Chapter 6

WoodFrameSolver Architecture

The purpose of the chapter is to present the architecture of the WoodFrameSolver program. The program is designed to read finite element input files generated either by the WoodFrameMesh pre-processor or by SAP 2000 version 7.xx. The input files are written in .S2K format. WoodFrameSolver is an object oriented finite element analysis platform developed in C++ by the WoodFrameSolver team at Virginia Tech. Under the current capability, the program can perform linear elastic static finite element analysis, but the platform developed is very flexible and may be extended to incorporate new elements and analysis capabilities. In addition to the architecture discussion, this chapter also presents some ideas for extending the platform to incorporate new features. Discussion of the comparison of the results obtained by analyzing example models using WoodFrameSolver and SAP 2000 is made at the end of the chapter. The detailed results and the comparison charts (for bandwidth and analysis speed) are presented in Appendix B.

6.1 WoodFrameSolver Overview

The program performs several operations to analyze a finite element model, but this entire process can be broken down into three key major operations which are as follows: (1) reading the input file and generating the analytical model, (2) analysis of the

generated model, and (3) printing output to a file. These operations are universal and are applicable to any structural analysis program irrespective of the nature of the design philosophy being followed in developing the code i.e. procedural or object oriented. As already mentioned, WoodFrameSolver is designed with object oriented design philosophies, and the above steps are achieved by performing these operations on various class objects instantiated at run time. Table 6.1 presents the list of all the classes implemented in WoodFrameSolver required in the above three steps. Table 6.2 presents the list of all the other classes (matrix, axes-transformation and string classes) implemented in WoodFrameSolver. A detailed discussion of these steps is made in the following sub-sections.

Table 6.1: List of classes

1.) Classes involved in the generation of an analytical model from the input data

- class KSS2KFileReader
- class KSModelBuilder
- class KSModel
  - class KSNode
  - class KSRestraint
  - class KSSpring
  - class KSConstraint
    - class KSEqualConstraint
    - ….
  - class KSMaterial
    - class KSElasticMaterial
    - ….
  - class KSSection
    - class KSFrameSection
    - class KSShellSection
    - class KSNonLinearLinkSection (currently linear properties)
    - ….

Table 6.1: List of classes (…Continued)

class KSElement

class KSFrameElement

class KSShellElement

class KSNonLinearLinkElement (currently linear)

…..

class KSLoad

class KSNodalLoad

class KSElementalLoad

class KSFramePointLoad

class KSFrameDistributedLoad

class KSShellPressureLoad

….

class KSLoadCase

class KSStaticForceLC

….

class KSEcho

2.) Classes involved in the analysis of the model

class KSAnalysisCase

class KSStaticAnalysisCase

….

class KSDOFNumberer

class KSPlainNumberer

class KSPacManReNumberer

….

class KSSOEandSolver

class KSBandedSOEnSolver

….

3.) Classes involved in printing output to file

class KSResponse

Table 6.2: List of other classes

1.) Matrix classes

class KSMatrixBase

class KSFullMatrix

….

2.) Axes-Transformation class

class KSAxesTransformation

3.) String class

class KSString

4.) Purge utility class

class KSUtilitySTL

6.1.1 Modeling

Generation of a valid analytical model is done using the *KSS2KFileReader*, *KSModelBuilder*, *KSEcho* and *KSModel* class objects. The *KSS2KFileReader* class is designed to read SAP 2000, version 7.40 input formats. This class requires the input file and the *KSModelBuilder* object as input. This input is provided to the class inside the main program via its constructor. The method ReadFile is called inside the main program once the *KSS2KFileReader* is instantiated and the above two parameters are passed. The ReadFile method calls several methods to read different blocks of input data and passes the input data to the *KSModelBuilder* class as they are read. The checks to ensure the valid format, i.e., SAP 2000 version 7.40, are performed inside the *KSS2KFileReader* class. So far, the design of the *KSFileReader* class has proved very extensible in adding the methods to read new blocks of data.

The *KSModelBuilder* class acts as an interface between the *KSS2KFileReader* and the *KSModel* and the *KSEcho* class as shown in Figure 6.1. This class is instantiated inside the main program and requires a *KSEcho* object to be passed via its constructor. This

class performs the next level of error checks on the input data. If any error is found in the input data, it is not passed to *KSModel*, and instead the error is passed to *KSEcho* object. In any case, the input data is printed inside the *KSEcho* file[1]. The map[2] container used inside *KSModel* to contain the input data facilitates the process of error checking, inside the *KSModelBuilder* class. The data is stored, printed and checked for errors on the fly as compared to first storing, then performing error checks by iterating and then printing. This makes the process more efficient both in terms of code writing and execution time. Table 6.3 lists the error checks that are currently performed on the input data. Similarly, additional errors may be identified and trapped as the program is further developed.

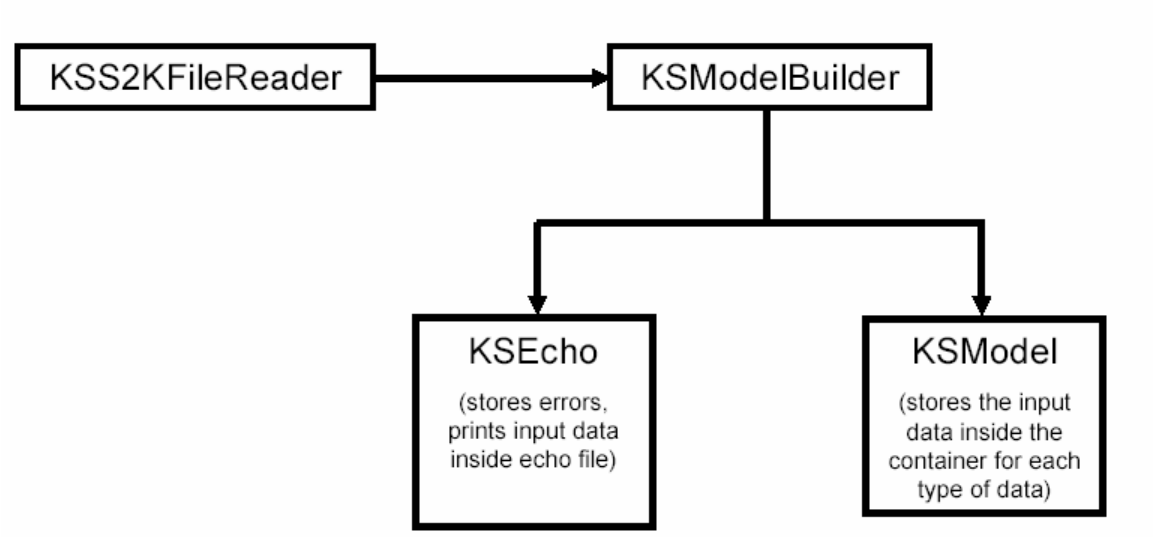

Figure 6.1: KSModelBuilder acting as interface between KSEcho and KSModel

Table 6.3: Errors trapped inside KSModelBuilder class

1.) Duplicate node no or node no <= 0.
2.) Check if local axes node exists.
3.) Check if restrained node exists.
4.) Check if duplicate restraints are applied.

[1] An echo file contains the input data read from the input file and the list of errors in the input data (if any)
[2] A storage class in C++ standard template library

Table 6.3: Errors trapped inside KSModelBuilder class (…Continued)

5.) Check if constraints have non existent nodes.
6.) Check for invalid spring stiffnesses.
7.) Check if springs are attached to non existent nodes.
8.) Check for invalid material properties i.e. elasticity modulus < = 0, poisson's ratio < 0 etc.
9.) Check for invalid frame section properties.
10.) Check for non existing material for a frame section.
11.) Check for duplicate frame sections.
12.) Check for invalid shell section properties.
13.) Check for non existing material for a shell section.
14.) Check for duplicate shell sections.
15.) Check for invalid node link properties
16.) Check for duplicate node link section
17.) Check for frame element errors i.e. incorrect releases, node numbers and section name.
18.) Check for shell element errors i.e. node numbers and section name
19.) Check for quadrilateral shell nodes not in one plane.
20.) Check for node link element errors i.e. node numbers and section name.

The *KSModel* class acts as storage for all the input data required for the analysis. This class uses an associative container (map) to store the input data. This proves to be very efficient as the input data for the analysis is provided in blocks, and these blocks are associated with each other. For example, nodes are associated to elements; materials are associated to sections which in turn are associated to elements. Map containers require the user to define a key (accessor) object as the first entry and the value object as second. The values are accessed using their key values. The input data is stored inside these

associative containers in the form of pointers[3] and hence are dynamically linked with each other.

6.1.2 Analysis

Once the *KSModel* class object is filled with input data, the analysis class comes into action. The program currently provides the capability of linear static finite element analysis and hence this discussion is limited to that. The *KSStaticAnalysisCase* object is used to perform this analysis. This class is a derived class of *KSAnalysisCase*. The attributes and methods common to linear static, linear dynamic, nonlinear dynamic and other type of analysis are present in this base class. Linear static analysis is performed by instantiating an object of type *KSStaticAnalysisCase.* The constructor for this class requires a *KSModel* class object, a *KSDOFNumberer* class object, *KSSOESolver* class object and *KSResponse* class object to be passed. Various important methods defined in all of these classes are discussed in the WoodFrameSolver architecture commentary.

An instance of *KSDOFNumberer* class defines the type of equation numbering scheme to be used during the analysis. Currently, this class provides two numbering schemes. One is a simple plain numberer, *KSPlainNumberer*, which simply uses the same equation numbering as generated from the input data. The second equation numbering scheme implemented was developed by Dr. Ed Wilson[4] in FORTRAN. It was translated into C++ and is implemented inside *KSPacManRenumberer*. The use of this renumberer decreases the bandwidth of the global stiffness matrix and hence the requirement on storage. It also provides for vastly increased execution speeds.

The *KSSOESolver* is an abstract base class and currently defines only one derived class which is *KSBandedSOEnSolver*. An instance of *KSBandedSOEnSolver* class is used for forming the global stiffness matrices and load vectors, solving them, and returning the displacements. This banded solver acts as a wrapper and in reality it calls

[3] A pointer is a memory address of any object. Pointers are variables that hold memory addresses in C++. They provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages. They are also useful for passing parameters into functions in a manner that allows a function to modify and return values to the calling routine.

[4] Dr. Ed Wilson is one of the preeminent researchers in the field of computer aided structural analysis. The revolutionary SAP program was first released by him in 1970.

subroutines from Intel math kernel library[5], written in FORTRAN, to solve the system of equations. These routines are implemented through the use of dynamic link libraries (DLLs) and are extremely efficient.

An instance of *KSStaticAnalysisCase* class is used to analyze the input model. Table 6.4 presents the steps involved in the static analysis. These steps are performed inside a method named Analyze which is defined as a pure virtual method and has a different body for different derived classes.

Table 6.4: Steps involved in the static analysis

1.) Determine equation numbers
2.) Determine bandwidth
3.) Get static load cases (fill right hand side of the equation [A]{x} = {B})
4.) Form global stiffness matrix
5.) Solve system of equations
6.) Get displacement vector, {x}
7.) Get element forces

6.1.3 Response

Once the system of equations has been solved, the next step is the processing of results in .txt format. Inside the WoodFrameSolver program, these results are processed by the methods provided by class *KSResponse*. These methods are called inside the Analyze method of *KSAnalysisCase* child classes to print the node displacement and element force data at run time. Currently the program generates only this information. The other information which the *KSResponse* class generates is: (1) bandwidth, (2) separate analysis times for each operation, and (3) total analysis time etc.

[5] The Intel® Math Kernel Library (Intel® MKL) is a set of highly optimized, thread-safe, mathematical functions for engineering, scientific and financial applications requiring high performance. Intel® MKL utilizes automatic runtime processor detection to execute code that has been specifically optimized for Intel® Itanium® 2, Intel® Xeon®, Intel® Pentium® III, and Intel® Pentium® 4 processors.

6.2 Architecture Extensibility

One of the major aims of writing this finite element analysis code using object oriented C++ was the advantage of easy extensibility, and hence the program was designed with this requirement in mind. Discussion on the extensibility of the program for incremental analysis and for non-linear material is provided in the WoodFrameSolver architecture commentary. This section focuses on other extensible aspects of the program along with some example code.

6.2.1 Adding New Elements

Currently the program provides the following 3D elements: frame, 3 and 4 node shell, spring and non-linear link[6]. All of these elements are derived from a base class called *KSElement* which contains common attributes and methods for all the elements. The program is designed to accommodate new elements as desired by the user. Figure 6.2 presents an example of how a new element can be added to the program. The new element could be any element, such as a plane element, eight node brick element, etc. Depending upon the element type the programmer will need to write additional methods and attributes. The methods shown in the Figure 6.2 are some of the methods which are defined as virtual in the base class and must be defined in the derived classes.

[6] Currently limited to linear behavior

```
class KSNewElement: public KSElement
{
public:
        KSNewElement();
        KSNewElement(int        Tag, std::string  Name,
                     std::deque<KSNode*>   Nodes,
                     const KSNewSection       *Sect,
                     const KSMaterial         *Mat,
                       ……..
                       ……..);
        ~KSNewElement();

        KSFullMatrix*                GetTangent(void);
        const std::deque<double>*    GetElementForces(int RHSLoc);
        void                         UpdateElementForces(int RHSLoc);
        void                         AddLoad(const std::string      &LoadType,
                                             std::deque<double> &LoadData,
                                             const double           &Factor,
                                             int                     RHSLoc);

        void                         DisplaySelf(std::ofstream &OutputFile);
        …….
        …….
};
```

Figure 6.2: Interface of a new derived element class

6.2.2 Adding New Constraints

The program defines an abstract base class *KSConstraint*. The methods necessary to be implemented in all the derived classes are defined in this class. The program currently provides only one constraint class called *KSEqualConstraint*. Figure 6.3 presents an interface for a new constraint class. The programmer may need to define additional methods or attributes depending upon the requirement.

```
class KSNewConstraint : public KSConstraint
{
public:
        KSNewConstraint();
        KSNewConstraint(int          Tag,
                        std::string  Name,
                        KSNode   *Master,
                        KSNode   *Slave,
                        bool       *Constr,
                        …….
                        …….);

        KSNewConstraint(int          Tag,
                        std::string  Name,
                        KSNode   *Master,
                        KSNode   *Slave,
                        bool         dx,
                        bool         dy,
                        bool         dz,
                        bool         rx,
                        bool         ry,
                        bool         rz,
                        …….
                        …….);

        ~KSNewConstraint();
        void             SetTag(const int &Tag);
        void             SetName(const std::string &Name);
        void             SetSlaveNode(KSNode *Slave);
        void             SetMasterNode(KSNode *Master);
        int              GetTag(void) const;
        std::string      GetName(void) const;
        void             ApplyConstraint(void);
        void             UpdateDOFs(void);
        void             DisplaySelf(std::ofstream &OutputFile);
        int              ReturnFlag(void) const;
        …….
        …….
};
```

Figure 6.3: Interface for a new constraint derived class

6.2.3 Adding New Sections

The WoodFrameSolver program currently provides three types of section classes i.e. shell, frame and non-linear link. The base class *KSSection* is defined which contains all possible common methods and attributes for any type of section. Figure 6.4 presents an

interface for a new section. The programmer may need to define additional attributes and methods depending upon the requirement.

```
class KSNewSection: public KSSection
{
public:
        KSNewSection();
        KSNewSection(int         Tag,
                     std::string  SectionName,
                     std::string  MaterialName,
                     ……
                     ……);
        ~KSNewSection();
        void            DisplaySelf(ofstream &OutputFile);
        int             ReturnFlag(void) const;
        …….
        …….
};
```

Figure 6.4: Interface of new section class

6.2.4 Adding New Equation Solver

As mentioned in the Analysis section, the program currently provides only a banded solver class called *KSBandedSOEnSolver*. This class is a derived class of an abstract base class called *KSSOEandSolver*. This class contains the definition of all common methods and attributes for its child classes. Figure 6.5 presents an interface for a new solver child class. The programmer may need to define additional attributes and methods depending upon the requirement.

```
class KSNewSOEnSolver : public KSSOEandSolver
{
public:
        KSNewSOEnSolver();
        KSNewSOEnSolver(KSModel *Model);
        KSNewSOEnSolver(KSModel *Model,
                        int         BandWidth,
                        int         NumEqns,
                        int         NumRHS,
                        …….
                        …….);
        virtual ~KSNewSOEnSolver();
        void                    FormTangent(void);
        void                    FormRHS(void);
        bool                    Solve(void);
        std::deque<double>  GetX(void) const;
        virtual void            SetDimensions(int BandWidth, int numEqns, int numRHS, …..);
        virtual void            SetTangentStorage(void);
        virtual void            SetRHSStorage(void);
        virtual void            DeleteTangent(void);
        ……..
        ……..
};
```

Figure 6.5: Interface of new solver class

6.2.5 Adding New Equation Numberer

Two numbering schemes are implemented in the WoodFrameSolver program; *KSPlainNumberer*, and *KSPacManRenumberer*. The base class of these classes contains all the common methods and attributes. Figure 6.6 presents an interface of a new numberer child class. The programmer may need to define additional attributes and methods depending upon the requirement.

```
class KSNewNumberer : public KSDOFNumberer
{
public:
        KSNewNumberer();
        KSNewNumberer(…..);
       ~KSNewNumberer();
        void    Number(KSModel *Model);
        ……..
        ……..
};
```

Figure 6.6: Interface of new numberer class

Thus, we see how easy it is to add new capabilities in the WoodFrameSolver program. Adding new features doesn't affect the code in other classes. Several examples have been presented in the above sections. A similar approach can be followed in writing other new child classes i.e. *KSDynamicAnalysisCase, KSNonLinearMaterial* etc. New independent classes may also be added with ease. If the programmer intends to add new elements, constraints or sections to the WoodFrameSolver program then he or she will need to write some additional methods in *KSS2KFileReader* class to read these new elements, constraints or sections respectively, from the input file. Also, the programmer will need to write additional methods in *KSModelBuilder* class to add these elements, constraints, or sections to the containers in the *KSModel* class. Addition of new solver, renumberer or analysis child classes will require the addition of some conditional statements to instantiate the requested type of solver, renumberer or analysis object. Figure 6.7 presents an example for such a case.

```
////Renumber is an integer parameter and defines the type of numberer one
////might wants to use. Possible Renumber values are 0, 1, 2…

KSDOFNumberer *Numberer;

if(Renumber == 0)
{
   Numberer = new KSPlainNumberer();
}
else if(Renumber == 1)
{
   Numberer = new KSPacManReNumberer();
}
else if(Renumber == 2)
{
   Numberer = new KSNew2Numberer();
}
else if(Renumber == 3)
{
   Numberer = new KSNew3Numberer();
}
```

Figure 6.7: Example showing addition of new code (shown in bold) in the main program if a new numberer child class is added

6.3 Analysis Results

Testing and verification of the program was done by the WoodFrameSolver team on a wide variety of FE and non FE models using frame, shell, non-linear links, springs and equal constraints. SAP 2000 was chosen as a benchmark for comparing results. The results obtained are accurate when compared to theory and SAP 2000. Appendix B presents the comparison of these results on some example models. Some of these models are generated automatically by WoodFrameMesh and the others are generated in SAP 2000. For detailed verification of the WoodFrameSolver program one can refer to the WoodFrameSolver verification manual.

Table B.1 shows all the implemented features in the WoodFrameSolver program. Testing of the program shows that it is capable of solving problems with up to 50,000 degrees of freedom without the "pacman" renumberer. Implementation of the "pacman" renumberer and writing data on core (hard disk) has facilitated in successful testing of

models up to 100,000 degrees of freedom. Testing has also shown that the analysis speed of WoodFrameSolver is much faster than SAP 2000. Figures B.2 and B.3 in Appendix B, presents the bandwidth comparison and speed comparison of WoodFrameSolver with SAP 2000.