OpenSees Dynamic API

Frank McKenna

University of California, Berkeley

Abstract

This document presents the dynamic OpenSees API that can be used to add new materials, elements, and commands into the OpenSees interpreter through the use of modules created by the developer and loaded during runtime.

DRAFT January 22, 2009

1 Introduction

The OpenSees interpreter is an object-oriented application that is created by linking against static libraries. When an application is linked with static libraries, the executable that is generated contains all the code in the library that may be needed by the application. When the application is launched, the application's code, which includes code from the static library, is loaded into the applications address space. Applications linked with dynamic libraries on the other hand do not load the code from the library into the address space of the application until it is actually needed. This results in smaller application size and memory footprint. The advantage of this is that it results in smaller application sizes and memory footprint requirements. The disadvantage being that time is required when running to load the libraries from the filesystem and that the libraries must be packaged and placed in an accessible location.

The OpenSees interpreter has itself been created with static libraries for ease of distribution and performance reasons. However, code has been provided in the application, to allow the use of dynamic libraries. Unlike traditional applications linked using dynamic libraries, the OpenSees interpreter need not know these dynamic libraries at compile time. Dynamic libraries can be used in OpenSees to add new elements, new materials and new commands into the interpreter. This will have a number of advatages: (1)This will allow developers to create new extensions without needing to compile and link with the rest of the OpenSees code. (2) Users can use external code provided by more than one developer, (3) Users and developers can use the latest version of the OpenSees application without the need to obtain/recompile the modules. and (4) the OpenSees development team can focus on the core of the application, allowing new code to be introduced firstly through the use of modules until it has been tested and verified by the external users.

Versions 2.1.0 and above of the OpenSees interpreter has been modified to allow the inclusion of these dynamic modules. The idea is rather simple. When the interpreter encounters a command, an element type or a material type that is unknown it will look for a dynamic library with a specific name. If the library exists and if specific procedures exist within the library, the library will be loaded, procedures stored, and the appropriate procedure will be invoked. These new elements and material modules can be written as a C++ class or they can be written as either a C or Fortran procedure. The only constraint is that they must follow the appropriate abstract class interface if C++, or a procedural interface if C or Fortran.

In the following sections we will go over the material and element interfaces that must be adhered to. Then we will cover the additional procedures available to C and Fortran developers and global objects available to C++ programmers. Finally we will provide example code for adding new modules, all of which are available in the OpenSees source code under the OpenSees/PACKAGE directory.

DRAFT January 22, 2009

2 Materials and Elements in OpenSees for C and Fortran Programmers

OpenSees is an object-oriented application. Whether elements and/or material routines are written in the object-oriented language C++, or the procedural languages C or fortran, when a new material or element is introduced, an object of this type is created. In object-oriented programming, objects are responsible for maintaining their data and for providing operations that work on that data. For C++ programmers this is automatically provided through use of class methods and variables defined in class interface. For procedural programmers this is done through new data types that we introduce. The data types for materials and elements contain fields for the data and they also contain a field for a pointer to a procedure. For example the new material object data type we introduce is as shown below. tag, nParam, nState, theParam, cState and tState are fields for the data, matFunctPtr is the field for a pointer to the procedure that uses this object and matObjectPtr is another field containing a pointer to an OpenSees object needed that is used in the OpenSees code. The details of what the developer does with the fields is left to later sections.

```
struct matObject {
  int tag;
  int nParam;
  int nState;
  double *theParam;
  double *cState;
```

```
  double *tState;
  matFunct matFunctPtr;
  void *matObjectPtr;
};
```

What C and Fortran programmers need to know, and what is different from other codes, is that the procedure they provide are passed this data structure (which contains a pointer to itself!) in the procedure call. In the body of the procedure, the developer must extract the parameter and history data from the data structure, work on that data and return the results. The procedural interfaces, element and material, contain a switch which is used to determine which operation is required and what result fields need to be filled in.

3 Introducing New Material into OpenSees

In OpenSees a material object is an object responsible for determining the stress-strain relationship at a point in the element. There are currently three basic material types in OpenSees:

- Uniaxial materials which provides one dimensional stress-strain or force-deformation relationship. The relationship depends on usage of material.
- nD materials which provides stress-strain relationship at gauss-points in a solid element.
- Section ForceDeformation materials which provides force-deformation relationship at gauss-points in beams and plates.

There are two ways in which a developer can add a new material into OpenSees: (1) Develop a new material in C++ using the existing class interface, and (2) Develop a new material in C or Fortran using a procedural interface.

3.1 Class Interface

3.2 Procedural Interface

The procedural interface is set up for programmers who do not wish to learn C++. However, it is set up so that the paradigm is still object-oriented. Object data and object methods that operate on the data still exist, it's just that methods must now be called with the object-data as an argument. the The api that all material developers must adhere to is:

DRAFT January 22, 2009

```
void matFuncName(matObj *thisObj, modelState *model, double *strain,
                 double *tang,  double *stress, int *isw, int *result)
```

where

- thisObj is a pointer to a struct of type matObj, the struct holding the material data.
- model is a pointer to a struct of type modelState.
- strain is a pointer to a double array of input strains, the dimension of the input depends on dimensionalty of material.
- tang is a pointer to a double array of output tangent quantities for input strain.
- stress is a pointer to a double array of output stress quantities for input strain.
- isw is a pointer to an integer switch.
- result is a pointer to the output error flag.

the matObj struct is defined in the file elementAPI.h as:

```
struct matObject {
    int tag;
    int nParam;
    int nState;
    double *theParam;
    double *cState;
    double *tState;
    matFunct matFunctPtr;
    void *matObjectPtr;
};
```

the corresponding fortran definition found in the file elementAPI.f is the same.

```
 public matObject
    type, bind(C) :: matObject
    integer(c_int) ::tag;
    integer(c_int) ::nParam;
    integer(c_int) ::nState
    type (c_ptr) :: theParam;
```

DRAFT January 22, 2009

```
    type (c_ptr) :: cState;
    type (c_ptr) :: tState;
    type(c_funptr) ::matFunctPtr
    type(c_ptr) ::matObjectPtr
  end type matObject;
```

where tag is material tag, nParam and nState are two integer variables declaring number of parameter and state variables. The theParam is a pointer to the array of nParam parameter variables. The cState and tState are pointers to the arrays for the committed and trial state variables. The matFunctPtr is a pointer to the material function associated with this material object and the matObjectPtr is used to store a pointer to a C++ Material object, should one be needed.

the modeState struct is defined as:

```
typedef struct {
  double time;
  double dt;
} modelState;
```

where time and dt are the time and time increment in the domain. The fortran definition is the same.

In addition to the material struct definitions there are a number of procedures that the material developers must use. These are:

- a method to allocate space for the element arrays

  ```
  extern "C" int  OPS_AllocateMaterial(matObject *);
  ```

- methods obtain input args from the command line:

  ```
  extern "C" int  OPS_GetIntInput(int *numData, int*data);
  extern "C" int  OPS_GetDoubleInput(int *numData, double *data);
  ```

3.3 Example C++ Material

The goal of this example is to ensure that the command

```
uniaxialMaterial ElasticPPcpp 1 3000 0.03
```

DRAFT January 22, 2009

is understood by the compiler. ElasticPPcpp is a new elastic perfectly plastic uniaxial material type. The C++ code for this example comprises 2 files, the header file, ElasticPPcpp.h, and the implementation file, ElasticPPcpp.cpp In the implementation file there is also provided the C routine that is searched for on loading of the dynamic library by the interpreter. This routine contains the code which will read from the input the args necessary for the material, construct the material and add it to the model builder. The naming of this function is important, it must be OPS_ followed by the name of the unknown material, in this example OPS_ElasticPPcpp.

The header file, ElasticPPcpp.h is as shown below. It first declares that the class ElasticPPcpp is a subclass of UniaxialMaterial. In then presents the lists the methods avaialable to other classes through it's public interface. In the private section it defines those class variables that are unaccessible to all other classes.

```
class ElasticPPcpp : public UniaxialMaterial
{
  public:
    ElasticPPcpp(int tag, double E, double eyp);
    ElasticPPcpp();

    ~ElasticPPcpp();

    int setTrialStrain(double strain, double strainRate = 0.0);
    double getStrain(void);
    double getStress(void);
    double getTangent(void);

    double getInitialTangent(void) {return E;};

    int commitState(void);
    int revertToLastCommit(void);
    int revertToStart(void);

    UniaxialMaterial *getCopy(void);

    int sendSelf(int commitTag, Channel &theChannel);
    int recvSelf(int commitTag, Channel &theChannel,
                 FEM_ObjectBroker &theBroker);

    void Print(OPS_Stream &s, int flag =0);

  protected:

  private:
    double fyp, fyn;    // positive and negative yield stress
    double E;             // elastic modulus
    double trialStrain; // trial strain
    double ep;            // plastic strain at last commit

    double trialStress;      // current trial stress
    double trialTangent;     // current trial tangent
};
```

DRAFT January 22, 2009

The methods are defined in the ElasticPPcpp.cpp file. The first two methods are constructors. The first constructor takes 3 args, the material tag, young modulus and the yield strain. The second constructor has 0 args. The UniaxialMaterial classes constructor is invoked in these constructors with the material tag and a class tag defined in the header file. The material parameters are also inititialized in the constructors. The two constructors are shown below:

```
ElasticPPcpp::ElasticPPcpp(int tag, double e, double eyp)
:UniaxialMaterial(tag,MAT_TAG_ElasticPPcpp),
 ezero(0.0), E(e), trialStrain(0.0), ep(0.0),
 trialStress(0.0), trialTangent(E)
{
  fyp = E*eyp;
  fyn = -fyp;
}

ElasticPPcpp::ElasticPPcpp()
:UniaxialMaterial(0,MAT_TAG_ElasticPPcpp),
 fyp(0.0), fyn(0.0), ezero(0.0), E(0.0), trialStrain(0.0), ep(0.0),
 trialStress(0.0), trialTangent(0.0)
{

}
```

The next method is called the destructor. For this class the destructor contains no code. If the class had requested that memory be dynamically allocated, it is here that the memory would be deallocated.

```
ElasticPPcpp::~ElasticPPcpp()
{
  // does nothing
}
```

The next method setTrialStrain() delas with the material state determination. The stress and tangent are determined in this method and stored in the trialStress and trialTangent variables. It is these variables that are returned in the getStress() and getTangent() methods. The trialStrain variable is also set equal to the incoming strain. This variable is returned when te getStrain() method is invoked.

```
int
ElasticPPcpp::setTrialStrain(double strain, double strainRate)
{
    trialStrain = strain;

    double sigtrial;    // trial stress
    double f;           // yield function

    // compute trial stress
```

DRAFT January 22, 2009

```
    sigtrial = E * ( trialStrain - ep );

    // evaluate yield function
    if ( sigtrial >= 0.0 )
        f =  sigtrial - fyp;
    else
        f = -sigtrial + fyn;

    double fYieldSurface = - E * DBL_EPSILON;
    if ( f <= fYieldSurface ) {

      // elastic
      trialStress = sigtrial;
      trialTangent = E;

    } else {
      // plastic
      if ( sigtrial > 0.0 ) {
        trialStress = fyp;
      } else {
        trialStress = fyn;
      }

      trialTangent = 0.0;
    }

    return 0;
}

double
ElasticPPcpp::getStrain(void)
{
  return trialStrain;
}

double
ElasticPPcpp::getStress(void)
{
  return trialStress;
}

double
ElasticPPcpp::getTangent(void)
{
  return trialTangent;
}
```

The next set of methods deal with the path dependent nature of the material. The method commitState() causes and updates the values of the commited plastic yield point.

```
int
ElasticPPcpp::commitState(void)
{
    double sigtrial;    // trial stress
    double f;           // yield function
```

DRAFT January 22, 2009

```
    // compute trial stress
    sigtrial = E * ( trialStrain  - ep );

    // evaluate yield function
    if ( sigtrial >= 0.0 )
        f =  sigtrial - fyp;
    else
        f = -sigtrial + fyn;

    double fYieldSurface = - E * DBL_EPSILON;
    if ( f > fYieldSurface ) {
      // plastic
      if ( sigtrial > 0.0 ) {
        ep += f / E;
      } else {
        ep -= f / E;
      }
    }

    return 0;
}

int
ElasticPPcpp::revertToLastCommit(void)
{
    return 0;
}


int
ElasticPPcpp::revertToStart(void)
{
    ep = 0.0;
     fyp = E*eyp;
     fyn = -fyp;

    trialStrain = 0.0;
    trialStress = 0.0
    trialTangent = 0.0;


    return 0;
}
```

The next method getCopy() is called in the element constructor. The method returns an exact copy of the material to he element. It is the responsibility of the element to invoke the desctrutor when it is finished with the material object.

```
UniaxialMaterial *
ElasticPPcpp::getCopy(void)
{
  ElasticPPcpp *theCopy =
    new ElasticPPcpp(this->getTag(),E,fyp/E);
  theCopy->ep = this->ep;

  return theCopy;
```

DRAFT January 22, 2009

```
}
```

Finally in the ElasticPPcpp.cpp file is the proedure OPS_ElasticPPcpp. This is the method that is invoked when the dynamic library is loaded into the application. This procedure is called whenever a new material object of type ElasticPPcpp is required by the interpreter. The procedure uses the OPS_GetIntInput() and OPS_GetDoubleInput() are called to read from input integer and double values from the input line.

```
extern "C" DllExport void *
OPS_ElasticPPcpp(void)
{
  // Pointer to a uniaxial material that will be returned
  UniaxialMaterial *theMaterial = 0;

  if (argc < 5) {
    opserr << "WARNING insufficient arguments\n";
    opserr << "Want: uniaxialMaterial ElasticPP tag? E? epsy?" << endln;
    return 0;
  }

  int    iData[1];
  double dData[2];
  int numData;

  numData = 1;
  if (OPS_GetIntInput(&numData, iData) != 0) {
    opserr << "WARNING invalid uniaxialMaterial ElasticPP tag" << endln;
    return 0;
  }

  numData = 2;
  if (OPS_GetDoubleInput(&numData, dData) != 0) {
    opserr << "WARNING invalid E & ep\n";
    return 0;
  }

  theMaterial = new ElasticPPcpp(iData[0], dData[0], dData[1]);

  if (theMaterial == 0) {
    opserr << "WARNING could not create uniaxialMaterial " << argv[1] << endln;
    return 0;
  }

  return theMaterial;
}
```

3.4 Example C Material

The goal of this example is to ensure that the command

```
uniaxialMaterial ElasticPPC 1 3000 0.03
```

DRAFT January 22, 2009

is understood by the compiler. ElasticPPC is a new elastic perfectly plastic uniaxial material type that is implemented using a c procedure. The example code for this comprises a single procedure in the file ElasticPPC.c. This file contains a single procedure ElasticPPC. The name of the procedure is important for the OpenSees interptreter when it encounerts a uniaxial material it knows nothing about will look for a library with the name ElasticPPC containing the procedure ElasticPPC. The ElasticPPC procedure interface follows that of the new material interface discussed previously. The output args are the tangent, the stress and an error code. The input args are the material object on which the procedure acts, the model state, the strain and the isw switch. The isw switch informs the method what actions it needs to perform for each invocation of the procedure. Comparing this code with the C++ code presented in the previous section, it is clear that the single procedure contains all the methods of the C++ code and the isw switch is responsible for determining which method is to be invoked. For example when isw is equal to ISW_COMMIT, the code of the commit method is implemented on the material object. The only case where the code is not identical is the case when isw is equal to ISW_INIT. For this case the procedure is responsible for reading from input using the procedure OPS_GetIntInput() and OPS_GetDoubleInput() the variables on the command line for the material, for allocating memory for the parameter and history variables in the object and for then initializing these variables. It is important to note that once the materials tag, number of parameters and number of state variables are defined that the procedure calls OPS_AllocateMaterial with the material object so that space for these variables is set aside. The procedure does not have to call a procedure to release this memory when the isw switch is set to ISW_DELETE, this switch is only used in materials that use additional memory not provided in the interface.

```
extern "C" DllExport void
ElasticPPC (matObj *thisObj, modelState *model, double *strain, double *tang, double *st
{
  if (*isw == ISW_INIT) {

    double dData[2];
    int    iData[1];

    /* get the input data  - tag? E? eyp? */
    int numData = 1;
    OPS_GetIntInput(&numData, iData);
    numData = 2;
    OPS_GetDoubleInput(&numData, dData);

    /* Allocate the element state */
    thisObj->tag = iData[0];

   /* Allocate the element state */
    thisObj->tag = iData[0];
    thisObj->nParam = 2;  /* E, eyp */
```

DRAFT January 22, 2009

```
  thisObj->nState = 2;  /* strain, ep */
  OPS_AllocateMaterial(thisObj);

  double E = dData[0];
  double eyp = dData[1];

  thisObj->theParam[0] = E;
  thisObj->theParam[1] = eyp;

} else if (*isw == ISW_DELETE) {
  ;
} else if (*isw == ISW_COMMIT) {

  double trialStrain = thisObj->tState[0];
  double f;           // yield function

  double E = thisObj->theParam[0];
  double eyp = thisObj->theParam[1];
  double ep = thisObj->cState[1];

  double fyp = E*eyp;
  double fyn = -fyp;

  // compute trial stress
  double sigtrial = E * ( trialStrain - ep );

  // evaluate yield function
  if ( sigtrial >= 0.0 )
      f =  sigtrial - fyp;
  else
      f = -sigtrial + fyn;

  double fYieldSurface = - E * DBL_EPSILON;
  if ( f > fYieldSurface ) {
    // plastic
    if ( sigtrial > 0.0 ) {
      ep += f / E;
    } else {
      ep -= f / E;
    }
  }
 thisObj->cState[0] = trialStrain;
  thisObj->cState[1] = ep;

} else if (*isw == ISW_REVERT) {

  for (int i=0; i<3; i++)
    thisObj->tState[i] = thisObj->cState[i];

} else if (*isw == ISW_REVERT_TO_START) {

  for (int i=0; i<2; i++) {
    thisObj->cState[i] = 0.0;
    thisObj->tState[i] = 0.0;
  }
} else if (*isw == ISW_FORM_TANG_AND_RESID) {
  double trialStrain = *strain;
```

DRAFT January 22, 2009

```
    double f;           // yield function
    double trialStress, trialTangent;

    double E = thisObj->theParam[0];
    double eyp = thisObj->theParam[1];
    double ep = thisObj->cState[1];

    double fyp = E*eyp;
    double fyn = -fyp;

    // compute trial stress
    double sigtrial = E * ( trialStrain - ep );

    // evaluate yield function
    if ( sigtrial >= 0.0 )
        f =  sigtrial - fyp;
    else
        f = -sigtrial + fyn;

    double fYieldSurface = - E * DBL_EPSILON;
    if ( f <= fYieldSurface ) {

      // elastic
      trialStress = sigtrial;
      trialTangent = E;

    } else {

      // plastic
      if ( sigtrial > 0.0 ) {
        trialStress = fyp;
      } else {
        trialStress = fyn;
      }

      trialTangent = 0.0;
    }

    thisObj->tState[0] = trialStrain;
    *stress = trialStress;
    *tang = trialTangent;
  }

  *result = 0;
}
```

3.5 Example Fortran Material

The goal of this example is to ensure that the command

```
uniaxialMaterial elasticppf 1 3000 0.03
```

is understood by the compiler. Elasticppf is a new elastic perfectly plastic uniaxial material type that is implemented using a fortran procedure. The example code for this comprises

DRAFT January 22, 2009

a single procedure in the file ElasticPPF.f. This file contains a single procedure ElasticPPf. Again and for the same reasons presented in the previous c example, the name of the procedure is important. It should be noted that unlike the c example, the command contains all underscores. This is a consequence of the fortran compiler, which for the compiler used output the procedure in all lower case. The code in the example is virtually the same as for the previous c example. The only thing to note is that before calling memory allocated for the pointers the fortran code must make a call to the fortran routine c_f_pointer().

```
      SUBROUTINE elasticPPf(matObj,model,strain,tang,stress,isw,error)
      use materialTypes
      use materialAPI
      implicit none

      type(matObject)::matObj
      type(modelState)::model
      real *8::strain
      real *8::tang
      real *8::stress
      integer::isw
      integer::error;

      real *8, pointer::theParam(:)
      real *8, pointer::cState(:)
      real *8, pointer::tState(:)

      real *8 E, eyp, ep, fyp, fyn, f, fYieldSurface
      real *8 sigtrial, trialStrain, trialStress, trialTangent

      integer, target :: iData(2)
      integer, pointer ::iPtr(:)
      integer numData, err
      real *8, target :: dData(2)
      real *8, pointer:: dPtr(:)

c     outside functions called
      integer OPS_GetIntInput, OPS_GetDoubleInput, OPS_AllocateMaterial

      IF (isw.eq.ISW_INIT) THEN

c     get the input data  - tag? E? eyp?

         numData = 1
         iPtr=>iData;
         err = OPS_GetIntInput(numData, iPtr)
         numData = 2
         dPtr=>dData;
         err = OPS_GetDoubleInput(numData, dPtr)

c     Allocate the element state
         matObj%tag = idata(1)
         matObj%nparam = 2
         matObj%nstate = 2

         err = OPS_AllocateMaterial(matObj)
```

DRAFT January 22, 2009

```
         call c_f_pointer(matObj%theParam, theParam, [2]);

c     Initialize the element properties
         theParam(1) = dData(1);
         theParam(2) = dData(2);

      ELSE

         call c_f_pointer(matObj%theParam, theParam, [2]);
         call c_f_pointer(matObj%cState, cState, [2]);
         call c_f_pointer(matObj%tState, tState, [2]);

         IF (isw == ISW_COMMIT) THEN
         IF (isw == ISW_COMMIT) THEN

            trialStrain = tState(1)

            E = theParam(1)
            eyp = theParam(2)
            ep = cState(2)

            fyp = E*eyp
            fyn = -fyp

c     compute trial stress
            sigtrial = E * ( trialStrain - ep );

c     evaluate yield function
            IF ( sigtrial >= 0.0 ) THEN
               f =  sigtrial - fyp
            ELSE
               f = -sigtrial + fyn
            END IF

            fYieldSurface = - E * DBL_EPSILON;

            IF ( f > fYieldSurface ) THEN
c     plastic
               IF ( sigtrial > 0.0 ) THEN
                  ep = ep + f / E
               ELSE
                  ep = ep - f / E
               END IF
            END IF

            cState(1) = trialStrain;
            cState(2) = ep;
          ELSE IF (isw == ISW_REVERT_TO_START) THEN
            cState(1) = 0.0;
            cState(2) = 0.0;
            tState(1) = 0.0;
            tState(2) = 0.0;

         ELSE IF (isw == ISW_FORM_TANG_AND_RESID) THEN

            trialStrain = strain;

            E = theParam(1);
```

DRAFT January 22, 2009

```
            eyp = theParam(2);
            ep = cState(2);

            fyp = E*eyp;
            fyn = -fyp;

c     compute trial stress
            sigtrial = E * ( trialStrain - ep );

c     evaluate yield function
            IF ( sigtrial >= 0.0 ) THEN
               f =  sigtrial - fyp;
            ELSE
               f = -sigtrial + fyn;
            END IF

            fYieldSurface = - E * DBL_EPSILON;

            IF ( f <= fYieldSurface ) THEN

               trialStress = sigtrial;
               trialTangent = E;

            ELSE

c     plastic
                  IF ( sigtrial > 0.0 ) THEN
                     trialStress = fyp;
             ELSE
                     trialStress = fyp;
             ELSE
                     trialStress = fyn;
                  END IF

                  trialTangent = 0.0;
            END IF

            tState(1) = trialStrain;
            stress = trialStress;
            tang = trialTangent;
         END IF
      END IF

c     return error code
      error = 0

      END SUBROUTINE elasticPPf
```

DRAFT January 22, 2009

4 Adding New Element to OpenSees

In OpenSees an element is responsible for determining it's tangent and resisting force given the changes in response at it's nodes.

There are two ways in which a developer can add a new element into OpenSees: (1)

Develop a new material in C++ using the existing class interface, and (2) Develop a new material in C or Fortran using a procedural interface.

4.1 Class Interface

4.2 Procedural Interface

The procedural interface is set up for programmers who do not wish to learn C++. However, it is set up so that the paradigm is still object-oriented. Object data and object methods that operate on the data still exist, it's just that methods must now be called with the object-data as an argument. the The api that all element developers must adhere to is:

```
void eleFunctName(struct eleObject *, modelState *, double *tang, double *resid, int *is
```

where

- thisObj is a pointer to a struct of type eleObj, the struct holding the element data.
- model is a pointer to a struct of type modelState.
- tang is a pointer to a double array of output tangent.
- resid is a pointer to a double array of output residual.
- isw is a pointer to an integer switch.
- error is a pointer to the output error flag.

the eleObj struct is defined in the file elementAPI.h as:

```
struct eleObject {
  int tag;
  int nNode;
  int nDOF;
  int nParam;
  int nState;
  int nMat;
  int *node;
  double *param;
  double *cState;
  double *tState;
  matObject **mats;
  eleFunct eleFunctPtr;
};
```

DRAFT January 22, 2009

where tag is the element unique tag in the model, nNode the number of nodes, nDOF the number of DOF at all the nodes, nParam the number of parameter variables, nState the number of trial and committed state variables and nmat the number of materials used by the element. The array variables include the node tags in node, the parameter variables in param, the committed and trial state variables in cState and tState, and an arry of pointers to the different materials used by the element in mats. Finally the object stores a pointer to it's own function in eleFunctPtr.

A similar fortran type exists for fortran in elementAPI.f

In addition to the element struct definitions there are a number of procedures that the element developers can use. These include

- a method to allocate space for the element arrays

```
extern "C" int  OPS_AllocateElement(eleObject *, int *matTags, int *maType);
```

- methods obtain input args from the command line:

```
extern "C" int  OPS_GetIntInput(int *numData, int*data);
extern "C" int  OPS_GetDoubleInput(int *numData, double *data);
```

- methods to obtain nodal coordinates and response quantaties.

```
extern "C" int  OPS_GetNodeCrd(int *nodeTag, int *sizeData, double *data);
extern "C" int  OPS_GetNodeDisp(int *nodeTag, int *sizeData, double *data);
extern "C" int  OPS_GetNodeVel(int *nodeTag, int *sizeData, double *data);
extern "C" int  OPS_GetNodeAcc(int *nodeTag, int *sizeData, double *data);
```

- method to invoke the material objects for fortran developers.

- method to send a message to output.

```
extern "C" int  OPS_Error(char *, int length);
```

4.3 Example C++ Element

The goal of this example is to ensure that the command

```
element TrussCPP 1 1 4 10.0 1
```

is understood by the compiler. TrussCPP is a new truss element type for two dimensional problems with two dof at each node. The C++ code for this example comprises 2 files, the header file, TrussCPP.h, and the implementation file, TrussCPP.cpp In the implementation file there is also provided the C routine that is searched for on loading of the dynamic library

DRAFT January 22, 2009

by the interpreter. This routine contains the code which will read from the input the args necessary for the material, construct the material and add it to the model builder. The naming of this function is important, it must be OPS_ followed by the name of the unknown element on the command line, in this example OPS_TrussCPP.

The header file, TrussCPP.h is as shown below. It first declares that the class TrussCPP is a subclass of Element. In then presents the lists the methods avaialable to other classes through it's public interface. In the private section it defines those class variables that are unaccessible to all other classes.

```
class TrussCPP : public Element
{
  public:
    // constructors
    TrussCPP(int tag,
             int Nd1, int Nd2,
             UniaxialMaterial &theMaterial,
             double A);

    TrussCPP();

    // destructor
    ~TrussCPP();


    // public methods to obtain inforrmation about dof & connectivity
    int getNumExternalNodes(void) const;
    const ID &getExternalNodes(void);
    Node **getNodePtrs(void);
    int getNumDOF(void);
    void setDomain(Domain *theDomain);

    // public methods to set the state of the element
    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 &getInitialStiff(void);
    const Matrix &getDamp(void);
    const Matrix &getMass(void);

    const Vector &getResistingForce(void);

    // public methods for output
    int sendSelf(int commitTag, Channel &theChannel);
    int recvSelf(int commitTag, Channel &theChannel, FEM_ObjectBroker &theBroker);
    void Print(OPS_Stream &s, int flag =0);

    Response *setResponse(const char **argv, int argc, Information &eleInfo, OPS_Stream
```

DRAFT January 22, 2009

```
    int getResponse(int responseID, Information &eleInformation);

  protected:

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

    // 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, could use a Vector
                        // if ever bother to change the Vector interface for
                        // x-product.

    double L;           // length of truss (undeformed configuration) - set in setDomain
    double A;           // area of truss
    Node *theNodes[2];  // node pointers

    // static data - 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
};
#endif


// typical constructor
TrussCPP::TrussCPP(int tag,
                   int Nd1, int Nd2,
                   UniaxialMaterial &theMat,
                   double a)
:Element(tag,ELE_TAG_TrussCPP),
 externalNodes(2),
 trans(1,4), L(0.0), A(a)
{
  // get a copy of the material object for our own use
  theMaterial = theMat.getCopy();
  if (theMaterial == 0) {
    opserr << "FATAL TrussCPP::TrussCPP() - out of memory, could not get a copy of the M
    exit(-1);
  }

  // fill in the ID containing external node info with node id's
  if (externalNodes.Size() != 2) {
    opserr << "FATAL TrussCPP::TrussCPP() - out of memory, could not create an ID of siz
    exit(-1);
  }

  externalNodes(0) = Nd1;
  externalNodes(1) = Nd2;

  theNodes[0] = 0;
  theNodes[1] = 0;
}
```

DRAFT January 22, 2009

```
// constructor which should be invoked by an FE_ObjectBroker only
TrussCPP::TrussCPP()
:Element(0,ELE_TAG_TrussCPP),
 theMaterial(0),
 externalNodes(2),
 trans(1,4), L(0.0), A(0.0)
{
  theNodes[0] = 0;
  theNodes[1] = 0;
}

//  destructor - provided to clean up any memory
TrussCPP::~TrussCPP()
{
    // clean up the memory associated with the element, this is
    // memory the TrussCPP objects allocates and memory allocated
    // by other objects that the TrussCPP object is responsible for
    // cleaning up, i.e. the MaterialObject.

    if (theMaterial != 0)
delete theMaterial;
}

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

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

Node **
TrussCPP::getNodePtrs(void)
{
  return theNodes;
}

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

 void
TrussCPP::setDomain(Domain *theDomain)
{
    // check Domain is not null - invoked when object removed from a domain
    if (theDomain == 0) {
return;
    }

    // first ensure nodes exist in Domain and set the node pointers
    Node *end1Ptr, *end2Ptr;
    int Nd1 = externalNodes(0);
    int Nd2 = externalNodes(1);
```

DRAFT January 22, 2009

```
    end1Ptr = theDomain->getNode(Nd1);
    end2Ptr = theDomain->getNode(Nd2);
    if (end1Ptr == 0) {
      opserr << "WARNING Truss::setDomain() - at truss " << this->getTag() << " node " <
Nd1 << "  does not exist in domain\n";

return;  // don't go any further - otherwise segemntation fault
    }
    if (end2Ptr == 0) {
      opserr << "WARNING Truss::setDomain() - at truss " << this->getTag() << " node " <
Nd2 << "  does not exist in domain\n";

return;  // don't go any further - otherwise segemntation fault
    }
    theNodes[0] = end1Ptr;
    theNodes[1] = end2Ptr;
    // call the DomainComponent class method THIS IS 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)) {
      opserr << "TrussCPP::setDomain(): 2 dof required at nodes\n";
      return;
    }

    // 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) {
      opserr << "WARNING TrussCPP::setDomain() - TrussCPP " << this->getTag() <<
" has zero length\n";
      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;
}


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

DRAFT January 2, 2009

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

int
TrussCPP::revertToStart()
{
    return theMaterial->revertToStart();
}

int
TrussCPP::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;
}


const Matrix &
TrussCPP::getTangentStiff(void)
{
    if (L == 0.0) { // length = zero - problem in setDomain() warning message already pr
trussK.Zero();
return trussK;
    }

    // get the current E from the material for the last updated strain
    double E = theMaterial->getTangent();

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

    // return the matrix
    return trussK;
}

const Matrix &
TrussCPP::getInitialStiff(void)
{
    if (L == 0.0) { // length = zero - problem in setDomain() warning message already pr
trussK.Zero();
return trussK;
    }

    // get the current E from the material for the last updated strain
    double E = theMaterial->getInitialTangent();

    // form the tangent stiffness matrix
    trussK = trans^trans;
```

DRAFT January 22, 2009

```
    trussK *= A*E/L;

    // return the matrix
    return trussK;
}

const Matrix &
TrussCPP::getDamp(void)
{
  return trussD;
}

const Matrix &
TrussCPP::getMass(void)
{
  trussM.Zero();
  return trussM;
}

const Vector &
TrussCPP::getResistingForce()
{
    if (L == 0.0) { // if length == 0, problem in setDomain()
trussR.Zero();
return trussR;
    }

    // want: 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;
}

int
TrussCPP::sendSelf(int commitTag, Channel &theChannel)
{
    int res;

    // 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();

    // TrussCPP 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(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.
```

DRAFT January 22, 2009

```
    if (matDbTag == 0) {
matDbTag = theChannel.getDbTag();
if (matDbTag != 0)
    theMaterial->setDbTag(matDbTag);
    }
    data(3) = matDbTag;

    res = theChannel.sendVector(dataTag, commitTag, data);
    if (res < 0) {
      opserr << "WARNING TrussCPP::sendSelf() - failed to send Vector\n";
      return -1;
    }

    // TrussCPP then sends the tags of it's two end nodes
    res = theChannel.sendID(dataTag, commitTag, externalNodes);
    if (res < 0) {
      opserr << "WARNING TrussCPP::sendSelf() - failed to send ID\n";
      return -2;
    }

    // finally TrussCPP asks it's material object to send itself
    res = theMaterial->sendSelf(commitTag, theChannel);
    if (res < 0) {
      opserr << "WARNING TrussCPP::sendSelf() - failed to send the Material\n";
      return -3;
    }

    return 0;
}

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

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

    Vector data(5);
    res = theChannel.recvVector(dataTag, commitTag, data);
    if (res < 0) {
      opserr << "WARNING TrussCPP::recvSelf() - failed to receive Vector\n";
      return -1;
    }

    this->setTag((int)data(0));
    A = data(1);

    // TrussCPP now receives the tags of it's two external nodes
    res = theChannel.recvID(dataTag, commitTag, externalNodes);
    if (res < 0) {
      opserr << "WARNING TrussCPP::recvSelf() - failed to receive ID\n";
      return -2;
    }

    // we create a material object of the correct type,
```

DRAFT January 22, 2009

```
    // sets its database tag and asks this new object to recveive itself.
    int matClass = data(2);
    int matDb = data(3);

    theMaterial = theBroker.getNewUniaxialMaterial(matClass);
    if (theMaterial == 0) {
      opserr << "WARNING TrussCPP::recvSelf() - failed to create a Material\n";
      return -3;
    }

    // we set the dbTag before we receive the material  - this is important
    theMaterial->setDbTag(matDb);
    res = theMaterial->recvSelf(commitTag, theChannel, theBroker);
    if (res < 0) {
      opserr << "WARNING TrussCPP::recvSelf() - failed to receive the Material\n";
      return -3;
    }

    return 0;
}

void
TrussCPP::Print(OPS_Stream &s, int flag)
{
      s << "Element: " << this->getTag();
      s << " type: TrussCPP  iNode: " << externalNodes(0);
      s << " jNode: " << externalNodes(1);
      s << " Area: " << A << endln;

      s << " \t Material: " << *theMaterial;
}

Response *
TrussCPP::setResponse(const char **argv, int argc, Information &eleInformation, OPS_Stre
{
     // 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,s);

    else
return 0;
}

int
TrussCPP::getResponse(int responseID, Information &eleInfo)
{
 double strain;

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

    case 1:
```

DRAFT January 23, 2009

```
      theMaterial->setTrialStrain(strain);
      return eleInfo.setDouble(A * theMaterial->getStress());

    default:
      return 0;
  }
}

double
TrussCPP::computeCurrentStrain(void) const
{
    // NOTE this method will never be called with L == 0.0

    // determine the strain
    const Vector &disp1 = theNodes[0]->getTrialDisp();
    const Vector &disp2 = theNodes[1]->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;
}

extern "C" DllExport void *
OPS_TrussCPP(int argc, const char **argv)
{
  // check the number of arguments is correct
  if (argc < 6) {
    opserr << "WARNING insufficient arguments\n";
    opserr << "Want: element truss eleTag? iNode? jNode? A? matTag?\n";
    return 0;
  }

  // get the id and end nodes
  int iData[4];
  double dData[1];
  int numData;

  numData = 3;

  if (OPS_GetIntInput(&numData, iData) != 0) {
    opserr << "WARNING invalid element data\n";
    return 0;
  }

  int eleTag = iData[0];

  numData = 1;
  if (OPS_GetDoubleInput(&numData, dData) != 0) {
    opserr << "WARNING error reading element area for element" << eleTag << endln;
    return 0;
  }
```

DRAFT January 22, 2009

```
  numData = 1;
  if (OPS_GetIntInput(&numData, &iData[3]) != 0) {
    opserr << "WARNING error reading element material tag for element " << eleTag << end
    return 0;
  }

  int matID = iData[3];
  UniaxialMaterial *theMaterial = OPS_GetUniaxialMaterial(matID);

  if (theMaterial == 0) {
    opserr << "WARNING material with tag " << matID << "not found for element " << eleTa
    return 0;
  }

  // now create the truss and add it to the Domain
  Element *theTruss = new TrussCPP(eleTag, iData[1], iData[2], *theMaterial, dData[0]);


  if (theTruss == 0) {
    opserr << "WARNING ran out of memory creating element with tag " << eleTag << endln;
    return 0;
  }

  return theTruss;
}
\ssp\begin{verbatim}
```

DRAFT January 22, 2009

4.4 Example C Element

The goal of this example is to ensure that the command

```
element trussC 1 1 4 10.0 1
```

is understood by the compiler. trussC is a new element type that is implemented using a c procedure. The example code for this comprises a single procedure in the file trussC.c. This file contains a single procedure trussC. The name of the procedure is important for the OpenSees interptreter when it encounerts a new element type it knows nothing about will look for a library with the name trussC containing the procedure trussC. The trussC procedure interface follows that of the new element interface discussed previously. The output args are the tangent, the residual and an error code. The input args are the element object on which the procedure acts, the model state, and the isw switch. The isw switch informs the method what actions it needs to perform for each invocation of the procedure.

Comparing this code with the C++ code presented in the previous section, it is clear that the single procedure contains all the methods of the C++ code and the isw switch is responsible for determining which method is to be invoked. For example when isw is equal to ISW_COMMIT, the code of the commit method is implemented on the material

object. The only case where the code is not identical is the case when isw is equal to ISW_INIT. For this case the procedure is responsible for reading from input using the procedure OPS_GetIntInput() and OPS_GetDoubleInput() the variables on the command line for the material, for allocating memory for the parameter and history variables in the object and for then initializing these variables. It is important to note that once the materials tag, number of parameters and number of state variables are defined that the procedure calls OPS_AllocateMaterial with the material object so that space for these variables is set aside. The procedure does not have to call a procedure to release this memory when the isw switch is set to ISW_DELETE, this switch is only used in materials that use additional memory not provided in the interface.

```
extern "C" DllExport void
trussC (eleObj *thisObj, modelState *model, double *tang, double *resid, int *isw, int *
{
  double matStrain[1];
  double matTang[1];
  double matStress[1];

  if (*isw == ISW_INIT) {

    double dData[1];
    int    iData[4];

    /* get the input data  - tag? nd1? nd2? A? matTag? */
    int numData = 3;
    OPS_GetIntInput(&numData, iData);
    numData = 1;
    OPS_GetDoubleInput(&numData, dData);
    OPS_GetIntInput(&numData, &iData[3]);

    /* Allocate the element state */
    thisObj->tag = iData[0];
    int n1 = iData[1];
    int n2 = iData[2];
    int matTag = iData[3];

    thisObj->nNode = 2;
    thisObj->nParam = 4;
    thisObj->nDOF = 4;
    thisObj->nState = 0;
    thisObj->nMat = 1;
    iData[0] = matTag;

    int *matData = iData;

    int matType = OPS_UNIAXIAL_MATERIAL_TYPE;
    OPS_AllocateElement(thisObj, matData, &matType);

    /* fill in the element state */
    thisObj->node[0] = n1;
    thisObj->node[1] = n2;
    double nd1Crd[2];
```

DRAFT January 22, 2009

```
  double nd2Crd[2];
  int numCrd = 2;

  thisObj->param[0] = dData[0];

  OPS_GetNodeCrd(&n1, &numCrd, nd1Crd);
  OPS_GetNodeCrd(&n2, &numCrd, nd2Crd);

  double dx = nd2Crd[0]-nd1Crd[0];
  double dy = nd2Crd[1]-nd1Crd[1];
  double L = sqrt(dx*dx + dy*dy);

  thisObj->param[1] = L;
  if (L == 0.0) {
    OPS_Error("Warning - truss element has zero length\n", 1);
    return;
  }

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

  thisObj->param[2] = cs;
  thisObj->param[3] = sn;

  /* ****************************
     placed in AllocateElement
     matObj *theMat = OPS_GetMaterial(&(iData[3]));
     if (theMat == 0) {
     //      OPS_Error("Warning - truss element could not find material\n",1);
     return;
     }
     thisObj->mats[0] = theMat;
     ******************************************** */

  *errFlag = 0;

} else if (*isw == ISW_COMMIT) {

    matObj *theMat = thisObj->mats[0];
    theMat->matFunctPtr(theMat, model, matStrain, matTang, matStress, isw, errFlag);

} else if (*isw == ISW_REVERT) {

  matObj *theMat = thisObj->mats[0];
 theMat->matFunctPtr(theMat, model, matStrain, matTang, matStress, isw, errFlag);

} else if (*isw == ISW_REVERT_TO_START) {

  matObj *theMat = thisObj->mats[0];
  theMat->matFunctPtr(theMat, model, matStrain, matTang, matStress, isw, errFlag);

} else if (*isw == ISW_FORM_TANG_AND_RESID) {

  double L = thisObj->param[1];
  if (L == 0.0) {
    //      OPS_Error("Warning - truss element has zero length\n", 1);
    return;
  }
```

DRAFT January 22, 2009

```
    double d1[2];
    double d2[2];

    int nd1 = thisObj->node[0];
    int nd2 = thisObj->node[1];

    int numDOF = 2;
    OPS_GetNodeDisp(&nd1, &numDOF, d1);
    OPS_GetNodeDisp(&nd2, &numDOF, d2);

    double A = thisObj->param[0];
    double cs = thisObj->param[2];
    double sn = thisObj->param[3];

    double tran[4];
    tran[0] = -cs;
    tran[1] = -sn;
    tran[2] = cs;
    tran[3] = sn;

    double dLength = 0.0;
    for (int i=0; i<2; i++){
      dLength -= (d2[i]-d1[i]) * tran[i];
    }

    matStrain[0] = dLength/L;

    matObj *theMat = thisObj->mats[0];
    theMat->matFunctPtr(theMat, model, matStrain, matTang, matStress, isw, errFlag);

    /* ****************** instead of call material funtion
    *errFlag = OPS_InvokeMaterialDirectly(theMat, model, matStrain,
                                         matStress, matTang, isw);
    ********************************************************** */

    if (*errFlag == 0) {

      double force = A*matStress[0];

      for (int i=0; i<4; i++)
        resid[i] = tran[i]*force;

      double k = A*matTang[0]/L;

      // tang(j,i)
      for (int i = 0; i<4; i++)
        for (int j=0; j < 4; j++)
          tang[i+j*4] = k * tran[i]*tran[j];

    }

  } else if (*isw == ISW_FORM_MASS) {

    double L = thisObj->param[1];
    if (L == 0.0) {
      //      OPS_Error("Warning - truss element has zero length\n", 1);
```

DRAFT January 22, 2009

```
      return;
    }
    double A = thisObj->param[0];

    double rho = thisObj->param[4];
    for (int i=0; i<16; i++)
      tang[i] = 0.0;

    if (rho != 0.0) {
      double massV  = rho * A * L/2;
      tang[0] = massV;
      tang[1+1*4] = massV;
      tang[2+2*4] = massV;
      tang[3+3*4] = massV;
    }

    *errFlag = 0;
  }
}
```

4.5 Example Fortran Element

The goal of this example is to ensure that the command

```
element trussf_ 1 1 4 10.0 1
```

is understood by the compiler. Trussf is a new element type that is implemented using a fortran procedure. The example code for this comprises a single procedure in the file TrussF.f. This file contains a single procedure TrussF. Again and for the same reasons presented in the previous c example, the name of the procedure is important. It should be noted that unlike the c example, the command contains all underscores. This is a consequence of the fortran compiler, which for the compiler used output the procedure in all lower case. The code in the example is virtually the same as for the previous c example. The only thing to note is that before calling memory allocated for the pointers the fortran code must make a call to the fortran routine c_f_pointer().

```
    SUBROUTINE TrussF(eleObj,modl,u,tang,resid,isw,error)
      use elementTypes
      use elementAPI
      implicit none

      type(eleObject)::eleObj
      type(modelState)::modl
      double precision u(1)
      double precision tang(4, *)
      double precision resid(1)
      integer::isw;
      integer::error;
```

DRAFT January 22, 2009

```
      integer :: tag, nd1, nd2, matTag, numCrd, i, j, numDOF
      real *8, pointer::theParam(:)
      integer, pointer::theNodes(:)

      double precision A, dx, dy, L, cs, sn
      double precision dLength, force, k

      integer :: iData(3);
      integer :: matTags(2);
c      integer (c_int), target :: matTags(2);

      type(c_ptr) :: theCMatPtr
      type(c_ptr), pointer :: theCMatPtrPtr(:)
      type(matObject), pointer :: theMat

      double precision dData(1), nd1Crd(2), nd2Crd(2)
      double precision d1(2), d2(2), tran(4)
      double precision strs(1), strn(1), tng(1)

      integer numData, err, matType

c     outside functions called
      integer OPS_GetIntInput, OPS_GetDoubleInput, OPS_InvokeMaterial
      integer OPS_GetNodeCrd, OPS_AllocateElement, OPS_GetNodeDisp

      IF (isw.eq.ISW_INIT) THEN

c     get the input data  - tag? nd1? nd2? A? matTag?

         numData = 3
         err = OPS_GetIntInput(numData, iData)
         tag = iData(1);
         nd1 = iData(2);
         nd2 = iData(3);

         numData = 1
         err = OPS_GetDoubleInput(numData, dData)
         A = dData(1);

         numData = 1
         err = OPS_GetIntInput(numData, iData)
         matTag = iData(1);

c     Allocate the element state

         eleObj%tag = tag
         eleObj%nnode = 2
         eleObj%ndof = 4
         eleObj%nparam = 4
         eleObj%nstate = 0
         eleObj%nmat = 2

         matTags(1) = matTag;
         matType = OPS_UNIAXIAL_MATERIAL_TYPE;
         err = OPS_AllocateElement(eleObj, matTags, matType)
```

DRAFT January 22, 2009

```
c         theCMatPtr = theCMatPtrPtr(2);
c         j=OPS_InvokeMaterialDirectly(theCMatPtr, modl, strn, strs,
c     +        tng, isw)

c         element sets material functions
c         call c_f_pointer(eleObj%mats, theCMatPtrPtr, [1]);
c         theCMatPtrPtr(1) = theCMatPtr;

c     Initialize the element properties

         call c_f_pointer(eleObj%param, theParam, [4]);
         call c_f_pointer(eleObj%node, theNodes, [2]);

         numCrd = 2;
         err = OPS_GetNodeCrd(nd1, numCrd, nd1Crd);
         err = OPS_GetNodeCrd(nd2, numCrd, nd2Crd);

         dx = nd2Crd(1)-nd1Crd(1);
         dy = nd2Crd(2)-nd1Crd(2);

         L = sqrt(dx*dx + dy*dy);
         if (L == 0.0) then
c            OPS_Error("Warning - truss element has zero length\n", 1);
            return;
         end if

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

         theParam(1) = A;
         theParam(2) = L;
         theParam(3) = cs;
         theParam(4) = sn;

         theNodes(1) = nd1;
         theNodes(2) = nd2;

         write(*,*) "nd1,nd2,A,L,cs,sn",tag, nd1, nd2, A, L, cs, sn

      ELSE

         IF (isw == ISW_COMMIT) THEN

            call c_f_pointer(eleObj%mats, theCMatPtrPtr, [1]);
            theCMatPtr = theCMatPtrPtr(1);

            j=OPS_InvokeMaterialDirectly(theCMatPtr, modl, strn, strs,
     +       tng, isw)

         ELSE IF (isw == ISW_REVERT_TO_START) THEN

            call c_f_pointer(eleObj%mats, theCMatPtrPtr, [1]);
            theCMatPtr = theCMatPtrPtr(1);

            j=OPS_InvokeMaterialDirectly(theCMatPtr, modl, strn, strs,
     +       tng, isw)
```

DRAFT January 22, 2009

```
         ELSE IF (isw == ISW_FORM_TANG_AND_RESID) THEN

            call c_f_pointer(eleObj%param, theParam, [4]);
            call c_f_pointer(eleObj%node, theNodes, [2]);
            call c_f_pointer(eleObj%mats, theCMatPtrPtr, [1]);
            theCMatPtr = theCMatPtrPtr(1);

            A = theParam(1);
            L = theParam(2);
            cs = theParam(3);
           nd1 = theNodes(1);
            nd2 = theNodes(2);

            numDOF = 2;
            err = OPS_GetNodeDisp(nd1, numDOF, d1);
            err = OPS_GetNodeDisp(nd2, numDOF, d2);

            tran(1) = -cs;
            tran(2) = -sn;
            tran(3) = cs;
            tran(4) = sn;

            dLength = 0.0;
            do 10 i = 1,2
               dLength = dLength - (d2(i)-d1(i)) * tran(i);
 10         continue

            strn(1) = dLength/L;

c            i = 0
c            i=OPS_InvokeMaterial(eleObj, i, modl, strn, strs, tng, isw)

            j=OPS_InvokeMaterialDirectly(theCMatPtr, modl, strn, strs,
     +       tng, isw)

            force = A*strs(1);
            k = A*tng(1)/L;

c            write(*,*) "A,L,cs,sn,K,force",A, L, cs, sn, k, force

            do 20 i =1,4
               resid(i) = force * tran(i);
               do 30 j = 1,4
                  tang(i,j) = k * tran(i)*tran(j);
 30            continue
 20         continue

         END IF

      END IF

c     return error code
      error = 0

      END SUBROUTINE trussf
```

DRAFT January 22, 2009

5 C and Fortran API

- Methods to allocate space for the material and element objects once the initial data has been set in the object, e.g. tag, nParam, nState variables:

```
extern "C" int  OPS_AllocateMaterial(matObject *);

PUBLIC OPS_AllocateMaterial
  interface
  function OPS_AllocateMaterial(mat)
     use elementtypes
     integer :: OPS_AllocateMaterial
     type(matObject) :: mat
   end function OPS_AllocateMaterial
   end interface
```

Allocates the theParam, cState and tState arrays in the matObject. The theParam array's size is set equal to nParam. The cState and tState arrays sizes are set to nState. The method returns 0 if successfull, a negative number if not.

```
extern "C" int  OPS_AllocateElement(eleObject *, int *matTags, int *matType);

PUBLIC OPS_AllocateElement
  interface
  function OPS_AllocateElement(ele, matTags, matType)
     use elementtypes
     use iso_c_binding
     integer :: OPS_AllocateElement
     type(eleObject) :: ele
     integer :: matTags(*)
     integer :: matType
   end function OPS_AllocateElement
end interface
```

Allocates the node, param, cState, tState, and mats arrays in the eleObject. The function will also allocate the materials. The node array array's size is set equal to nNode The cState and tState arrays sizes are set to nState. The mats array is set equal to nMat. The method returns 0 if successfull, a negative number if not. Copies of the materials are obtained. The tags for the materials that are obtained are specified in the matTags array. The material type, e.g. Uniaxial, nD_PlaneStrain, etc are specified in the integer matType.

```
extern "C" matObj *OPS_GetMaterial(int *matTag, int *matType) ;

PUBLIC OPS_GetMaterial
interface
   function OPS_GetMaterial(matTag, matType)
     use iso_c_binding
     use elementtypes
```

DRAFT January 22, 2009

```
     type(c_ptr) :: OPS_GetMaterial
     integer     :: matTag
     integer     :: matType
   end function OPS_GetMaterial
end interface
```

A procedure to return a copy of a matObject with matTag and of matType that exists. Returns 0 if fails to find a material.

- methods obtain input args from the command line:

```
extern "C" int  OPS_GetIntInput(int *numData, int*data);
```

```
PUBLIC OPS_GetIntInput
interface
   function OPS_GetIntInput(numData, iData)
     integer       :: OPS_GetIntInput
     integer       :: numData
     integer       :: iData(*)
   end function OPS_GetIntInput
end interface
```

Obtains the next numData values from the input, treats them as integers and places them in the data array.

```
extern "C" int  OPS_GetDoubleInput(int *numData, double *data);
```

Obtains the next numData values from the input, treats them as doubles and places them in the data array.

- methods to obtain nodal coordinates and response quantaties.

```
extern "C" int  OPS_GetNodeCrd(int *nodeTag, int *sizeData, double *data);
```

```
PUBLIC OPS_GetNodeCrd
interface
   function OPS_GetNodeCrd(nodeTag, numData, dData)
     integer   :: OPS_GetNodeCrd
     integer   :: nodeTag
     integer   :: numData
     real *8   :: dData(*)
   end function OPS_GetNodeCrd
   end interface
```

Method that returns the nodal coordinates at node nodeTag. The first sizeData coordinates are placed in the data array.

DRAFT January 22, 2009

```
extern "C" int  OPS_GetNodeDisp(int *nodeTag, int *sizeData, double *data);

PUBLIC OPS_GetNodeDisp
interface
   function OPS_GetNodeDisp(nodeTag, numData, dData)
     integer   :: OPS_GetNodeDisp
     integer   :: nodeTag
     integer   :: numData
     real *8   :: dData(*)
   end function OPS_GetNodeDispend interface
```

Method that returns the trial nodal displacements at node nodeTag. The first sizeData displacements are placed in the data array.

```
extern "C" int  OPS_GetNodeVel(int *nodeTag, int *sizeData, double *data);

PUBLIC OPS_GetNodeVel
interface
   function OPS_GetNodeVel(nodeTag, numData, dData)
     integer   :: OPS_GetNodeVel
     integer   :: nodeTag
     integer   :: numData
     real *8   :: dData(*)
   end function OPS_GetNodeVel
end interface
```

Method that returns the trial nodal velocities at node nodeTag. The first sizeData velocities are placed in the data array

```
extern "C" int  OPS_GetNodeAcc(int *nodeTag, int *sizeData, double *data);

PUBLIC OPS_GetNodeAccel
interface
   function OPS_GetNodeAccel(nodeTag, numData, dData)
     integer   :: OPS_GetNodeAccel
     integer   :: nodeTag
     integer   :: numData
     real *8   :: dData(*)
   end function OPS_GetNodeAcel
end interface
```

Method that returns the trial nodal accelerations at node nodeTag. The first sizeData accelerations are placed in the data array

- method to invoke the material objects for fortran developers.

```
extern "C" int
OPS_InvokeMaterialDirectly(matObject **theMat, modelState *model,
   double *strain,  double *stress,double *tang, int *isw);

PUBLIC OPS_InvokeMaterialDirectly
```

DRAFT January 22, 2009

```
interface
  function OPS_InvokeMaterialDirectly(mat, md, strn, strs, tang, i)
     use elementtypes
     use iso_c_binding
     integer   :: OPS_InvokeMaterialDirectly
     type(c_ptr) :: mat
     type(modelState) :: md
     real *8   :: strn(*)
     real *8   :: strs(*)
     real *8   :: tang(*)
     integer   :: i
   end function OPS_InvokeMaterialDirectly
end interface
```

To invoke the material procedure in the matObject theMat, with theMat as the input matObject, model as input modelState, strain as input strain, and isw as input isw.

- method to send a message to output.

```
extern "C" int  OPS_Error(char *msg, int length);
```

Method to send to output the first length characters of the character array msg.

6 Additional C++ API

Additional methods available for the C++ developer.

```
UniaxialMaterial * OPS_GetUniaxialMaterial(int matTag);
```

To return a copy of a previously entered UniaxialMaterial with tag matTag. Returns 0 if th material does not exist.

```
NDMaterial * OPS_GetNDMaterial(int matTag);
```

To return a copy of a previously entered nDMaterial with tag matTag. Returns 0 if th material does not exist.

Corresponding Author For further information, please contact Frank McKenna, Department of Civil and Environmental Engineering, University of California, Berkeley, Berkeley CA 94720-1711. Email:fmckenna@ce.berkeley.edu

DRAFT January 22, 2009