How to Introduce a New Material into OpenSees

Version 1.1

August 21, 2001

Michael H. Scott and Gregory L. Fenves
PEER, University of California, Berkeley

1 Introduction

This document shows how to add a new material implementation to OpenSees. The hierarchical nature of the OpenSees software architecture allows new material models to be seamlessly added to the framework. By keeping element and material implementations separate, a new material model can be used in an existing element without modifying the element implementation, and vice versa. The programming language C++ directly supports the data encapsulation and run-time binding necessary to achieve this complete separation of material from element.

2 Material Abstractions

Currently, there are three Material abstractions in OpenSees, each of which can be used across a wide range of element implementations:

1. **UniaxialMaterial** - Provides the interface for all one-dimensional material models, either stress-strain or force-deformation. UniaxialMaterial models define the stress-strain response of a truss element, uniaxial fiber behavior in a beam-column section, or the force-deformation response of a beam section or zero-length element.

2. **NDMaterial** - The multi-dimensional generalization of UniaxialMaterial; provides the stress-strain response at a point in a solid element, or multi-dimensional fiber behavior in a plate or beam-column section.

3. **SectionForceDeformation** - Defines the interface for stress resultant models which are used to describe both plate and beam-column force-deformation response as well as the constitutive response of more general zero-length elements, e.g., for isolator bearings.

Each interface listed above is essentially the same with minor differences. The NDMaterial and SectionForceDeformation abstractions both represent multi-dimensional constitutive response. However, a distinction is made between stress and stress *resultant* response to allow for safer element implementations. Furthermore, the stress-strain equations for continuum material models can be written in terms of tensors. This is not the case for stress resultant models. Lastly, to avoid returning matrices and vectors or tensors of size one, the UniaxialMaterial abstraction is made distinct for reasons of efficiency, as scalar values describe the behavior of a one-dimensional model.

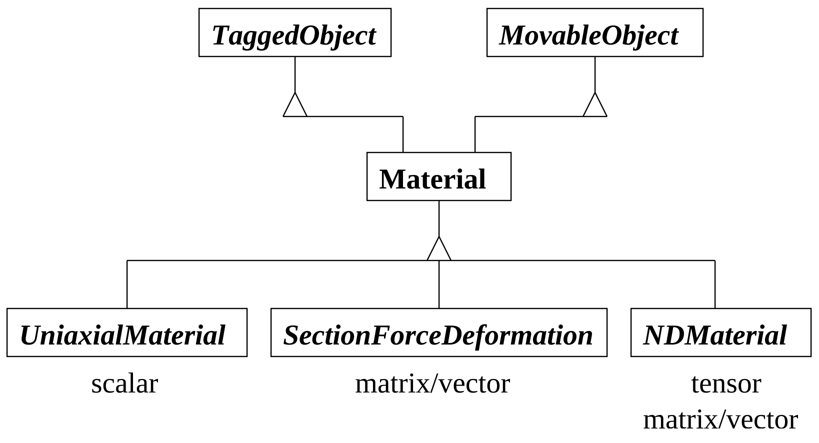

Figure 1: Material class hierarchy

As indicated in figure 1, each material abstraction is a subclass of Material. The Material class is a subclass of both the TaggedObject and MovableObject classes, and therefore inherits the functionality of these two classes. As a result, it can be said that a Material "is a" TaggedObject as well as a MovableObject. Furthermore, since each of UniaxialMaterial, NDMaterial, and SectionForceDeformation "is a" Material, each is also a TaggedObject and a MovableObject. The TaggedObject class provides functionality for identifying materials, through a tag, during model building; and the MovableObject class provides functionality for parallel processing and database programming.

Rather than show examples of implementing a material model under each interface, only the UniaxialMaterial interface is covered herein. The basic concepts of adding a material model to OpenSees carry directly over from UniaxialMaterial to NDMaterial and SectionForceDeformation.

The remainder of this document is laid out as follows. First, the UniaxialMaterial interface is listed and explained. Then, an example UniaxialMaterial implementation, HardeningMaterial, is presented. Along with the C++ implementation, it is shown how to 1) add the new model to the OpenSees Tcl model builder, and 2) make the new model "movable" for parallel processing and database programming. Finally, a FORTRAN interface for programming UniaxialMaterial models in OpenSees is described.

3 UniaxialMaterial Interface

Implementations of the UniaxialMaterial interface are used in several contexts within the OpenSees modeling framework. Due to their simplicity, these models can define both stress-strain and force-deformation relationships. It is up to the calling object, be it an element object or another material object, to interpret the meaning appropriately.

Listed below is the UniaxialMaterial class interface. All methods in the UniaxialMaterial interface are public, there are no protected or private data or methods. Following the UniaxialMaterial class interface listing, each method in the interface is described.

```
#include <Material.h>

class Response;
class Information;

class UniaxialMaterial : public Material
{
   public:
      UniaxialMaterial(int tag, int classTag);
      virtual ~UniaxialMaterial(void);

      virtual int setTrialStrain(double strain, double strainRate = 0.0) = 0;
      virtual double getStrain(void) = 0;
      virtual double getStrainRate(void);
      virtual double getStress(void) = 0;
      virtual double getTangent(void) = 0;
      virtual double getDampTangent(void);
      virtual double getSecant(void);

      virtual int commitState(void) = 0;
      virtual int revertToLastCommit(void) = 0;
      virtual int revertToStart(void) = 0;

      virtual UniaxialMaterial *getCopy(void) = 0;

      virtual Response *setResponse(char **argv, int argc, Information &matInfo);
      virtual int getResponse(int responseID, Information &matInfo);

   protected:

   private:
};
```

A note about the C++ syntax seen in the UniaxialMaterial interface. The keyword "virtual" at the start of a method declaration indicates this method may be overridden by a subclass of UniaxialMaterial. The UniaxialMaterial base class provides default implementations for its virtual methods. The notation "= 0" at the end of the method declaration indicates the method is *pure* virtual, meaning it *must* be defined by subclasses because the UniaxialMaterial base class does not provide a default implementation.

The UniaxialMaterial base class constructor takes a tag and classTag as its arguments. The tag passed to the constructor identifies this UniaxialMaterial as unique among all other UniaxialMaterial objects, and the classTag is used primarily for parallel processing and database programming. Class tags are defined in the file classTags.h. The tag and classTag arguments are passed to the Material class constructor, where they are in turn passed to the TaggedObject and MovableObject class constructors, respectively. The UniaxialMaterial destructor is declared, but does not do anything as the UniaxialMaterial base class contains no data.

The method *setTrialStrain()* takes one or two arguments, an updated strain and strain rate. The strain rate is an optional argument, with default value 0.0. This method is pure virtual, so it must be implemented in all subclasses of UniaxialMaterial. The next two methods, *getStrain()* and *getStrainRate()*, are to return the current strain and strain rate of this UniaxialMaterial. The method *getStrain()* is pure virtual, while *getStrainRate()* is only virtual; by default it returns 0.0, but may be overridden in subclasses if needed.

```
double
UniaxialMaterial::getStrainRate(void)
{
   return 0.0;
}
```

The next method is *getStress()*, which is to return the current stress of this UniaxialMaterial. The current stress is a function of the current strain, ε, and the current strain rate, $\dot{\varepsilon}$,

$$\sigma = \sigma(\varepsilon, \dot{\varepsilon})\ . \tag{1}$$

The *getStress()* method is pure virtual and must be implemented by subclasses of UniaxialMaterial.

The current material tangent is returned by the next method, *getTangent()*. The material tangent is the partial derivative of the material stress with respect to the current strain,

$$D_t = \frac{\partial \sigma}{\partial \varepsilon}\ . \tag{2}$$

The *getTangent()* is also pure virtual and must be implemented in all UniaxialMaterial subclasses.

The *getDampTangent()* method is next, and is to return the current damping tangent, which is the partial derivative of the current stress with respect to the current strain rate,

$$\eta = \frac{\partial \sigma}{\partial \dot{\varepsilon}}\ . \tag{3}$$

By default, this method returns 0.0, and it may be overridden in subclasses of UniaxialMaterial where there is strain rate dependence.

```
double
UniaxialMaterial::getDampTangent(void)
{
   return 0.0;
}
```

Finally, the *getSecant()* method is provided to return the material secant, which is the current stress divided by the current strain,

$$D_s = \frac{\sigma}{\varepsilon}\ . \tag{4}$$

By default, this method returns the result of dividing the current stress by the current strain. If the current strain is zero, the current tangent is returned instead.

```
double
UniaxialMaterial::getSecant(void)
{
   double strain = this->getStrain();
   double stress = this->getStress();

   if (strain != 0.0)
      return stress/strain;
   else
      return this->getTangent();
}
```

The next set of methods deal with possible path dependent behavior of UniaxialMaterial models. All Material objects in OpenSees are responsible for keeping track of and updating their own history variables. First, the method *commitState()* is invoked to inform a UniaxialMaterial object that its current state is on the converged solution path and its internal history variables should be updated accordingly. Next, the method *revertToLastCommit()* is provided to let a UniaxialMaterial object know that it should return to its last committed state at. Finally, *revertToStart()* informs the UniaxialMaterial object to revert to its initial state, i.e., at the start of the analysis. All three of these methods are pure virtual, and thus must be implemented in all subclasses of UniaxialMaterial.

The *getCopy()* method is declared so a calling object, be it an Element, Fiber, or another Material object, can obtain an exact copy of this UniaxialMaterial object. A pointer to the new object is returned by this function, and the calling object is responsible for deleting this dynamically allocated memory. This method is pure virtual because only a subclass of UniaxialMaterial knows the internal representation of its data.

The final two methods, *setResponse()* and *getResponse()*, are declared for recording UniaxialMaterial state information. These methods have default implementations to record the material stress, strain, and tangent. These methods may be overridden, but their implementations are not shown in this document.

4 Example – HardeningMaterial

In this section, it is shown how the rate-independent uniaxial hardening material model given in Simo & Hughes, *Computational Inelasticity* (1998) is implemented in OpenSees. First, the class implementation is shown, followed by its inclusion in the Tcl model builder.

4.1 Class Implementation

The HardeningMaterial class interface is shown below. Here, no methods are virtual since this class provides implementations for the corresponding methods inherited from the UniaxialMaterial class.

Note, three additional methods not declared in the UniaxialMaterial interface, *sendSelf()*, *recvSelf()*, and *Print()*, must be defined in implementations of UniaxialMaterial. These methods are inherited from higher level classes in the OpenSees framework, particularly,

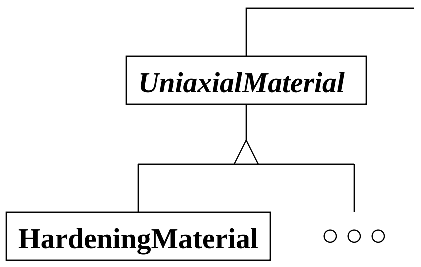

Figure 2: HardeningMaterial class hierarchy

TaggedObject and MovableObject. An explanation of these methods is provided in what follows.

```
#include <UniaxialMaterial.h>

class HardeningMaterial : public UniaxialMaterial
{
   public:
      HardeningMaterial(int tag, double E, double sigmaY, double Hiso, double Hkin);
      HardeningMaterial();
      ~HardeningMaterial();

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

      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(ostream &s, int flag = 0);

   protected:

   private:
      // Material parameters
      double E;       // Elastic modulus
      double sigmaY;  // Yield stress
      double Hiso;    // Isotropic hardening modulus
      double Hkin;    // Kinematic hardening modulus

      // Committed history variables
      double CplasticStrain;  // Committed plastic strain
      double CbackStress;     // Committed back stress;
      double Chardening;      // Committed internal hardening variable

      // Trial history variables
      double TplasticStrain;  // Trial plastic strain
      double TbackStress;     // Trial back stress
      double Thardening;      // Trial internal hardening variable

      // Trial state variables
      double Tstrain;   // Trial strain
      double Tstress;   // Trial stress
      double Ttangent;  // Trial tangent
};
```

The first two methods defined for HardeningMaterial are the constructors. The first constructor takes the material tag and the material parameters: elastic modulus, E, yield stress, σ_y, isotropic hardening modulus, H_{iso}, and kinematic hardening modulus, H_{kin}. The UniaxialMaterial base class constructor is invoked with the arguments tag and MAT_TAG_Hardening (defined in classTags.h). The material parameters for this object are initialized in the initialization list with the arguments passed to the constructor, and all history variables are initialized by invoking *revertToStart()*. The second constructor is a default constructor which sets all material parameters to 0.0 then invokes *revertToStart()*.

```
HardeningMaterial::HardeningMaterial(int tag, double e, double s,
                                     double hi, double hk)
:UniaxialMaterial(tag,MAT_TAG_Hardening),
 E(e), sigmaY(s), Hiso(hi), Hkin(hk)
{
   // Initialize variables
   this->revertToStart();
}

HardeningMaterial::HardeningMaterial()
:UniaxialMaterial(0,MAT_TAG_Hardening),
 E(0.0), sigmaY(0.0), Hiso(0.0), Hkin(0.0)
{
   // Initialize variables
   this->revertToStart();
}
```

The next method defined is the destructor, which does nothing since no memory is dynamically allocated by a HardeningMaterial object.

```
HardeningMaterial::~HardeningMaterial()
{
   // Does nothing
}
```

The following methods deal with the material state determination. The return mapping algorithm is coded in *setTrialStrain()*. The stress and tangent of this HardeningMaterial object are computed and stored in the instance variables Tstress and Ttangent and returned by the methods *getStress()* and *getTangent()*, respectively. The trial strain, stored in the instance variable Tstrain, is returned by the method *getStrain()*.

```
int
HardeningMaterial::setTrialStrain(double strain, double strainRate)
{
   // Set total strain
   Tstrain = strain;

   // Elastic trial stress
   Tstress = E * (Tstrain-CplasticStrain);

   // Compute trial stress relative to committed back stress
   double xsi = Tstress - CbackStress;

   // Compute yield criterion
   double f = fabs(xsi) - (sigmaY + Hiso*Chardening);

   // Elastic step ... no updates required
   if (f <= 0.0) {
      // Set trial tangent
      Ttangent = E;
   }
   // Plastic step ... perform return mapping algorithm
   else {
      // Compute consistency parameter
      double dGamma = f / (E+Hiso+Hkin);

      // Find sign of xsi
      int sign = (xsi < 0) ? -1 : 1;

      // Bring trial stress back to yield surface
      Tstress -= dGamma*E*sign;

      // Update plastic strain
      TplasticStrain = CplasticStrain + dGamma*sign;

      // Update back stress
      TbackStress = CbackStress + dGamma*Hkin*sign;
      // Update internal hardening variable
      Thardening = Chardening + dGamma;

      // Set trial tangent
      Ttangent = E*(Hkin+Hiso) / (E+Hkin+Hiso);
   }

   return 0;
}

double
HardeningMaterial::getStress(void)
{
   return Tstress;
}

double
HardeningMaterial::getTangent(void)
{
   return Ttangent;
}

double
HardeningMaterial::getStrain(void)
{
   return Tstrain;
}
```

The next set of methods deal with the path dependent behavior of this HardeningMaterial object. The method *commitState()* sets the committed history variables to be their corresponding trial values. Nothing needs to be done in the method *revertToLastCommit()*, and all history variables are set to 0.0 in *revertToStart()*.

```
int
HardeningMaterial::commitState(void)
{
   // Commit trial state variables
   CplasticStrain = TplasticStrain;
   CbackStress = TbackStress;
   Chardening = Thardening;

   return 0;
}

int
HardeningMaterial::revertToLastCommit(void)
```

```
{
   // Nothing to do here
   return 0;
}

int
HardeningMaterial::revertToStart(void)
{
   // Reset committed history variables
   CplasticStrain = 0.0;
   CbackStress = 0.0;
   Chardening = 0.0;

   // Reset trial history variables
   TplasticStrain = 0.0;
   TbackStress = 0.0;
   Thardening = 0.0;

   // Initialize state variables
   Tstrain = 0.0;
   Tstress = 0.0;
   Ttangent = E;

   return 0;
}
```

The *getCopy()* method is defined so this HardeningMaterial object can provide a clone of itself to a calling object, be it an Element, Fiber, or other Material object. The constructor is invoked to create a new object, then all instance variables are copied to the new object. The calling object is responsible for deleting this dynamically allocated memory.

```
UniaxialMaterial*
HardeningMaterial::getCopy(void)
{
   HardeningMaterial *theCopy =
      new HardeningMaterial(this->getTag(), E, sigmaY, Hiso, Hkin);

   // Copy committed history variables
   theCopy->CplasticStrain = CplasticStrain;
   theCopy->CbackStress = CbackStress;
   theCopy->Chardening = Chardening;

   // Copy trial history variables
   theCopy->TplasticStrain = TplasticStrain;
   theCopy->TbackStress = TbackStress;
   theCopy->Thardening = Thardening;
   // Copy trial state variables
   theCopy->Tstrain = Tstrain;
   theCopy->Tstress = Tstress;
   theCopy->Ttangent = Ttangent;

   return theCopy;
}
```

The next two methods are defined for parallel processing and database programming, and are inherited from MovableObject. The first method, *sendSelf()*, packs the material properties and committed history variables in a Vector, then sends the Vector across the Channel object passed as an argument to the method. The second method, *recvSelf()*, receives data from the Channel object, then populates the data of this HardeningMaterial object with the received data.

```
int
HardeningMaterial::sendSelf(int cTag, Channel &theChannel)
{
   static Vector data(8);

   data(0) = this->getTag();
   data(1) = E;
   data(2) = sigmaY;
   data(3) = Hiso;
   data(4) = Hkin;
   data(5) = CplasticStrain;
   data(6) = CbackStress;
   data(7) = Chardening;

   int res = theChannel.sendVector(this->getDbTag(), cTag, data);
   if (res < 0)
      cerr << "HardeningMaterial::sendSelf() - failed to send data\n";

   return res;
}

int
HardeningMaterial::recvSelf(int cTag, Channel &theChannel,
                            FEM_ObjectBroker &theBroker)
{
   static Vector data(8);

   int res = theChannel.recvVector(this->getDbTag(), cTag, data);

   if (res < 0) {
      cerr << "HardeningMaterial::recvSelf() - failed to receive data\n";
      this->setTag(0);
   }
   else {
      this->setTag((int)data(0));
      E = data(1);
      sigmaY = data(2);
      Hiso = data(3);
      Hkin = data(4);
      CplasticStrain = data(5);
      CbackStress = data(6);
      Chardening = data(7);

      // Set the trial state variables
      revertToLastCommit();
   }

   return res;
}
```

The final HardeningMaterial method is *Print()*, which writes the material name, tag, and parameters to the output stream passed as an argument. This method is inherited from TaggedObject.

```
void
HardeningMaterial::Print(ostream &s, int flag)
{
   s << "HardeningMaterial, tag: " << this->getTag() << endl;
   s << "  E: " << E << endl;
   s << "  sigmaY: " << sigmaY << endl;
   s << "  Hiso: " << Hiso << endl;
   s << "  Hkin: " << Hkin << endl;
}
```

4.2 Tcl Model Builder

The new HardeningMaterial model must be added to the OpenSees Tcl model builder in order for it to be used by analysis models defined in Tcl script files. The general from of the uniaxialMaterial command is as follows:

```
   uniaxialMaterial materialType tag <specific material parameters>
```

So, for a HardeningMaterial object, it is necessary to read in the material parameters that are passed to its constructor, namely the elastic modulus, yield stress, and isotropic and kinematic hardening moduli. The general form of the command will be:

```
   uniaxialMaterial Hardening tag E sigmaY Hiso Hkin
```

An example command to add a HardeningMaterial object with tag 1, elastic modulus of 30000.0, yield stress of 60.0, isotropic hardening modulus of 0.0, and kinematic hardening modulus of 1000.0 may then look like:

```
uniaxialMaterial Hardening 1 30000.0 60.0 0.0 1000.0
```

How these values are parsed and used to construct a HardeningMaterial object is described next. The parsing of input data for all UniaxialMaterial models is done in the function *TclModelBuilderUniaxialMaterialCommand* contained in the file TclModelBuilderUniaxialMaterialCommand.cpp. In this file there are multiple if/else statements, one for each UniaxialMaterial that can be added to the model builder. To add the new model only requires adding an additional case with the accompanying code to parse the Tcl command line.

The above command is split into an array of character strings (argv) by the Tcl interpreter, then sent to the *TclModelBuilderUniaxialMaterialCommand* function. argv[0] contains the command name "uniaxialMaterial", argv[1] holds the material keyword "Hardening", argv[2] contains the material tag, and the remaining entries in the argv array hold the specific material parameters. These parameters are the arguments needed to call the HardeningMaterial constructor. The number of elements in the argv array is stored in the variable argc.

Calls are made to the Tcl routines *Tcl_GetInt* and *Tcl_GetDouble* to get integer and double values from the character strings contained in argv. These routines perform error checking and return the pre-defined value TCL_OK if there was no error. Once the UniaxialMaterial has been allocated, it is added to the Tcl model builder at the end of the *TclModelBuilderUniaxialMaterialCommand* function, after the multiple if/else statement has ended.

```
int
TclModelBuilderUniaxialMaterialCommand(ClientData clienData, Tcl_Interp *interp,
                                       int argc, char **argv,
                                       TclModelBuilder *theTclBuilder)
{

   // Pointer to a UniaxialMaterial that will be added to the model builder
   UniaxialMaterial *theMaterial = 0;

   if (strcmp(argv[1],"Elastic") == 0) {
      //
      // Additional code not shown
      //
   }

   else if (strcmp(argv[1],"Hardening") == 0) {
      if (argc < 7) {
         cerr << "WARNING insufficient arguments\n";
         printCommand(argc,argv);
         cerr << "Want: uniaxialMaterial Hardening tag E sigmaY Hiso Hkin" << endl;
      return TCL_ERROR;
   }

   int tag;
   double E, sigmaY, Hiso, Hkin;

   if (Tcl_GetInt(interp, argv[2], &tag) != TCL_OK) {
      cerr << "WARNING invalid uniaxialMaterial Hardening tag" << endl;
      return TCL_ERROR;
   }

   if (Tcl_GetDouble(interp, argv[3], &E) != TCL_OK) {
      cerr << "WARNING invalid E\n";
      cerr << "uniaxialMaterial Hardening: " << tag << endl;
      return TCL_ERROR;
   }

   if (Tcl_GetDouble(interp, argv[4], &sigmaY) != TCL_OK) {
      cerr << "WARNING invalid sigmaY\n";
      cerr << "uniaxialMaterial Hardening: " << tag << endl;
      return TCL_ERROR;
   }

   if (Tcl_GetDouble(interp, argv[5], &Hiso) != TCL_OK) {
      cerr << "WARNING invalid Hiso\n";
      cerr << "uniaxialMaterial Hardening: " << tag << endl;
      return TCL_ERROR;
   }

   if (Tcl_GetDouble(interp, argv[6], &Hkin) != TCL_OK) {
      cerr << "WARNING invalid Hkin\n";
      cerr << "uniaxialMaterial Hardening: " << tag << endl;
      return TCL_ERROR;
   }

   // Parsing was successful, allocate the material
   theMaterial = new HardeningMaterial(tag, E, sigmaY, Hiso, Hkin);
}

//
// Additional code not shown
//

// Now add the material to the modelBuilder
if (theTclBuilder->addUniaxialMaterial(*theMaterial) < 0) {
   cerr << "WARNING could not add uniaxialMaterial to the model builder\n";
   cerr << *theMaterial << endl;
```

```
      delete theMaterial; // invoke the material objects destructor,
                          // otherwise memory leak
      return TCL_ERROR;
   }

   return TCL_OK;
}
```

4.3 FEM_ObjectBroker

In order for the new HardeningMaterial object to be used for parallel processing and database programming, the *getNewUniaxialMaterial()* method in the FEM_ObjectBroker class must be modified. An additional case statement should be added, as shown below. The MAT_TAG_Hardening classTag is the same pre-defined value passed to the UniaxialMaterial constructor by the HardeningMaterial constructor described earlier. The FEM_ObjectBroker simply returns a blank HardeningMaterial object, whose data can be subsequently populated by invoking *recvSelf()*.

```
UniaxialMaterial*
FEM_ObjectBroker::getNewUniaxialMaterial(int classTag)
{
   switch(classTag) {

   case MAT_TAG_Hardening:
      return new HardeningMaterial();

   //
   // Additional cases not shown
   //

   default:
      cerr << "FEM_ObjectBroker::getPtrNewUniaxialMaterial - ";
      cerr << " - no UniaxialMaterial type exists for class tag ";
      cerr << classTag << endl;
      return 0;
   }
}
```

5 A FORTRAN Interface for UniaxialMaterial Models

Subclasses of UniaxialMaterial hide their implementation details from calling objects, i.e., calling objects only see the public interface defined in the UniaxialMaterial base class. How the interface is implemented is encapsulated by each subclass of UniaxialMaterial. Thus, a particular UniaxialMaterial implementation need not be written in C++ as long as the implementation conforms to the UniaxialMaterial interface.

As an example, the FEDEAS uniaxial material library developed by F.C. Filippou is used as a FORTRAN interface for UniaxialMaterial models in OpenSees. This example is meant to demonstrate the process of linking OpenSees with other material libraries and is not limited to just the FEDEAS library. Material libraries with any well-defined interface, e.g. DRAIN, may be linked in a similar manner. Similar concepts carry directly over to implementing NDMaterial and SectionForceDeformation models in FORTRAN.

5.1 FEDEAS subroutine interface

The subroutine interface defined for a FEDEAS uniaxial material model named "ML1D" is shown below.

```
subroutine ML1D(matpar,hstvP,hstv,epsP,sigP,deps,sig,tang,ist)
```

The subroutine arguments are given as follows:

1. matpar - a double array of material parameters (in)
2. hstvP - a double array of committed history variables (in)
3. hstv - a double array of trial history variables (out)
4. epsP - strain at the last committed state (in)
5. sigP - stress at the last committed state (in)
6. deps - change in strain from the last committed state (in)
7. sig - the stress at the current trial state (out)
8. tang - the tangent at the current trial state (out)
9. ist - integer indicating the operation to be performed (in): 0 - return number of material parameters and history variables, 1 - compute stress and tangent, 2 - compute stress and secant

5.2 FedeasMaterial Implementation in OpenSees

This section presents an implementation of UniaxialMaterial capable of wrapping any subroutine that conforms to the FEDEAS interface described in the previous section. This implementation, FedeasMaterial, manages the data arrays sent to the FEDEAS subroutine, i.e., FedeasMaterial is responsible for storing the material parameters and history variables, as well as for swapping trial and committed history variables.

FedeasMaterial is a subclass of UniaxialMaterial, as shown in figure 3. Subclasses of FedeasMaterial are responsible for defining constructors which take the appropriate material parameters. All other functionality (state determination, swapping of history variables, etc.) is common to all subclasses of FedeasMaterial. Therefore, this functionality is defined in the base class, FedeasMaterial.

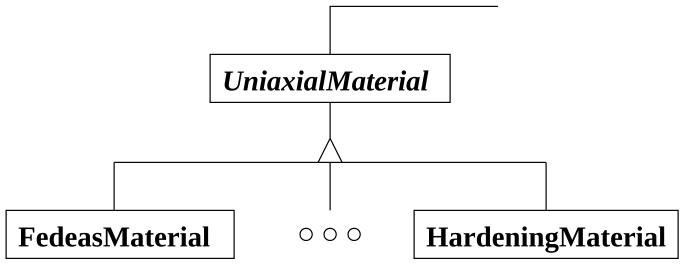

Figure 3: FedeasMaterial class hierarchy

The FedeasMaterial class interface is shown below. All methods are declared as virtual so they may be overridden by subclasses of FedeasMaterial.

```
#include <UniaxialMaterial.h>

class FedeasMaterial : public UniaxialMaterial
{
   public:
      FedeasMaterial(int tag, int classTag, int nhv, int ndata);
      virtual ~FedeasMaterial();

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

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

      virtual UniaxialMaterial *getCopy(void);

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

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

   protected:
      // Invokes the FORTRAN subroutine
      virtual int invokeSubroutine(int ist);

      double *data;         // Material parameters array
      double *hstv;         // History array: first half is committed, second is trial

      int numData;          // Number of material parameters
      int numHstv;          // Number of history variables

      double epsilonP;      // Committed strain
      double sigmaP;        // Committed stress

   private:
      double epsilon;       // Trial strain
      double sigma;         // Trial stress
      double tangent;       // Trial tangent
};
```

Instance variables are declared in FedeasMaterial to store history variables and material parameters. First, data is a double array of size numData, the number of material parameters for this object. Next, hstv is a double array of size 2*numHstv, where numHstv is the number of history variables needed for this FedeasMaterial object. Note that committed history variables are stored in the first half of the hstv array, while the trial values are kept in the second half. The values epsilonP and sigmaP are the committed strain and stress, respectively, of this FedeasMaterial object, as they are required by the FEDEAS subroutine interface. Finally, three trial state variables, epsilon, sigma, and tangent are defined to store the current strain, stress, and tangent.

The FedeasMaterial constructor initializes the number of history variables and number of material parameters with arguments passed to the constructor. The UniaxialMaterial base class constructor is invoked with the tag and classTag arguments. The trial and committed strain and stress are initialized to 0.0. Then, the history variable and material parameter arrays are allocated. All entries in the history variable and material parameter arrays are initialized to 0.0.

```
FedeasMaterial::FedeasMaterial(int tag, int classTag, int nhv, int ndata)
:UniaxialMaterial(tag,classTag),
 data(0), hstv(0), numData(ndata), numHstv(nhv),
 epsilonP(0.0), sigmaP(0.0),
 epsilon(0.0), sigma(0.0), tangent(0.0)
{
   if (numHstv < 0)
      numHstv = 0;

   if (numHstv > 0) {
      // Allocate history array
      hstv = new double[2*numHstv];
      if (hstv == 0)
         g3ErrorHandler->fatal("%s -- failed to allocate history array -- type %d",
                              "FedeasMaterial::FedeasMaterial", this->getClassTag());

      // Initialize to zero
      for (int i = 0; i < 2*numHstv; i++)
         hstv[i] = 0.0;
   }
   if (numData < 0)
      numData = 0;

   if (numData > 0) {
      // Allocate material parameter array
      data = new double[numData];
      if (data == 0)
         g3ErrorHandler->fatal("%s -- failed to allocate data array -- type %d",
                              "FedeasMaterial::FedeasMaterial", this->getClassTag());

      // Initialize to zero
      for (int i = 0; i < numData; i++)
         data[i] = 0.0;
   }
}
```

The FedeasMaterial destructor deallocates the memory allocated in the constructor to hold the history variables and material parameters.

```
FedeasMaterial::~FedeasMaterial()
{
   if (hstv != 0)
      delete [] hstv;

   if (data != 0)
      delete [] data;
}
```

The next group of FedeasMaterial methods deals with material state determination. First, *setTrialStrain()* stores the trial strain, then invokes the FEDEAS subroutine with ist = 1, indicating that normal stress and tangent quantities should be computed. The methods *getStrain()*, *getStress()*, and *getTangent()* return the strain, stress, and tangent of this FedeasMaterial.

```
int
FedeasMaterial::setTrialStrain(double strain, double strainRate)
{
   // Store the strain
   epsilon = strain;

   // Tells subroutine to do normal operations for stress and tangent
   int ist = 1;

   // Call the subroutine
   return this->invokeSubroutine(ist);
}
```

```
double
FedeasMaterial::getStrain(void)
{
   return epsilon;
}

double
FedeasMaterial::getStress(void)
{
   return sigma;
}

double
FedeasMaterial::getTangent(void)
{
   return tangent;
}
```

The next three methods deal with the path dependent behavior of this FedeasMaterial object. The *commitState()* method copies the trial history variables from the second half of the hstv array to the first half, where the committed values are stored. The committed values are copied to the trial values in *revertToLastCommit()*, and all values are set to 0.0 in the *revertToStart()* method.

```
int
FedeasMaterial::commitState(void)
{
   // Set committed values equal to corresponding trial values
   for (int i = 0; i < numHstv; i++)
      hstv[i] = hstv[i+numHstv];

   epsilonP = epsilon;
   sigmaP = sigma;

   return 0;
}

int
FedeasMaterial::revertToLastCommit(void)
{
   // Set trial values equal to corresponding committed values
   for (int i = 0; i < numHstv; i++)
      hstv[i+numHstv] = hstv[i];

   epsilon = epsilonP;
   sigma = sigmaP;
   return 0;
}

int
FedeasMaterial::revertToStart(void)
{
   // Set all trial and committed values to zero
   for (int i = 0; i < 2*numHstv; i++)
      hstv[i] = 0.0;

   epsilonP = 0.0;
   sigmaP = 0.0;

   return 0;
}
```

A copy of this FedeasMaterial object is returned by *getCopy()*. First, the FedeasMaterial constructor is called with the necessary tag, type, and array size data. Then, the committed strain and stress, all history variables, and material parameters are copied to the new object before it is returned.

```
UniaxialMaterial*
FedeasMaterial::getCopy(void)
{
   FedeasMaterial *theCopy =
      new FedeasMaterial(this->getTag(), this->getClassTag(), numHstv, numData);

   // Copy history variables
   int i;
   for (i = 0; i < 2*numHstv; i++)
      theCopy->hstv[i] = hstv[i];

   for (i = 0; i < numData; i++)
      theCopy->data[i] = data[i];

   theCopy->epsilonP = epsilonP;
   theCopy->sigmaP = sigmaP;

   return theCopy;
}
```

The next two methods are defined for parallel processing and database programming. The first method, *sendSelf()*, packs the tag and array size information for this FedeasMaterial object into an ID vector and sends it across the Channel. Then, the material properties and committed history variables are put in a Vector, and sent as well. The second method, *recvSelf()*, receives both the ID and Vector data from the Channel object, then populates the data of this FedeasMaterial object with the appropriate data.

```
int
FedeasMaterial::sendSelf(int commitTag, Channel &theChannel)
{
   int res = 0;

   static ID idData(3);

   idData(0) = this->getTag();
   idData(1) = numHstv;
   idData(2) = numData;

   res += theChannel.sendID(this->getDbTag(), commitTag, idData);
   if (res < 0)
      cerr << "FedeasMaterial::sendSelf() - failed to send ID data\n";

   Vector vecData(numHstv+numData+2);

   int i, j;
   // Copy only the committed history variables into vector
   for (i = 0; i < numHstv; i++)
      vecData(i) = hstv[i];

   // Copy material properties into vector
   for (i = 0, j = numHstv; i < numData; i++, j++)
      vecData(j) = data[i];

   vecData(j++) = epsilonP;
   vecData(j++) = sigmaP;

   res += theChannel.sendVector(this->getDbTag(), commitTag, vecData);
   if (res < 0)
      cerr << "FedeasMaterial::sendSelf() - failed to send Vector data\n";

   return res;
}

int
FedeasMaterial::recvSelf(int commitTag, Channel &theChannel,
                         FEM_ObjectBroker &theBroker)
{
   int res = 0;

   static ID idData(3);

   res += theChannel.recvID(this->getDbTag(), commitTag, idData);
   if (res < 0) {
      cerr << "FedeasMaterial::recvSelf() - failed to receive ID data\n";
      return res;
   }

   this->setTag(idData(0));
   numHstv = idData(1);
   numData = idData(2);

   Vector vecData(numHstv+numData+2);

   res += theChannel.recvVector(this->getDbTag(), commitTag, vecData);
   if (res < 0) {
      cerr << "FedeasMaterial::recvSelf() - failed to receive Vector data\n";
      return res;
   }

   int i, j;
   // Copy committed history variables from vector
   for (i = 0; i < numHstv; i++)
      hstv[i] = vecData(i);

   // Copy material properties from vector
   for (i = 0, j = numHstv; i < numData; i++, j++)
      data[i] = vecData(j);

   epsilonP = vecData(j++);
   sigmaP   = vecData(j++);

   return res;
}
```

The *Print()* method outputs the name of this FedeasMaterial object to the stream passed as an argument. More cases can be added to the switch statement as additional subroutines are added.

```
void
FedeasMaterial::Print(ostream &s, int flag)
{
   s << "FedeasMaterial, type: ";

   switch (this->getClassTag()) {
   case MAT_TAG_FedeasHardening:
      s << "Hardening" << endl;
      break;

   // Add more cases as needed
   default:
      s << "Material identifier = " << this->getClassTag() << endl;
      break;
   }
}
```

In order to link the FORTRAN subroutine with the OpenSees C++ libraries, the following external function declarations are needed. There are two syntactic styles for these declarations, one for Win32 and the other for everything else. The preprocessor directives put the proper declaration into the source code. Additional declarations may be added as more subroutines are included in OpenSees.

```
#ifdef _WIN32

extern "C" int _stdcall HARD_1(double *matpar, double *hstvP, double *hstv,
                               double *strainP, double *stressP, double *dStrain,
                               double *tangent, double *stress, int *ist);

#define hard_1_ HARD_1

// Add more declarations as needed

#else

extern "C" int hard_1_(double *matpar, double *hstvP, double *hstv,
                       double *strainP, double *stressP, double *dStrain,
                       double *tangent, double *stress, int *ist);

// Add more declarations as needed

#endif
```

The method *invokeSubroutine()* calls the appropriate subroutine based on the material classTag. The FedeasMaterial instance variables are passed to the FORTRAN subroutine from this method. Additional cases in the switch statement can be added as more subroutines are linked with OpenSees.

```
int
FedeasMaterial::invokeSubroutine(int ist)
{
   // Compute strain increment
   double dEpsilon = epsilon-epsilonP;

   switch (this->getClassTag()) {
   case MAT_TAG_FedeasHardening:
      hard_1_(data, hstv, &hstv[numHstv], &epsilonP, &sigmaP, &dEpsilon,
              &sigma, &tangent, &ist);
      break;

   // Add more cases as needed

   default:
      g3ErrorHandler->fatal("%s -- unknown material type",
                            "FedeasMaterial::invokeSubroutine");
      return -1;
   }

   return 0;
}
```

5.3 Example – FedeasHardeningMaterial

The material data array defined in FedeasMaterial is populated by its subclasses. As an example, consider the case where the uniaxial hardening material is coded in a FORTRAN subroutine. In order to link this subroutine with OpenSees, a subclass of FedeasMaterial, FedeasHardeningMaterial, must be created (see figure 4). The functionality of this subclass is to populate the material parameter array and to determine the number of history variables required for analysis.

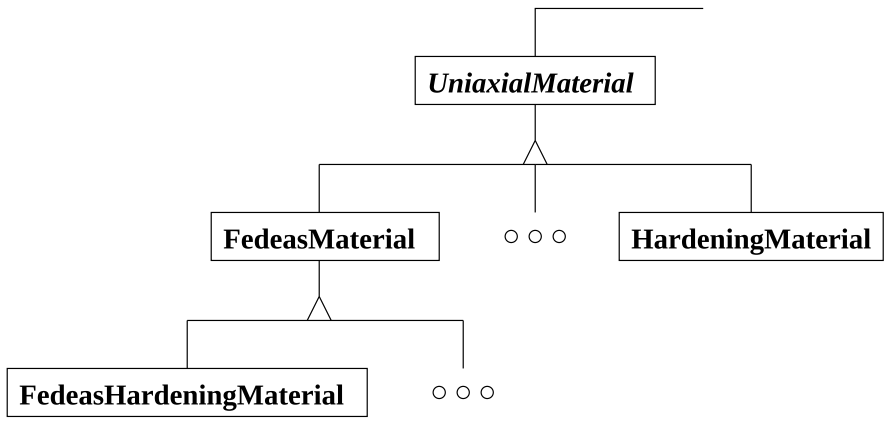

Figure 4: FedeasHardeningMaterial class hierarchy

As additional FEDEAS subroutines are added to OpenSees, new subclasses of FedeasMaterial must be added in order to populate the data array. This is all that need be done in the derived class as the base class, FedeasMaterial, contains all the computational code and keeps track of path dependent behavior. This functionality is inherited from the FedeasMaterial base class. However, the FedeasMaterial class must be modified such that the appropriate subroutine is called during state determination from the method *invokeSubroutine()*.

5.3.1 FEDEAS Hardening Subroutine

This section contains the implementation of the uniaxial hardening material coded as a FORTRAN subroutine using the FEDEAS interface. The subroutine declares local variables to store the material parameters passed through the matpar array. The committed history variables are received from hstvP, and the trial history variables are written to hstv upon return. The trial stress and tangent are also set upon return in the variables sig and tang.

```
      subroutine Hard_1(matpar,hstvP,hstv,epsP,sigP,deps,sig,tang,ist)
c I  matpar contains fixed properties (4)
c     E    = Elastic modulus             --> matpar(1)
c     sigY = Yield stress                --> matpar(2)
c     Hiso = Isotropic hardening modulus --> matpar(3)
c     Hkin = Kinematic hardening modulus --> matpar(4)
c
c I  hstvP contains committed history variables:
c     ep    = hstvP(1) --> plastic strain
c     alpha = hstvP(2) --> internal hardening variable
c     kappa = hstvP(3) --> back stress for kinematic hardening
c
c O  hstv will be set to the corresponding trial values of hstvP
c     hstv(1) = ep
c     hstv(2) = alpha
c     hstv(3) = kappa
c
c I  epsP: strain at last committed state
c I  sigP: stress at last committed state
c I  deps: current strain increment
c O  sig : updated stress
c O  tang: updated tangent
c I  ist : tangent calculation switch
c             1 = tangent, 2 = incremental secant, 3 = total secant

      implicit none

c     Arguments
      integer ist
      real*8  matpar(4),hstvP(3),hstv(3)
      real*8  epsP,sigP,deps
      real*8  sig,tang

c     Local variables
      real*8  E,sigY,Hiso,Hkin
      real*8  ep,alpha,kappa
      real*8  eps,f,xsi,dGamma
      integer sgn
```

```
c     Material parameters
      E    = matpar(1)
      sigY = matpar(2)
      Hiso = matpar(3)
      Hkin = matpar(4)

c     History variables
      ep    = hstvP(1)
      alpha = hstvP(2)
      kappa = hstvP(3)

c     Current strain
      eps = epsP + deps

c     Elastic predictor
      sig = E * (eps - ep)

c     Stress relative to back stress
      xsi = sig - kappa

c     Yield function
      f = dabs(xsi) - (sigY + Hiso*alpha)

c     Inside yield surface
      if (f <= 0.0) then
         tang = E
c     Outside yield surface ... do return mapping
      else
c     Consistency parameter
         dGamma = f / (E+Hiso+Hkin)

c     Normal to yield surface
         if (xsi <= 0.d0) then
            sgn = -1
         else
            sgn = 1
         endif

c     Updated stress
         sig = sig - dGamma*E*sgn

c     Updated plastic strain
         ep = ep + dGamma*sgn

c     Updated back stress
         kappa = kappa + dGamma*Hkin*sgn
```

```
c     Updated internal hardening variable
         alpha = alpha + dGamma

c     Elasto-plastic tangent
         tang = E*(Hkin+Hiso) / (E+Hkin+Hiso)
      endif

c     Update history variables
      hstv(1) = ep
      hstv(2) = alpha
      hstv(3) = kappa

c     Compute requested tangent
      if (ist==2.and.deps/=0.d0) then
       tang = (sig-sigP)/deps
      else if (ist==3.and.eps/=0.d0)then
       tang = sig/eps
      else
c     add additional cases, if needed
      endif

      return

      end subroutine
```

5.3.2 FedeasHardeningMaterial Subclass

As stated previously, the functionality of the FedeasHardeningMaterial class is to read in the material parameters required for the Hard_1 subroutine invoked from the FedeasMaterial method *invokeSubroutine()*. In addition, the required number of history variables must be passed to the FedeasMaterial base class.

The class interface for FedeasHardeningMaterial is shown below. The constructor takes the tag and material parameters as arguments. No other methods are declared in the FedeasHardeningMaterial interface as all functionality is inherited from FedeasMaterial.

```
#include <FedeasMaterial.h>

class FedeasHardeningMaterial : public FedeasMaterial
{
   public:
      FedeasHardeningMaterial(int tag, double E, double sigmaY,
                              double Hiso, double Hkin);
      FedeasHardeningMaterial(void);
      ~FedeasHardeningMaterial();

   protected:
```

```
   private:
};
```

The constructor takes the tag, elastic modulus, yield stress, and isotropic and kinematic hardening moduli as arguments. The FedeasMaterial class constructor is called with the tag and classTag MAT_TAG_FedeasHardening defined in classTags.h and the number of history variables and material parameters required for this particular material model. Then the material parameters are inserted into the data array. The default constructor simply invokes the base class constructor, and the destructor does nothing.

```
FedeasHardeningMaterial::FedeasHardeningMaterial(int tag,
                            double E, double sigmaY, double Hiso, double Hkin):
// 3 history variables and 4 material parameters
FedeasMaterial(tag, MAT_TAG_FedeasHardening, 3, 4)
{
   data[0] = E;
   data[1] = sigmaY;
   data[2] = Hiso;
   data[3] = Hkin;
}

FedeasHardeningMaterial::FedeasHardeningMaterial(void):
FedeasMaterial(0, MAT_TAG_FedeasHardening, 3, 4)
{
   // Does nothing
}

FedeasHardeningMaterial::~FedeasHardeningMaterial(void)
{
   // Does nothing
}
```

5.3.3 Important Polymorphic Note

Should any method in the FedeasMaterial class need to be overridden, e.g., if a subclass does not want all of its history variables set to 0.0 in *revertToStart()*, the *getCopy()* method must also be overridden to return a pointer to the subclass. If *getCopy()* is not overridden, the dynamic type of the returned pointer will be of the FedeasMaterial type and the overridden method, e.g., *revertToStart()*, will not be called.

5.4 Tcl Model Builder

Adding the FedeasHardeningMaterial model to the Tcl model builder is done in exactly the same manner as for HardeningMaterial since both models have the same material parameters. Only the material allocation would change. The following line in TclModelBuilderUniaxial-MaterialCommand.cpp

```
      // Parsing was successful, allocate the material
      theMaterial = new HardeningMaterial(tag, E, sigmaY, Hiso, Hkin);
```

would be changed to

```
      // Parsing was successful, allocate the material
      theMaterial = new FedeasHardeningMaterial(tag, E, sigmaY, Hiso, Hkin);
```

5.5 FEM_ObjectBroker

As for the HardeningMaterial class, an additional case needs to be added to the *getNewUniaxialMaterial()* method in FEM_ObjectBroker in order for the FedeasHardeningMaterial class to be used for parallel processing and database programming.

```
UniaxialMaterial*
FEM_ObjectBroker::getNewUniaxialMaterial(int classTag)
{
   switch(classTag) {

   case MAT_TAG_Hardening:
      return new HardeningMaterial();

   case MAT_TAG_FedeasHardening:
      return new FedeasHardeningMaterial();

   //
   // Additional cases not shown
   //

   default:
      cerr << "FEM_ObjectBroker::getPtrNewUniaxialMaterial - ";
      cerr << " - no UniaxialMaterial type exists for class tag ";
      cerr << classTag << endl;
      return 0;
   }
}
```