.27: Using prototype pattern in IO for creating `Element`s.

IO uses lookup table to find the object prototype for any component name. This table consists of representative names and their corresponding prototype. Encapsulating this table introduces the new `KratosComponents` class to our structure. `KratosComponents` class encapsulates a lookup table for a family of classes in a generic way. Prototypes must be added to this table by unique names to be accessible by IO. These names can be created automatically using C++ RTTI or given manually for each component. In this design the manual approach is chosen, so shorter and more clear names can be given to each component and also there is a flexibility to give different names to different states of an object and create them via different prototypes. For example having, `TriangularThermalElement` and `QuadrilateralThermalElement` both as different instances of `2DThermalElement` initializing with a Triangle or a Quadrilateral.

This structure allows us to create any registered object just by knowing its representative name. But sometimes it is useful to know the family which an object belongs to. For example at the time of reading `Element`s there is no need to search in `Variable`s and `Condition`s and put all of them together can slow down the parsing process unnecessarily. Dividing the lookup table to three family of classes: `Variable`s, `Element`s and `Condition`s helps to distinguish them in the time of search. Doing this also eliminates unnecessary type casting and makes the implementation easier and clearer. Figure 9.28 shows the resulting structure.

Unfortunately storing `Element`s and `Condition`s using their prototyped name requires each `Element` and `Condition` to store their names. This introduces an unnecessary overhead and it is better to use the RTTI and not manual prototyping.

Conflict arises between accepting new types and multi format supporting. The restriction comes from format which not only have to support the type but also to indicate the type to IO. In fact `Element`s and `Condition`s provide a uniform initializing interface which is not possible for different data values. So `Variable` can create data or clone it from some existing sources but the IO to set the value of a given `Variable` needs its type information. However in finite element

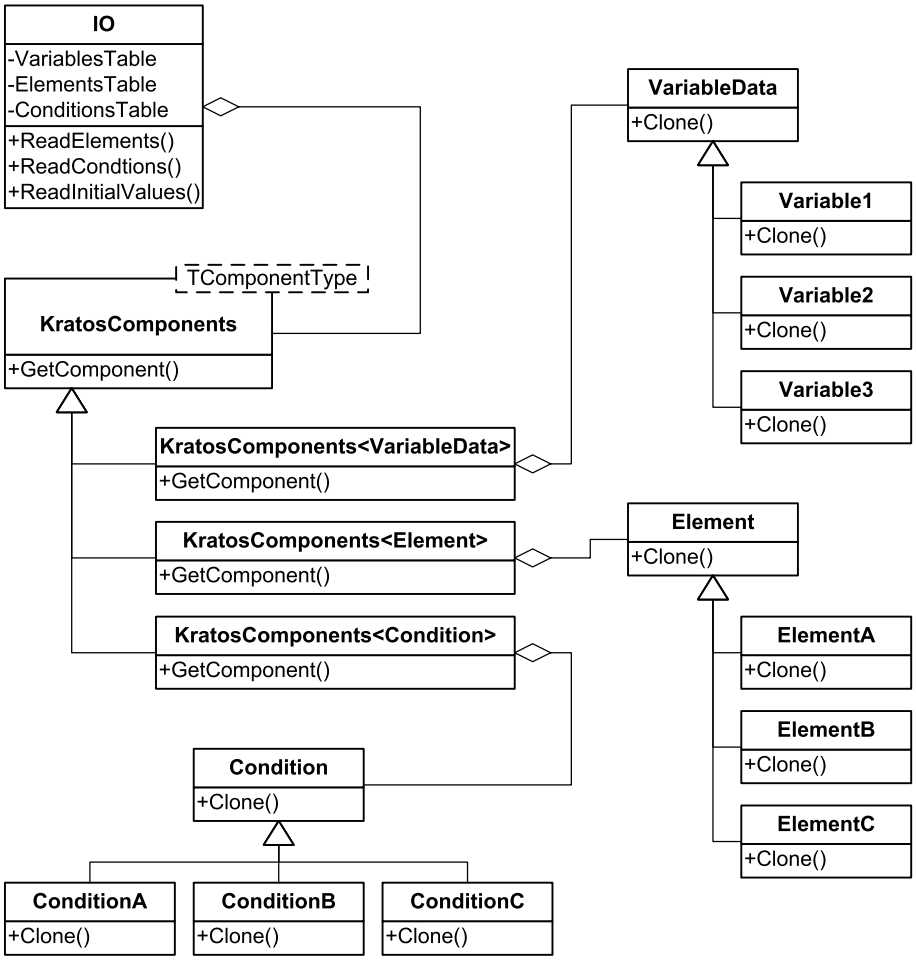

Figure 9.28: Using `KratosComponents` in IO

formats usually few number of types are available and one can handle them separately at the time of implementing the input.

9.3.5 Serialization Support

As mentioned before this feature gives new abilities to the program but it is not a very essential part of IO in our design. Serialization can be implemented separately and parallel to our design. So it is possible to use an external serialization library without changing our design as shown in figure 9.29.

9.3.6 Process IO

Understanding a given process by IO needs an interpreter which constructs dynamically the algorithm statements and executes them. The interpreter concept and its design will be described

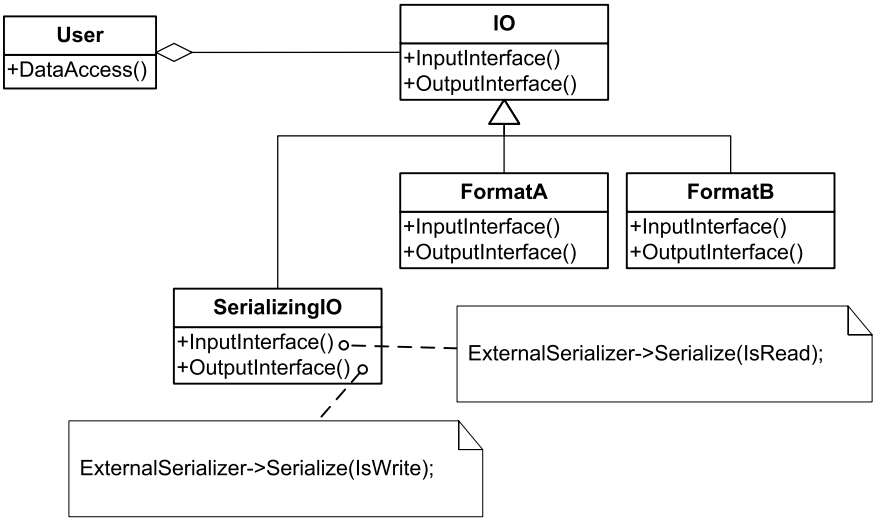

Figure 9.29: Using an external serialization library

later. But the important point to mention here is the difference of interpreting inside or outside the program.

One can implement an interpreter inside any IO subclass and use it in concepts of the IO module. In this way any application which uses this library can use the interpreter and accepts scripts without calling any other executable. Also encapsulating the interpreter in IO allows to implement and use more than one interpreter simultaneously. This allows the user to use subprograms written in languages different than the main script itself.

Another approach is to implement or use an interpreter like an application with close binding to our library. In this approach more complex interpreters can be used easier while there is even no need to compile them. This makes the library more portable and easier to compile.

Finally in practice there is no such a difference between the two approaches. The implementation cost in both cases is more or less the same. Also modern interpreters can run another interpreter which make it indifferent to put them inside IO or not. Usually it is better to put the small interpreters inside the IO and binding to the complex ones as an external application. In Kratos both approaches are used however this can be changed easily in future.

After designing the main structure of IO, now it is time to go through more details. The first part will be input interface designing and following that will be the output interface description.

9.3.7 Designing an Input Interface

Looking to the global scheme of a finite element application it can be seen that `Node`s, `Properties`, `Element`s, `Condition`s and initial values are the common input data. So a straightforward design for input interface consist of methods to extract these objects from an input source. However, providing these methods is sufficient to start working but still the interface can be extended more to ease its use. In a finite element program reading data usually consists of filling some finite element container like `Mesh` or `Modelpart`. So it is useful to include also some methods to handle directly this containers which eases the use of IO and in some cases increases the performance.

As mentioned before the `IO` class represents the interface to be implemented in derived classes. The input part of its interface is defined as shown below:

ReadNode Retrieves the next `Node` from the input and stop reading. This is useful in manually reading of especial `Node`s. This method gets a reference to the `Node` to be read and set it by input information.

```
ReadNode(NodeType& rThisNode)
```

ReadNodes Reads all the `Node`s in a source until the end of sequence is reached. Normal reading of `Node`s array can be performed using this method. It gets a `Node`s array and puts all the input `Node`s inside it.

```
ReadNodes(NodesContainerType& rThisNodes)
```

ReadProperties Reads the next `Properties` from the input and stops reading. It is useful to update a `Properties`. For example to read a modified `Properties` in an optimization procedure. It takes a `Properties` and put given information inside it.

```
ReadProperties(Properties& rThisProperties)
```

ReadProperties Reading different `Properties` from input can be done by this method. This method reads all the `Properties` until the end of sequence. It takes an array of `Properties` and put all the given `Properties` inside it.

```
ReadProperties(PropertiesContainerType& rThisProperties)
```

ReadElement Reads the next `Element` in the input. This method recognizes any `Element` registered in Kratos and reads the first one given as input. This method needs an array of `Node`s and Properties to extract the necessary information to create an `Element`. It also takes an `Element` pointer and assigns it to the new created `Element`.

```
ReadElement(NodesContainerType& rThisNodes,
            PropertiesContainerType& rThisProperties,
            Element::Pointer pThisElement)
```

ReadElements `Element`s reading method. This method recognizes any `Element` registered in Kratos and reads all the `Element`s given in the input. This method needs an array of `Node`s and `Properties` to extract necessary information to create `Element`. It also takes an `Element`s array to put created `Element`s in it.

```
ReadElements(NodesContainerType& rThisNodes,
             PropertiesContainerType& rThisProperties,
             ElementsContainerType& rThisElements)
```

ReadCondition Reads the next `Condition` in the input sequence. Like the `ReadElement` method, needs an array of `Node`s and `Properties` to extract the necessary information to create a `Condition`. It also takes a `Condition` pointer and if the read `Condition` is not Dirichlet type, it assigns it to the new created `Condition`.

```
ReadCondition(NodesContainerType& rThisNodes,
              PropertiesContainerType& rThisProperties,
              Condition::Pointer pThisCondition)
```

ReadConditions Conditions reading method. This is very similar to ReadElements but look for Conditions instead of Elements. This method also reads Dirichlet type conditions given as input. Also like ReadElements it takes arrays of Nodes and Properties to create Conditions. The generated Conditions are placed in the given Conditions array.

```
ReadConditions(NodesContainerType& rThisNodes,
               PropertiesContainerType& rThisProperties,
               ConditionsContainerType& rThisConditions)
```

ReadInitialValue Reads the next initial value from the input and puts it in data structure. It takes Nodes, Elements and Conditions and assign them their initial values.

```
ReadInitialValues(NodesContainerType& rThisNodes,
                  ElementsContainerType& rThisElements,
                  ConditionsContainerType& rThisConditions)
```

ReadInitialValues Reads all initial values and put them in the data structure. It takes Nodes, Elements and Conditions to assign their initial values given by input.

```
ReadInitialValues(NodesContainerType& rThisNodes,
                  ElementsContainerType& rThisElements,
                  ConditionsContainerType& rThisConditions)
```

ReadMesh Reads all Nodes, Properties, Elements and Conditions and store them in the given Mesh. This method is the easiest way to fill the Mesh at the beginning of program.

```
ReadMesh(MeshType & rThisMesh)
```

ReadModelPart Reads all Nodes, Properties, Elements and Conditions and store them in the given ModelPart. It also initialize the ModelPart name and its attributes.

```
ReadModelPart(ModelPart & rThisModelPart)
```

It is important to mention that above methods are just the interface and each derive class may implement all or just a set of them. So to avoid unexpected problems it is useful to put an error message in all IO methods and inform the user about calling some unimplemented feature in the derived class.

```
virtual void ReadModelPart(ModelPart & rThisModelPart)
{
  KRATOS_ERROR(std::logic_error,
    "Calling base class member. Please check the ...", "");
}
```

9.3.8 Designing the Output Interface

The output interface is defined by IO and output format is encapsulated in IO derived classes. The first part of the interface is devoted to writing basic objects. These methods are helpful for example to write a restart file. Also methods to write finite element containers are provided. These methods are useful in term of exporting the model or giving additional information to thepost processors. Finally there is a part of the interface dedicated to writing results. Here is the list of available output methods:

WriteNode Writes given `Node` into the output. This method may also write nodal data if the output format require it.

```
WriteNode(NodeType const& rThisNode)
```

WriteNodes This method writes an array of `Node`s into the output. It can be used also to create the restart file. Its argument is simply a `Node` container.

```
WriteNodes(NodesContainerType const& rThisNodes)
```

WriteProperties Takes a `Properties` and writes it into the output. This method is useful to communicate some `Properties` or communicate it via an interface.

```
WriteProperties(Properties const& rThisProperties)
```

WriteProperties Writes all given `Properties` to the output. Takes a `Properties` container and write all of them. This method can be used to create restart file by giving writing all `Properties` in it.

```
WriteProperties(PropertiesContainerType const& rThisProperties)
```

WriteElement Writes an `Element` into the output. It takes an `Element` as its argument. It is useful to do a selective writing between `Element`s.

```
WriteElement(Element const& rThisElement)
```

WriteElements This method takes a container of `Element`s and write all of them into the output. So it is a useful method to write a restart file by passing all `Element`s to it.

```
WriteElements(ElementsContainerType const& rThisElements)
```

WriteCondition Writes the given `Condition` into the output.

```
WriteCondition(Condition const& rThisCondition)
```

WriteConditions Takes a container of `Node`s and writes the Dirichlet conditions assigned to each `Node`. This method can be used to write the restart file.

```
WriteConditions(NodesContainerType const& rThisNodes)
```

WriteConditions An overload of previous method which takes a container of `Condition`s and write them into the output. This method also is useful to write the restart file.

```
WriteConditions(ConditionsContainerType const& rThisConditions)
```

WriteMesh This method writes all the `Node`s, `Properties`, `Element`s and `Condition`s in the given `Mesh` to the output. This makes it one of the best methods to write the restart file or it can be also used to create additional information for post processing task.

```
WriteMesh(MeshType const& rThisMesh)
```

WriteModelPart Writes all the `Node`s, `Properties`, `Element`s and `Condition`s for a given `ModelPart` into the output. This method also puts the `ModelPart` attributes into the output.

```
WriteModelPart(ModelPart const& rThisModelPart)
```

WriteNodalResults This method writes the nodal value of a given `Variable` for all given `Nodes` into the output. This method can be used to write nodal results for post processing. It takes the variable to be written and a container of `Nodes` which their values must be written. Also it takes a `ProcessInfo` which gives extra information for writing results, like the time step.

```
WriteNodalResults(VariableData const& rVariable,
                  NodesContainerType& rNodes,
                  ProcessInfo & rCurrentProcessInfo)
```

WriteElementalResults This method writes the elemental value of given variable into the output. This method can be use to write elemental results for post processing. It takes the variable to be write and a container of `Elements` which their values must be written. Also it takes a `ProcessInfo` which gives extra information for writing results, like the time step.

```
WriteElementalResults(VariableData const& rVariable,
                      ElementsContainerType& rElements,
                      ProcessInfo & rCurrentProcessInfo)
```

WriteIntegrationPointsResults This method writes the result calculated on integration points of given `Elements` into the output. It takes the variable to be write and a container of `Elements` which their integration points values must be written. Also it takes a `ProcessInfo` which gives extra information for writing results like time step.

```
WriteIntegrationPointsResults(VariableData const& rVariable,
                              NodesContainerType& rNodes,
                              ProcessInfo & rCurrentProcessInfo)
```

There is an important detail to be mentioned here. It can be seen that the last three methods are taking `VariableData` as their argument and not a typed variable or components. In C++ template function cannot be defined as virtual. This prevent us to make these methods a template. So to have it still extendible to new type the interface provide just a very general interface and each implementation has to check the type of variable via RTTI and then dispatch it to the proper implementation.

9.4 Writing an Interpreter

This section describes a step by step approach to write an interpreter. First the global structure and its components will be commented. Then creating an interpreter by using two different tools is described.

9.4.1 Global Structure of an Interpreter

An interpreter usually is divided into two parts as shown in figure 9.30.

The first part is the *lexical analyzer*. This part reads the input characters, for example a source file or given command line statement, and converts it to a sequence of *tokens*, like digits, names, keywords, etc., usable for parser. Since white spaces (like blank, tab and newline characters) and comments are not used in parsing the code, there is a secondary task for a lexical analyzer to

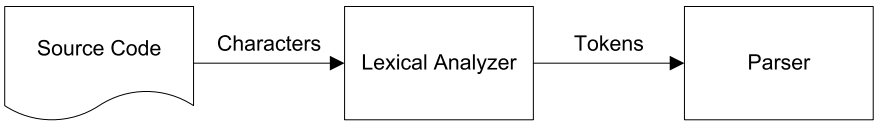

Figure 9.30: Global structure of an interpreter

eliminate all white spaces and also comments during the analysis. The lexical analyzer may also provide additional information for the parser to produce more descriptive error messages.

A *token* is a sequence of characters having a logical meaning or making a unit in our grammar. One can divide tokens in different categories depending on the grammar working with. Table 9.3 shows some examples of tokens.

Token's Type	Examples
INTEGER	1 34 610
REAL	0.23 4.57 2e-4
IF	if
FOR	for
LPARENTHESES	(
RPARENTHESES	)

Table 9.3: Some typical tokens with example of matching sequences

If more than one sequence of input characters matches to a token then this token must provide a value field to store the input data represented by it. To clarify the concept of token and lexical analyzer let make an example. Considering a simple C `if` statement:

```
if(x > 0.00)
  return 0;
```

passing it to a C lexical analyzer would result in the sequence of tokens:

IF	LPARENTHESES	ID(x)	GREATER	REAL(0.)
RPARENTHESES		RETURN	INTEGER(0)	SEMICOLON

The second part is the *parser*. Parser does the syntax analysis. Takes the tokens from lexical analyzer and tries to find their relation and meaning due to its grammar. Parser produces a parse tree by putting together recognized statements and their sub-operations. This tree can be used to execute the given source. For example passing above sequence of tokens to a C parser would result in a parse tree as presented in figure 9.31.

There are several reasons to separate the lexical analyzer from the parser. The first one is simplifying the design and reducing parser complexity. Creating a parser over separated tokens without any comments or whitespaces is easier than a parser over input characters. The second reason is improving the performance of the interpreter. Large amount of interpreter time spent to do the lexical analysis, separating it and using techniques like buffering can increase significantly the performance. Another reason is related to portability and reuse ability of the interpreter. Any problem due to the different character maps in different devices can be encapsulated in the lexical analyzer without changing the parser.

In some cases, the lexical analyzer is also divided into two phases. A scanner which does simple tasks like removing comments and whitespaces, and analyzer which does the real complex job.

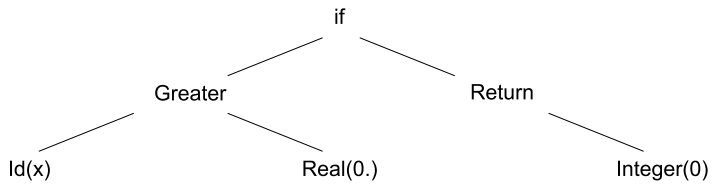

Figure 9.31: Parse tree for `if` statement example

Now let us go inside the each part and see how to implement it.

9.4.2 Inside a Lexical Analyzer

A lexical analyzer uses some patterns to find tokens in a given sequence of characters. Usually these patterns are specified by *regular expressions*. So before starting with the lexical analyzer pattern a brief description of regular expressions is necessary.

Regular expressions

A language can be considered as a set of strings with some special properties. So it is important to describe exactly these properties to define a language. Regular expressions are the common notation to define these concepts in a generic way.

This notation consist of some rules which let regular expression r denoting the language $L(r)$ and also some other rules to combine different expressions together in different ways. This combination rules makes it very powerful in term of dividing complex definition to simpler ones or to reuse some expressions to make a more complex one.

Regular expressions rules consist of symbol collective rules from a set of symbols called *alphabet*, which regular expression is defined over it. Here are the rules which define the regular expression over alphabet Σ:

Epsilon ϵ is a regular expression to denote the language $\{\epsilon\}$ containing just the empty string .

Symbol For a symbol a in Σ, the regular expression a denotes the language $\{a\}$, whose only string is `a`.

Alternative Alternative operator written with a vertical bar | combines two regular expressions r and s into new regular expression $r|s$. This expression represent the language $L(r) \bigcup L(s)$ that contains all strings contained by either r or s. For example $a|b$ represents the language $\{a, b\}$.

Concatenation Two regular expressions r and s can be combined into a new regular expression $r \cdot s$ using the concatenation operator $\cdot$. This new expression denotes the language $L(r)L(s)$ that contains all strings $\alpha\beta$ for all $\alpha \in L(r)$ and $\beta \in L(s)$. For example $a \cdot b$ denotes the language $\{ab\}$. In some notations the $\cdot$ symbol is simply omitted and putting two expressions simply one after other means concatenation, $a \cdot b \equiv ab$.

Repetition Repetition operator *, or *Kleene star*, applied to a regular expression r represents the language $L(r)^*$ that contains strings with zero or more instances of any symbol in $L(r)$. For example, $(a|b)^*$ represents the language $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, ...\}$.

Parentheses Parentheses can be use to group an operation and applying it to a single rule will not change the expression rule.

There is also some conventions in precedence of operators which helps to reduce ambiguities:

- Kleene star has the highest precedence.
- Concatenation has the second precedence.
- Alternative operator has the lowest precedence.
- All these operators are left associative.

Finally there are some abbreviations for frequent expressions which eases its use in practice:

Positive closure The positive closure operator $^+$ is left associative and applied to the regular expression r denotes the language $L(r)^+$ which contains one or more instances of any symbol in $L(r)$. This operator has the same precedence as the repetition operator *. $r^* = r^+|\epsilon$ and $r^+ = rr^*$ equations describes the relation between the positive closure operator and the repetition operator. For example, $(a|b)^+$ represents the language $\{a, b, aa, ab, ba, bb, ...\}$.

Zero or one instances Applying the unary operator ? to the regular expression r denotes the language $L(r) \bigcup \{\epsilon\}$ which contains zero or one instance of any symbol in $L(r)$. This operator is the abbreviation of $r|\epsilon$. For example, $(a|b)$? represents the language $\{\epsilon, a, b\}$.

Character classes For alphabet symbols a,b,...,z The notation $[abc]$ is equivalent to $a|b|c$ and the $[a-z]$ is equivalent to $a|b|...|z$. This two abbreviations can also be combined. For example a C identifier can be any letter followed by zero or more letters or digits. This definition can be written using regular expression $[A-Za-z][A-Za-z0-9]^*$.

After a brief description of regular expressions, now it is time to take a look at the lexical analyzer itself.

Lexical Analyzer

As mentioned before a lexical analyzer is in charge of converting a sequence of characters as its input to a sequence of tokens as its output using some patterns. Usually regular expressions are used to specify these patterns. For example to read the following node statements:

```
// A typical Node Statement
Node(1,1.00,0.00,0.00)
```

The lexical analyzer need to understand the `Node` keyword, integer, real parenthesis and also comments. Here are the regular expressions defining legal tokens and their corresponding actions for this case:

```
Node                                        {return NODE_TOKEN}
[0-9]+                                      {return INTEGER_TOKEN}
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)         {return REAL_TOKEN} "//"("
"|[A-Za-z0-9])*"\n"                  {/* It is just comment*/} "
"|"\n"|"\t"                               {/* white spaces */}
```

These rules are somehow ambiguous. For example reading the x coordinate `1.00` can create a problem while the first `1` can be recognized as an integer and the `.00` as a double after it! To reduce this ambiguities lexical analyzers use two additional rules:

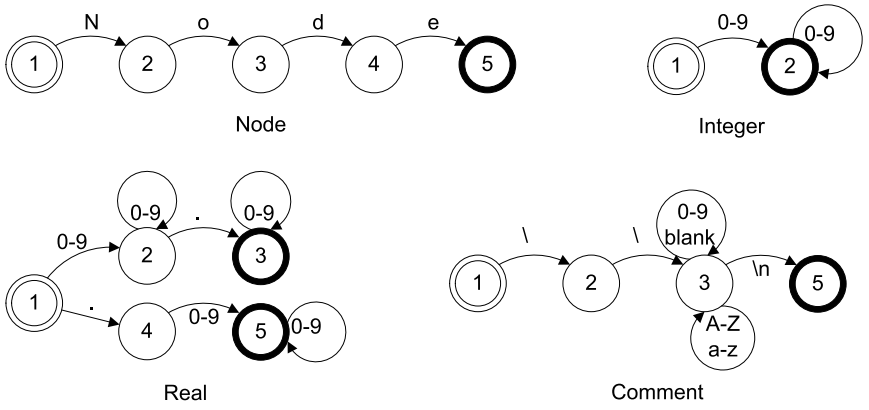

Figure 9.32: Finite automata example

Longest match Analyzer tries to find the longest sequence of input characters that can match to one of the patterns.

Pattern priority For a particular longest string which can much, the first regular expression which matches to it has priority to determine its type.

The idea here is to create a mechanism to convert a given regular expressions to a lexical analyzer automatically. The method is to convert given regular expressions to a graph which can be implemented easily.

First the regular expression has to be converted to *finite automata*. The finite automata is a graph consisting of finite set of *states* connected by *edges* and each edge is labeled with a symbol. This graph shows the way a sequence of character can traverse all states to be matched to a regular expression. Finite automata is *deterministic* if all its states have a maximum of one edge going out for a symbol and any state with more than one outward edge for a symbol converts it to a *nondeterministic finite automata*.Figure 9.32 shows a finite automata of the regular expressions in previous example.

Now let us combine all these separate finite automata in a single one which represents our lexical analyzer. For this small example this can be down manually but in practice manual combination can be impossible. Fortunately there are some ways to do this automatically. These methods are consist of converting the regular expressions to nondeterministic finite automata (NFA) and then reduce them to a deterministic finite automata (DFA) plus some optimizations. Detail information about automatic converting procedure of regular expressions into deterministic finite automata can be found in [12, 13, 16]. Here the manual combination is used to create the complete graph shown in figure 9.33.

Finally this combined graph can be converted to a *transition table*. This table consists of several rows, one for each state and also several columns, one for each symbol. Each field contains the target state number for its row state giving its column symbol.

Having a transition table, analyzing an algorithm is very easy to implement. The program first reads a character and goes to the field corresponding to the state 1's row and to this symbol's column. This field contains the target state which is for example state 7. Then it reads the next character and goes to state 7's row and to this new symbol's column to find the next step and so on. An auxiliary array which maps the state numbers to actions is also necessary to verify if a

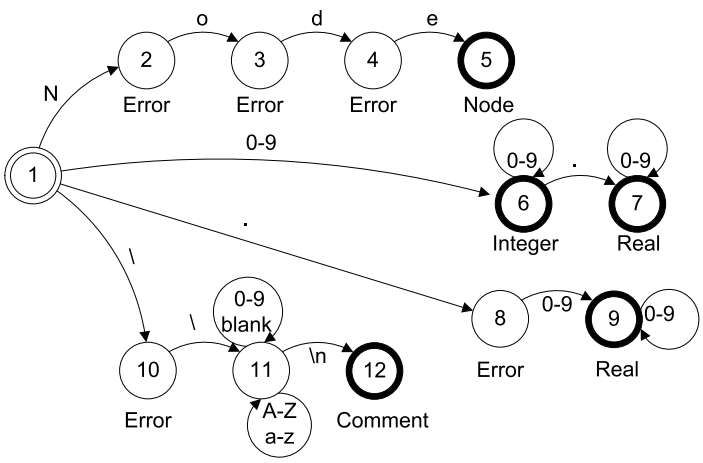

Figure 9.33: Combined finite automata example

pattern is matched what is the action to perform. To find the longest match is just need to store the last match found and update it each time a new match is found. This procedure continues until a dead state is reached. In this time the last match case, if there is any, is returned or if there is no match an error can be send.

9.4.3 Parser

A lexical analyzer prepares tokens to be analyzed by the parser and create the parse tree to be executed. The parser tries to match a given sequence of tokens to a pattern. Here a context free grammar is used to specify the patterns. So let see what it is before continuing with the parser.

Context Free Grammar

The idea of this notation is to group basic parts in some more complex ones like regular expressions and also to make a recursive use of symbols to enable recursively construction of complex statements.

A context free grammar consists of terminals, nonterminals, productions and also a start symbol.

Terminal Terminals are the basic units of grammar. In grammars for programming languages terminals are equivalent to tokens. In our case each token is a terminal in the context free grammar.

Nonterminal Nonterminals are syntatic variables that represents a group of terminals or nonterminals as a part of a language syntax.

Production Production is the manner of defining a nonterminal by combination of terminals and nonterminals. Production has a general form of:

$$nonterminal \rightarrow sting\ of\ terminals\ and\ nonterminals$$

In some notations symbol ::= is used instead of the arrow.

Start Symbol It is a nonterminal that is selected as a start symbol and denotes the language defined by the grammar.

To make this definition clear let is create a grammar to define the previous node input file language:

- *statement* → *statement* | *node_creator*
- *node_creator* → **Node** (**integer** , *coordinates*)
- *coordinates* → **real** , **real** , **real**

The third line creates a pattern of coordinates with three reals separated by a comma. The second line gives the pattern for construction a `Node`. Finally the first line define the statement which is constructing the `Node` or itself. This leads parser to recurse and reads all the node statements comes statements comes after.

It can be verified that every regular expression set is a context free grammar by itself. This may introduce a question, Why use regular expression to define lexical patterns and not the context free grammar by itself. Here are some reasons:

- Regular expressions are enough to define the syntax and there is no need to deal with the complexity of context free grammars.
- Regular expressions are easier to understand and they define a token more clear.
- Lexical analyzer can be automatically optimized to be more efficient when constructed over regular expression rather than a generic grammar.

Designing a Parser

To design a parser the *Interpreter* pattern is what we are looking for. A context free structure can be represented by this structure and then used to create the parse tree to execute the source code.

In general there are various ways to implement a parser in the literature [12, 13, 16]. Also there are some tools to generate parser. Here we will use this tools without going into the parser implementation details.

9.4.4 Using Lex and Yacc, a Classical approach

There are several tools to create a parser. Among these, *Lex* and *Yacc* are two classic ones.

Lex is a lexical analyzer generator. It takes a *lexical specification* as input and produces a lexical analyzer written in C. Lexical specification contains regular expression defining each token and also an action that will be executed when an input sequence is matched to a regular expression. These actions are normal C statements and usually returning the token for their case. The backward of Lex is its performance. *Fast lexical analyzer generator (Flex)* is significantly faster than Lex [16] and solves the problem of performance, But still it generates a C code. Though there is no problem in binding a C code to a C++ program, this still pollutes the scope by using several global variables and functions. This also make it difficult to maintain more than one lexical analyzer in the same code. In this work a C++ variation of Flex named *Flex++* is used to generate the lexical analyzer.

Yet Another Compiler-Compiler (Yacc) is a classic and widely used parser generator. There are several compilers implemented using Yacc. *Bison* is another parser generator which is more

modern than Yacc but is upward compatible with it. Both generate parsers which are implemented in C language. In this work *Bison++* is used which is a variety of Bison that generates parsers implemented in C++.

Using Flex

Flex++ like Lex uses lexical specifications to generate the lexical analyzer. This specifications must be prepared by creating an input file usually with extension "`l`", like "`input.l`", and consists of three parts separated by `%%`:

```
%{
C++ Definitions
%}
Lex Definitions

%%

Regular expressions and their actions

%%

Auxiliary procedures
```

The first part is for definitions which is also divided in two parts, C++ definitions part which is enclosed between `%{ %}` symbols and Lex definitions parts. FLex++ simply puts the C++ declaration part above the generated output file. This part usually contains the necessary include files and some macro definitions to customize the generated code. In our case the C++ declaration part is:

```
%{
#define YY_DECL \
  int Kratos::InputCScanner::yylex(yy_InputCParser_stype &yylval)

#include <iostream>
#include <malloc.h>
#include "includes/input_c_parser.h"
#include "includes/input_c_scanner.h"

int yyFlexLexer::yylex(){ return 0; }
%}
```

The lex definition part is to put macros for lex or conditions to be used afterwards. Here is an example of macro definition:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

In this case only a condition for comments is defined here:

```
%x COMMENT
```

The second part is to define rules. Each rule is a regular expression and its corresponding action. In Kratos four groups of rules are defined:

- Rules to handle comments. C++ comments are accepted in this format. One point here is to keep line numbering inside the comments. Another point is to give a warning in case of comments in comments.

```
\n                  { ++mNumberOfLines; }

"//".*$             /* remove one line coments */

"/*"                { BEGIN COMMENT;  }
<COMMENT>"/*"       { Warning("Comment in comment."); }
<COMMENT>\n         { ++mNumberOfLines; }
<COMMENT>[^*/\n]*   ;
<COMMENT>"*"+[^*/\n]*   ;
<COMMENT>"/"+[^*/\n]*   ;
<COMMENT>"*"+"/"    { BEGIN INITIAL; }
```

- The second group are symbols. Most of them are passed as they are. In more formal way each symbol must be passed by a representative token which is not done here for simplicity.

```
\[|\]|"("|")"|"{"|"}" { return yytext[0]; }

"="                   { return yytext[0]; }

","|";"|"."|":"       { return yytext[0]; }

"<"|">"               { return yytext[0]; }

"=="                  { return InputCParser::EQUAL_TOKEN; }

"!="                  { return InputCParser::NOT_EQUAL_TOKEN; }

"<="                  { return InputCParser::LESS_EQUAL_TOKEN; }

">="                  { return InputCParser::GREATER_EQUAL_TOKEN;}

"+"|"-"               { return yytext[0]; }
```

- After symbols there are rules to define Kratos variables. Here are some examples of these rules:

```
TEMPERATURE  {
                yylval.double_variable = &TEMPERATURE;
                return InputCParser::DOUBLE_VARIABLE_TOKEN;
             }

VELOCITY     {
                yylval.vector_double_variable = &VELOCITY;
                return InputCParser::VECTOR_DOUBLE_VARIABLE_TOKEN;
             }
```

- And finally rules for keywords:

```
NODES            { return InputCParser::NODES_TOKEN; }

PROPERTIES       { return InputCParser::PROPERTIES_TOKEN; }

Node             { return InputCParser::NODE_TOKEN; }
```

```
Fix             { return InputCParser::FIX_TOKEN; }

ElementsGroup   { return InputCParser::ELEMENTS_GROUP_TOKEN; }

ELEMENTS        { return InputCParser::ELEMENTS_TOKEN; }

LinearSolver    { return InputCParser::LINEAR_SOLVER_TOKEN; }

SolvingStrategy { return InputCParser::SOLVING_STRATEGY_TOKEN; }

Smooth          { return InputCParser::SMOOTH_TOKEN; }

Map             { return InputCParser::MAP_TOKEN; }

DataMapper      { return InputCParser::MAPPER_TOKEN; }

for             { return InputCParser::FOR_TOKEN; }

Solve           { return InputCParser::SOLVE_TOKEN; }

MoveMesh        { return InputCParser::MOVE_MESH_TOKEN; }

Print           { return InputCParser::PRINT_TOKEN; }

Execute         { return InputCParser::EXECUTE_TOKEN; }

CreateSolutionStep {return InputCParser::CREATE_SOLUTION_STEP_TOKEN; }

CreateTimeStep  { return InputCParser::CREATE_TIME_STEP_TOKEN; }
```

There are also some other rules defines to scape white spaces our returning not matches words.

The third part of Flex input is to put the auxiliary procedures which in this case is left empty.

Flex++ takes these information to generate the transition tables and also a generic code to be called by Bison++ later.

Using Bison++

Like Flex++, specifications for Bison++ must be prepared in a file but usually with extension "`y`", like "`input.y`". This file again consists of three parts separated by `%%`:

```
%{
C++ Definitions
%}
Parser Definitions

%%

grammar rules

 %%

Auxiliary codes
```

The C++ definition here is important because all definitions of the abstract tree classes goes there:

```
%{
  Kratos::String block_name;

  class Statement{
  public:
    Statement(){}
    virtual void Execute(Kratos::Kernel* pKernel){}
    virtual int Value(Kratos::Kernel* pKernel){return 0;}
  };

  class BlockStatement : public Statement{...};

  class GenerateNodeStatement : public Statement{...};

  template<class TDataType>
  class GeneratePropertiesStatement : public Statement{...};

  class SetElementsGroup : public Statement{...};

  class GenerateElementStatement : public Statement{...};

  class FixDofStatement : public Statement{...};

  template<class TDataType>
  class SetSourceStatement : public Statement{...};


  class GenerateLinearSolverStatement : public Statement{...};

  class GenerateSolvingStrategyStatement : public Statement{...};

  class GenerateMapperStatement : public Statement{...};

  class SolveStatement : public Statement{...};

  class MoveMeshStatement : public Statement{...};

  class ProcessStatement : public Statement{...};

  class ForStatement : public Statement{
    Statement* mpFirst;
    Statement* mpSecond;
    Statement* mpThird;
    Statement* mpForth;
  public:
    ForStatement(Statement* pFirst, Statement* pSecond,
                 Statement* pThird, Statement* pForth) :
      mpFirst(pFirst), mpSecond(pSecond),
      mpThird(pThird), mpForth(pForth) {}
```

```
    ~ForStatement(){
        delete mpFirst; delete mpSecond;
        delete mpThird; delete mpForth;
    }

    void Execute(Kratos::Kernel* pKernel){
        for(mpFirst->Execute(pKernel); mpSecond->Value(pKernel) ;
                                       mpThird->Execute(pKernel))
            mpForth->Execute(pKernel);
    }
};

template<class TDataType>
class PrintStatement : public Statement{...};

template<class TDataType>
class PrintOnGaussPointsStatement : public Statement{...};

template<class TDataType>
class SmoothStatement : public Statement{...};

template<class TDataType>
class MapStatement : public Statement{...};

class CreateTimeStepStatement : public Statement{...};

class CreateSolutionStepStatement : public Statement{...};

template<class TFunction, class TDataType>
class LogicalStatement : public Statement{...};

template<class TDataType>
class ValueStatement{...};

template<class TDataType>
class ConstantValueStatement : public ValueStatement<TDataType> {...};

template<class TDataType>
class VariableStatement : public ValueStatement<TDataType> {...};

template<class TFunction, class TDataType>
class BinaryVariableStatement : public ValueStatement<TDataType> {...};

template<class TDataType>
class AssigningVariableStatement : public Statement {...};

template<class TDataType>
class AssigningNodalVariableStatement : public Statement {...};
%}
```

The class implementations are removed to reduce the size of the document. The second division is for parser definitions. Macro definitions with all tokens and nonterminal declarations are placed here:

```
%define CONSTRUCTOR_PARAM Kratos::Kernel* ...
%define CONSTRUCTOR_INIT : mInputCScanner(pNewInput, ...
%define YY_InputGid_CONSTRUCTOR_CODE
%define LEX_BODY { return mInputCScanner.yylex(yylval); }
%define MEMBERS  Kratos::InputCScanner mInputCScanner;...

%token <integer_value> INTEGER_TOKEN
%token <double_value> DOUBLE_TOKEN
%token <string_value> WORD_TOKEN
%token <double_variable> DOUBLE_VARIABLE_TOKEN
%token <vector_double_variable> VECTOR_DOUBLE_VARIABLE_TOKEN
%token <matrix_variable> MATRIX_VARIABLE_TOKEN
%token OPEN_BRACKET_TOKEN
%token CLOSE_BRACKET_TOKEN
%token NODES_TOKEN
%token NODE_TOKEN
%token PROPERTIES_TOKEN
%token ELEMENTS_GROUP_TOKEN
%token ELEMENTS_TOKEN
%token ELEMENT_TOKEN
%token BEGIN_TOKEN
%token END_TOKEN
%token FIX_TOKEN
%token SOURCES_TOKEN
%token FOR_TOKEN
%token SOLVE_TOKEN
%token MOVE_MESH_TOKEN
%token PRINT_TOKEN
%token PRINT_ON_GAUSS_POINTS_TOKEN
%token EXECUTE_TOKEN
%token TRANSIENT_TOKEN
%token ALPHA_TOKEN
%token SMOOTH_TOKEN
%token MAP_TOKEN
%token MAPPER_TOKEN
%token LINEAR_SOLVER_TOKEN
%token SOLVING_STRATEGY_TOKEN
%token EQUAL_TOKEN
%token NOT_EQUAL_TOKEN
%token LESS_EQUAL_TOKEN
%token GREATER_EQUAL_TOKEN
%token CREATE_SOLUTION_STEP_TOKEN
%token CREATE_TIME_STEP_TOKEN

%type <statement_handler> statement
%type <statement_handler> expresion
%type <statement_handler> node_generating
%type <statement_handler> properties_adding
%type <statement_handler> set_elements_group
%type <statement_handler> element_generating
%type <statement_handler> nodes_variables_fixing
%type <statement_handler> nodes_variables_setting
%type <statement_handler> elements_variables_fixing
```

```
%type <statement_handler> linear_solver_generating
%type <statement_handler> solving_strategy_generating
%type <statement_handler> mapper_generating
%type <statement_handler> solve_process
%type <statement_handler> move_mesh_process
%type <statement_handler> execute_process
%type <statement_handler> print_process
%type <statement_handler> print_on_gauss_points_process
%type <statement_handler> smooth_process
%type <statement_handler> map_process
%type <statement_handler> step_creating
%type <statement_handler> time_step_creating
%type <statement_handler> for_loop
%type <statement_handler> assignment_expresion
%type <statement_handler> logical_expresion
%type <block_statement_handler> block_generating
%type <block_statement_handler> block
%type <double_value_statement_handler> double_variable_expresion
%type <integer_value> integer_expresion
%type <integer_value> integer_index
%type <integer_value> nodes_array
%type <integer_value> elements_array
%type <integer_value> properties_array
%type <double_value> double_expresion
%type <vector_integer_value> integer_sequence
%type <vector_integer_value> vector_integer
%type <vector_double_value> vector_double
%type <matrix_double_value> matrix
%type <vector_double_value> double_sequence
%type <vector_vector_double_value> vector_double_sequence
```

The second part holds grammar definitions in context free format and their corresponding semantic actions. Here is the organization grammar for input:

```
statement               :   expresion ';' {$$ = $1}
                        |   block {$$ = $1;}
                        |   for_loop {$$ = $1;}
                         ;

expresion               :   assignment_expresion {$$ = $1;}
                        |   logical_expresion {$$ = $1;}
                        |   node_generating {$$ = $1}
                        |   properties_adding {$$ = $1;}
                        |   set_elements_group {$$ = $1;}
                        |   element_generating {$$ = $1;}
                        |   nodes_variables_fixing {$$ = $1;}
                        |   nodes_variables_setting {$$ = $1;}
                        |   elements_variables_fixing {$$ =$1;}
                        |   solve_process {$$ = $1;}
                        |   move_mesh_process {$$ = $1;}
                        |   execute_process {$$ = $1;}
                        |   print_process {$$ = $1;}
                        |   print_on_gauss_points_process {$$ = $1;}
                        |   smooth_process {$$ = $1;}
```

```
                        |   map_process {$$ = $1;}
                        |   linear_solver_generating {$$ =$1;}
                        |   solving_strategy_generating {$$ = $1;}
                        |   mapper_generating {$$ = $1;}
                        |   step_creating {$$ = $1;}
                        |   time_step_creating {$$ = $1;}
                        ;

block                   :   block_generating '}'
                            {
                                $$ = $1;
                            }

block_generating        :   '{' statement
                            {
                                $$ = new BlockStatement();
                                $$->AddStatement($2);
                            }
                        |   block_generating statement
                            {
                                $1->AddStatement($2);
                                $$ = $1;
                             }
```

The rest of definitions have not brought here to reduce the size of listing. The third part is for the auxiliary function. In our case an error reporting function is defined here is:

```
using Kratos::Exception;

void InputCParser::yyerror( char *s) {
  Kratos::String buffer;
  buffer << "Pars error in line no "
         << mInputCScanner.rNumberOfLines()
         << ", last token was : "
         << mInputCScanner.GetYYText();
  KRATOS_ERROR(std::runtime_error, "Reading Input", buffer, "");
}
```

9.4.5 Creating a new Interpreter Using Spirit

Flex++ and Bison++ are great tools to generate an interpreter but they have been put aside from this work for mainly two reasons. The first was changes in the strategy of code development form creating a sophisticated interpreter to using an existing one. The idea was to have a simple but flexible data IO and using an existing interpreter for process IO. By the way for reading data still a simple parser was necessary, but these tools were too heavy to be used for this case. The other reason was portability and maintaining issues.Any small changes in grammar, for example introducing a new variable name, requires to regenerate the parser and then recompile the program. In practice this yields portability problems specially in Windows, due to the incompatibilities between Linux and Windows compilations of these tools.

Spirit is an object-oriented parser generator framework implemented in C++ using template meta-programming techniques. Spirit enable us to write the context free grammar directly in C++ code and compile it with a C++ code to generate the parser. In this way the translation step from

context free grammar to a parser implemented in C++ by an external tool is omitted. There are several small parsers already defined in Spirit which help users to implement its parsers easier.

Parsing in Spirit can be done using one of the overloaded parse functions. Some of these overloads work in character level while some others work in phrase level and take and skip parser. This skip parser is helpful for example in skipping white spaces and comments. The `parse` function is also overloaded respected to its input. There are overloads that accept first and last operator like STL algorithms, and others which accept a null terminating string. Phrase level parse functions with first and last iterator as input are used in this work:

```
template <typename IteratorT, typename ParserT, typename SkipT>
parse_info<IteratorT> parse
    (
        IteratorT const&         first,
        IteratorT const&         last,
        parser<ParserT> const&   p,    // Grammar rules
        parser<SkipT> const&     skip // Skip rules
    );
```

Now let us start to create a parser for reading our node statement example. This helps to see how a simple parser can be generated using Spirit. Here is the statement to be parsed:

```
// a node statement:
// Node(Index,X,Y,Z);
Node(1, 0.02, 1.00, 0.00);
```

First we need to use Spirit grammar components to define our grammar. Table 9.4 shows valid operators in Spirit. These operators are defined to be as close as possible, respecting to the C++ language restrictions, to their corresponding regular expression operators. So understanding regular expressions helps to use Spirit as well.

Operator	Description
`!P`	Zero or One P
`*P`	Zero or more P
`+P`	One or more P
`~P`	Anything except P
`P1 % P2`	One or more P1 separated by P2
`P1 - P2`	P1 but not P2
`P1 >> P2`	P1 followed by P2
`P1 & P2`	P1 and P2
`P1 ^ P2`	P1 or P2, but not both
`P1 | P2`	P1 or P2
`P1 && P2`	Synonym for P1 >> P2
`P1 || P2`	P1 or P2 or P1 >> P2

Table 9.4: Spirit's operators

Using this operators together with some predefined Spirit parsers creates the necessary grammar to read the node statements:

```
(
  str_p("Node")
  >> '('
```

```
    >> uint_p
    >> ','
    >> real_p
    >> ','
    >> real_p
    >> ','
    >> real_p
    >> ')'
  )
```

str_p is a predefined parser which matches input with given string which is "Node". uint_p is another parser which matches to an unsigned integer in input and used to read the index of the Node. real_p used to read coordinates. It matches to a real number from input. Finally any character matches to itself. This grammar can be used to parse the statement but still there is no action to do when input matches each part. So we have to add some actions to it:

```
  (
    str_p("Node")
    >> '('
    >> uint_p[assign_a(node.Id())]
    >> ','
    >> real_p[assign_a(node.X())][assign_a(node.X0())]
    >> ','
    >> real_p[assign_a(node.Y())][assign_a(node.Y0())]
    >> ','
    >> real_p[assign_a(node.Z())][assign_a(node.Z0())]
    >> ')'
  )
```

Here assign_a is an action which retrieves the value read by parser and assign it to its argument. Also it can be seen that actions can be cascaded when there are more than one action has to be performed for a rule. For handling comments and white spaces we need a skip parser. This parser can be written easily as below:

```
(space_p | comment_p("//") | comment_p("/*", "*/"))
```

space_p is a parser already defined in Spirit which matches to any white space character. There is another predefined parser in sprit which handles comment patterns. comment_p with just one argument matches to a sequence starting with given string and finished by end of line, like C++ comments. comment_p with two arguments matches with a sequence starting from the first argument and finished with second one, like C comments.

Now let us put every things together and create our ReadNode method:

```
void ReadNode(NodeType& rNode,
              IteratorType First,
              IteratorType Last)
{
   using namespace boost::spirit;
   parse(First, Last,
        // Begin grammar
        (
            str_p("Node")
            >> '('
            >> uint_p[assign_a(rNode.Id())]
```

```
            >> ','
            >> real_p[assign_a(rNode.X())][assign_a(rNode.X0())]
            >> ','
            >> real_p[assign_a(rNode.Y())][assign_a(rNode.Y0())]
            >> ','
            >> real_p[assign_a(rNode.Z())][assign_a(rNode.Z0())]
            >> ')'
        )
        // End grammar
        ,
        // Skip grammar
        (space_p | comment_p("//") | comment_p("/*", "*/")));
}
```

That's it! This function can parse any sequence of characters given by iterators `First` and `Last`. But we may need to parse an input data file. This can also be done by slightly modifying above function. Spirit provides a file iterator which can be used in the same way that normal the iterators are used. Here is the new version of `ReadNode` which is able to read a `Node` from an input file:

```
void ReadNode(NodeType& rNode,
              std::string Filename)
{
    using namespace boost::spirit;

    FileIterator first(Filename);
    FileIterator last= First.make_end();

    parse(first, last,
        // Begin grammar
        (
            str_p("Node")
            >> '('
            >> uint_p[assign_a(rNode.Id())]
            >> ','
            >> real_p[assign_a(rNode.X())][assign_a(rNode.X0())]
            >> ','
            >> real_p[assign_a(rNode.Y())][assign_a(rNode.Y0())]
            >> ','
            >> real_p[assign_a(rNode.Z())][assign_a(rNode.Z0())]
            >> ')'
        )
        // End grammar
        ,
        // Skip grammar
        (space_p | comment_p("//") | comment_p("/*", "*/")));
}
```

In this IO we also want to parse new components, for example new variables, `Elements`, or `Conditions`, without changing and updating the parser each time. Spirit provides a `symbols` class which fits well to our purpose. This class is a parser associated with a symbol table. This table can be initialized with different symbols and parser matches to any symbol stored in its table. So adding KratosComponents elements to a symbol class provides a parser to match these components.

```
template<class TComponentType>
  class  ComponentParser :
    public symbols<reference_wrapper<TComponentType const> >
  {
  public:
    ComponentParser()
    {
      typedef KratosComponents<TComponentType> ComponentsType;
      typedef ComponentsType::ComponentsContainerType ContainerType;
      typedef ContainerType::const_iterator iterator_type;

      iterator_type i_component;

      for(i_component = ComponentsType::GetComponents().begin() ;
          i_component != ComponentsType::GetComponents().end() ;
          i_component++)
            add(i_component->first.c_str(), i_component->second);
      }

  };
```

`ComponentParser` derived from `symbols` class and in its constructing time adds automatically all `TComponentsType` components to the symbol table. This make it a parser that match to any components stored in `KratosComponents` list of that type.

Finally all these concepts and tools are used together to create a flexible and extendible IO for Kratos.

9.5 Using Python as Interpreter

As we mentioned before during the evolution of Kratos, the strategy for implementing a complete interpreter for the IO part changed to use an existing one. This change in the development strategy, lead finally to using the Python interpreter.

9.5.1 Why another interpreter

In time of decisions about what to include in the IO features, a process IO was chosen as a way to introduce new algorithms without changing the code and recompile it. To be able to do this in practice IO has to provide a complete set of language flow control commands.

Implementing an interpreter is a hard work. From the management point of view, it introduces a significant overhead to project cost and time. Maintaining it is even harder and results in more overhead in the cost of the code. Also implementing an interpreter requires knowledge in different concepts like grammar notations, some tools and libraries usage and compiler implementation techniques. Finite element programmers usually are not familiar with most of these concepts and in many cases even do not like to deal with them. So in developing a finite element program it is not easy to share the implementation and maintenance of an interpreter with others due to this lack of experience. In practice, this can lead to longer implementation times and extra cost.

During the implementation of Kratos all these facts affected the implementation of an interpreter and made its development more and more difficult to the point that it became one of the bottlenecks of the project. Consequently the strategy changed by stop writing an interpreter and looking for an existing one.

Binding Kratos to Python makes it extremely flexible and extendible. New algorithms can be implemented using existing Kratos tools and methods without even recompiling the code. Interaction between domains and planning different staggered strategies to solve a coupled problem also can be perform easily at this level without changing the Kratos or its applications. Also it can be used as a small lab for testing new algorithms and formulations before programming it into the Kratos. The list of added features is unlimited and many complex tasks can be done easily using this interface.

9.5.2 Why Python

Selecting a script languages between the large amount of available script language is not an easy task. Each different language has its own advantages and disadvantages which can be useful for some cases but leaves uncovered some objectives. So an intention is to understand first the requirements and then find a language that fits more naturally to these tasks.

In this case there are different features required to be supported by the language:

- First of all any interpreter to be used has to have an interface to C++ or at least to C. It really makes no sense to choose an interpreter which cannot be bounded to our code.
- Portability is another essential point to be considered. Kratos is written in standard C++ and uses also portable libraries to enhance its portability. So the interpreter must continue this idea.
- To be object-oriented is an important feature. Though high level commands can be expressed by traditional languages but an object-oriented language can represent much better internal classes and data structures.
- Language syntaxes and its readability is another important feature. This depends on personal tastes but a self descriptive syntax considered to be clearer than the highly symbolized one.
- Ease of learning also is considered as an important point. A complex and difficult to learn language reduces the number of users who wants to use these IO features.
- Flexibility and extendibility are other features to be considered. Adding new types of data and also using them via the interface is essential in order to add finite element data and containers.
- An active language in sense of development will be preferred to an old but not active one due to the risk of new unresolved problems.
- Finally a popular language is preferable because the larger community of programmers helps a lot in dealing with day to day problems and questions.

From the long list of languages first some more popular ones were selected to be verified:

- *Lisp* is one of the widely used script languages. However its full of parenthesis syntax and lack of other features causes to be eliminated quickly from our list. Here is a square generator sample in Lisp:

```
(defun print-squares (upto)
    ( loop for i from 1 to upto
        do (format t "~A^2 = ~A~%" i (* i i))))
```

- *Perl* (Practical Extraction and Report Language) is a mature language with a fast interpreter. Also it is used to interpret data files with its built-in regular expressions, which is another added value for our case. Perl has a modular structure with some object-oriented features added to it later and is single thread without supporting multi-thread. Here is the print squares code in Perl:

```
for($i = 1 ; $i <= 10 ; ++$i) {
    print $i*$i, ' ';
}
```

- *Ruby* is a young but attractive full object oriented language including many features of Perl, with an intuitive syntax. It also supports multi-threading at the interpreting level which enables it to be used also in single thread operating systems.

```
for i in 1..10 do
    print i*i, " "
end
```

- *Tcl* is a widely used script language. It is portable and can be easily integrated with C. It is a modular language without object-oriented features. Here is the print squares example in Tcl:

```
set i 1
while {$i <= 10} {puts [expr $i*$i];set i [expr $i+1]}
```

- *Python* is a general purpose script language with object oriented features and very easy and intuitive syntax. It also support multi-threading. Here is the print squares example in Python:

```
for i in range(1,11): print i*i,
```

Finally Python was selected due to some details and also our background. Tcl was a good choice as it is already used in GiD and a large amount of experience was available. Lack of object-oriented features however prevented this choice. Perl also was considered for its maturity and performance, but again it is less object-oriented than Python and does not support multi-threading. Ruby at that time was completely new to us and also was considered to be somehow young.

There were some other reasons to chose Python. An existing well designed interface to Python was one important reason to be selected. Also there was some practical use of Python in finite element applications [10, 11].

9.5.3 Binding to Python

Boost Python library is used for binding Kratos to Python. This library provides an easy but powerful way to connect a C++ code to Python.

Now let us create a `Node` interface to Python and use it as a step by step introduction in using this library. The first step is to introduce the `Node` class itself. The following code introduces the `Node<3>` class as a `Node` identifier in Python:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer>("Node");
```

`class_` template introduces a class to Python. The first template parameter is the class to be imported. The second one is the handler in Python for objects of this type. In this case is the `Node<>::Pointer` which is a reference counted pointer. Boost Python library handles properly this type of pointer and binds it correctly to the memory manager of Python. The argument of constructor is the representative name of this class in Python. Python does not have templates so different instances of a template can be exposed to Python and not the template itself. For example here the instance `Node<3>` is exposed.

In Kratos `Node<>` is derived from `Point<>` and `IndexedObject`. We can keep this hierarchy by introducing `Node`'s bases in its definition:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType, IndexedObject > >("Node");
```

By introducing its bases, `Node` automatically inherites all imported methods of `Point<>` and `IndexedObject`. This class is exposed now and can be constructed by its default constructor. But we prefer to construct it by its id and coordinates. `init` provides a way to introduce a constructor and can be passed as another argument to the `class_` constructor:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>());
```

Another constructors can be added using `def` method of `class_`. For example a constructor by id and point can be exposed in the following way:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    ;
```

`def` uses a visitor pattern which makes it extendible to new concepts. Here `init` uses this pattern to expose a new constructor. Now we can create the `Node` in Python and calling its inherited methods from `Point<3>` and `IndexedObject`, but we cannot print it yet. To make Node printable, another statement must be added:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    .def(self_ns::str(self))
    ;
```

This statement uses the << operator already defined for Node to print its information. def can be used also to define member functions. For example Node.IsFixed(Variable) can be imported to Python as follow:

```
typedef Node<3> NodeType;

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    .def("IsFixed", &NodeType::IsFixed)
    .def(self_ns::str(self))
    ;
```

To import Kratos as a module, the following statements are necessary:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(Kratos) {
  // Module components will be defined here.
}
```

Adding the IndexedObject, the Point<3> and the Node<3> interfaces results in the first Kratos module:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(Kratos) {

AddPointsToPython();

class_<NodeType, NodeType::Pointer,
    , bases<NodeType::BaseType,
            IndexedObject > >("Node",
                init<int, double, double, double>())
    .def(init<int, const Point<3>& >())
    .def("IsFixed", &NodeType::IsFixed)
    .def(self_ns::str(self))
    ;

}
```

The Point interface is placed in another function and is just called here. This makes the code easier to read and also reduces the required memory to compile this file.

Compilation consists of compiling Boost Python library itself and above codes as a dynamic link library, then we must put them into Python dynamic link libraries folder.

In Python one must import this module to use all its components or just import the require components. Here is an example of using this module in Python:

```
from Kratos import *

# Constructing a node:
node = Node(1,2.00, 10.00, 0.00)
```

```
# Using X property inherited from Point:
node.X = 4.5

# Calling exposed IsFixed method
if(node.IsFix(TEMPERATURE)) :
    # Using << operator of Node
    print node
```

This simple case was just an introduction. Many other concepts from the Boost Python library are necessary to implement a real interface. Call policies, virtual and overloaded methods, exception handling and so on are examples of these concepts which are described in the library documentations.