An adaptable finite element modelling kernel

M.D. Rucki *, G.R. Miller1

University of Washington, Box 352700, Seattle, WA 98195, U.S.A.

Received 9 January 1997; accepted 22 April 1998

Abstract

This paper describes a finite element kernel designed to provide high levels of adaptability and flexibility for modelling and analysis. The kernel is based on high-level abstractions that can be used to create a variety of interactive finite element-based modelling applications. The structure and functionality of the kernel are embedded in frameworks and classes of objects which encapsulate modelling and computational capabilities in modular form. Sample uses of the kernel are illustrated via prototype modelling applications, including a direct manipulation modelling environment, which exemplifies the type of application for which the modelling capabilities of the kernel are well suited.

1998 Elsevier Science Ltd. All rights reserved.

1. Introduction

This paper describes the structure and use of a general finite element modelling kernel designed to provide new levels of flexibility and adaptability for both program creation and use, with a particular focus on supporting rich user interaction. A direct manipulation modelling environment exemplifies the type of user interaction that the kernel has been designed to support. The kernel has been designed using object-oriented programming principles and supports a wide range of modelling capabilities which are convenient to use, modify and extend. The kernel incorporates many noteworthy features including:

1. Model definition and updating are expressed in terms of high-level abstractions—this facilitates the creation of rich user interfaces, and provides high levels of code portability, extendability, reusability and maintainability.
2. Coordinate-free data types are used consistently throughout the implementation—several benefits are realized, including: no explicit local–global coordinate transformations are necessary, calculations are independent of the embedding space, the underlying geometry is clearly represented in the code, computational simplifications can appear quite naturally, and run-time performance of algorithms can be improved.
3. Physical modelling classes are endowed with state, and algorithmic details are locally encapsulated—the modelling components have ongoing states that are updated based on their interaction with other objects. Numerical processing is seamlessly integrated into the modelling environment.
4. Algorithms are interchangeable and their initialization is automatic—partitions of modelling components and their corresponding updating algorithms can be arbitrarily associated in both time and space, providing opportunities for hybrid solution strategies and new modes of runtime interaction.
5. Algorithms can be embedded recursively—partitions of modelling components can behave like elements. These partitions can contain other partitions as elements so that algorithms can implicitly incorporate the algorithmic functionality of other algorithms.
6. The kernel is efficient—despite the flexibility and generality of the kernel, in terms of raw numerical
processing speed, it compares very well with traditional techniques.

Certain of these features have already been presented in previous papers: specifically Ref. [8] presented aspects of the use of coordinate-free geometry data types, and Ref. [5] presented details of the algorithmic features. The contribution of this paper is to describe, in terms of high-level abstractions, the kernel created from the integration of such features, and to demonstrate how the flexibility and adaptability of the kernel can produce benefits for developers and end users.

The modelling kernel has been designed to be general, but this does not mean that it is intended to serve solely as a monolithic, general problem solver. Rather, one aspect of the focus on flexibility has been to make it possible to customize and adapt the kernel to a wide range of focused applications and to support highly specialized interfaces, since in many circumstances these are the most useful to engineers. The ability to support convenient development of multiple special-purpose interfaces is a key feature of the kernel. As is illustrated later in the paper, another important aspect of the kernel’s flexibility is in the nature of the functionality itself one can achieve with these interfaces.

This work can be broadly classified with related efforts in object-oriented approaches to finite element analysis; see, for example, Refs [1]–[4] for descriptions of overall modelling systems, and Refs [10]–[12] for descriptions of work focusing on specific issues such as run-time performance and user interface design.

2. The structure of the kernel

The modelling kernel has been cast in terms of object-oriented frameworks [6], in which the system can be viewed as structured collections of classes. The classes in these frameworks are based on well defined abstractions and they collaborate in order to perform computational tasks. In particular, the kernel decomposes into two subframeworks: (1) a physical modelling framework representing the physical model and its components; and (2) an algorithmic framework that is responsible for the computations associated with solution algorithms. The physical modelling framework consists primarily of high level interfaces to the various physical modelling components, which define the creation and manipulation of modelling components such as elements, nodes, constraints, and so on. The algorithmic framework includes a wide range of algorithmic methods and techniques structured for convenient use in a wide range of modelling situations. There are algorithm classes in element and degree of freedom-based form of linear and nonlinear problems in quasi-static and dynamic contexts.

The subframeworks are described in more detail below following an overview of some fundamental utility classes.

2.1. Utility classes

2.1.1. Coordinate-free data types

From the discretization of the domain to the element formulation, and throughout the solution for the unknowns, the finite element method is intimately tied to the underlying differential geometry of the problem considered. While mathematical derivations are commonly expressed in coordinate-free notation, traditional implementations are based on processing matrices of explicit scalar components. There are advantages associated with removing the derivation/implementation dichotomy in this regard, and so the implementation presented here has been based consistently on a set of classes for performing calculations in coordinate-free form. The main classes model geometric points, vectors, and second-order tensors and define functions for their interaction. These classes have been defined so that geometric computations can be performed directly in terms of coordinate-free data types in the spirit of the work presented in Ref. [7]. A full consideration of the use of coordinate-free abstractions is beyond the scope of this paper (a presentation of coordinate-free element formulations can be found in Ref. [8], and a forthcoming paper will focus on coordinate-free implementations), but some of the more significant effects with respect to the current algorithmic framework can be summarized as follows:

1. significant efficiencies have resulted—computations often can be more efficient in coordinate-free form [5]. Also, simplifications appear in coordinate-free element formulations that are not readily apparent in coordinate-based ones;
2. explicit local-to-global coordinate transformations are not needed;
3. the dimension of the embedding space need not be specified—the same code is valid in 2-D and 3-D contexts;
4. the underlying geometry is expressed explicitly in the code.

While the development presented here has been based consistently on coordinate-free computations, this is not absolutely necessary to achieve the main features of modelling flexibility exhibited by the kernel. One could use traditional coordinate-based calculations as well.

2.1.2. Manager classes

Manager classes are used to facilitate the creation, deletion, storage and manipulation of collections of
objects. The manager classes effectively simplify coding by encapsulating the details of managing collections of modelling components. A base manager class provides simple access to generic collections of objects. Object-specific manager subclasses handle the details for creating specific types of objects and manipulating an entire collection. As a particular example, the call to AlgorithmMgr::UpdateAlgos() in Code Listing 1.B in Section 3.1 encapsulates all the details of proper ordering of calls to each Algorithm’s UpdateState method, which in general depends on the Algorithm object’s type as well as whether it is associated with the super or sub-Partition.

2.2. Physical modelling classes

A schematic overview of the classes defining the physical modelling framework and their relationships is presented in Fig. 1. In the following presentation, the set of physical modelling classes is decomposed into three sets: (1) the set of Elements, Nodes, and degrees of freedom (DOF’s) representing the discretized physical domain and its state; (2) the set of ConstraintSets and Loads representing the generalized boundary conditions; and (3) the Partition as the union of two of these sets.

2.2.1. Decomposition of the physical domain

A discretized domain is represented by Node and Element objects. A Node contains DOF objects, representing the degrees of freedom of the model, and is associated with a geometric location. A DOF object encapsulates all fundamental abstractions typically associated with degrees of freedom: displacement, external and internal forces, mass, etc.—all in coordinate-free form (i.e. vector quantities are represented using vector classes). The DOF’s play a key role in the modelling kernel: essentially they act as intermediaries between the physical modelling classes, described in this subsection, and the numerical representations of a finite element model, as represented by the algorithmic classes, described in Section 2.3.

Elements represent the underlying mechanical properties of the problem being modelled and are responsible for defining the relationships between degrees of freedom. The way in which an Element actually enforces degree of freedom connectivity in the code depends on the solution method considered: element-based updating methods require elements to update generalized restoring forces based on current displacements, while degree of freedom-based methods require Elements to install tangent stiffnesses at their associated degrees of freedom. Like many of the other objects in the overall framework, the details of the Elements’ behavior are encapsulated, allowing for simple interchange. While the specific details of Element updating are beyond the present scope (more details are given in Ref. [8] for example) it is worth mentioning that in the present framework the Element base class defines the generic functionality for accommodating both kinematic and material nonlinear calculation of forces and stiffnesses. The actual method
used by an element at any point in time depends on an internal flag that can be switched during an analysis by the engineer. In this way, one specialized version of an element, for instance a linear kinematic one, need not be destroyed and another created in order to change the details of force calculations. Such details need not be publicly visible in any case.

While nodes, degrees of freedom, and elements are abstractions that can be drawn easily from traditional finite element modelling, there is no similarly obvious set of abstractions for encapsulating material properties and behavior. In traditional implementations, materials typically are represented by simple sets of scalar parameters associated directly with particular elements. Such representations can be much richer when object-based abstractions are considered. In the present framework, material behavior is decomposed into distinct classes representing constitutive behavior and classes encapsulating material parameters. The MaterialState represents the strain and stress states at a point inside an element. Referring to Fig. 1, the MaterialModel is a simple container for the material and ConstitutiveModel classes. The idea is that the Material classes encapsulates a set of parameters used to describe a particular material, and the ConstitutiveModel encapsulates the algorithmic details of updating material properties and transforming stress states into corresponding strain states. In this way, MaterialState instances are associated with individual Elements, while ConstitutiveModels and Materials can be shared among general MaterialModels. This material framework provides a plug-and-play style of material switching allowing flexibility with respect to material implementation and use.

### 2.2.2. Boundary conditions

The second set of classes in the physical modelling framework represents boundary conditions. In the present implementation, boundary conditions are represented by the classes Load and ConstraintSet.

The Load class represents external forces in coordinate-free form applied to degrees of freedom—either interactively, or according to a prescribed, time-dependent pattern. Current classes in the hierarchy can be used to apply generalized point loads to Nodes. Classes modelling distributed loads and general internal element loads must cooperate with Element classes to assign loads to DOFs appropriately.

To explain the classes implementing generalized constraints, it is helpful to examine the following mathematical representation of a constraint:

\[ u_i = f(u_j) + c \]  \hspace{1cm} (1)

where \( u_i \) represents the constrained degree of freedom's value, \( \{ u_j \} \) is a collection of degree of freedom values, \( f \) represents some general imposed relationships between the degrees of freedom, and \( c \) is a prescribed value.

The classes for enforcing generalized constraints are indicated in Fig. 1; namely Constraint, DOFConnector, DOFFilter, and ConstraintSet. Each Constraint object is responsible for enforcing a generalized constraint condition of a particular DOF via encapsulated DOFConnector and DOFFilter objects, which, in turn, enforce the actions represented by \( f \) and \( c \) respectively in Eq. (1). A ConstraintSet is essentially a container for multiple Constraint objects to implement specific types of constraint conditions such as rollers. This modular approach accommodates a high degree of flexibility in implementing particular types of generalized constraint conditions.

#### 2.2.3. Partitions

The third set of classes in the physical modelling framework is the Partition. As indicated in Fig. 1, a Partition contains each type of physical component manager: ElementManager, NodeManager, LoadManager and ConstraintSetManager. During an analysis a Partition plays a relatively passive role, essentially acting as a manager of physical modelling component managers.

A Partition can represent a complete model or alternatively, a model partition to support partitioned solution schemes. Essentially, such partitioned solution schemes are divide and conquer strategies whereby the original model, or alternatively its corresponding numerical representation, is partitioned into subsets which can be treated individually to varying degrees.

In this modelling kernel, different algorithmic approaches can be associated with each Partition and an overall analysis is based on treating each Partition conceptually as a generalized super-element. Thus, relative to other Partitions, Partitions can exhibit element-like behavior. This is reflected in the fact that Elements and Partitions both inherit from the abstract class GeneralElement. The accommodation of recursive algorithms is based on embedding Partitions within Partitions.

#### 2.3. Algorithmic framework

A complete description of the algorithmic framework has been presented in Ref. [5], including detailed consideration of its efficiency. This section provides an abbreviated summary of the framework’s structure and functionality.

The algorithmic framework has been designed to support efficient and convenient development, implementation, selection, assignment, and blending of algorithms. To this end, the construction of the framework is based on the idea that the job of an algorithm is to update states. The goal in updating states is to
satisfy generalized equilibrium in an approximate, but suitably accurate manner. In an abstract sense, an algorithm can be viewed as an operator, \( A_i \), that updates the state of an associated system or subsystem, \( S_j \), from state \( n \) to \( n+1 \):

\[
A_i : S_j^n \rightarrow S_j^{n+1}.
\]

What appears in this equation as the operator \( A_i \) becomes an active agent in the framework, embodied in an Algorithm class.

General algorithmic flexibility is provided by offering a number of classes of algorithms that can be easily accessed, used, extended, and adapted. All Algorithm classes support a shared external interface that is compact and simple. Algorithms can be selected and changed dynamically, and different Algorithms can be used in different parts of a model. As mentioned earlier, degree of freedom and element-based, iterative and direct Algorithms are supported for solving linear and non-linear problems in quasi-static and dynamic contexts.

While the underlying computations performed by the algorithmic framework can be based on standard techniques, the integration of the framework into the overall modelling kernel is markedly different from what one finds in a traditional implementation, in which the role of an algorithm is simply to solve a temporarily existing set of equations which have been extracted from the model. The fundamental role of the Algorithm class is to orchestrate the incremental updating of the state of a system or subsystem, without necessarily constructing a separate numerical representation. Effectively, the state of a model is updated via the direct collaboration of the various modelling components. This requires an embedding of functionality within the model itself, and it is this “in-place” solution approach that gives the framework its unique flavor.

The algorithmic framework is based on encapsulating a variety of computational functionality into a basic Algorithm class and associated helper classes as shown in Fig. 2. The classes shared with the physical modelling framework, Partition and DOF, are italicized to distinguish them from the classes unique to the algorithmic framework. Each of the classes in Fig. 2 are summarized below:

1. Algorithms provide a uniform mechanism for updating states while encapsulating various solution strategies;
2. Partitions act in the solution process as liaisons between their associated Algorithms and the corresponding physical modelling objects;
3. DOF objects contain the fundamental data and methods associated with the abstraction of a generalized vector degree of freedom;
4. AlgorithmAgents provide DOF’s with the specific algorithmic capabilities necessary to support local embedding of computational functionality;
5. AlgorithmAgentManagers handle the details of structuring and controlling an Algorithm’s AlgorithmAgents;
6. BoundaryFilters act as agents between connected Partitions, enforcing their connection by communicating stiffness, displacement, and force information as required;
7. AlgorithmManager provides high-level control over model updating, and performs general management of contained Algorithms;
8. Clocks are provided so that Algorithms may keep track of local and global time as necessary.

The generic and interchangeable nature of these classes allows for flexible selection, modification, and combination of algorithms. This plug-and-play aspect can be useful in many ways, including run-time algorithm swapping in response to behavioral cues, spatially heterogeneous distribution of algorithms, and recursive algorithm embedding. An example of the latter two cases is illustrated in Fig. 3, in which a structure has been divided into three partitions, each of which has its own associated internal algorithm, while an additional super-algorithm combines the partition behaviors into a global response. Numerical examples demonstrating the utility of such combinations from a computational point of view are provided in Ref. [5]—here the emphasis is on the fact that the capability itself exists.

3. Kernel use

To illustrate the use of the kernel, two example applications are presented in this section. The first application is very simple and is intended to show the style of code necessary to define, manipulate, and update a model. The second application is an example of a direct-manipulation analysis program, demonstrating...
some of the more interesting types of capabilities that can be realized conveniently using the kernel.

3.1. Coding with the Kernel

The objective here is to demonstrate the actual use of the various computational classes defining the modelling kernel. A simple program demonstrates how the various high-level modelling classes can be used to easily create, modify, associate, and direct Partitions and Algorithms.

An example program is presented in Listing 1. The code's syntax is that of C++ [9], and some declarations have been elided for brevity. The code has been broken into four segments identified by the symbols to the right of the listing: O, A, B, & C. The code in segment O defines the highest level objects, PartitionManager and AlgorithmManager, as well as objects that are meant to be shared between Partitions such as material models and element cross-sections. The modelling events in Segments A, B, and C are pictorially represented in Fig. 4—segment A creates three Partitions; one represents a two-story frame, one models a region of soil, and one is a super-Partition containing the other two as GeneralElements. Segment B creates Algorithms, associates each with a Partition and updates the state of the model. Segment C modifies the structure and switches algorithms to demonstrate automatic Algorithm reinitialization after Partition modification and after Algorithm switching. Other modelling capabilities, such as manipulation of boundary conditions are similarly straightforward to use but are not included here for brevity.

As demonstrated in the sample code, the collection of high-level abstractions defined by the framework essentially defines a structural programming language. The commands to the various modelling objects clearly convey their intent. As a result, the code is relatively easy to understand, and many of the comments are somewhat redundant. Further, creating user interfaces in terms of such high-level abstractions is reasonably straightforward: computational details are well encapsulated, and creating the interface involves designing a layer around the existing interfaces of the different classes.

3.2. Direct manipulation modelling with the kernel

This section presents a prototype direct manipulation modelling environment, an exemplar of the type of application for which the modelling capabilities of the kernel are well suited. Such an environment allows

---

**Fig. 3.** An example illustrating spatially heterogeneous algorithm distribution and recursive algorithm embedding.

**Fig. 4.** Modelling events included in Sample Code Listing 1.
void main()
{
    PartitionMgr* thePartitionMgr = new PartitionMgr;
    AlgorithmMgr* theAlgOMgr = new AlgorithmMgr;

    //--- Set the global Clock---
    gClock ->SetIncrement (dt);

    //--- Make a Material and MaterialModel---
    IsotropicMaterial theMaterial (rho, E, nu, sigY, E2);
    SimpleIsomatModel theMatModel (theMaterial);

    //--- Make Element CrossSections---
    GeneralCrossSection theGenCS (scalarDim);
    BeamCrossSection theBeamCS (csarea, Ixx, Iyy, polarI, inPlaneE1);

    //--- Make Partitions ---
    Partition* theFramePartition = thePartitionMgr->MakePartition ();
    Partition* theSoilPartition = thePartitionMgr->MakePartition ();
    Partition* theSuperPartition = thePartitionMgr->MakePartition ();

    //--- Make Nodes ---
    GPoint theNodeLoc (xi, xj, xk),...
    Node* node1 = theFramePartition ->MakeNode (theNodeLoc);
    ... define rest of Frame's Nodes
    theNodeLoc.SetComponents(xi, xj, xk),...
    Node* node5 = theSoilPartition ->MakeNode (theNodeLoc);
    ... define rest of Soil's Nodes
    theNodeLoc.SetComponents(xi, xj, xk),...
    Node* node27 = theSuperPartition ->MakeNode (theNodeLoc);
    ... define rest of SuperPartition's Nodes

    //--- Identify shared nodes for sub-Partitions ---
    theFramePartition ->ConsiderAsBoundary (node27);
    theSoilPartition ->ConsiderAsBoundary (node27);

    //--- Make BeamElements ---
    theFramePartition ->MakeElement (eBeam, node1, node2, mat1, mat2, theBeamCS);
    ... define rest of Frame's Elements

    //--- Make Plane Elements ---
    theSoilPartition ->MakeElement (ePlaneTri, node1, node2, node3, mat1, theGenCS);
    ... define rest of Soil's Elements

    //--- Sub-Partitions are considered as GeneralElements by super-Partition ---
    theSuperPartition ->AddGeneralElement (theFramePartition);
    theSuperPartition ->AddGeneralElement (theSoilPartition);

    //--- Make a Load ---
    GVector theLoadVect (vi, vj, vk);
    Load* theLoad = theFramePartition ->MakeLoad (eForce, node1, theLoadVect);

    //--- Make pinned constraints ---
    theSoilPartition ->MakeConstraintSet (ePin, node5);
    ... define rest of Soil's ConstraintSets
an exploratory, hands-on approach to engineering analysis and design by allowing a user to interact with a model as if it were on his or her desk. A high level of flexibility in such an environment supports a style of analysis that is qualitatively different from the traditional—engineers interactively modify and analyse models in quasi-realtime and select, adapt, and combine algorithms based on factors such as the required

Fig. 5. A prototype direct manipulation environment.
levels of accuracy and efficiency, variations in the model in space and over time, and changing modelling assumptions.

The prototype environment is briefly introduced here to indicate its general mode of operation—details of its implementation are not of particular importance in the present discussion. Fig. 5 shows the basic interface components that compose the prototype environment. The main window shows the current state of the model being considered. The global origin and the three basis vectors are shown below and to the left of the model. The text pane at the bottom of the main window displays messages to the user. The floating window on the right side of the figure is the solution control window; it provides access to the global clock, and the solution is controlled via the icons on the left which represent the red, yellow, and green lights of a traffic signal. The floating window on the left contains a tool palette (the upper grid containing the various tool icons), a view control pane (in the middle with only horizontal and vertical bisecting lines) for controlling the view of the model, and a manipulation pane (at the bottom with a boxed X in the center) for orienting the 2-dimensional plane by which the user interacts with the 3-dimensional model. Also, various functionalities are provided via the menus shown at the top of the figure.

Figs. 6 and 7 show screen shots from the prototype implementation demonstrating the type of modelling situations that the kernel/environment combination is well suited to. In particular, the examples demonstrate the simultaneous analysis of independent partitions, and the transparent re-initialization of the Algorithms which is checked independently by each Algorithm with modelling components retaining their current state.

Fig. 6 demonstrates the use of the kernel’s modelling capabilities for behavioral exploration via comparative modelling. Five similar 2-story structures are shown subjected to the same load applied at the top left node of each. Each structure is a Partition that is associated with a distinct Algorithm object. As the figure indicates, the Partitions differ slightly from each other; various boundary conditions are considered as well as the effects of trussed bracing, and framing elements (beams and trusses).

The next example, shown in Fig. 7, demonstrates how model evolution can be handled using the modelling kernel. The figure shows sequential screen shots of the partition as it is evolving. The particular details of the model are not important for this discussion. The important points with respect to the use of the modelling kernel are (1) that algorithm re-initialization is transparent to the user, and (2) that modelling component state data, in particular that encapsulated by the degree of freedom and material state objects, are locally encapsulated and are independent of the applied Algorithm. Each Algorithm is responsible for checking whether or not it needs to be reinitialized before each update. In general, the extent of component and algorithm re-initialization depends on the change to the model and the type of algorithm considered.

![Fig. 6. Behavioral exploration using multiple algorithms.](image-url)
What cannot be made clear in any of these static figures is the dynamic nature of the interface. As a consequence of the integration of the modelling and analysis, results are displayed directly in response to user actions. Depending on the size and complexity of the model, one can achieve quasi-realtime, animated simulation and visualization. For linear problems, one currently can deal comfortably with thousands of degrees of freedom on contemporary PC-class hardware in this manner. This means many practical problems already can be modelled in this way, and improved processing power will make it possible to handle many more.

4. Summary and concluding remarks

This article has presented the structure and use of a finite element modelling kernel designed to support new levels of modelling adaptability and flexibility. The structure of the kernel includes coupled physical modelling and algorithmic frameworks, which provides for integrated modelling and analysis. By replacing the typical file-based solver metaphor with that of an environment of interacting objects endowed with state, it was found that support for interactive, direct-manipulation analysis can be achieved relatively easily. By replacing global data structures and solution schemes with localized versions based on algorithm agents, it was found that mixing and modifying algorithms in space and time can be accommodated conveniently and such capabilities can be useful in many modelling situations. The use of high-level abstractions makes it straightforward to use the kernel in applications, and the example applications presented here demonstrate the type of non-traditional, interactive style of modelling to which the kernel is particularly well suited. Already, it is possible to analyse practically-sized problems in a very different manner than has been the case to date, and the domain of problems for which direct manipulation/visualization is practical continues to grow. The techniques described in this paper provide a suitable foundation upon which this new style of analysis tool can be built.

Acknowledgements

This material is based upon work supported by the National Science Foundation under grant no. CMS-9203140 while the first author was a PhD candidate at the University of Washington.
References


