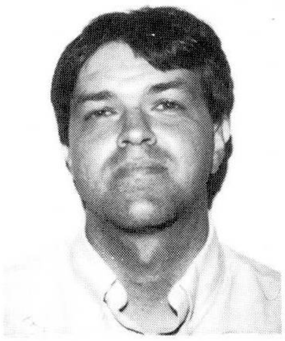
Alternative Programming Techniques for Finite Element Program Development

Autres techniques de programmation pour le développement de programmes d'éléments finis

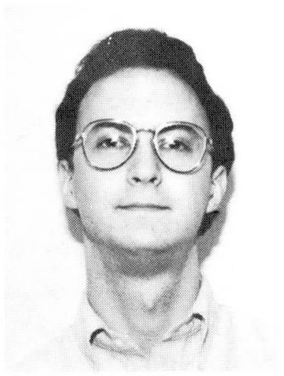
Alternative Programmiertechniken für die Entwicklung von Finite Elemente Programmen

Daniel R. REHAK
Assoc. Professor
Carnegie Mellon University
Pittsburgh, PA, USA

John W. BAUGH Jr.
Graduate Res. Assist.
Carnegie Mellon University
Pittsburgh, PA, USA



Daniel R. Rehak, born in 1951, received Bachelor's and Master's degrees in Civil Engineering from Carnegie Mellon University, and a Ph.D. from the University of Illinois. His research interests center on the applications of emerging computer technologies to large-scale engineering software development, especially in the civil engineering domain.



John W. Baugh Jr., born in 1960, received the Bachelor's degree from Auburn University, and the Master's degree from Carnegie Mellon University, both in Civil Engineering. He worked as a Research Engineer in structural mechanics at Battelle, Pacific Northwest Laboratory before returning to pursue the Ph.D.

SUMMARY
Finite element program development is hard; the translation of a page of matrix algebra, integrals and derivatives into code results in several tens of thousands of lines of non-trivial code. The difficulty arises because implementations specifiy *how* to solve the problem, rather than *what* the solution entails. Alternative programming approaches, based on formal specifications and data abstractions, let the programmer deal with a more declarative and abstract representation of the finite element solution process. These techniques are being used in an objet-oriented environment to provide a tool-kit for researchers implementing new finite element programs.

RESUME
Le développement d'un programme d'éléments finis est complexe; la traduction d'une page d'algèbre matricielle, d'intégrales et de dérivées en codes résulte en plusieurs dizaines de milliers de lignes de codes non-triviaux. La difficulté survient du fait que des réalisations spécifient *comment* résoudre le problème, plutôt que *qu-est-ce que* les solutions impliquent. D'autres approches de programmation, basées sur des spécifications formelles et sur des abstractions de données, laissent le programmeur travailler avec une représentation plus spécifique et abstraite de la méthode de résolution par éléments finis. Ces techniques sont couramment utilisées dans un environnement «orienté objet» constituant ainsi une «trousse à outil» pour les chercheurs mettant en oeuvre de nouveaux programmes d'éléments finis.

ZUSAMMENFASSUNG
Die Entwicklung von Finite Elemente Programmen ist schwierig. Die Übersetzung einer Seite von Matrix Algebra, Integralen und Ableitungen in Programm iersprache resultiert gewöhnlich in zehntausenden von nicht trivialen Zeilen. Die Schwierigkeiten entstehen, weil die Implementierungen spezifizieren *wie* das Problem zu lösen ist, und nicht *was* die Lösung zur Folge hat. Alternative Programmierwege, die auf formalen Spezifikationen und Datenabstraktionen basieren, erlauben es dem Programmierer mehr mit deklarativen und abstrakten Darstellungen der Finite Elemente Lösung zu arbeiten. Diese Techniken werden in einer objektorientierten Umgebung verwendet, um dem Forscher ein Werkzeug für die Entwicklung von neuen Finite Elemente Programmen zur Verfügung zu stellen.

1 Introduction

The finite element method is an extremely powerful and popular analysis tool which is used in a variety of engineering domains, including mechanical and civil structures, aerospace, electrical fields, nuclear, and shipbuilding. The basic description of the method, in terms of general formulations, element derivations, material models, solvers, etc., can be presented in terms of simple and elegant mathematics. These concise descriptions (often on the order of one page of mathematics) belie the computational complexity of the method. Programs which implement the method are complex, both in their structure and in their computational requirements. There is a dichotomy between the elegance of the basics of the method and the "dirty code" used to realize programs.

Development of finite element programs, simply put, is *hard*. Programmers have to deal with a variety of issues. First they must translate the mathematics of the method into numerical procedures and algorithms, coupled with appropriate data representations. In doing this, they must handle problem domain issues, e.g., selection of material models, shape approximations, order of integration, etc. The problems of the method and the domain are only a part of the complexity. Programs must execute on real machines (often with special architectures) with finite resource limits. Complex data structures and memory and storage management obscure other parts of program. Lastly, from the user's perspective, the goal is problem solving. Inputting a problem description and reviewing results are most important, and a major portion of any program must deal with user interfaces.

These characteristics result in large, complex programs, often measured in tens-of-thousands of lines of (convoluted, unmaintainable, nonportable) code. This situation is mitigated by the programming methodology used. Current programs are imperative—"word at a time". All details of the method, domain characteristics and resource management issues have to be stated in explicit detail (specifying how to move every word of problem data through the solution process, one simple operation at a time). This is quite different from the general, *abstract*, mathematical statements used to represent the method. The mathematics present *what* the finite element method is, while a program describes *how* to use the method in problem solving.

The central thesis of this work is that this distinction of *what* from *how* is the cause of much difficulty in implementing finite element programs, and that alternative (non-traditional) programming techniques can be used to lessen this distinction. In terms of finite element programs, the goals are:

- to make programs more *declarative*, such that they represent more of the abstract statement of the finite element method and less of the details of how to process the data;
- to lessen program development effort by letting programmers work at a higher, more abstract level, closer to the description of the method, again without dealing with low-level implementation details;
- to improve program reliability by placing responsibility for determining many of the details of the implementation with the program (or programming environment) instead of with the programmer; and secondarily
- to exploit parallelism (prevalent in emerging hardware systems) by uncoupling flow control from implementation.

The programming techniques used to reach these goals, along with an overview of the work to date, are presented in the sequel. The approach to developing abstract, declarative programs arises from, and is influenced by, the general methodology of knowledge-based systems and artificial intelligence. In both cases the goals are the same—to provide a declarative representation of the components of the problem-solving domain and to let the computer use those representations as needed. The actual techniques used in the work come from the domains of artificial intelligence, programming languages and software engineering.

2 Techniques for Finite Element Programming

There are a variety of alternative programming techniques available which can be of value in developing finite element programs. Before giving an overview of those being considered, it is necessary to outline the requirements placed on the technologies. As stated above, the overall objective is to develop a programming approach which concentrates on telling the computer what to do, and letting it decide how to do it. To meet this objective, the technologies used must fulfill three requirements. They must provide:

- representation—what type of knowledge of the finite element method can be expressed;
- expression—how the knowledge is expressed; and
- use—how the information is used to solve a problem.

A brief overview of each of the key techniques being explored follows, including a discussion of their role in finite element program development. While not explicitly described, the three requirements described above are considered in determining the applicability of the techniques.

2.1 Abstraction

Programming methodologies have evolved from simple (unstructured) programming through structured programming (which emphasizes the decomposition of programs into procedures and data structures) to data abstraction. Simply put, abstraction is the hiding of (appropriate) details. In terms of programming, abstraction consists of developing a set of (*abstract*) *data types* and the set of *all* operators associated with the data type.

Thus an abstraction consists of a data item or entity (akin to a data structure) and the procedures which operate on the data. The key is the inseparability of the entity from the operators. What is hidden is *all* internal details of how the data is actually represented and how the operators perform their tasks. The abstraction presents an external view of what it represents (i.e., the data type) and what it computes (i.e., the operators).

An abstraction is defined by a specification of what it represents and what it does. A program is built by using operators of the abstractions to perform tasks and manipulate the data entities represented by the abstract types. The actual implementation of the abstraction (representation and operators) is totally hidden, and can be changed without impacting other parts of the program as long as the implementation conforms to the specification.

As a simple example, consider an abstraction which represents a strain-displacement matrix for an element. Some operator might return the matrix. Internally the operator might compute the entire matrix each time it is needed, or compute and save it the first time it is requested and return the stored version when needed. Similarly, for a 2–D plane-stress problem, the abstraction might represent the matrix as a two-dimensional array, or as the two unique terms. What matters is that on demand, the abstraction will return the matrix: how it decides to perform the task or represent the data is immaterial to the user of the abstract type.

Abstraction techniques directly address the major stated objective of separation of *what* from *how*. The implementation provides, but hides, the details of how to do things, while the specification of the abstraction provides a (formal) definition of what the abstract type represents and what operators do. Abstractions are still lacking in that the specification is not (necessarily) executable, but must be transformed into running code. Thus a specification does not provide all of the expressional power needed to develop a fully declarative programming environment.

2.2 Dataflow Representations

In terms of this work, dataflow is a representational strategy (not to be confused with dataflow or reduction hardware architectures, although the associated problems are the same). Dataflow provides a graphical representation of problem data dependencies (i.e., what data items are produced and consumed by each computational step). Dataflow represents only the essential temporal constraints on data computation.

A dataflow graph does not represent a single sequence of *control flows* through a set of computations. Rather, some process must interpert the data dependencies in some fashion to drive computations. Thus a dataflow representation provides a declarative form for representing how to sequence the computations in a finite element program.

Dataflow representations can be processed either in a forward (data-driven) or a backward (demand-driven) manner. In forward or *dataflow* processing, computations proceed when data is available. Results propagate, (i.e., data *flows*) through the graph. As soon as an operator has all of its ingredient data, a computation is performed and the result is propagated to downstream operators.

In backward or demand processing, computations are invoked only on demand. When a data item is needed, the operator which produces it is invoked. If the ingredient data is available, the result is computed. If it is not available, a recursive process is used to compute that item. Demand-driven and data-driven dataflow are analogous to backward and forward chaining in rule-based systems.

Besides providing a declarative presentation of control, dataflow representations can be used to implicitly represent the inherit parallelism in finite element computations. Consider the control problem of stiffness matrix assembly and solution. A few of the alternative control strategies include:

- Sequentially form all element matrices, sequentially assemble the generated matrices, then solve the entire set of equations;
- Sequentially form and assemble matrices one element at a time, solve after all matrices have been assembled;
- Sequentially form and assemble matrices one element at a time, solve after any row is complete;
- Form element matrices in parallel, sequentially assemble the generated matrices, then solve the entire set of equations; or
- Form element matrices in parallel, synchronize, assemble the generated matrices in parallel, synchronize, then solve the entire set of equations (serially or in parallel);

There are other alternatives, including any of the above done on a nodal or degree of freedom (DOF) basis instead of an element basis.

A dataflow graph which represents the assembly and solution process for static analysis is shown in Figure 1 (individual nodes of this graph can be further decomposed). The processor which interprets this graph can implement any of the control strategies described above. Based on the data types used, the same graph represents the solution at various levels of data granularity (e.g., element, node, or dof). Thus dataflow representations provide a powerful declarative technique for uncoupling control flow from program structure and hardware architecture.

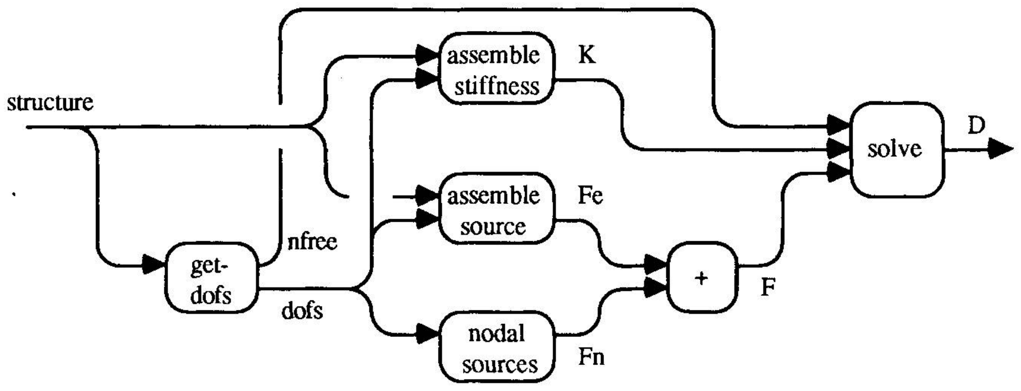


Figure 1: Static Analysis Dataflow Graph

2.3 Functional Programming

Functional programming is another paradigm which might be used to represent a finite element program. In functional programming, all expressions are of the form of a function applied to data items: functions are the primary entities in the language. The structure of a functional language is based on λ-calculus (*pure* Lisp is an example). The languages use abstraction to represent the data types, and binding in combination with function invocation to perform computations.

Functional programming provides a declarative, *applicative* form of representing problem solving. This form of a program excludes assignments, side effects, loops, branching, and representation of *state* (i.e., the program cannot rely on any knowledge of computational order). Used directly, functional programming can be used to provide another declarative representation of the finite element process.

In addition, the order of execution need not be that of simple sequential evaluation. Rather, the functional representation can be considered as one which encodes parallelism. A functional program can be compiled directly into a dataflow representation. Thus the functional form can be processed to yield either a demand-driven or data-driven problem-solving order.

2.4 Constraint Programming

Constraint programming provides yet another declarative representation strategy. Constraints can be used to represent desired relationships between data objects. As such, a constraint need not be satisfied, but if not satisfied, the fact that there is a violation is known. In addition, a constraint may, or may not, represent the needed information used to insure its satisfaction.

For example, a constraint might state that two elements must be in the same coordinate system for their stiffness matrices to be added. Such a constraint is sufficient to check that the condition exist before the elements are added, but does not provide any information on what to do if the condition is not true (e.g., apply a rotational transformation to one of the matrices).

The most desirable representation of constraints is a *nondirectional* one. For example, the relationship *area* = *width* x *height* is not an assignment statement to compute the area, but rather a relationship which can be used to compute either *area*, *width*, or *height* given the other two values.

Constraints can be used either locally or globally. In a local representation, individual constraints are used to verify pre-conditions (and post-conditions) of individual operations. In a global representation, a set of constraints can be used to represent all of the relationships which must hold in solving the problem. In this case, the entire problem can be solved by applying some constraint satisfaction procedure (e.g., *relaxation*), or the constraints can be used in a forward inference (e.g., greedy) strategy or a backward (e.g., lazy) search to drive computations as in dataflow representations.

2.5 Knowledge-Based Methods

The last declarative representational technology considered is the general area of knowledge-based methods. Of particular interest are knowledge representation strategies, such as rule-based programming. Rules provide an explicit declarative representation of problem-solving knowledge. Frame-based representations provide similar capabilities.

Different types of knowledge might be encoded in a knowledge representation. *Causal* knowledge represents the basics of the finite element method. Constraints (as described above) represent relationships and can be encoded in a knowledge representation. *Process* knowledge represents details of problem solving and constraint satisfaction (e.g., a rotation transform is used to align coordinate systems).

An important use of knowledge is as (rule-based) meta-knowledge used to control the details of the problem-solving process. For example, meta-knowledge might be used to select one type of algorithm or representation from those available (e.g., a non-sparse matrix representation is appropriate for a "small" problem). In addition to the rules used to make such selections, other knowledge must be represented, such as the characteristics of the available solution mechanisms or data representations, or the resource utilization characteristics of processes (used in making task-to-processor assignments in a multi-processor environment).

2.6 Object-Oriented Programming

As a technique for finite element program development, object-oriented programming is the implementation methodology. It provides the expression of the operational program. Object-oriented programming is a desirable implementation methodology due to four characteristics it provides: (1) data abstraction, (2) data type completeness, (3) inheritance, and (4) polymorphism.

One of the most fundamental characteristics of an object-oriented language is that the language forms and programming style provide and encourage abstraction. An *object* consists of a (hidden) local representation and an associated set of procedures which manipulate the object to produce result objects used by other operators. As such, an object is an instance of an abstract data type.

Data type completeness is an important characteristic which is missing from some "object-oriented" programming languages (e.g., C++). Data type completeness results in a situation where every entity manipulated by the program is treated equally, and can be used in any situation (e.g., can be assigned to, passed as a parameter, returned from a function, or used as components of other data structures). Object-oriented systems which provide this capability often do so by providing only a single underlying concept, the *object*. (Data types, operators, the compiler, data representations, etc., are all objects.) This approach is beneficial in that it provides a single, uniform underlying structure for all components of a system.

An inheritance mechanism is useful in that it simplifies programming. Rather than create a unique type of abstraction for each logical entity the program manipulates, sets of related concepts are created. These are organized hierarchically, with more general concepts (abstractions representing both storage and operators) at the *top* of the hierarchy. A generalized concept is specialized into more specific types by adding or overriding representations and operators. At the lowest level of the hierarchy are the most specific types of individual entities. An example of an inheritance tree (tangle) for some finite element concepts is shown in Figure 2.6.

An actual object may have parents from more than one hierarchy (*tangle* inheritance). This multi-level graphical structure of abstract types lets the programmer structure objects so that shared concepts are not repeated, and lets different concepts remain *orthogonal* rather than being combined into a single piece of code (a detailed example of orthogonality for matrix types and representations is presented below).

Polymorphism is present (in a limited form) in many languages. For example, the "+" operator in FORTRAN is polymorphic in that it can be applied to real, integer or complex variables. Polymorphism consists of using a single operator to represent an operation on a variety of data types (operator *overloading*) while providing the means to differentiate between operators with the same name that are applied to different data types (operator *dispatching*). Object-oriented languages provide the capabilities to define an operator that can be applied to a variety of data types (e.g., define "+" for reals, integers, time, matrices, linked-lists), and to automatically invoke (at run-time) the correct

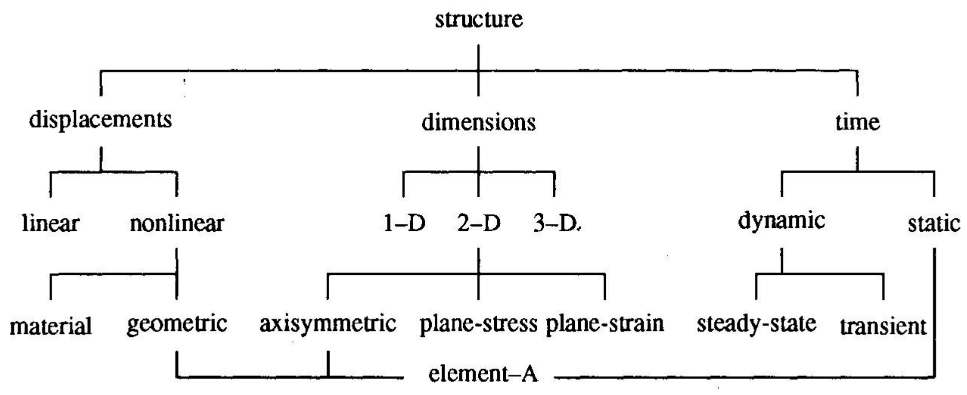


Figure 2: Example Inheritance Tangle

code based on the types of objects to which the operator is applied (i.e., the code used to add two time variables is different from the code used to add two matrices, but this is not apparent from the program).

Together these aspects of object-oriented languages are useful in simplifying programming. Creating abstract data types is natural; type completeness makes programs conceptually "cleaner"; inheritance simplifies organization and promotes reuse of code; and polymorphism can be used to implement constraints and to provide a single notation for programming mathematically similar operations on distinct types.

No other methodology provides all of these characteristics. Using alternative implementation strategies is feasible, but requires more complex programming (essentially their use would require building what are the inherit characteristics of an object-öriented language). The current work uses CLOS (Common Lisp Object System) as the actual implementation language.

3 Solution Approach

The techniques outlined above are being used, individually and together, in developing finite element programs[1,2]. At this time, work is centered on detailed explorations of the proper use of the techniques individually; the integration of the techniques into a unified framework for finite element programming is still under study.

The current target use of the programming environment is for FEM researchers. This community is more demanding in their need for a flexible environment which simplifies programming, and while resource utilization is important, it can be treated as a secondary issue.

The key components of the solution are the use of abstract types and an associated formal specification. Additionally, representation of control and knowledge are important components, as is an approach which is applicable to a parallel hardware base.

The conceptual model of how to integrate these components is centered on a constraint-based or data dependency-based task scheduler. A program is represented as a set of abstract data types. Associated is a representation of fundamental dependency relationships and constraints on the data items. A computational task is selected on the basis of user requests for results. Heuristics are used as needed to select from competing alternatives, and resource information is used to allocate the task to one of many alternative processors in a parallel implementation. As the computations proceed, declarative information from constraints and data dependencies are used to select additional tasks which must be completed to solve the problem.

In this mode of problem solving, the program is dynamically generating the problem-solving strategy. As such, components of the program can be changed or replaced without concern for the implications of the change. A single global program does not exist, but is generated from the available components as needed on the basis of those components available, the knowledge, and the problem characteristics.

The control strategy outlined above has not yet been implemented. In the current implementation, control remains procedural, with imbedded implicit knowledge of problem-solving sequences. While the implemented approach lacks the elegance one might like, it is still important in that it is derived from a declarative representation, based on abstract data types.

As noted above, most of the techniques described form the basis of how to express problem-solving knowledge, with object-oriented programming being the implementation strategy. What information is represented, both in terms of finite element domain information and problem-solving control information, must be "programmed" (represented) using these techniques to provide the complete programming environment. Formal specifications of the abstract data types are used to provide this information and knowledge independently from the types' programmatic implementation.

Numerous abstract data types are needed to define the finite element method. This work is aimed at providing a framework for program development, and is not attempting to define *all* possible abstract types. Some representative abstraction classes include:

- Engineering concepts—dimensions, units
- Mathematical concepts—vectors, matrices, linear algebra
- Structural concepts—load, displacement, stiffness
- Modeling concepts—nodes, DOF, elements, structures
- Representation concepts—sparseness
- Resource concepts—processors, memory, communications bandwidth
- Finite element concepts—specific element types

These items form a hierarchy of concepts. For example, a *load* can be represented as a vector of values with units of force (i.e., a vector of force objects). Similarly, a structure load matrix can be represented as a matrix of loads (each term is a vector instance of the load type) stored in some representation (full matrix or some sparse representation).

Some examples of these data types and their specifications, along with their use in problem solving, are described below.

3.1 Matrix Abstraction

As an example, consider an abstraction for a matrix data type (a more detailed presentation of this example is contained in [3]). The matrix abstraction must provide a representation and a set of operators. Since in terms of problem solving, matrices with special properties are common (e.g., symmetric, triangular), the abstract type is divided into classes for the various type of matrices.

Independent of the class, the same set of operators must be available for all types of matrices (*completeness* is essential so type changes have no other impact). The types of operators provided include: *constructors* (used to create and build matrix objects), *observers* and *mutators* (used to access and change elements of a matrix), coercion (used to change the class of a matrix), mathematics (addition, transposition, multiplication, etc.), control (looping over all elements and *mapping*), and utility operators (direct copy, I/O).

An example of the specification of one operator, solution of simultaneous equations, is:

```
solve = proc (A: matrix[float], C: vector[float], n: integer) returns (B: vector[float])
   requires nrows(A) = ncols(A) = length(C) and A is non-singular and n > 0.
   effects Returns B such that length(B) = min(n,nrows(A)) and A B = C for indices
         0 ≤ i < min(n,nrows(A)).
```

This specification only describes the behavior of the operator: that the equations must satisfy **A B** = **C**. The actual implementation can decide how to solve the equation (direct, relaxation), or in fact if the implementation is imperative instead of simply a constraint used by a constraint satisfaction procedure.

A number of classes of matrices can be envisioned (general, symmetric, upper or lower triangular, diagonal, square, singular, etc.); the classes are not mutually exclusive. What is important is that the operators can be consistently applied to all instances of a class. For some classes, this implies error detection (inversion should not be attempted on a matrix which is known to be singular). For others, special action by the operators are required (an accessor trying to return a value from the upper triangle of a lower triangular matrix should return a zero). Insuring proper behavior in all situations improves program reliability, and lets the programmer concentrate on the more important aspects of problem solving, not on the details of matrix representation and manipulation.

Independent of the class of matrix is the storage of the matrix elements. Typical examples of storage include full matrices, packed representations (a triangular matrix stored in a vector), linked representations, hypermatrices, banded and skyline. Any class can be stored in any representation. In selecting a representation, the programmer must consider several issues: access efficiency, storage efficiency, and flexibility of modifying terms. An example of this orthogonality of class and representation is shown in Figure 3.

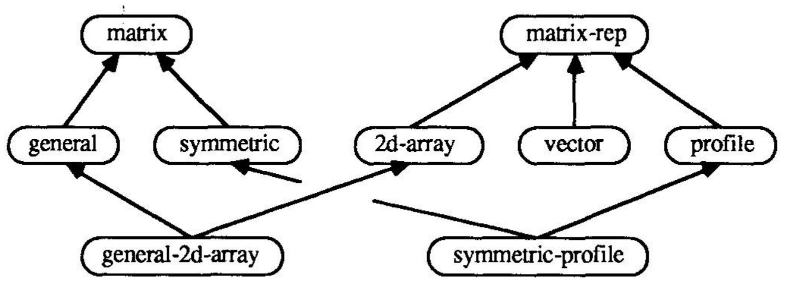


Figure 3: Orthogonality of Matrix Class and Representation

Clearly delineating the class from the representation lets any combination be used: the program (programmer) is free to choose what is best for the problem at hand. Consistency of the operators implies that these choices can be changed (even dynamically with coercion as needed), without any other changes to the program. Separating specification from implementation lets the programmer *tune* the performance of the program as needed, again without impacting other aspects.

3.2 Solution Abstraction

The representation of a structure or element is a graph, with the graph considered in its mathematical sense. Graph vertices are nodes, and links are elements. A full set of graph abstractions and topological operations are provided to manipulate the representation of structures and elements and to form the basis for problem solving. Given such a representation, it is possible to define a procedure for solving a static, linear-elastic problem.

The general solution procedure is to follow a number of steps. Note that this is imperative, procedural control, and is but one of the many alternative sequencing of the steps. The name of the operator used in each step is given in brackets.

- order the nodes of the graph (Reverse Cuthill-McKee) [rcm]
- determine the degrees of freedom of the structure from the set of nodes in the graph [build-dofs]
- partition the ordered nodes based on the boundary constraints [partition]
- generate nodal indices for the assembled system equations [set-indices]
- create the empty stiffness matrix [make-matrix]
- assemble the element matrices by processing all edges in the graph [transform / assemble-stiffness]
- generate the nodal load vector
- solve the resulting equations [solve]

The procedure described above is the process performed by the static-linear-elastic operator. As noted, the steps performed by the operator are implementation-specific. The translation of the procedure into code follows.

```
node-list := rcm(g)
ordered-dofs := build-dofs(node-list)
dofs, nfree := partition(ordered-dofs, source-prescribed?)
set-indices(dofs)
ndofs := vector-length(dofs)
k := make-matrix(list(ndofs, ndofs), symmetric-matrix-1)
for e in edges(g) do
  transform(e, orientation(global))
  assemble-stiffness(e, k)
f := nodal-sources(dofs)
for i := 0 below ndofs do
  f[i] := source(dof[i])
d := solve(k, f, nfree)
```

The specification of the routines used to implement this operator are shown below.

```
static-linear-elastic = **proc** (g: graph)
  requires ∀ (v : member(v,vertex-names(g)) : ∀ (dof : nodal-dof(get-vertex(v,s)) :
      (source-prescribed?(dof) or state-prescribed?(dof)) and ¬state-set?(dof))).
  modifies g.
  effects Normally computes and sets the state of: all elements in g, and the free dofs in g. Signals
      singular-matrix if the assembled stiffness of s is singular.

rcm = proc (g: graph) returns (list[node])
  modifies all nodes in g.
  effects The reverse Cuthill-Mckee algorithm. Orders nodes in g by breadth-first-search, reversing and
      returning the result. Uses mark, marked?, and clear on nodes (thus modifying them) to determine
      whether or not they have been visited.

build-dofs = proc (l: list[node]) returns (vector[dof])
  effects Returns a vector of all the dofs defined on the nodes in l.

partition = proc (v: vector[type], p: proc (t: type) returns (bool)) returns (v-new: vector[type], n: integer
or nil)
  modifies v.
  effects If ∀ (i : 0 ≤ i < length(v) : p(v[i])) then return v unchanged and nil, otherwise produce a
      stable rearrangement of v such that ∀ (i : 0 ≤ i < n : p(v-new[i])) and ∀ (i : n ≤ i < length(v) :
      ¬p(v-new[i])) such that n is the index of the first element not satisfying the predicate p.

set-indices = proc (v: vector[dof])
  requires ∀ (i : 0 ≤ i < length(v) : ¬index-set?(v[i])).
  modifies all dofs in v.
  effects Sets the index of each dof in v such that v[index(dof)] = dof.

nodal-sources = proc (v: vector[dof]) returns (s: vector[float])
  requires ∀ (i : 0 ≤ i < length(v) : source-prescribed?(v[i]) or state-prescribed?(v[i])) and v is partitioned
      by source-prescribed?
  effects A bijection using source on each of the elements of v. When ¬source-prescribed?(v[i]), sets
      s[i] = 0.0.
```

Each of the components of the specification must be implemented to provide the running program. For example, an implementation of the assemble-stiffness operator in CLOS is:

```
(defun assemble-stiffness (e k)
  (let* ((ke (stiffness e))
         (id (incident-dofs e))
         (n (array-dimension id 0)))
    (dotimes (i n)
      (dotimes (j (1+ i))
        (incf (symref k (index (svref id i)) (index (svref id j)))
              (aref ke i j))))))
```

4 Closure

The work described above is just a portion of that underway. To date, the issues of formal specification and data abstraction based on an object-oriented methodology have formed the kernel of work. The most concrete results have been the formal specification of the data abstractions and the implementation of these in the working system. The role of the specification must be emphasized. It is a clear statement of *what* behavior the program must exhibit. It is completely independent from the actual implementation of the program, and a variety of implementations (ranging from declarative to imperative) can be used to transform the specification into the *how* of problem solving.

To meet the goals of a completely declarative finite element system, a number of other topics must be explored in more detail. These include: (1) the issue of alternative control abstractions (other ways of "how to do it"); (2) parallelism and task scheduling for multi-processor hardware bases; (3) incorporation of resource management in a program while keeping resource issues orthogonal and uncoupled from other concepts; and (4) use of heuristics (e.g., which equation solver to use when) in control of problem solving. Beyond these tasks, the most challenging component of the remaining work is to combine all of these approaches into a powerful, yet "clean" environment (a limited number of clearly defined and distinct concepts) for finite element programming.

This work has demonstrated that the use of alternative programming methodologies can yield more abstract and declarative finite element programs[1,2]. Using the underlying technology and basic tools developed, the implementation and modification of a program require less effort than using conventional imperative programming methodologies.

As stated, the approach is motivated by and has its roots in knowledge-based systems and artificial intelligence, but it is not AI per se, in the classical sense of a problem solver which behaves as a human. When viewed at an abstract level, however, programs built using these techniques do exhibit *intelligent* behavior. At this abstract level, concepts are represented in a declarative form, and details are hidden. The underlying support mechanism processes these declarative forms to perform problem solving, just as classical inference strategies process knowledge.

In closing, it must be noted that the concepts in such an approach are not limited to finite element programming. Rather, the finite element method provides a rich domain to demonstrate a different approach to the development of numerical problem solvers.

Acknowledgements: This work was supported in part by a U.S. National Science Foundation Presidential Young Investigator Award, Grant number ENG-8451533.

5 References

[1] Baugh, J. W., Jr., *Computational Abstractions for Finite Element Programs*, unpublished Ph.D. Dissertation, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, PA, August 1989.

[2] Rehak, D. R., and Baugh, J. W., "Development of an Intelligent Finite Element System," *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 1989, in preparation.

[3] Baugh, J. W., Jr., and Rehak, D. R., "Implementation of a Finite Element Programming System — A Declarative Approach," *Computer Utilization in Structural Engineering*, ASCE Structures Congress '89, San Francisco, CA, ASCE, pp. 91-100, May 1989.