

Introducing a New Element into OpenSees

Version 1.0

August 21, 2000

Frank McKenna and Gregory L. Fenves
Pacific Earthquake Engineering Research Center
University of California, Berkeley

1 Introduction

This document is intended to demonstrate the steps necessary to introduce a new element into the OpenSees interpreter. OpenSees is an object-oriented framework under construction for finite element analysis. A key feature of OpenSees is the interchangeability of components and the ability to integrate existing libraries and new components into the framework (not just new element classes) without the need to change the existing code. Core components, that is the abstract base classes, define the minimal interface (minimal to make adding new component classes easier but large enough to ensure all that is required can be accommodated).

The OpenSees interpreter is an extension of the Tcl scripting language. Tcl is a string based procedural command language which allows substitution, loops, mathematical expressions, and procedures. The OpenSees interpreter adds commands to Tcl to allow users to create objects from the OpenSees framework and invoke methods on those objects once they have been created. Each of these commands is associated (bound) with a C++ procedure that is provided. It is this procedure that is called upon by the interpreter to parse the command.

In this document we provide a simple example of a script that can be used with the OpenSees interpreter to analyze a simple model. We then outline the C++ code necessary to introduce a new element into the framework. In addition, we demonstrate what must be done in addition to use the element with the interpreter. Finally, we show the changes to the script that are needed to use this new element. Note, that all the code and example scripts mentioned in this document can be found in OpenSees/EXAMPLES/NewElement.

2 A Simple Truss Example

In this section the example script, `example1.tcl`, for the static analysis of the simple linear three bar truss example shown in figure 1 is presented. For a more comprehensive set of examples showing the reader should be referred to the OpenSees examples manual, which can be found at <http://opensees.berkeley.edu/OpenSees/OpenSeesExamples.pdf>.

In the script, the analyst first creates a ModelBuilder object. In this example a BasicBuilder object is created. The construction of this object adds new commands to the interpreter, i.e. `node`, `material`, `element`, `fix` and `load`. It is these commands which can be used by the analyst to construct the model.

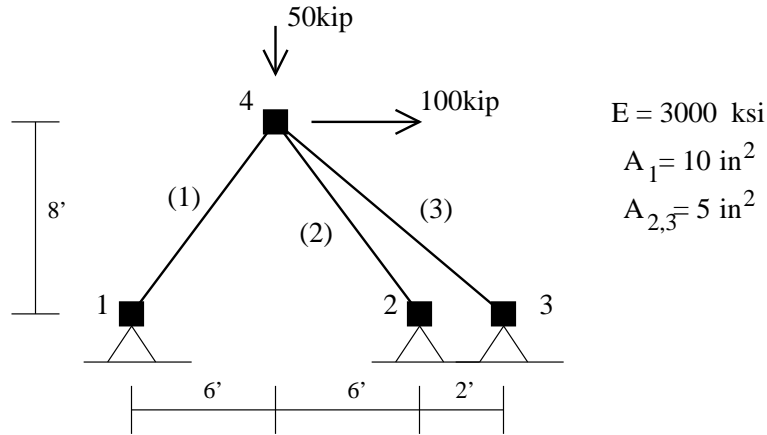


Figure 1: Example 1

```
# create the ModelBuilder object
model BasicBuilder -ndm 2 -ndf 2
```

The analyst then constructs the model. This is done by creating the four Node objects, a Material object, three Element objects, some Constraint objects, and finally a Load object.

```
# build the model
# node nodeId xLoc yLoc
node 1 0 0
node 2 144 0
node 3 168 0
node 4 72 96

# material matId type <type args>
uniaxialMaterial Elastic 1 3000

# element truss trussId iNodeId jNodeId Area matId
element truss 1 1 4 10 1
element truss 2 2 4 5 1
element truss 3 3 4 5 1

# constraint nodeId xFix? yFix?
fix 1 1 1
fix 2 1 1
fix 3 1 1

# pattern type patternID TimeSeries
pattern Plain 1 Linear {
  # load nodeID xForce yForce
  load 4 100 -50
}
```

After the model has been defined, the analyst then constructs the Analysis. This is done by first constructing the components of the Analysis object. In this example a BandSPD linear system of equation and a lapack solver (default for BandSPD), a ConstraintHandler object which deals with homogeneous single point constraints, an Integrator object of type LoadControl with a load step increment of one, an Algorithm object of type Linear, and a DOF_Numberer object of type RCM (reverse Cuthill-McKee). Once these objects have been created, the StaticAnalysis object is constructed.

```
# build the components for the analysis object
system BandSPD
constraints Plain
integrator LoadControl 1
algorithm Linear
numberer RCM

# create the analysis object
analysis Static
```

After the Analysis object is constructed a Recorder object is created. In this example we create a NodeRecorder to record the load factor and the two nodal displacements at Node 4, the results are stored in the file example.out.

```
# create a Recorder object for the nodal displacements at node 4
recorder Node example.out disp -load -nodes 4 -dof 1 2
```

Finally the analysis is performed and the results are printed.

```
# perform the analysis
analyze 1

# print the results at node 4 and at all elements
print node 4
print ele
playback 1
```

When OpenSees is run and the commands outlined above are input by the analyst at the interpreter prompt, or are sourced from a file using the source filename command, the following is output by the program.

```
Node: 4
  Coordinates : 72 96
  commitDisps: 0.530093 -0.177894
  unbalanced Load: 100 -50
```

```
Element: 1 type: Truss iNode: 1 jNode: 4 Area: 10
```

```
strain: 0.00146451 axial load: 43.9352
unbalanced load: 26.3611 35.1482 -26.3611 -35.1482
Material: ElasticMaterialModel: 1 E: 3000
```

```
Element: 2 type: Truss iNode: 2 jNode: 4 Area: 5
strain: -0.00383642 axial load: -57.5463
unbalanced load: 34.5278 -46.0371 -34.5278 46.0371
Material: ElasticMaterialModel: 1 E: 3000
```

```
Element: 3 type: Truss iNode: 3 jNode: 4 Area: 5
strain: -0.00368743 axial load: -55.3114
unbalanced load: 39.1111 -39.1111 -39.1111 39.1111
Material: ElasticMaterialModel: 1 E: 3000
```

```
1 0.530093 -0.177894
```

3 Introducing a New Element into OpenSees

We will look at the C++ code that is required to introduce a new Truss element, `MyTruss`, into the framework and a new command `myTruss` into the interpreter. The new class will work in planar problems where each node has two degrees-of-freedom. The new type will be complicated by the fact that each instance will be associated with a `UniaxialMaterial` object, this is done to show what this entails for parallel and database processing.

To introduce the new class into OpenSees three new files must be created, `MyTruss.h` and `MyTruss.cpp`, to define the class interface and implementation, and `TclMyTrussCommand.cpp`, to define the procedure to be invoked when the Tcl command `myTruss` is invoked. In addition two existing files must be modified, `TclElementCommands.cpp`, and `FEM_ObjectBroker.C` and `FEM_ObjectBroker.C`. The new files and modifications to existing files are outlined in the following subsections. All the files can be found in the `OpenSees/EXAMPLES/ExampleNewElement` directory.

3.1 `MyTruss.h`

The file `MyTruss.h` defines the class interface and details information about the instance variables associated with the objects of type `MyTruss`. The interface first declares that the `MyTruss` class inherits from the `Element` class.

```
class MyTruss : public Element {
```

The interface then defines two constructors and a destructor. The first constructor is used to construct each object by the analyst. The arguments passed include the elements id, the id's of the two end nodes, a reference to a `Material` object (a copy of which is created by the `MyTruss` object), and the Area of the bar. In addition the analyst may specify a mass per unit volume, if none is specified 0 is assumed. The second is used in parallel and database programming. The destructor is the method called when the object is being destroyed. It is called so that memory associated with the object is returned to the system.

```

public:
    // constructors
    MyTruss(int tag, int Nd1, int Nd2, UniaxialMaterial &theMat,
            double A, double rho = 0.0);
    MyTruss();
    // destructor
    ~MyTruss();

```

After the destructor comes the public member functions, these define the methods that all other objects in the program will be able to invoke on objects of type MyTruss. These methods are all inherited from the Element class and each subclass of Element must declare them.

```

public:

    // public methods to obtain information about dof & connectivity
    int getNumExternalNodes(void) const;
    const ID &getExternalNodes(void);
    int getNumDOF(void);

    // public methods to set the state of the element
    void setDomain(Domain *theDomain);
    int commitState(void);
    int revertToLastCommit(void);
    int revertToStart(void);
    int update(void);

    // public methods to obtain stiffness, mass, damping and residual information
    const Matrix &getTangentStiff(void);
    const Matrix &getSecantStiff(void);
    const Matrix &getDamp(void);
    const Matrix &getMass(void);

    void zeroLoad(void);
    const Vector &getResistingForce(void);
    const Vector &getResistingForceIncInertia(void);

    // public methods for output
    int sendSelf(int commitTag, Channel &theChannel);
    int recvSelf(int commitTag, Channel &theChannel, FEM_ObjectBroker &theBroker);
    int displaySelf(Renderer &theViewer, int displayMode, float fact);
    void Print(ostream &s, int flag =0);
    Response *setResponse(char **argv, int argc, Information &eleInformation);
    int getResponse(int responseID, Information &eleInformation);

```

Following the declaration of the public member functions comes the declaration of a private member function, computeStrain(), which can only be called by objects of type MyTruss.

```
private:
    // private member function - only available to objects of the class
    double computeCurrentStrain(void) const;
```

After the private member function is defined all the private variables, data which can only be accessed by objects of this type. First the instance variables associated with each object of this class are defined. Each object of this type will store its length, area, mass per unit volume, a pointer to a UniaxialMaterial object, a pointer to each of its end nodes, a pointer to a transformation matrix and an ID (integer array) object containing the node identifiers.

```
private:
    // private attributes - a copy for each object of the class
    UniaxialMaterial *theMaterial; // pointer to a material
    ID externalNodes;             // contains the id's of end nodes
    Matrix trans; // hold the transformation matrix
    double L; // length of MyTruss based on undeformed configuration
    double A; // area of MyTruss
    double M; // weight per unit volume
    Node *end1Ptr, *end2Ptr; // two pointers to the trusses nodes.
```

After the instance variable is defined some class variables. There is one instance of each of these variables shared by all objects of this class. The class variables comprise of a 3 matrices to return the stiffness, mass and damping matrices from the member functions and a vector to return the residual force. While the class variables could also be declared as instance variables, declaring them as class variables reduces the amount of memory required by each object of type MyTruss.

```
private:
    // private class attributes - single copy for all objects of the class
    static Matrix trussK; // class wide matrix for returning stiffness
    static Matrix trussD; // class wide matrix for returning damping
    static Matrix trussM; // class wide matrix for returning mass
    static Vector trussR; // class wide vector for returning residual
```

3.2 MyTruss.C

The MyTruss.C file contains the implementation. The file first initializes the class variables defined in the MyTruss.h file. Here the three matrices are initialized to be of size 4*4 and the Vector of size 4. It should be noted that access to the data stored in these objects is done using the C indexing notation, which starts at 0.

```
// initialize the class wide variables
Matrix MyTruss::trussK(4,4);
Matrix MyTruss::trussM(4,4);
Matrix MyTruss::trussD(4,4);
Vector MyTruss::trussR(4);
```

After the initialization of the class variables, the constructors for the class are implemented. The first constructor takes as arguments the objects identifier, the identifiers of the two end nodes, a reference to the objects material and the objects area. The constructor initializes the instance variables with the appropriate values. In addition, the constructor makes a copy of the Material object; it is this copy that will be used by the MyTruss object. The second constructor, which takes no arguments, is used in parallel and database programming for constructing an empty object whose instance variables will be filled in by the object itself when `recvSelf()` is invoked on the object.

```
MyTruss::MyTruss(int tag,
                int Nd1, int Nd2,
                UniaxialMaterial &theMat,
                double a, double rho)
:Element(tag,ELE_TAG_MyTruss),
  externalNodes(2),
  trans(1,4), L(0.0), A(a), M(rho), end1Ptr(0), end2Ptr(0)
{
    // create a copy of the material object
    theMaterial = theMat.getCopy();

    // fill in the ID containing external node info with node id's
    externalNodes(0) = Nd1;
    externalNodes(1) = Nd2;
}

// constructor which should be invoked by an FE_ObjectBroker only
MyTruss::MyTruss()
:Element(0,ELE_TAG_MyTruss),
  theMaterial(0),
  externalNodes(2),
  trans(1,4), L(0.0), A(0.0), M(0.0), end1Ptr(0), end2Ptr(0)
{
    // does nothing
}
```

After the two constructors comes the destructor for the class. This is the function that is invoked when the object is removed from the system. Each object is responsible for cleaning up after itself, anything the object created using the new operator. The destructor is also responsible for memory allocated by other objects, in this case the UniaxialMaterial object.

```
MyTruss::~MyTruss()
{
    if (theMaterial != 0)
        delete theMaterial;
}
```

After the destructor comes a number of utility methods that each Element type must provide. These include the methods `getNumberExternalNodes()` to return the number of

end nodes, `getExternalNodes()` an ID object containing the identifiers of these nodes, and `getNumDOF()` to return the number of degrees of freedom associated with the Element. In addition, there is the method `setDomain()` that is called when the object is added to a Domain object. When this method is invoked each MyTruss object completes the initialization of its instance variables. For example, if the two end nodes exist in the Domain the pointers to these end nodes are now set.

```

int
MyTruss::getNumExternalNodes(void) const
{
    return 2;
}

const ID &
MyTruss::getExternalNodes(void)
{
    return externalNodes;
}

int
MyTruss::getNumDOF(void) {
    return 4;
}

void
MyTruss::setDomain(Domain *theDomain)
{
    // first ensure nodes exist in Domain and set the node pointers
    int Nd1 = externalNodes(0);
    int Nd2 = externalNodes(1);
    end1Ptr = theDomain->getNode(Nd1);
    end2Ptr = theDomain->getNode(Nd2);

    if (end1Ptr == 0)
        return; // don't go any further - otherwise segmentation fault
    if (end2Ptr == 0)
        return; // don't go any further - otherwise segmentation fault

    // call the DomainComponent class method. note: THIS IS VERY VERY IMPORTANT
    this->DomainComponent::setDomain(theDomain);

    // ensure connected nodes have correct number of dof's
    int dofNd1 = end1Ptr->getNumberDOF();
    int dofNd2 = end2Ptr->getNumberDOF();
    if ((dofNd1 != 2) || (dofNd2 != 2))
        return; // don't go any further - otherwise segmentation fault
}

```

```

// now determine the length & transformation matrix
const Vector &end1Crd = end1Ptr->getCrds();
const Vector &end2Crd = end2Ptr->getCrds();

double dx = end2Crd(0)-end1Crd(0);
double dy = end2Crd(1)-end1Crd(1);

L = sqrt(dx*dx + dy*dy);

if (L == 0.0)
    return; // don't go any further - otherwise divide by 0 error

double cs = dx/L;
double sn = dy/L;

trans(0,0) = -cs;
trans(0,1) = -sn;
trans(0,2) =  cs;
trans(0,3) =  sn;

// determine the nodal mass for lumped mass approach
M = M * A * L/2; // remember M was set to rho in the constructor
}

```

Following the methods comes a number of methods related to the solution algorithm. These methods include `commitState()`, which is invoked when a point on the solution path has been achieved for a step in the analysis, `revertToLastCommit()` which is a method invoked to inform the Element that convergence to an acceptable solution was not achieved and that the Element is to return to the state it was at when the last `commit()` method was invoked on it. The method `revertToStart()` is invoked to tell the Element that it is to return to the state it was at the beginning of the analysis. For a `MyTruss` element type, which has no state information, the object itself will simply invoke the corresponding method on it's associated material object.

```

int
MyTruss::commitState()
{
    return theMaterial->commitState();
}

int
MyTruss::revertToLastCommit()
{
    return theMaterial->revertToLastCommit();
}

int

```

```

MyTruss::revertToStart()
{
    return theMaterial->revertToStart();
}

int
MyTruss::update()
{
    // determine the current strain given trial displacements at nodes
    double strain = this->computeCurrentStrain();

    // set the strain in the materials
    theMaterial->setTrialStrain(strain);

    return 0;
}

```

After these come the implementation of the typical Element methods to obtain the current linearized stiffness, mass and damping matrices, and the residual vector. In each of these methods, a call is made to the objects `computeCurrentStrain()` method. Once the strain has been computed the tangent matrices and residual vector are computed using standard matlab like operations. It should be noted that the results are placed into the class matrices and vectors.

```

const Matrix &
MyTruss::getTangentStiff(void)
{
    if (L == 0) { // if length == zero - we zero and return
        trussK.Zero();
        return trussK;
    }

    // get the current E from the material for the strain that was set
    // at the material when the update() method was invoked
    double E = theMaterial->getTangent();

    // form the tangent stiffness matrix
    trussK = trans^trans;
    trussK *= A*E/L;

    return trussK;
}

const Matrix &
MyTruss::getSecantStiff(void)
{

```

```

    if (L == 0) { // if length == zero - we zero and return
        trussK.Zero();
        return trussK;
    }

    // get the current strain from the material
    double strain = theMaterial->getStrain();

    // get the current stress from the material
    double stress = theMaterial->getStress();

    // compute the tangent
    double E = stress/strain;

    // form the tangent stiffness matrix
    trussK = trans^trans;
    trussK *= A*E/L;

    return trussK;
}

const Matrix &
MyTruss::getDamp(void)
{
    // no damping associated with this type of element
    return trussD;
}

const Matrix &
MyTruss::getMass(void)
{
    if (L == 0) { // if length == zero - we zero and return
        trussM.Zero();
        return trussM;
    }
    // determine mass matrix assuming lumped mass
    double nodeMass = M * A * L/2;
    for (int i=0; i<4; i++)
        trussM(i,i) = nodeMass;

    return trussM;
}

void
MyTruss::zeroLoad(void)
{

```

```

    // does nothing - no element loads associated with this object
}

const Vector &
MyTruss::getResistingForce()
{
    if (L == 0) { // if length == zero - we zero and return
        trussR.Zero();
        return trussR;
    }

    // R = Ku - Pext
    // force = F * transformation
    double force = A*theMaterial->getStress();
    for (int i=0; i<4; i++)
        trussR(i) = trans(0,i)*force;

    return trussR;
}

const Vector &
MyTruss::getResistingForceIncInertia()
{
    // R = Ku - Pext + Ma

    // determine the resisting force sans mass
    this->getResistingForce();

    // now include the mass portion
    if (L != 0 && M != 0) {
        double nodeMass = M * A * L/2;

        const Vector &accel1 = end1Ptr->getTrialAccel();
        const Vector &accel2 = end2Ptr->getTrialAccel();

        for (int i=0; i<2; i++) {
            trussR(i) = trussR(i) - nodeMass*accel1(i);
            trussR(i+2) = trussR(i+2) - nodeMass*accel2(i);
        }
    }

    return trussR;
}

```

After the more standard Element methods come two pairwise methods that are required for parallel and database processing, `sendSelf()` and `recvSelf()`. Each Element object is responsible for sending enough information to a Channel object such that an Element of similar

type, on the other end of the Channel, will be able to initialize itself so that the geometry and state information in the two objects are the same. When sending this data to a Channel a MyTruss object initially sends its tag, it's unique database tag, it's area, the Material objects tag in a vector object to the Channel. The MyClass object then sends it's ID to the Channel and finally the MaterialObject is told to send itself. When receiving itself from a Channel a MyTruss object will receive this information. It should be noticed that before the object can invoke `recvSelf()` on its Material object it must create the appropriate type of Material object, this it does using the `FEM_ObjectBroker` object and the Material objects class tag.

```

int
MyTruss::sendSelf(int commitTag, Channel &theChannel)
{
    // note: we don't check for dataTag == 0 for Element
    // objects as that is taken care of in a commit by the Domain
    // object - don't want to have to do the check if sending data
    int dataTag = this->getDbTag();

    // MyTruss packs it's data into a Vector and sends this to theChannel
    // along with it's dbTag and the commitTag passed in the arguments

    Vector data(5);
    data(0) = this->getTag();
    data(1) = A;
    data(4) = M;
    data(2) = theMaterial->getClassTag();
    int matDbTag = theMaterial->getDbTag();
    // NOTE: we do have to ensure that the material has a database
    // tag if we are sending to a database channel.
    if (matDbTag == 0) {
        matDbTag = theChannel.getDbTag();
        if (matDbTag != 0)
            theMaterial->setDbTag(matDbTag);
    }
    data(3) = matDbTag;

    theChannel.sendVector(dataTag, commitTag, data);

    // MyTruss then sends the tags of it's two end nodes
    theChannel.sendID(dataTag, commitTag, externalNodes);

    // finally MyTruss asks it's material object to send itself
    theMaterial->sendSelf(commitTag, theChannel);

    return 0;
}

```

```

int
MyTruss::recvSelf(int commitTag, Channel &theChannel, FEM_ObjectBroker &theBroker)
{
    int dataTag = this->getDbTag();

    // MyTruss creates a Vector, receives the Vector and then sets the
    // internal data with the data in the Vector

    Vector data(5);
    theChannel.recvVector(dataTag, commitTag, data);

    this->setTag((int)data(0));
    A = data(1);
    M = data(4);
    // MyTruss now receives the tags of it's two external nodes
    theChannel.recvID(dataTag, commitTag, externalNodes);

    // we create a material object of the correct type,
    // sets its database tag and asks this new object to receive itself.
    int matClass = data(2);
    int matDb = data(3);

    theMaterial = theBroker.getNewUniaxialMaterial(matClass);

    // we set the dbTag before we receive the material - this is important
    theMaterial->setDbTag(matDb);
    theMaterial->recvSelf(commitTag, theChannel, theBroker);

    return 0;
}

```

To display graphically an Element in an image the `displaySelf()` method is invoked on an Element object. What is displayed depends on the flag and display factor values provided as arguments. The display factor is used to magnify the nodal displacements at the member ends so that distortion of the structure is visible. The flag is used to indicate what measure is to be displayed. A MyTruss object will display the strain in the object if a 1 is passed, otherwise the force in the object is displayed.

```

int
MyTruss::displaySelf(Renderer &theViewer, int displayMode, float fact)
{
    // first determine the two end points of the truss based on
    // the display factor (a measure of the distorted image)
    // store this information in 2 3d vectors v1 and v2
    const Vector &end1Crd = end1Ptr->getCrds();
    const Vector &end2Crd = end2Ptr->getCrds();
    const Vector &end1Disp = end1Ptr->getDisp();

```

```

const Vector &end2Disp = end2Ptr->getDisp();

Vector v1(3);
Vector v2(3);
for (int i=0; i<2; i++) {
    v1(i) = end1Crd(i)+end1Disp(i)*fact;
    v2(i) = end2Crd(i)+end2Disp(i)*fact;
}
if (displayMode == 3) {
    // use the strain as the drawing measure
    double strain = theMaterial->getStrain();
    return theViewer.drawLine(v1, v2, strain, strain);

} else if (displayMode == 2) {
    // otherwise use the material stress
    double stress = A*theMaterial->getStress();
    return theViewer.drawLine(v1,v2, stress, stress);

} else { // use the axial force
    double force = A * theMaterial->getStress();
    return theViewer.drawLine(v1,v2, force, force);
}
}

```

To print the state of the Element, to a file or to the screen, the Print() method is invoked on the Element object. What is printed by the object, depends on the flag passed as an argument. A MyTruss object prints a detailed description of its state if 0 is passed, a shortened version if 1 is passed, and nothing at all if anything else is passed.

```

void
MyTruss::Print(ostream &s, int flag) const
{
    // compute the strain and axial force in the member
    double strain, force;
    if (L == 0.0) {
        strain = 0;
        force = 0.0;
    } else {
        strain = theMaterial->getStrain();
        force = A * theMaterial->getStress();
    }

    for (int i=0; i<4; i++)
        trussR(i) = trans(0,i)*force;

    if (flag == 0) { // print everything
        s << "Element: " << this->getTag();
    }
}

```

```

s << " type: MyTruss iNode: " << externalNodes(0);
s << " jNode: " << externalNodes(1);
s << " Area: " << A;
if (M != 0) s << " Mass (PerUnitVolume): " << M;

s << " \n\t strain: " << strain;
s << " axial load: " << force;
s << " \n\t unbalanced load: " << trussR;
s << " \t Material: " << *theMaterial;
s << endl;
} else if (flag == 1) { // just print ele id, strain and force
s << this->getTag() << " " << strain << " " << force << endl;
}
}
}

```

To allow the analyst to obtain information specific to the Truss element the methods `setResponse()` and `getResponse()` are provided. The `setResponse()` method is used to obtain an integer code that is used in subsequent requests to the `MyTruss` object for information. The integer returned depends on the array of strings passed to the element. A `MyTruss` object responds to requests for the axial force, 'axialForce', and the tangent stiffness matrix, 'stiffness'. Requests for material information are passed to the material object. A `-1` is returned if the `MyTruss` does not supply this information. it should be noted that for information returned in ID, vector and Matrix objects the `MyTruss` elements allocate these objects from the heap, this is done to reduce memory overhead in the elements. The `getResponse()` method is used to obtain the information.

Response *

```

MyTruss::setResponse(char **argv, int argc, Information &eleInformation)
{
    //
    // we compare argv[0] for known response types for the Truss
    //

    // axial force
    if (strcmp(argv[0], "axialForce") == 0)
        return new ElementResponse(this, 1, 0.0);

    // a material quantity
    else if (strcmp(argv[0], "material") == 0)
        return theMaterial->setResponse(&argv[1], argc-1, eleInformation);

    else
return 0;
}

```

```

int
MyTruss::getResponse(int responseID, Information &eleInformation)

```

```

{
  switch (responseID) {
    case -1:
      return -1;

    case 1:
      return eleInfo.setDouble(A * theMaterial->getStress());

    default:
      return 0;
  }
}

```

Finally the private member function for determining the current strain in the element is defined. This method uses the nodal pointers to look at the current trial displacements at the nodes from which the strain can be determined.

```

double
MyTruss::computeCurrentStrain(void) const
{
  // determine the strain
  const Vector &disp1 = end1Ptr->getTrialDisp();
  const Vector &disp2 = end2Ptr->getTrialDisp();

  double dLength = 0.0;
  for (int i=0; i<2; i++)
    dLength += (disp2(i)-disp1(i)) * trans(0,i);

  double strain = dLength/L;

  return strain;
}

```

3.3 TclElementCommands.cpp

The TclElementCommands.cpp file contains the C++ procedure TclModelBuilderElementCommand(). It is this procedure that is invoked every time the **element** command is invoked. This file is modified to allow the analyst to construct a new element of type MyTruss with the command:

```

element myTruss eleId iNodeID jNodeID area materialID massPerUnitVolume

```

To do this in the prototypes section of the file the the procedure TclModelBuilder_addMyTruss() is declared to be an externally defined procedure. (NOTE that the whole element myTruss command could be parsed in this file. However, to keep things modular and allow the command to work with other interpreters that may be developed it has been the convention to place the procedure in a separate file).

```
extern int
TclModelBuilder_MyTruss(ClientData , Tcl_Interp *, int, char **,
                        Domain*, TclModelBuilder *, int);
```

Then in the body of the procedure we inform the interpreter that, if the second argument of the **element** command is the string **myTruss**, that the external procedure is to be invoked. This is done by adding the following 4 lines of code:

```
} else if (strcmp(argv[1],"myTruss") == 0) {
    int result = TclModelBuilder_MyTruss(clientData, interp, argc, argv,
        theTclDomain, theTclBuilder);
    return result;
```

3.4 TclMyTrussCommand.cpp

In this file we place the procedure, `TclModelBuilder_MyTruss()`, that was defined to be external in `TclElementCommands.cpp`. In this procedure a check is first made to insure that the correct number of arguments have been provided. The arguments are then parsed. A `MyTruss` element object is finally constructed with the parsed arguments and added to the domain.

```
int
TclModelBuilder_MyTruss(ClientData clientData, Tcl_Interp *interp,
                        int argc, char **argv) {
    // make sure at least one other argument to contain type of system
    if (argc != 7 && argc != 8) {
        interp->result = "WARNING bad command - myTruss eleId iNode jNode Area matID";
        return TCL_ERROR;
    }

    // get the id, x_loc and y_loc
    int trussId, iNode, jNode, matID;
    double A, M = 0.0;
    if (Tcl_GetInt(interp, argv[2], &trussId) != TCL_OK) {
        interp->result = "WARNING invalid eleId- myTruss eleId iNode jNode Area matID";
        return TCL_ERROR;
    }
    if (Tcl_GetInt(interp, argv[3], &iNode) != TCL_OK) {
        interp->result = "WARNING invalid iNode- myTruss eleId iNode jNode Area matID";
        return TCL_ERROR;
    }
    if (Tcl_GetInt(interp, argv[4], &jNode) != TCL_OK) {
        interp->result = "WARNING invalid jNode- myTruss eleId iNode jNode Area matID";
```

```

    return TCL_ERROR;
}

if (Tcl_GetDouble(interp, argv[5], &A) != TCL_OK) {
    interp->result = "WARNING invalid A- myTruss eleId iNode jNode Area matID";
    return TCL_ERROR;
}
if (Tcl_GetInt(interp, argv[6], &matID) != TCL_OK) {
    interp->result = "WARNING invalid matID- myTruss eleId iNode jNode Area matID";
    return TCL_ERROR;
}
if (argc == 8 && Tcl_GetDouble(interp, argv[7], &M) != TCL_OK) {
    interp->result = "WARNING invalid matID- myTruss eleId iNode jNode Area matID";
    return TCL_ERROR;
}

UniaxialMaterial *theMaterial = theModelBuilder->getUniaxialMaterial(matID);

if (theMaterial == 0) {
    cerr << "WARNING TclPlaneTruss - truss - no Material found with tag ";
    cerr << matID << endl;
    return TCL_ERROR;
}

// now create the truss and add it to the Domain
MyTruss *theTruss = new MyTruss(trussId,iNode,jNode,*theMaterial,A,M);
if (theTruss == 0) {
    cerr << "WARNING TclPlaneTruss - addMyTruss - ran out of memory for node ";
    cerr << trussId << endl;
    return TCL_ERROR;
}
if (theDomain->addElement(theTruss) == false) {
    delete theTruss;
    cerr << "WARNING TclPlaneTruss - addTruss - could not add Truss to domain ";
    cerr << trussId << endl;
    return TCL_ERROR;
}

// if get here we have successfully created the node and added it to the domain
return TCL_OK;
}

```

3.5 FEM_ObjectBroker.cpp

An FEM_ObjectBroker is the object responsible for creating blank objects of a specific type. This object is required for parallel and database programming. The MyTruss.h file is first included in the list of element header files. Then code in the `getNewElement()` method is revised to return a new MyTruss element if requested.

```
Element *
FEM_ObjectBroker::getNewElement(int classTag) {
    switch(classTag) {
        // existing code
        .....
        // new two lines of code added for MyTruss element
        case ELE_TAG_MyTruss:
            return new MyTruss();
    }
}
```

4 Modification to Example Script

To use the new element in the new OpenSees interpreter, the code in `example1.tcl` is modified. The modified code can be found in `example2.tcl`. The lines with the element command in `example1.tcl`:

```
# element truss trussId iNodeId jNodeId Area matId
element truss 1 1 4 10 1
element truss 2 2 4 5 1
element truss 3 3 4 5 1
```

are replaced with the following lines in `example2.tcl`:

```
# element truss trussId iNodeId jNodeId Area matId
element myTruss 1 1 4 10 1
element myTruss 2 2 4 5 1
element myTruss 3 3 4 5 1
```