Chapter 8

Finite Element Implementation

8.1 Elements

`Element`s and `Condition`s are the main extension points of Kratos. New formulations can be introduced into Kratos by implementing a new `Element` and its corresponding `Condition`s. This makes the `Element` an special object in our design.

8.1.1 Element's Requirements

An `Element` is used to introduce a new formulation to Kratos. To guarantee the extendibility of Kratos, adding an `Element` must be an easy task and without large modifications in the code. Encapsulating all data and procedures necessary to calculate local matrices and results in one object and using a clear interface is required to achieve this objective.

A user may use different `Element`s for different parts of the model and then assemble them separately or together. For example in modeling a multi floor structure, the user would use beam elements for frames and shell elements for floors. So these `Element`s must be compatible to be solved together as a complex system. For this reason the ability to use any `Element` in any part, or even mix them, is another requirement to be considered in the Kratos design.

`Element` has to have a very flexible interface due to the wide variety of formulations and different requirement they have. For example, some formulations need to calculate the stiffness matrix and also the righthand side vector in each solution step. Some others just need to calculate the stiffness matrix once and right hand side for each step. Sometimes having the damping matrix separately is needed to handle different time dependent strategies. These formulations are not only different in their local matrices, but also they need different data for their calculations. For example some need time and time step, some other the number of nonlinear iterations, or other parameters depending on the strategy chosen for analysis.

Easy to implement is another requirement for an `Element`. Finite element developers are usually less familiar with advanced programming language features and they would like to focus more on their finite element developing tasks. For this reason the main intention in designing Kratos is to isolate this parts from working with memory or an excessive use of templates. The idea is to provide a clear and simple structure for an `Element` to be implemented by finite element developers wishing to introduce a new formulation to Kratos.

Performance is a very important point to be considered at the time of designing an `Element`. In a finite element code `Element`s methods are called in nearly most inner loops of code. This means that any small fault in `Element`'s performance can cause great overhead in the program execution time. It is obvious that performance of a new `Element` is highly depended on its implementations, but sometimes a weak design can lead to serious bottlenecks in the performance of all `Element`s. We will see later how an elegant but not optimized interface can highly decrease the performance of `Element`s.

Memory efficiency is also important for `Element`s. Modeling a real problem with the FEM usually needs a large number of `Element`s to be created. For this reason any unnecessary overhead in memory used by each `Element` can cause a significant overhead in the whole memory used by program. This overhead, by the way, can restrict the maximum size of the model that can be analyzed in a machine. So efficiency in memory is considered to be very important and less important features must be reduced to keep `Element` as small as possible.

8.1.2 Designing Element

After reviewing the `Element`'s requirements, the next step is to design it. In Kratos an `Element` is an object which holds its data and calculates elemental matrices and vectors to be assembled and also can be used to calculate local results after the analysis. For example a thermal element calculates the local stiffness matrix and the mass matrix (if necessary) and give it to Kratos for assembly process. Also it can be used to calculate thermal flow after solving the problem. This definition provides a good isolation for `Element` related to rest of the code which is helpful for the proper encapsulation of `Element`.

`Element`s must be designed to be implemented independently and added easily to Kratos in order to guarantee the extendibility of Kratos. Also they must be compatible with each other in order to let users interchange them or even mix them together in a complex model. According to these two requirements the strategy pattern described in section 3.4.1 is what we are looking for. Applying this pattern to our problem results in the `Element`s' structure shown in Figure 8.1.

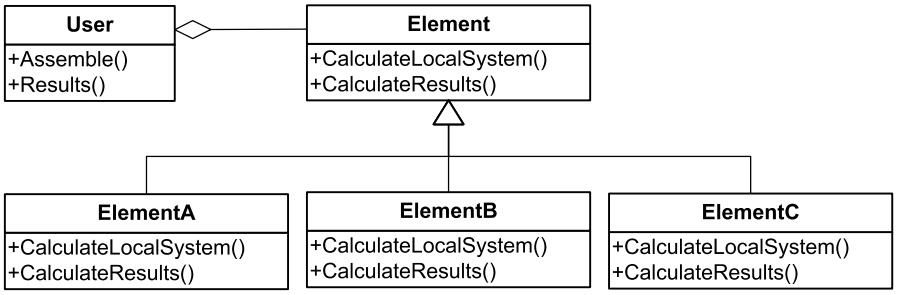

Figure 8.1: `Element`s' structure using strategy pattern.

Using this pattern each `Element` encapsulates one algorithm separately and also make them interchangeable as we want. User keeps a pointer to `Element` class which may point to any member of `Element`'s family and use the interface of `Element` to call different procedures.

The next concept in `Element`s design is its relation with the geometry. As described in section 5.3 a geometry holds a set points or `Node`s and provides a set of common operations to ease the implementation of `Element`s and `Condition`s. Each `Element` has to work with geometry and from

many points of view its an extended geometry with a finite element formulation as it is extension part. This relation can be translated in an object oriented philosophy as a parent and derived class relationship as can be seen in figure 8.2.

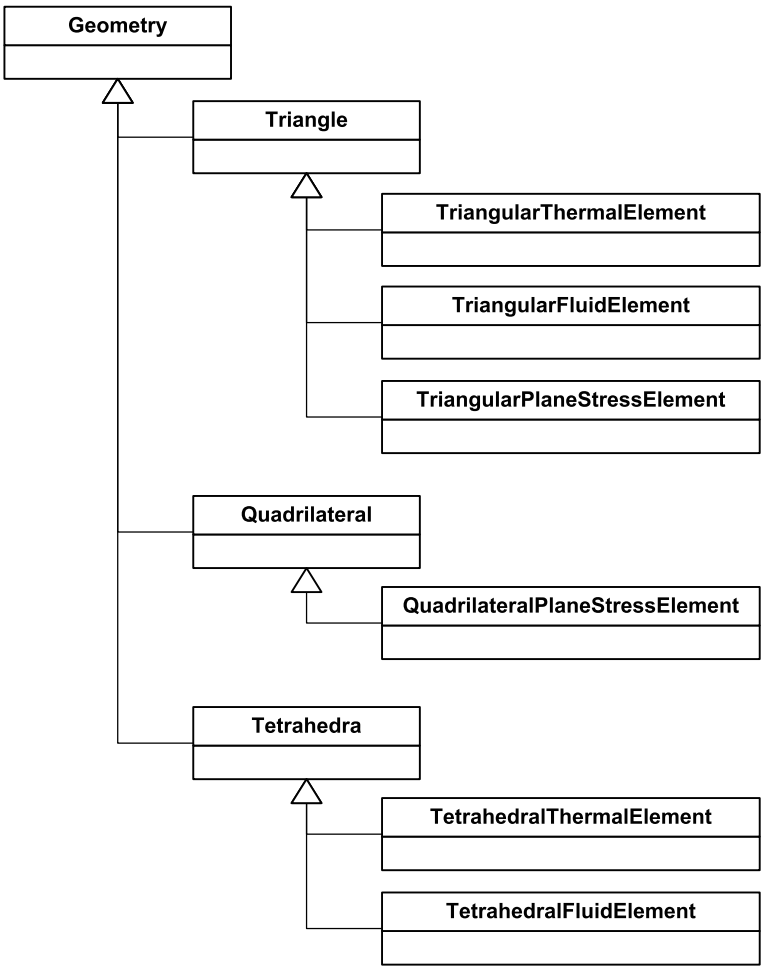

Figure 8.2: Deriving `Element` from geometry requires several `Element`s with same formulation but different geometries to be created.

This structure has the advantage that `Element`s access to geometry data is fast and increases the performance of elemental procedures. Beside this advantage there are two main disadvantages that make this structure unsuitable for our purpose. The first disadvantage is that applying a formulation to different geometries, requires several `Element`s to be implemented. The second drawback is that `Element`s with different formulation cannot share a geometry in memory.

In this structure each `Element` can be implemented to add a formulation to the geometry which is derived from. So different `Element`s must be implemented to extend a formulation to different geometries. For example a triangular plane stress element is derived from a triangle and has a plane stress formulation. Applying the same formulation to a quadrilateral requires another `Element`,

with nearly the same structure but derived from a quadrilateral to be written. This results in a significant overhead in the implementation and also in maintenance of `Element`s.

In a multi-disciplinary problem there are some situations when two interacting domains use the same mesh. In order to implement this possibility, different `Element`s should be able to share the same geometry. In this way the data transfer is minimized and the interpolation cost is eliminated. A simple example is a thermal and structural interaction. Mesh is used to create thermal elements and calculate the temperature over the domain. Then the same mesh is used to create structural elements and calculate the stresses and deformations in domain using the previously calculated temperature for temperature dependent materials. Without sharing geometries, a copy of all geometries must be created so that each set of geometries can be assigned to one domain. This increases the memory used by the application that can be avoided easily by sharing the geometries in mesh.

The alternative design is to use a bridge pattern. Introducing this pattern to our `Element`'s structure design results in the structure shown in Figure 8.3.

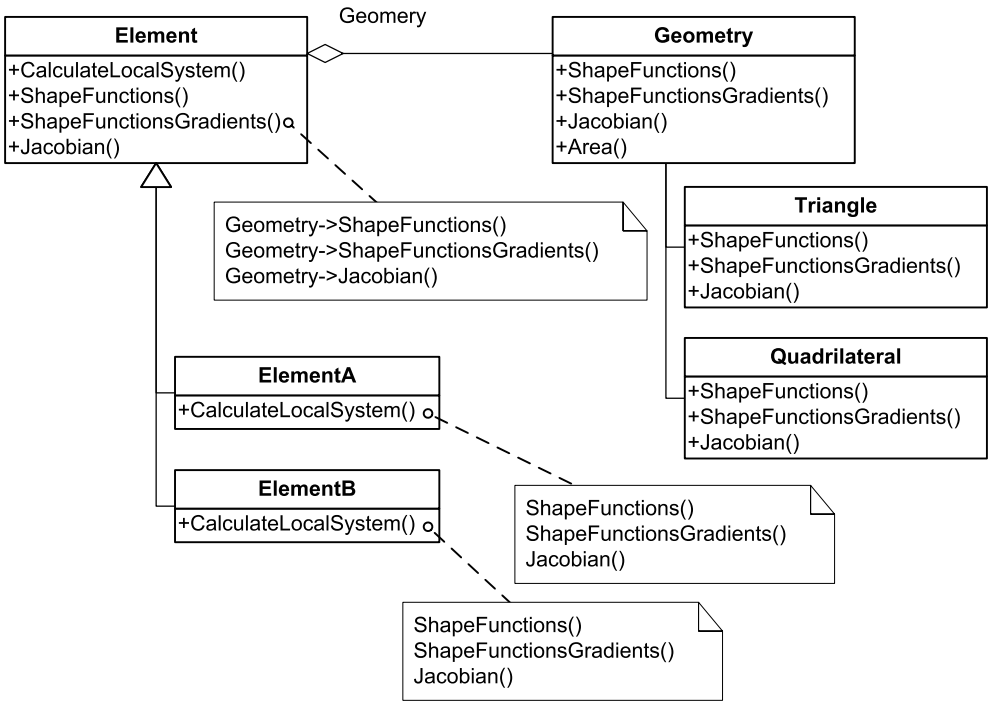

Figure 8.3: `Element`'s structure using the bridge pattern.

This pattern allows each `Element` to combine its formulation with any geometry. In this way less implementation is needed. Also having a pointer to geometry allows an `Element` to share its geometry with other ones. The only drawback of this structure is the time overhead comes from pointer redirection in memory. Having a pointer to geometry beside deriving from it creates a small overhead in accessing geometries' data respect to a direct derived `Element`. Though the efficiency in `Element` is crucial the complexity of the first approach imposes accepting the small different in performance and therefore we have implemented this second approach. A better solution is to

make `Element` a template of its geometry. Using templates provides good performance and also enough flexibility but it was considered to be too complex to be used by finite element users. As mentioned earlier an `Element` has to be easy to program with the less possible advanced feature of programming language. So finally the current structure with bridge pattern was selected.

There are some designs in which different `Element`s can be composed to create a more complex `Element` [70]. This approach can be simulated here using a *Composite* pattern. However this structure is not implemented yet in Kratos.

After designing the global structure now it is time to define interfaces. Here the finite element methodology helps in designing a generic interface. According to the finite element procedure, the strategy asks `Element` to provide its local matrices and vectors, its connectivity in form of equation id, and after solving also calls `Element` to calculate the elemental results. So `Element` has to provide three set of methods:

Calculate Local System The first set of methods are required to calculate local matrices and vectors.

Assembling Information These methods give information about the position of each row and column of the local system in the global system. This information comes from dof and `Element` provide it by giving its dofs or just their equation id.

Calculate It is used to calculate any variable related to an `Element` which usually are the results depending on gradients within the element.

An important issue here is the efficiency of these methods. An attractive form is to make these methods take their necessary parameters as their arguments and return their results as their return values:

```
Matrix CalculateLeftHandSide(ProcessInfo& rCurrentProcessInfo)
{
    //calculating stiffness matrix

    return stiffness_matrix;
}

// Assembling
for(int i = 0 ; i < number_of_elements ; i++)
    Assemble(elements[i].CalculateLeftHandSide(process_info));
```

It can be seen that this design is very natural and easy to use, but in practice produces a significant overhead in performance. Calling each method consists in creating a new matrix or vector, fill it an finally pass it by value as result. Creating a dynamic matrix or vector is a very slow process and passing them by value needs temporaries to be created which is time consuming. All these steps make this design very slow and therefore unacceptable. A better idea is passing the result matrix or vector by reference to these methods as additional arguments. In this way there are no temporaries for passing by value and there is no need to create a variable for the result inside each calculation method. A performance issue for this new design is the resizing of result matrices and vectors. In practice resizing dynamic matrices and vectors results to be very slow. A simple control of a given matrix or vector size before resize it can reduce the resizing overhead for cases that the given size is correct.

A set of methods are necessary for calculating local system matrices and vectors. Different procedures in finite element methods require different information in different analysis points from `Element`. For example a simple linear strategy requires the local matrices and vectors once to

assemble the global system. So a method which calculates local system components is enough to handle this strategy. But for a non-linear analysis, the strategy needs to get also the right hand side to calculate convergence. For these cases a method to calculate this right hand side component is necessary. Also there are some cases that the right hand side is not changing during the analysis and strategy only needs to update its left hand side component which requires a method for calculating only left hand side components.

Implementing only the first method to calculate local system components results in a calculation overhead for non-linear cases. For example calculating the convergence of the solution with a residual criteria, only needs the right hand side components to be updated and calculating all components can apply an unacceptable overhead to this procedure. Keeping only the interface for the left and right hand side components also produces calculation overhead. Usually for calculating each part of, local system the jacobian of the elements must be calculated which is a time consuming operation. Calculating right and left hand side components separately implies that jacobian must be calculated twice. So an optimum design is to keep both interfaces in parallel.

The drawback of this decision is the need for implementing duplicated methods. This problem can be solved by a more carefully implementation. One can create two private auxiliary methods: `LeftHandSide` and `RightHandSide` to calculate left hand side and right hand side matrices and vectors with the calculated jacobian as their input. Then `CalculateLocalSystem` can calculate the jacobian once and call these methods with this jacobian to calculate local system components. Also `CalculateLeftHandSide` and `CalculateRightHandSide` would calculate the jacobian and call their related method to calculate the local contribution. Here is an example of this implementation:

```
class MyElement
{
public:
  virtual void
  CalculateLocalSystem(MatrixType& rLeftHandSideMatrix,
                       VectorType& rRightHandSideVector,
                       ProcessInfo& rCurrentProcessInfo)
  {
    Matrix jacobian;

    Jacobian(jacobian);
    LeftHandSide(rLeftHandSideMatrix, jacobian,
                 rCurrentProcessInfo);
    RightHandSide(rRightHandSideVector, jacobian,
                 rCurrentProcessInfo);
  }

  virtual void
  CalculateLeftHandSide(MatrixType& rLeftHandSideMatrix,
                        ProcessInfo& rCurrentProcessInfo)
  {
    Matrix jacobian;

    Jacobian(jacobian);
    LeftHandSide(rLeftHandSideMatrix, jacobian,
                 rCurrentProcessInfo);
  }

  virtual void
```

```
  CalculateRightHandSide(VectorType& rRightHandSideVector,
                         ProcessInfo& rCurrentProcessInfo)
   {
    Matrix jacobian;

    Jacobian(jacobian);
    RightHandSide(rRightHandSideVector, jacobian,
                  rCurrentProcessInfo);
   }

  private:

   void LeftHandSide(MatrixType& rLeftHandSideMatrix,
                     Matrix& rJacobian,
                     ProcessInfo& rCurrentProcessInfo)
   {
       // Calculating left hand side matrix using given jacobian.
   }

   void RightHandSide(VectorType& rRightHandSideVector,
                     Matrix& rJacobian,
                     ProcessInfo& rCurrentProcessInfo)
   {
       // Calculating right hand side vector using given jacobian.
   }

};
```

Also it is important to mention that `Element`s not necessarily have to implement all these interfaces and they can be compatible with just one way and not providing the other. By the way, calling two separate methods in `CalculateLocalSystem` method of `Element` class can keep more compatible the `Element`s which are not providing the `CalculateLocalSystem` method and just provide `CalculateLeftHandSide` and `CalculateRightHandSide` methods.

Another issue is optimizing for symmetric or diagonal matrices. In Kratos local system matrices are defined as dense matrices in order to be more general. `Element`s with symmetric formulation also have to fill this dense matrix. The optimization can be done at the strategy level by assembling only half of this matrix in a symmetric global matrix to reduce memory usage and also assembling time. However the redundant time of filling all components of the dense matrix is unavoidable in order to keep `Element`s compatible with nonsymmetric strategies. Diagonal matrices can be treated as symmetric ones by keeping the optimization level in strategy and not in `Element`.

This interface is designed to be generic but its flexibility to support new algorithms also depends on its ability in passing different parameters necessary for different formulations. For this reason a variable base container is used to enable users pass any parameter to an `Element` using the VBI described before. `ProcessInfo` can be used to pass any parameter which is necessary for calculating local systems in an `Element`. The usual parameters are time, time increment, time step, non-linear iteration number, some global norms which are calculated over the domain, etc. Using `ProcessInfo` guarantees the flexibility which is necessary for the `Element` to be an extension point of Kratos.

According to the previous comments the following methods are designed:

`CalculateLocalSystem` This method calculates all local system components. It takes a left hand side matrix and a right hand side vector to put its result in them. `ProcessInfo` is passed to

provide the analysis parameters.

`CalculateLeftHandSide` This method calculates only the left hand side matrix. It takes a matrix to put its result in it. `ProcessInfo` is passed to provide the analysis parameters. `ProcessInfo` which provides analysis parameters.

`CalculateRightHandSide` Calculates right hand side component of local system. It takes a vector to put its result in it. `ProcessInfo` is passed to provide the analysis parameters.

`Element` also has to provide assembling information for `Strategy`. It has to provide the corresponding position of each local system row and column in the global equation system. `Strategy` then uses this information to properly assemble the local matrices and vectors in global equation system. This information comes from the `Dof` associated with each row or column of local system. `Strategy` by itself cannot find these equation because each `Element` may have different dofs and also may arrange them in different order. For example an structural element can define a local system with all displacement's components for the first `Node` then second `Node` and so on. Another structural element can arrange its local system by placing first the displacement's x component of all `Node`s, then the y components and their z components. So an `Element`'s task is to give its local system arrangement to `Strategy`.

`Element` can give an array of `Dof`s with the same order that local system is constructed, or get their associated equation ids and give them to `Strategy` as an array of indices. `Strategy` uses these indices to assemble a given local system into the global equation system. This part of `Element`'s interface consists of two methods:

`EquationIdVector` This method is used to directly give the global equation id related to each row or column of local system matrices and vectors. For example giving a vector $i = \{24, 5, 9\}$ means that the first element of the right hand side vector must be added to the 24th row of global system's right hand side or component k_{23} of the local stiffness matrix must be added to the component K_{59} of the global left hand side matrix. A `ProcessInfo` is passed to this method to provide any addition parameter needs for this procedure.

`GetDofList` This method gives `Element`'s `Dof`s in the same order as local system is defined. `Strategy` can use this list to extract the equation id related to each local position and then use them to assemble the `Element`'s local system components correctly. Like the previous method, it takes a `ProcessInfo` object as its argument which can be used to pass any additional information needed for this procedure.

It can be seen that both methods take `ProcessInfo` as their argument. This argument seems to be redundant but in practice there are situations that is really necessary. For example in solving a fluid using a fractional steps method [26], `Element` must know which is the current fractional step for providing the corresponding list of dofs or equation ids. Passing a `ProcessInfo` to these methods provides these additional parameters and guarantees the generality of the design.

Here is an example of `EquationIdVector` implemented for a generic structural element which can be used with different geometries in 2D and 3D spaces:

```
virtual void EquationIdVector(EquationIdVectorType& rResult,
                              ProcessInfo& rCurrentProcessInfo)
{
  unsigned int number_of_nodes = GetGeometry().size();
  unsigned int dimension = GetGeometry().WorkingSpaceDimension();
```

```
  unsigned int number_of_dofs = number_of_nodes * dimension;

  if(rResult.size()!= number_of_dofs)
    rResult.resize(number_of_dofs);

  for (int i = 0 ; i < number_of_nodes ; i++)
  {
    unsigned int index = i * dimension;

    rResult[index] =
      GetGeometry()[i].GetDof(DISPLACEMENT_X).EquationId();
    rResult[index + 1] =
      GetGeometry()[i].GetDof(DISPLACEMENT_Y).EquationId();
    if(dim == 3)
      rResult[index + 2] =
        GetGeometry()[i].GetDof(DISPLACEMENT_Z).EquationId();
  }
}
```

The third category of methods are devoted to calculating elemental variable which are used mainly for calculating post-analysis results. A simple example is calculating stresses in structural elements after obtaining the displacements in the domain. Users can ask `Element` to calculate additional results using its internal information and solving results. A flexible interface here is very important and can increase the generality of the code. A VBI can be used to provide a clear but flexible interface for these methods. Element developers can define a set of methods to calculate variables related to its `Element` and users can use them for specifying the variable they wants to calculate. Similarly to methods for calculating the local system, the result is passed as an additional argument in order to increase the performance and eliminate the redundant time necessary to create temporaries. Two sets of methods are defined for this task:

`Calculate` Can be used to calculate elemental variables. These methods are overloaded to support different types of variables to be calculated. Element developer can override them to implement the procedure necessary to calculate each elemental variable. They take the variable which a user wants to be calculated as their argument. If the variable is supported by `Element` it will give the result and it will do nothing if the variable is not related to this `Element`. The result also is passed as an additional argument to increase the performance and eliminate the overhead produced by creating temporary objects. Passing `ProcessInfo` to these methods provides a generic way to pass additional calculation parameters.

`CalculateOnIntegrationPoints` This set of methods calculate variables not for the whole `Element` but specifically at each integration point. The interface is the same as for previous methods. The variable to be calculated is given as an argument and the results as another argument. `ProcessInfo` provides any additional information necessary for calculation procedure.

Providing an standard way to access neighbors of `Element`s can be very useful for some algorithms. The problem is that for the rest of algorithms keeping the list of neighbors results in large overhead in total memory used. Keeping in mind the importance of memory efficiency in `Element`s these features are considered to be optional. So the first solution was to have arrays for neighbor `Node`s and neighbor `Element`s which are empty and fill them when they are necessary. This implementation was good but still the empty containers was producing memory overhead for simple `Element`s. In the current implementation these containers are omitted and neighbor `Node`s

and Elements are stored inside the elemental data container. In this way the overhead of empty containers are eliminated and the existing container is reused to hold this information. This solution can be used for any other feature that must be provided optionally but without any overhead for other Elements.

8.2 Conditions

Condition is defined to represent the conditions applied to boundaries or to the domain itself. In many codes conditions or specially boundary conditions are represented by an element with a formulation modified for boundary conditions. In Kratos also Conditions are designed very similar to Elements. They interact with Strategy in the same way as Elements. Strategy ask their local system components and also information for assembling process. The reason of using a different type and not Element itself is to clarify the different purpose of these two objects. In a usual finite element model, there are much more Elements than Conditions. For this reason some features that are considered to be too expensive in performance or memory consuming for Elements can be used for Conditions. Making Element and Condition two independent types allows additional features to be added to Condition without affecting Element.

8.2.1 Condition's Requirements

Condition like an Element is used to introduce new formulations into Kratos. So adding a new Condition must be an easy task and without great modification in code. Encapsulating all data and procedures necessary to calculate local matrices and results in one object and using a clear interface is required to achieve this objective.

A complex model usually has different type of Conditions in its boundary or domain. This requires Conditions to be compatible with each other in order to be assembled and solved together in a complex system. Another design point is to let users change Conditions or mix them in the model without problem.

Like Element, Condition has to have a very flexible interface due to the wide variety of algorithms and their different requirement. For example, most Conditions are applied to the right hand side component of the system but in some cases, like thermal radiation, they affect also the left hand side matrix of equation system. For this reason creating the interface only for right hand side component results in sever restriction in adding some Conditions. Conditions are not only different in local components, but also they need different data for they calculations. Hence a generic interface is necessary to guarantee the flexibility required for implementing different Conditions.

Condition must be easy to implement. Finite element developers are usually less familiar with advance programming language features and they like to focus more on their finite element developing task. For this reason excessive use of templates or other difficult concepts of programming language cannot be used for the Condition's implementation. The idea is to provide a clear and simple structure for a Condition to be filled by finite element developers easily.

For Condition performance is important but not so crucial as for Element. Its performance is important because it is usually called in very inner loops of global procedure. So any small fault in Condition's performance can cause large overhead in the program execution time. However its less important than the performance of Element because there are less Conditions in the model and the global overhead is less. So in designing Condition the intention is to avoid features producing bottleneck in the performance.

Memory efficiency is another design point to keep in mind. As mentioned before `Element`s have to avoid any redundant memory usage due to their large quantity in a model. Number of `Condition`s in a model usually is far less than number of `Element`s. This lets `Condition`s to provide features that are considered too expensive for `Element`s. However abusing memory by `Condition` can also produce a large overhead in memory usage and has to be avoided.

8.2.2 Designing Condition

`Condition` is very similar to `Element`, and hence the same methodology is used to design it. `Condition` is defined as an object which holds its data and calculates its local matrices and vectors to be assembled and also can be used to calculate local results after analysis. Defining `Condition` in this way isolates it from the rest of the code and helps towards its encapsulation.

Like `Element`s, `Condition`s must be designed to be implemented independently and added easily to Kratos in order to guarantee the extendibility of Kratos. Also they must be compatible with each other in order to let users mix them together in a complex model. The same strategy pattern used for `Element` is reused here. The figure 8.4 shows the structure for `Condition` applying the strategy pattern.

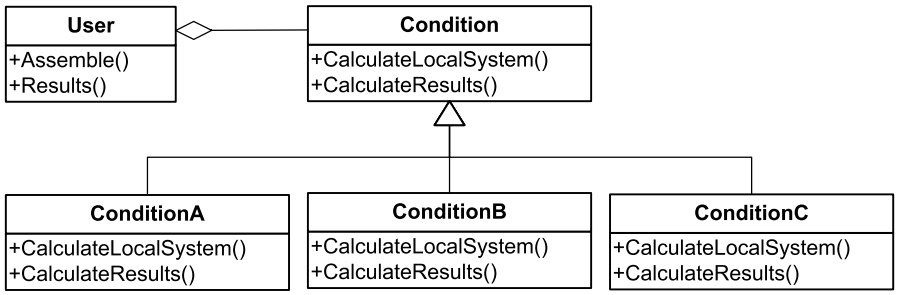

Figure 8.4: `Condition`'s structure using strategy pattern.

In this structure each `Condition` encapsulates one algorithm separately and also make them interchangeable as we want. The interface established by the `Condition` base class also make its derived class compatible with each other and enable user in mixing them together to model a multi-disciplinary problem.

`Condition` has a close relation to geometry. As explained for `Element`, deriving `Condition` from geometry, can increase the performance of geometries' data access but requires different `Condition`s to be implemented for a formulation applied to different geometries and also prevents `Condition` to share a geometry with `Element`s or other `Condition`s.

This structure reduces the flexibility of geometry and also produces unnecessary implementation overhead. So this structure is considered to be unsuitable because for `Condition` the flexibility is more important than a small increase in performance.

The alternative design is to use the bridge pattern. Introducing this pattern to our design, results in the structure shown in Figure 8.5.

This pattern allows each `Condition` to change its geometry and omits the strong relation of previous design. In this way less implementation is needed. Also having a pointer to geometry allows `Condition` to share its geometry with other `Condition`s or even with `Element`s without problem. The only drawback is the time overhead coming from pointer redirection in memory.

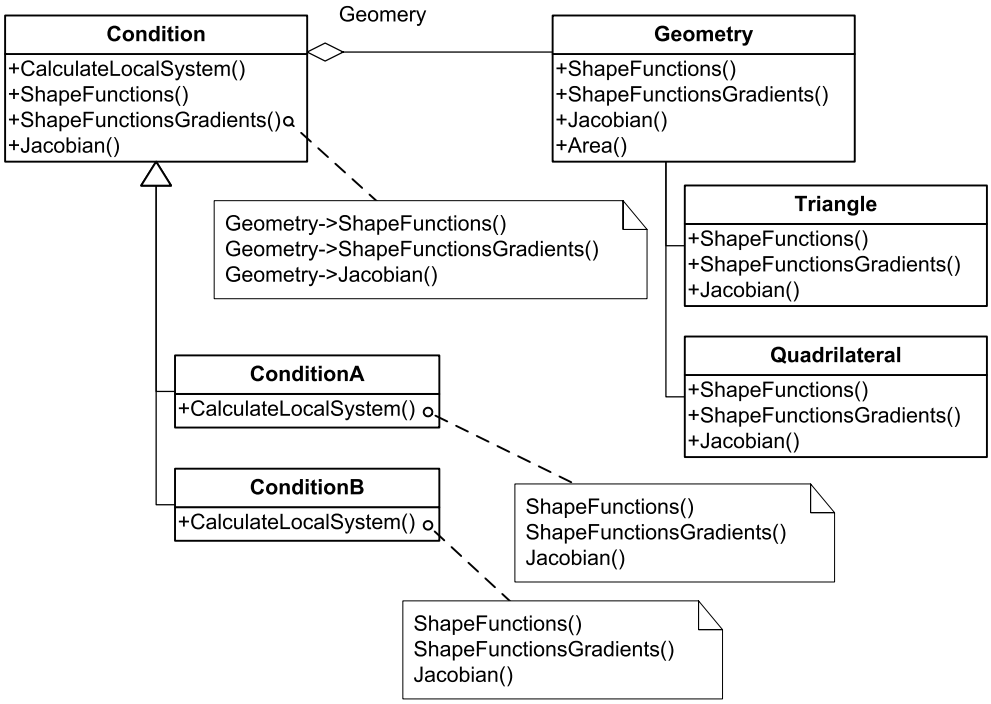

Figure 8.5: `Condition`'s structure using bridge pattern.

Having a pointer to geometry beside deriving from it produces an overhead in accessing geometries' data. There are other alternatives in designing `Condition`'s structure with enough flexibility and better performance but requires introducing advance features of programming language to `Condition` which are not acceptable in our design.

The interface of `Condition` is similar to the one designed for `Element`. Again here there are three categories of methods:

Calculate Local System The first set of methods are required to calculate local matrices and vectors.

Assembling Information These methods give information about the position of each row and column of the local system in the global system. This information comes from dof and `Condition` provide it by giving its dofs or just their equation id.

Calculate Is used to calculate any variable related to this `Condition` which usually are the results depending on gradients in the condition.

As described before in designing the `Element` interface, passing calculation parameters to `Condition` methods and getting the result as their return value produces a significant reduction the performance. To avoid this problem the result variable is also passed to each method. Passing the result vector or matrix by reference prevents the program from making temporaries and increases the performance. Also controlling the size of a given result matrix or vector and resize them if necessary can optimize the code performance.

`Condition` uses the same set of methods as `Element` to calculate the local system's matrices and vectors. `ProcessInfo` is used to pass any parameter which is necessary for calculating the local system in `Condition`. All methods are defined for working with dense matrix and strategies working with symmetric or other types of matrices must use a dense matrix to communicate with `Condition`. This part of the interface is defined by the following methods:

`CalculateLocalSystem` This method calculates all local system components. It takes a left hand side matrix and a right hand side vector to store the results and a `ProcessInfo` which provides the analysis parameters.

`CalculateLeftHandSide` This method calculates only the left hand side matrix. It takes a matrix to store the results and a `ProcessInfo` which provides the analysis parameters.

`CalculateRightHandSide` Calculates right hand side component of local system. It takes a vector to store the result and a `ProcessInfo` which provides the analysis parameters.

The second set of methods provide assembling information for `Strategy` which is the corresponding position of each local system row and column in global equation system. `Condition` can give an array of `Dof`s with the same order that local system is constructed, or get their associated equation ids and give them to strategy as an array of indices. Strategy uses these indices to assemble the local system into the global equation system. This part of `Condition`'s interface consists of two methods:

`EquationIdVector` This method is used to directly give the global equation id related to each row or column of local system matrices and vectors. A `ProcessInfo` is passed to this method to provides any additional parameter needs for this procedure.

`GetDofList` This method gives `Condition`'s Dofs in the same order as the local system is defined. Strategy can use this list to extract the equation id related to each local position and then use them to assemble the `Condition`'s local system components correctly. Like the previous method, it takes a `ProcessInfo` object as its argument which can be used to pass any additional information needed for this procedure.

Like `Element`, `Condition` uses its data container to store references to its neighbor `Node`s, `Element`s, or `Condition`s. This solution also can be extended to store the references to nearest `Element` or `Condition` in contact problems or other similar information.

8.3 Processes

Creating a finite element application consists of implementing several algorithms for solving different problems. In practice, each set of problems has their own solving algorithms. For example an steady state analysis algorithm is not the same as a transient algorithm. A one domain process is also different from a multi domain one and so on. While these algorithms are the heart of the code and flexibility and power of the code is depended on them, a good design to handle them in a generic way becomes very important.

A possible approach to handle algorithms in a finite element code is to provide some high level classes to handle different tasks in the code [33]. In Kratos, the `Process` class and its derived classes are defined to implement different algorithms and handle different tasks. Different processes may be used to handle a very small task like setting a nodal value to some complex one like solving a fluid structure interaction problem. Grouping some processes in a bigger one is also helpful specially to make a pack of small processes in order to handle a complex algorithm.

8.3.1 Designing Process

`Process` can be considered as a function class. `Process` is created and executed just like a function is called. The strategy pattern is used to design the family of processes. Figure 8.6 shows this pattern applied to `Process` structure.

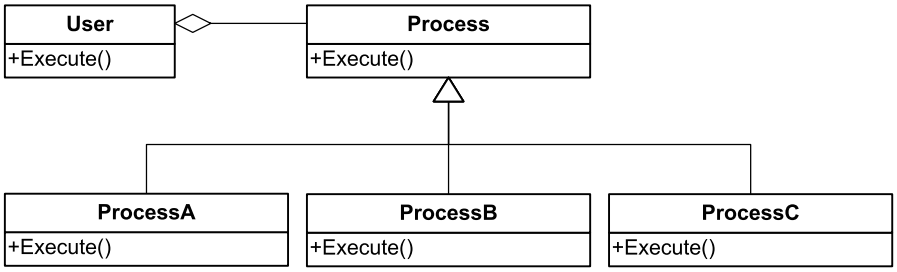

Figure 8.6: `Process` structure using strategy pattern.

Applying this pattern lets a `Process` to encapsulate an algorithm independently and also provide an standard interface which makes them to be replaceable with each other. Encapsulating each algorithm in one `Process` without modifying other parts of the code makes adding a new `Process` very easy and increases the extendibility of the library to new algorithms. The compatibility of processes with each other helps to customize the program flow and is useful in cases when user wants to interchange some algorithms.

Another feature to be provided by `Process` is the ability to combine different processes in one and use the resulting `Process` like a normal one. The composite pattern can be used to achieve this requirement. Applying this pattern to `Process` results in the extended structure shown in figure 8.7.

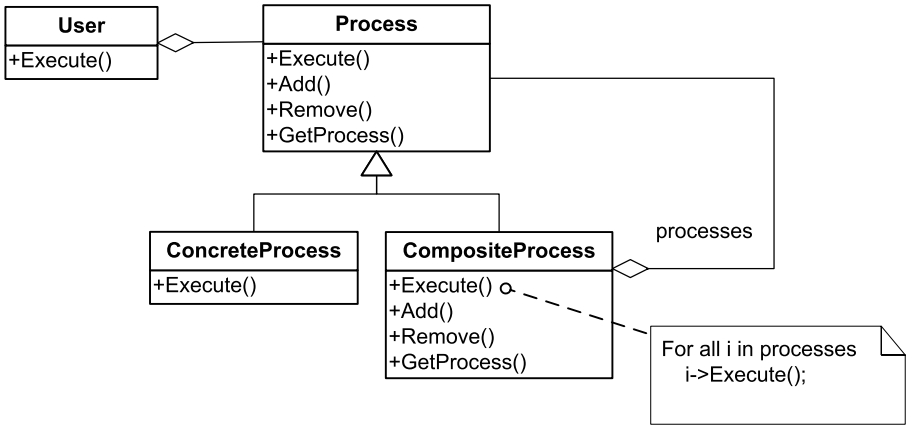

Figure 8.7: Applying composite pattern to the `Process` structure.

This structure allows users to merge different process in one and use it like an ordinary process.

In practice this structure is considered to be too sophisticated for our purpose. The composite pattern provides an interface for changing the children of each composite object. In order to simplify the implementation of new processes and the total implementation of the structure, the interface for changing sub-processes has been removed and `CompositeProcess` must get all its sub-processes with their other parameters at creation time. However this interface can be added in the future. Figure 8.8 shows the reduce structure.

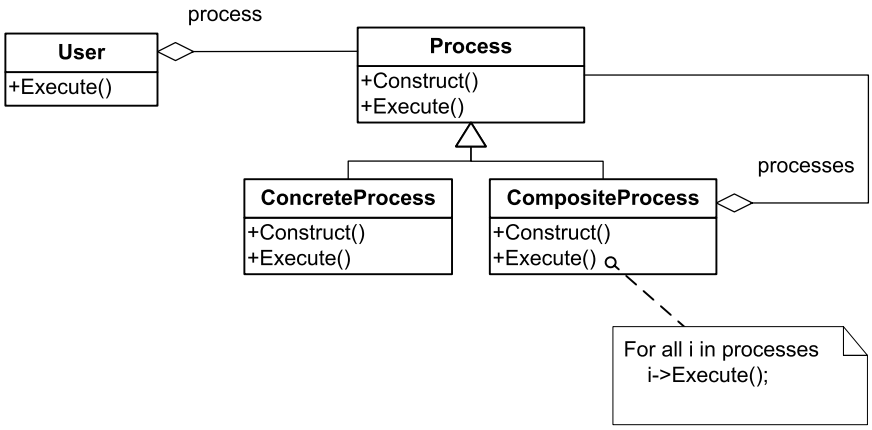

Figure 8.8: The reduced composite structure for `Process`.

The process interface is relatively simple. `Execute` method is used to execute the `Process` algorithms. While the parameters of this method can be very different from one `Process` to other there is no way to create enough overridden versions of it. For this reason this method takes no argument and all `Process` parameters must be passed at construction time. The reason is that each constructor can take different set of argument without any dependency to other processes or the base `Process` class.

8.4 Solving Strategies

After designing `Process` and its derived classes, we will focus in an important family of processes which are dedicated to manage the solving task in the program.

The `SolvingStrategy` is the object demanded to implement the "order of the calls" to the different solution phases. All the system matrices and vectors will be stored in the strategy, which allows to deal with multiple LHS and RHS. Trivial examples of these strategies are the linear strategy and the Newton Raphson strategy.

`SolvingStrategy` is derived from `Process` and use the same structure as shown in figure 8.9. Deriving `SolvingStrategy` from `Process` lets users to combine them with some other processes using composition in order to create a more complex `Process`. The strategy pattern used in this structure lets users to implement a new `Strategy` and add it to Kratos easily which increases the extendability of Kratos. Also lets them selecting an strategy and use it instead of another one in order to change the solving algorithm, which increases the flexibility of Kratos.

Composite pattern is used to let users combining different strategies in one. For example a fractional step strategy can be implemented by combining different strategies used for each step in

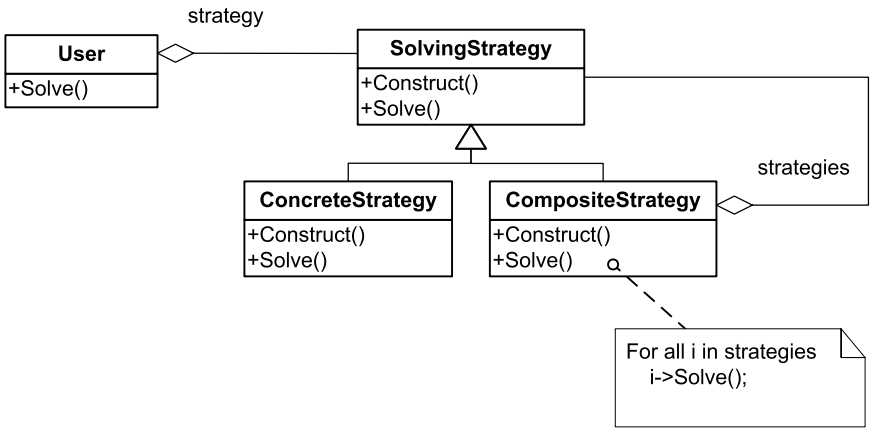

Figure 8.9: `SolvingStrategy` uses the structure designed for `Process`.

one composite strategy. Like for `Process`, the interface for changing the children of the composite strategy is considered to be too sophisticated and is removed from the `Strategy`. So a composite structure can be constructed by giving all its components at the constructing time and then it can be used but without changing its sub algorithms.

The interface of `SolvingStrategy` reflects the general steps in usual finite element algorithms like prediction, solving, convergence control and calculating results. This design results in the following interface:

`Predict` A method to predict the solution. If it is not called, a trivial predictor is used and the values of the solution step of interest are assumed equal to the old values.

`Solve` This method implements the solving procedure. This means building the equation system by assembling local components, solving them using a given linear solver and updating the results.

`IsConverged` It is a post-solution convergence check. It can be used for example in coupled problems to see if the solution is converged or not.

`CalculateOutputData` Calculates non trivial results like stresses in structural analysis.

Strategies sometimes are very different from each other but usually the global algorithm is the same and only some local steps are different. The template method pattern helps to implement these cases in a more reusable form. As mentioned before, this pattern defines the skeleton of an algorithm separately and defers some steps to subclasses. In this way the template method pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Applying this pattern to `SolvingStrategy` results in the structure shown in figure 8.10.

This structure is suitable when the algorithm is not changing at all but in our case the algorithm varies from one category of strategies to another. For this reason in order to reduce the dependency of the algorithm and its steps a modified form of the bridge pattern is applied to this structure. Different steps for solving template methods are deferred to two other objects which are not derived from `Strategy`: `BuilderAndSolver` and `Scheme`. Figure 8.11 shows this structure.

The main idea of using these two additional set of objects was to increase the reusability of the code and prevent users from implementing a new `Strategy` from scratch. In practice this

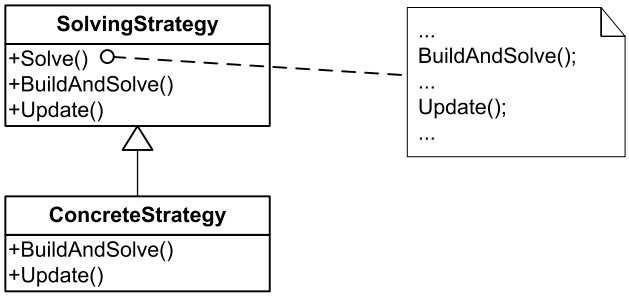

Figure 8.10: Template Method pattern applied to solving strategy.

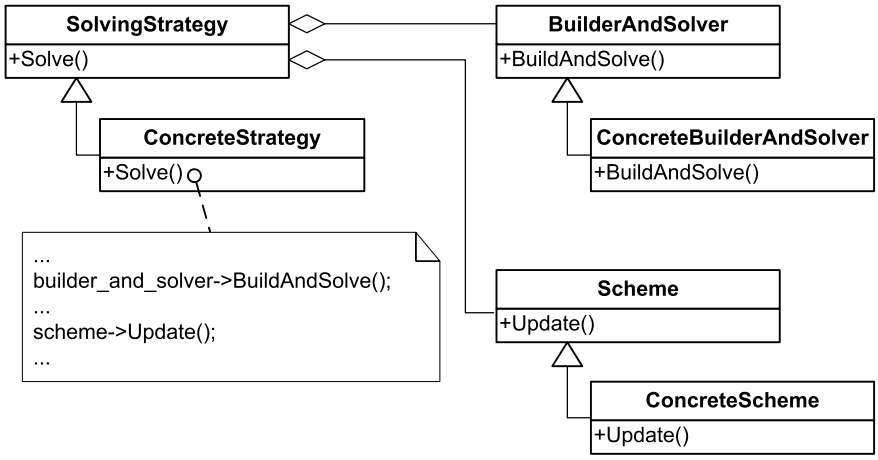

Figure 8.11: Deferring different parts of the algorithm to `BuilderAndSolver` and `Scheme`.

structure can support usual cases in finite element methodology but still advanced developers have to configure their own `Strategy` without using `BuilderAndSolver` or `Scheme`. For this reason in the current structure both approaches can be used to implement a solving algorithm.

8.4.1 BuilderAndSolver

The `BuilderAndSolver` is the object demanded to perform all of the building operations and the inversion of the resulting linear system of equations. The choice of grouping together the solution and the building step is not necessarily univocal. This choice was made in order to allow a future parallelization of the code, which should involve both the linear system solution and the Building Phase.

Due to its features `BuilderAndSolver` covers the most computational intensive phases of the overall solution process. This will clearly require low level tuning in order to ensure high performance. A typical user is not required to understand the implementation details for this class. Nevertheless the comprehension of the role of this object is necessary.

`BuilderAndSolver` needs a linear solver to solve its constructed equation system. In order to

give the possibility of assigning any linear solver to any `BuilderAndSolver` a bridge pattern is used to connect these two sets of classes. In this way `BuilderAndSolver` can use any linear solver available.

The interface of `BuilderAndSolver` provides a complete set of methods to build the global equation system or its components separately. It also provides methods for building the system and solving it or rebuilding just the left hand side or the right hand side and solve the updated equation system. This interface consists of the following methods:

`BuildLHS` Calculate the left hand side matrix of the global equation system.

`BuildLHS_CompleteOnFreeRows` Builds the rectangular matrix related to all free dofs, adding also the columns related to fixed dofs.

`BuildLHS_Complete` Gives the complete left hand side matrix regardless to fixed dofs.

`BuildRHS` Calculates and gives the right hand side vector of the global equation system. This method is useful in cases that left hand side matrix is the same for different solution steps but the right hand side is changing.

`Build` Builds the whole equation system. This method gives the possibility to calculate both sides at the same time and avoids duplicated calculation that must be done when calculating each component separately.

`ApplyDirichletConditions` In some strategies, for example for standard linear solutions, the Dirichlet condition can be applied efficiently by some operation over Dirichlet partition of the equations system. This can be done by this method.

`SystemSolve` Uses the linear solver to solve the prepared equation system.

`BuildAndSolve` Calling this method is equivalent to calling `Build` and then `SystemSolve` for most algorithms. It can be also used to implement algorithms that build the system while solving it, like the advancing front solution method.

`BuildRHSAndSolve` This methods is useful for updating just the right hand side and solve the equation system.

`CalculateReactions` Calculates the reaction at fixed degrees of freedom.

There are also several methods for initializing the internal system matrices and vectors and also to remove them from memory if it is necessary. `Strategy` can use this interface to implement its algorithm using any of the procedures defined above.

8.4.2 Scheme

`Scheme` is designed to be the configurable part of `Strategy`. It encapsulates all operations over the local system components before assembling and updating of results after solution. This definition is compatible with time integration schemes, so `Scheme` can be used for example to encapsulate the Newmark scheme. By the way definition is more general and can be used to encapsulate other similar operation over solution component.

According to the template method pattern the important steps of the solving procedure in usual finite element strategies is used to design the interface of scheme. Usually a finite element solving strategy consists of several steps like: initializing, initializing and finalizing solution steps, initializing and finalizing non linear iterations, prediction, update and calculating output data. Considering the steps mentioned before, the interface of `Scheme` is designed as follows:

`Initialize` This method is used for initializing `Scheme`. This method is intended to be called just once when `Strategy` is initializing.

`InitializeElements` Is used to initialize the `Element` by calling its `Initialize` method when `Strategy` is initializing.

`InitializeSolutionStep` `Strategy` calls this method at the beginning of each solution step. This method can be used to manage variables that are constant over time step. For example time-scheme constants depending on the actual time step.

`FinalizeSolutionStep` This method is called by `Strategy` at the end of a solution step.

`InitializeNonLinIteration` It is designed to be called at the beginning of each non linear iteration.

`FinalizeNonLinIteration` This method is called at the end of each non linear iteration.

`Predict` Performs the prediction of the solution.

`Update` Updates the results value in the data structure.

`CalculateOutputData` This method calculates the non trivial results.

8.5 Elemental Expressions

Finite element methodology usually consists of first converting the governing differential equation to its weak form, then its discretization over an appropriate approximation space, and finally the derivation of matrix forms as elemental contributions. Zimmermann and Eyheramendy [107, 39, 40, 38] have developed an environment for automatic symbolic derivation from the variational form to matrix form and integrate it into a unified environment with modeling tools [105]. Nowadays several computer algebra systems like *Matlab* [68], *Mathematica* [103], and *Maple* [66] can do this type of symbolic derivations. In Kratos the first part of changing the variational equation to weak form is dedicated to previous tools and only a set of tools is designed and implemented to help users converting their weak form to matrix form as elemental contributions.

Elemental expressions are designed and implemented to help users in writing their weak form expressions in `Element`. The main idea is to create a set of classes and overloaded operator to understand a weak form formulation and calculate the local matrices and vectors according to it.

For example in a simple heat conduction problem the governing equation is:

$$-\nabla^T \mathbf{k} \nabla T + Q = 0$$

where T is the temperature over domain and Q is the heat sources over domain. Converting this equation to its weak form results in the following equation [104]:

$$\mathbf{ST} + \mathbf{f} = \mathbf{0}$$

where the elemental matrix $\mathbf{S}$ is:

$$S_{ij} = \int_\Omega (\nabla N_i)^T \mathbf{k} \nabla N_j d\Omega$$

and the elemental right hand side vector $\mathbf{f}$ is defined as follows:

$$f_i = \int_\Omega N_i Q d\Omega + \int_{\Gamma_q} N_i \bar{q} d\Gamma$$

For an isotropic material the conductivity k can be extracted from the integral and the resulting equation is:

$$S_{ij} = k \int_\Omega (\nabla N_i)^T \mathbf{I} \nabla N_j d\Omega$$

or:

$$S_{ij} = k(\nabla_i N_l, \nabla_j N_l)$$

This equation can be implemented in `Element` by the following code:

```
for(int i=0 ; i < nodes_number ; i++)
  for(int j=0 ; j < nodes_number ; j++)
    for(int l=0 ; l < integration_points_number ; l++)
    {
      Matrix const& g_n = shape_functions_gradients[l];
      for(int d=0 ; d < dimension ; d++)
        rLeftHandSideMatrix(i,j) += k * g_n(i,d)*g_n(j,d) * w_dj;
    }
```

Using elemental expressions the same formulation can be written in a simpler form as:

```
KRATOS_ELEMENTAL_GRAD_N(i,l) grad_Nil(expression_data);
KRATOS_ELEMENTAL_GRAD_N(j,l) grad_Njl(expression_data);

noalias(rLeftHandSideMatrix) = k * (grad_Nil, grad_Njl) * w_dj ;
```

It can be seen that the later form is conforming with the symbolic notation of equations which makes it much easier to implement. The overloading operators provided in C++ is the start point for implementing the code necessary to understand this notation, but simple overloading results, poor performance due to the redundant temporary objects that creates. Expression template technique described in section 3.4.2 can be used to convert above expression to previous hand written form automatically. Template metaprogramming described in section 3.4.2 also is used to impose the tensorial notation. All these techniques are used to evaluate the symbolic notation and generate an specialized code for each case. Here are examples of vector ”,” overloaded operators:

```
template<unsigned int TIndex1,
         unsigned int TIndex2,
         class TExpression1,
         class TExpression2,
         class TVectorType1,
         class TVectorType2>
typename result_type
  operator,(Elemental1DExpression<TIndex1,
                                  TExpression1,
                                  TVectorType1> const& rVector1,
            Elemental1DExpression<TIndex2,
                                  TExpression2,
                                  TVectorType2> const& rVector2)
  {
    return outer_prod(rVector1()(), rVector2()());
```

```
  }

template<unsigned int TIndex1,
         class TExpression1,
         class TExpression2,
         class TVectorType1,
         class TVectorType2>
double
  operator,(Elemental1DExpression<TIndex1,
                                  TExpression1,
                                  TVectorType1> const& rVector1,
            Elemental1DExpression<TIndex1,
                                  TExpression2,
                                  TVectorType2> const& rVector2)
  {
    return inner_prod(rVector1()(), rVector2()());
  }
```

The first overloaded version will be used in cases when two vectors have different indices and implements an outer product of these two vectors. While the second version will be used when two given expressions have a same index and implements an inner product of these two vector. It is important to mention that the first version returns the expression and not the calculated matrix and uses the expression template technique to optimize its efficiency.

There are similar operators implemented to handle different cases of matrix operations depending on their indices. Also the integration over domain is added for simplifying the elemental expressions even further.

In the current version of Kratos, elemental expressions are still in experimental phase. However some benchmarks have shown that their efficiency is comparable with hand coded `Elements` as supposed to be.

8.6 Formulations

Kratos was designed to support elemental approaches in finite element methods. For some problems elemental approach results to be less suitable than other approaches like nodal formulations. `Formulation` is defined as a place for implementing all these approaches. `Formulation` is not implemented yet, but is considered to be one of the future features of Kratos.