FOOBAR: An Artificial Intelligence Based Finite Element System

by

J. W. Baugh, Jr. and D. R. Rehak

EDRC-12-11-87

FOOBAR: An Artificial Intelligence Based Finite Element System

John W. Baugh Jr. and Daniel R. Rehak

Department of Civil Engineering, Carnegie Mellon University,
Pittsburgh, PA 15213-3890, USA

Abstract

This paper presents altematives to the traditional programming methodologies found in current finite element analysis systems. Artificial intelligence based representations are discussed, as are their incorporation in FOOBAR, a finite element system under development by the authors. Particularly useful architectures include constraint satisfaction paradigms, object-oriented techniques, and declarative knowledge representation. The development of FOOBAR is based on these ideas, which functionally decompose "what" to compute from "how" to compute it. We expect the resulting system to provide an intelligent, flexible environment for exploring research issues in finite element analysis.

FOOBAR: An Artificial Intelligence Based Finite Element System

John W. Baugh Jr. and Daniel R. Rehak

**Department of Civil Engineering, Carnegie Mellon University,
Pittsburgh, PA 15213-3890, USA**

1. Introduction

Finite element analysis systems are difficult to write, maintain and modify. Conventional systems have control strategies, along with data structures, computational algorithms, etc., that are dependent upon each other. The application of existing models of computation from the artificial intelligence community, however, can greatly simplify program development, ease the burden of maintenance, and result in a more robust system. For example, in knowledge-based systems, knowledge is represented in a manner that is independent from control. FOOBAR is an example of a new class of finite element system that is based on a declarative representation of knowledge and an object-oriented style of programming.

The finite element method is described in FOOBAR using various types of knowledge, including *process knowledge, constraints,* and *meta-knowledge,* which are implemented in a declarative form. Knowledge about the overall solution process, including the fundamental relationships of mechanics, alternative solution techniques, etc., is used to guide numerical computations. Thus, the actual sequence of processing for a given problem is not known a priori since it is not explicitly coded in the system. Knowledge about constraints insures that certain relationships exist in the data, for instance, that quantities to be added have the same units. Meta-knowledge about equation solvers and other algorithms is used to select the most appropriate solution techniques and data representations for a given problem.

Control is based on process knowledge, and is therefore independent of the routines that actually perform numerical computation. Thus, the explicit representation of knowledge and constraints in FOOBAR provides independence between levels of abstraction. As a result, the overall solution process can be changed through modification of knowledge without affecting the underlying procedures which perform the computations. In the same manner, the computational procedures may be modified without affecting the solution process.

Routines that perform numerical computation in FOOBAR are implemented in an object-oriented fashion so that data, and the procedures that operate on it, are encapsulated. Thus, information hiding is guaranteed, and illegal operations prevented. The object-oriented methods used for numerical computations are based on a lattice of objects that define a set of abstract data types. FOOBAR has at the highest level a complete set of basic engineering quantities (force, length, etc.), which are based

on primitive data types (integers and reals). Subsequent levels define objects from mechanics (stress, strain, etc.), mathematics (matrix, tensor, etc.), and so on. Operations performed on the objects via sending messages are therefore inherently consistent. For instance, a message sent to add a stress to a length would be refused.

A prototype of FOOBAR is being developed by the authors as a test bed for many of these ideas [Rehak 86]. This paper presents our current thinking on some of the approaches that seem promising and does not reflect an actual implementation. Ideas at the conceptual level are addressed, and pieces of low-level details are presented along with various examples. The following section identifies some of the organizational methodologies that we believe are useful - in particular, object-oriented techniques, knowledge-based representations, and constraint networks. Following that is a description of how these ideas may be incorporated in FOOBAR.

2. Organizational Methodologies

Many of the goals in the development of FOOBAR are the same that occur throughout computer science, particularly in artificial intelligence and logic programming. That is, it is desired to separate "what" is to be computed from "how" to compute it. (These same concepts are also addressed in database management systems, where the "conceptual" level is separated from the "internal" level, so that performance and other machine-dependent issues are isolated from the rest of the system.) This section addresses some of the high level control issues and alternatives that arise in implementing a finite element system with independent representations of knowledge, control, and processing.

2.1. Objects and Message Sending

Although the operator/operand model found in conventional languages treats operators and operands as independent, operators are limited in the types of operands they can accommodate. This is particularly true of scientific and engineering computations. In the message/object approach [Stefik 86], however, objects record their class (type) explicitly, and methods are invoked based on the classes of operands they receive. An example is given by Cox [Cox 84] of electrical systems, in which incompatibility between plugs and sockets make it safe (you can't do the wrong thing). The object-based approach is similar: type dependencies cannot spread through the system since they are permanently encapsulated within classes. *Encapsulation* is provided by dynamic binding (at runtime) which results in greater flexibility. Conventional languages, however, use static binding. The user achieves "dynamic binding" using conditionals, which must be changed with any change or addition of data types. In object-oriented languages, the multiplicative operator, for instance, can have the same power that it does in mathematical notations. That is, it is capable of operating on scalars, such as integers, reals, or complex numbers, as well as collections of data, such as vectors and matrices. This ability of objects from different classes to respond appropriately to the same message is referred to as *polymorphism.*

In addition, most object-oriented languages allow classes of data to be organized into hierarchies and lattices. This in itself is a useful mechanism for classifying and organizing data in large software systems. Because of this arrangement, subclasses may *inherit* methods defined on super classes. Therefore, redundancy is reduced since more general operations are automatically invoked when a method is not defined on a particular class.

2.2. Handling Constraints

Much of the pioneering work in constraint-based models of computation was performed by Abelson and Sussman [Abelson 85] and their students at MIT [Steele 80], as well as by Alan Boming [Borning 79] at Stanford. Some of the simple models of local constraint satisfaction turn out to be particularly suitable for representing the finite element method. A special class of constraints, dataflow, serves as a useful introduction to constraints in general.

2.2.1. **Dataflow**

The use of dataflow compilers on previously sequential Fortran programs is a common approach for achieving parallelism in scientific computations. The compiler looks for data dependencies in the code, and imposes the constraint that no operation be attempted before its ingredient data items are known. The concept of *explicitly defining* the pre- and post-conditions to a task within the code itself is attractive: it allows a modularity not found in the conventional imperative programming paradigm where procedures and functions are the basic unit of program decomposition. Dataflow constraints may be stated in a declarative manner. That is, the required input and subsequent output of a task (or procedure) can be defined without actually invoking it. A task scheduler can then locate and execute tasks whose input is available. Examples of such declarations in the context of a rule-based system might consist of the following facts:

```
(task name solver
      inputs force stiffness
      output displacement)

(task name stiffness-assembly
      inputs stiffness-i stiffness-i+1 ...
      output stiffness)

(task name element-stiffness-generation
      inputs element-type nodal-coordinates elastic-modulus
      output stiffness-i)
```

as well as a set of rules that schedule tasks, activating those whose preconditions have been satisfied. In this way, the control is separated from the actual computations. In addition, parallelism comes at no extra charge since multiple tasks may simply be activated when more than one has its preconditions satisfied. For example, in the case of element stiffness generation, the inputs for many elements may be available. Therefore, their stiffnesses can be generated simultaneously. But more important than the parallelism (at least in terms of development and maintenance issues) is the modularity of the the system. By using dataflow techniques, the *dependency graph* is explicitly represented as facts which may be conveniently modified as needed.

An alternative to a global task scheduler is a message sending scheme in which the ingredient data items signal the appropriate tasks when its value is known. For instance, if *force* obtains a value (as in the previous declarations), it sends task *solver* (and any other tasks which rely on the value of *force)* a message indicating it has been computed. When a given task has received such indications from all its constituent data items, the task activates itself, producing its output, and then sends subsequent tasks a message that its output has been computed, and so on until all possible computations have been made.

To represent the dependency graph using classical message sending semantics, we define two classes, one for data types and one for the tasks themselves. The *data* class is the root for all data represented in the dependency graph and is defined as follows in the proposed Common Lisp Object System [Bobrow 87, Bobrow 85]:

```
(dafdass data ()
    ((•aim  :initform nil)
     (tasks  :ia±tfoxm nil)
     (informant  :initform nil)))
```

where the *value* slot contains the actual value of a piece of data; the *tasks* slot is a list of the tasks that use the data; and the *informant* slot keeps track of how the value was obtained (i.e., who set it). If the value is already set and the new value is different, setp complains.

Then, a method called *setp* is defined for assigning the value of a variable and propagating it to the appropriate tasks:

```
(dafmathod satp ((d data) naw-valua sattar)
    (with-slots (d)
        (oond ((null informant)
                  (••tq valua naw-valua)
                  (•atq informant sattar)
                  (for-aach-axcapt Mttar 'notify tasks))
              ((/• valua naw-valua)
                  (error "Contradiction: ~s *•"
                      valua naw-valua)))))
```

Notice that *setp* first insures that the value is not already set (informant would be non-null if it was). Then it sets the value and the informant based on the supplied input, notifying tasks that use the data.

The general class for *task data* is simply defined as:

```
(dafclass task () ())
```

Specific tasks (e.g., a solver), are created by defining subclasses of *task:*

```
(dafclass solvar (task)
    (forca
     stiffnass
     displacamant))
```

and a method that performs the desired task:

```
(dafmathod notify ((• solvar))
    (with-slots (a)
        (if (and (known-p forca)  (known-p stiffnass))
            (satp displacamant
                  (/ (gat-slot forca 'valua)
                     (gat-slot stiffnass 'valua))
                  •))))
```

When both the force and stiffness have been defined, the displacement is computed and propagated to tasks using it, such as ones for computing stresses and strains. In this way, data are propagated throughout the dependency graph until all satisfiable tasks have been performed.

2.2.2. Constraint Satisfaction

A problem with dataflow, however, is that computations are directional. Consider again the example of the force-displacement relationship using dataflow. The code for actually computing the displacements may be defined as follows in Common Lisp [Steele 84]:

```
(dafun solvar (forca stiffnass)
    (/ forca stiffnass))
```

Yet, the meaning of the basic relationship between forces and displacements is obscured. It is desirable to define the relationship without specifying any order of computation, as in:

force = stiffness x displacement

where the "=" represents equality and not assignment. Thus, tasks with specified input and output may be replaced with *constraints* that define relationships between data, establishing the "what" without specifying the "how". For instance, if the quantities here are matrix quantities, the relationship is still valid, and the relationship says nothing about how to compute the quantities. If the stiffness and displacement matrices are known, then the force may be determined by matrix multiplication. Or if the stiffness and force are known, then the displacement may be determined by Gaussian elimination, or perhaps by some other solution technique. The point is simply that a relationship has been established between the data, and that there may be more than one way to satisfy the constraint.

This simple statement of relationships constitutes the declarative programming paradigm we consider useful. Thus, the competence ("what to compute") and performance ("how" to compute it) are separated [Steele 80]. *Competence* deals with factual information (relationships) and is responsible for correctness. *Performance,* on the other hand, deals with strategy (manipulation) and is responsible for efficiency. Ideally, a program with little or no information about performance should run correctly, though inefficiently. Predicate calculus is like this. It inherently makes no commitment to a computational technique.

It is possible to represent general constraints (e.g., that a linear spring obeys Hooke's law) in the same message sending style as dataflow constraints. That is, constraints may be represented as objects consisting of rules, where the rules consist of a procedural test plus a set of methods that can be invoked to satisfy the constraint when it is violated. Applying any of the methods causes the constraint to be satisfied. For example, an abstract class for constraints may be defined as:

```
(d«fcltft constraint () ())
```

Then, to specify a multiplicative relationship between a, *b,* and c, such that *a x b = c,a multiplier can* be defined as:

```
(def class Multiplier (constraint) (a b c))
```

with the method *notify* to compute the third value when any two are known:

```
(defmethod notify ((a multiplier))
    (with-slots (a)
        (oond ((or (and (known-p a) (zerop a))
                   (and (known-p b) (zerop b)))
               (Mtp o 0 a))
              ((and (known-p a) (known-p b))
               (Mtp o (* a b) a))
              ((and (known-p a) (known-p o) (invertible-p a))
               (Mtp b (/ o a) a))
              ((and (known-p b) (known-p c) (invertible-p b))
               (setp a (/ o b) a)))))
```

Constraints such as *adders* and *multipliers* are referred to as *primitive constraints* and may be used to specify entire networks of algebraic constraints which compute in any direction. Now, the force-displacement relationship described earlier may be defined as:

```
(Mtf stiffness (make-instance data))
(setf displacement (make-instance data))
(setf force (make-instance data))
```

```
(make-instance multiplier
    :a stiffness
    :b displacement
    :o foroe)
```

It should be noted that the above quantities may be either matrix or scalar classes, since the "*" operator is actually a set of methods defined on each *data* class.

The use of constraints as described is commonly referred to as constraint satisfaction or propagation of known states by degrees-of-freedom; constraints are satisfied by finding a degree-of-freedom (i.e., a relationship) that can be satisfied locally, and then propagating its new value through the network, looking for other locally satisfied degrees-of-freedom. However, this technique doesni always work, (e.g., when there are cycles in the graph, as in simultaneous equations) in which case other approaches, such as relaxation, multiple views, symbolic propagation, etc., must be used [Borning 79].

Although the dataflow and general constraint models are similar in many ways, dataflow constraints are directional, having specified input data and output data. Constraints in general, however, are adirectional. Thus, dataflow is a special case of constraints [Steele 80]. It turns out that the constraint model, like dataflow, is also a good for multiprocessor computation since each constraint only computes when all the relevant information is available. This is because constraints perform locally defined tasks on locally available information.

3. Description of FOOBAR

FOOBAR is an experimental object-oriented finite element system that combines the local constraint satisfaction paradigm previously discussed with knowledge-based reasoning. All system components are represented as objects, and are organized as in Figure 3-1, where *constraint* is a library of constraint types, such as multipliers and adders; *data* is a lattice of abstract data types used in the system, such as forces and displacements; and structure is a hierarchy of structural characteristics, such as static, non-linear, etc. Each of the classes has particular methods that may be used to manipulate class instances as needed. FOOBAR uses principles of constraint satisfaction to enforce fundamental relationships between objects. In addition, knowledge modules (deductive retrieval systems) are used to select the appropriate solver, method of data representation, etc.

3.1. Encoding Fundamental Relationships

The basis of the finite element method is a set of fundamental relationships from laws of engineering mechanics. The approach used in FOOBAR is to encode these relationships in the most general way possible. A constraint network is used to enforce fundamental relationships between data. The result is a high level mathematical description of the finite element method. To solve practical problems, the description is specialized based on the problem at hand (this is one way to use meta-knowledge). No heuristics are needed for this since the most general relationships are always valid. However, if some assumptions can be made about the system, then specialization is used to simplify the relationships for efficient computations. For example, particular methods are invoked based on having a linear material and plane strain.

The advantages of explicitly encoding the fundamental relationships include:

- *Correctness.* It may be easier to demonstrate and verify program correctness.

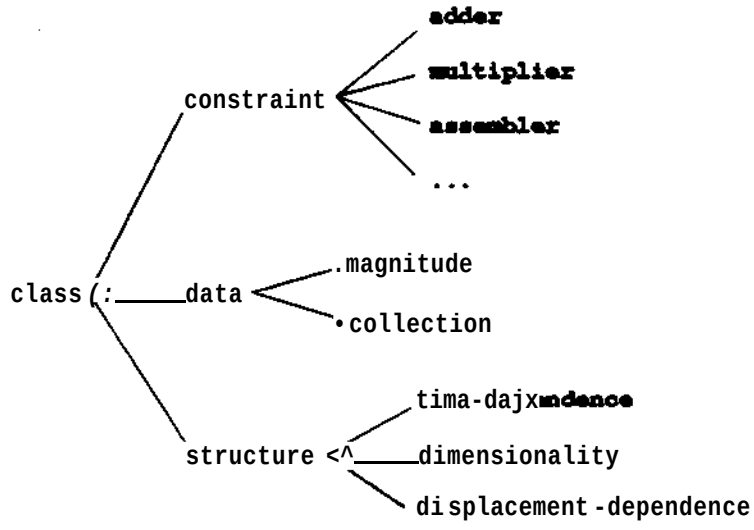

Figure 3-1: Organization of Classes in FOOBAR

- *Modularity.* Constraints can be specified as independent sources of knowledge - it's up to the constraint satisfaction system to satisfy them. (This is the same idea found in knowledge-based systems, namely that the inference engine operate on independent sources of knowledge.) Thus, the finite element researcher can add new capabilities to the system via data types and methods to solve a specific type of problem, or improve the performance of an existing capability by specializing methods - either of these can be done without changing other parts of the system. Thus, new methods can be incrementally added without invalidating existing ones.

- *Explanation facilities.* Constraint satisfaction systems have the ability to record dependencies of how information was propagated within the network. This can reveal how certain data were obtained. For instance, when a new piece of data has been computed, such as nodal displacements, a user might be interested in the basis of its computation, and ask "how was $\{d\}_f$ computed?" To receive meaningful replies such as: "from the given force-displacement relationship for linear static problems," or, perhaps, "from you, it was input as a displacement constraint" may be helpful. Also, the structure of the constraint network can reveal what potential histories would be even if the known states cannot be fully propagated.

The following fundamental relationships from mechanics are defined and enforced in FOOBAR:

- Force-displacement relations
- Stress-strain relations
- Strain-displacement relations

in addition to relationships useful in the generation of element stiffness matrices:

- One degree-of-freedom elements:
  - Direct Equilibrium
- Two and three degree-of-freedom elements:

- Virtual Work
- Minimum Potential Energy
- Weighted Residuals

Elements and assembled structures, sub-structures, etc., are represented in FOOBAR as objects with their own little pieces of private state. Each of these structural objects records information about external forces acting on it, relevant stress-strain transformations, etc. Since elements and assembled structures share the same characteristics, they each belong to the same class, *structure.* The structure class is defined in FOOBAR as:

```
(defclass structure ()
    Internal-fore*
    •sternal-force
    displacement
    stiffness
    stress
    stress-strain
    strain
    strain-displacement
    nodal-mapping)
```

Its immediate subclasses define a set of abstract structural characteristics, which consist of time dependence, dimensionality, and displacement dependence. These in turn have specific subclasses, as in Figure 3-2, that are instantiated by user-defined structural elements to provide a complete description of their characteristics. Subclasses from each of the three abstract classes must be instantiated by any given element (e.g., "my-element" in Figure 3-2) so that the required methods are available. The resulting graph of structural characteristics for each element thus becomes a lattice, instead of a pure hierarchy.

As an example, suppose an element is defined to be of classes *linear, static,* and *2-D.* The methods used to compute its stress-strain transformation matrix are then general enough for any linear, 2-D problem. Whereas, if its dimensionality is defined to be of class *plane stress,* the methods are only general enough to handle plane stress problems. In this way, efficiency can be achieved when specifics of a problem are known. Yet, when specifics are uncertain, a more general class of structure may be chosen with the assurance that a correct and appropriate (albeit less efficient) method is used to compute its properties. With this organization, users can easily define their own types and specialize them as desired, or use any of the default methods provided. For example, one might define *my-plane-strain* to be a subclass of *plane-strain.* If no new methods are defined on the class, its instances will behave exactly as instances of its super class, *plane-strain.* On the other hand, new methods may be defined to compute the stress-strain relationship in an alternate way without modifying existing code.

When an element is created, the user has the option of selecting any of the fundamental relations previously described to impose on the element. Therefore, if the stress-strain relationship is not imposed, stresses cannot be computed for that particular element. When the structure is composed of other structures, the force-displacement matrix is assembled (rather than generated) from its constituent element stiffness matrices. If it comprises some larger structure, this too is specified so that the element is assembled into the appropriate structure. That is, a mapping is defined for element forces, stiffnesses, and displacements to and from their global counterparts. The mapping, or assembly process, is a function of the connectivity

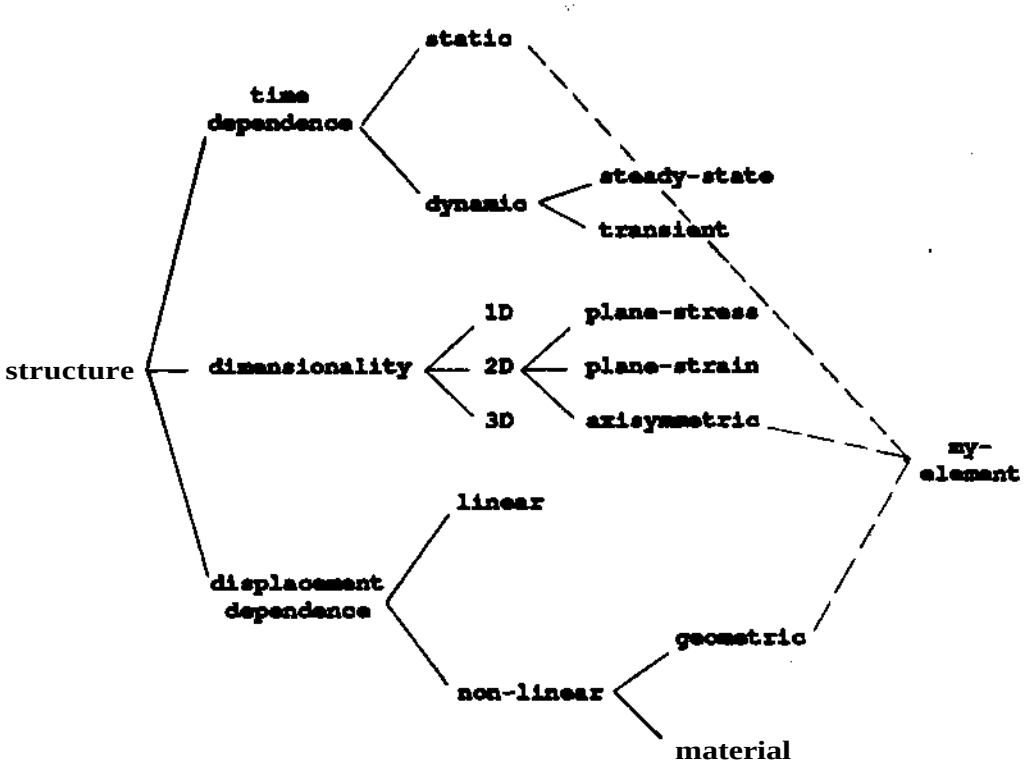

Figure 3-2: Structural Characteristics Defined in FOOBAR

information between local and global nodes in the structure. This information is provided by creating *assembler* constraints, which are defined as:

```
(de£dass assembler (constraint)
    (sub-structure
     super-structure))
```

Thus, the actual *structure* objects are assigned to the sub- and super-structure stots as in:

```
(setq element-1 (make-instance structure))
(setq element-2 (make-instance structure))
(setq structure-1 (make-instance structure))

(make-instance assembler
    :sub-structure element-1
    :super-structure structure-1)

(make-instance assembler
    :sub-structure element-2
    :super-structure structure-1)
```

Although the elements above are defined individually, it is possible, of course, to collectively define them as members of sets, arrays, or other structures.

Multipliers, assemblers, etc., may be used to specify a constraint network that describes the finite element method at a conceptual level, independent of issues such as control, representation, and numerical algorithms. Other constraints must also be incorporated in this high level description. For instance, assembly may take place only when matrices are in the same coordinate system. As in other parts of the system, it is expected that these components can be incrementally added throughout program development.

3.2. Invoking Numerical Computations

At this point, specific data structures and algorithms must be defined before computations can proceed. For instance, satisfying the constraint relating force to displacement **when** given both stiffness and force requires that the multiplicative definition of *displacement = stiffness xforce* be inverted. Based on the mathematical class of the components, the inversion operation is determined either to be division in the case of scalars, or solution of simultaneous equations (instead of inversion) when the data are matrices. At some computational cost, FOOBAR incorporates a library of representations and algorithms without being committed to any one. The most appropriate method of computation (e.g., Gaussian elimination, Sidel iteration, direct computation for a **2x2,** etc.) is selected for a given situation. This information is provided by "knowledge modules" with domain specific knowledge about solution methods, internal representations, element stiffness generation, etc.

A simple deductive retrieval system determines the appropriate solution method and internal representations to use. It consists of rules such as:

```
(if (and (available-ooze ?a)
         (required-core ?r)
         (eval « ?a ?r)))
    (stiffness-representation banded)
    "Banded solvers execute with nothing else in core, unlike
     wavefront solvers, which simultaneously generate element
     stiffnesses.")

(if (stiffness-representation ?s)
    (solution-method ?s)
    "Use a solution method consistent with the internal
     representation of the stiffness matrix.")

(if (and (eval (singular-p stiffness))
         (known-displacements ?d)
         (known-forces ?£)
         (number-of-dofs ?n)
         (eval (» (+ ?d ?f) ?n))
    (recovery-method (partition stiffness)))
    "Try partitioning the stiffness matrix if it is singular
     and the required forces and displacements are known.")
```

Then, before assembling the global stiffness matrix, the knowledge-base is queried to determine its most appropriate representation:

```
(infer '(stiffness-representation ?x) solution-module)
```

Heuristics are also used to select methods to use in stiffness generation, selecting weights when using Galerkin's method, etc. Typical rules include:

```
(if (eval (one-dimensional-p ?•))
    (stiffness-generation-msthod ?s direct-equilibrium)
    "Use the direct-equilibrium method to generate stiffnesses
     for one-dimensional structures.")

(if (and (eval (not (one-dimensional-p ?•)))
         (eval (elastic-p ?•)))
    (stiffness-generation-method ?s minimum-potential-energy)
    "Use the principle of minimum potential energy to generate
     stiffnesses for elastic, 2-D or 3-D structures.")
```

By providing separate knowledge modules in key system components, FOOBAR can not only select among any of various data structures and algorithms, but also maintain flexibility for incorporating other alternatives in the future. New heuristics may

also be added along with computational methods for determining when and how they should be used.

4. Closure

The goal of FOOBAR is to incorporate advanced models of computation into a finite element analysis system, with the expectation that the end result will be a flexible, self-descriptive tool that is suitable for use in research environments. This paper has described several of the techniques that we are using to bring this about. As development of the system continues, we intend to maintain a methodology that separates logical levels of abstraction and functionality. Additional meta-knowledge, constraints, objects and methods, and any other needed features, may be added easily in an incremental manner as a result of this.

References

[Abelson 85] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs,* MIT Press, 1985.

[Bobrow 85] Daniel G. Bobrow, et al., "CommonLoops: Merging Common Lisp and Object-Oriented Programming," *Proceedings, Object-Oriented Programming Systems, Languages, and Applications,* Vol. 21, Special Issue of SIGPLAN Notices, pp. 17-29, 1985, [Also available as Xerox Palo Alto Research Center (PARC) Report ISL-85-8.].

[Bobrow 87] Daniel G. Bobrow, et. al., *Common Lisp Object System Specification,* Technical Report 87-002, Xerox Palo Alto Research Center (PARC), Palo Alto, CA, 1987.

[Boming 79] Alan H. Borning, *ThingLab - A Constraint-Oriented Simulation Laboratory,* Technical Report SSL-79-3, Xerox Palo Alto Research Center (PARC), Palo Alto, CA, July 1979, [Also available as Stanford University, Computer Science Department Report STAN-CS-79-746.].

[Cox 84] Brad J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," *IEEE Software,* Vol.1, No. 1, pp. 50-61, January 1984.

[Rehak86] Daniel R. Rehak, "Artificial Intelligence Based Techniques for Finite Element Program Development," *Proceedings, Symposium on Reliability of Methods for Engineering Analysis,* Swansea, U.K., pp. 515-532,1986.

[Steele 80] Guy L. Steele Jr., *The Definition and Implementation of a Computer Programming Language based on Constraints,* Technical Report MTA-595, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August 1980.

[Steele 84] Guy L. Steele Jr., *Common Lisp: The Language,* Digital Press, 1984.

[Stefik86] Mark Stefik and Daniel G. Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine,* Vol.6, No.4, pp. 40-62, Winter 1986.