Aspects of an object-oriented finite element environment

Th. Zimmermann, P. Bomme, D. Eyheramendy, L. Vernier, S. Commend

Laboratory of Structural and Continuum Mechanics (LSC), Civil Engineering Department, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland

Received 14 November 1995; received in revised form 10 August 1997

Abstract

The object-oriented approach provides an appropriate context for an integrated description of finite element related techniques within a single unified environment, combining symbolic and numerical manipulations, graphics and expertise. This paper discusses and illustrates the key features of such an integrated environment which includes an object-oriented graphic interactive environment, object-oriented operators for symbolic mathematical derivations, an object-oriented finite element environment, and its extension to object-oriented knowledge based expert assistance. © 1998 Elsevier Science Ltd and Civil-Comp Ltd. All rights reserved.

Keywords: Object-oriented, Finite element methods, Graphics, Automatic programming, Rule-based expert system

1. Introduction

Since the early work of some pioneers [1–5], the application of the object-oriented approach to scientific programming has been gaining popularity, as recent literature indicates [6–13]. The research work carried out by the authors, which initially dealt with object-oriented finite element programming [14–16], was later extended to pre-processing [17], to automatic programming [18], and to object intelligence for scientific and technical applications [19]. The purpose of this paper is to put these different aspects in perspective and to illustrate their relationships.

Application of the object-oriented approach to mechanical problems simulated by finite elements is the first aim of the research described herein; the selection of classes and objects is of course influenced by this priority and this aspect is discussed first in Section 2, pre- and post-processing are then described in Section 3, followed by automatic coding in Section 4 and finally object intelligence in Section 5.

2. The classes and objects of the finite element method

An appropriate selection of the classes of objects is essential in the implementation of the object-oriented approach to finite elements. Classes should group a set of data and corresponding methods in a way which appears the most natural to every object-oriented programmer knowledgeable in the field of application.

The language of implementation, Smalltalk for most of the work described herein, provides the basic environment. Some of the additional classes needed for the implementation of finite elements appear as obvious choices, such as class Element, with subclasses such as PlaneStrain, Truss2D, etc. Other classes, although less evident from the finite element point of view, impose themselves as a necessity, e.g. class Dof. In fact, when the set of data to encapsulate and the associated methods are easy to identify, for example when a physical correspondence exists, then the chances of building a class which is inappropriate are small.
An initial hierarchy of classes was proposed in refs [14, 15] (Fig. 1); until now it has proved to be appropriate and remained almost unchanged except for extensions. The major classes of the class hierarchy are FEMComponent, Dof (the degree of freedom), Domain (which manages the problem), LinearSystem, and GaussPoint. FEMComponent groups the key features of finite element analysis: Element, Load, Material, Node, all identifiable almost naturally at first inspection.

Some classes, like TimeIntegrationScheme, with corresponding subclasses Newmark or Static, have the apparent character of a method. Their implementation is therefore slightly less straightforward. Class Domain, and class Newmark are typical examples of these ‘algorithmic’ classes and their implementation is briefly described next.

2.1. Class Domain

The domain achieves two main tasks: the management of the components (elements, nodes, etc.) and problem solving. Both tasks are algorithmic in nature, the latter is discussed below. The solution procedure is, in principle, under the control of the user who sends a sequence of messages to the system. A typical sequence of user instructions is illustrated in Fig. 2.

The problem to be solved is materialized here by an instance of class Domain called MyStructure. The first message will create a new instance of class Domain called MyStructure. The second associates this new instance of class Domain with a data file called ‘mystruct.dat’. The third starts the solution process which is described in more detail next and the fourth requests the display of the solution.

The solution procedure itself is illustrated in Fig. 3. It consists essentially in forming and solving repeatedly the linear system. Notice that there is no anticipation of which time step or which element is currently to be processed.

2.2. Class TimeIntegrationScheme

Class TimeIntegrationScheme groups classes corresponding to specific time integration schemes: class Newmark for the Newmark scheme, and class ParabolicNewmark for the Newmark scheme for parabolic equations. These classes have the two following main tasks:

- indicating if the left-hand-side of the linear system has to be computed at each time step;
- managing the algorithmic parameters (β, γ) (Fig. 4).
Notice that the time stepping itself is done in class Domain.

```smalltalk
solveYourself
"Create repeatedly a new time step, then solve the receiver at that time step; stop whenever the list of steps in the input file is exhausted."
1 currentStep 1
[(currentStep:=self getNextTimeStep) exists]
whileTrue: [self solveYourselfAt: currentStep].
solveYourselfAt: stepN
"solve the receiver at time step stepN."
self formLinearSystemAt: stepN.
linearSystem solveYourself.
self terminate: stepN.

formLinearSystemAt: stepN
"form at time step stepN the system of linear equations"
1 nElem 1
stepN requiresNewLhs
ifTrue: [
    [self giveLinearSystem
        giveLHS reinitalizeYourself].
nElem := self giveNumberOfElements.
1 to:nElem do: [:i |
    (self giveElementAt: i)
        assembleYourselfAt: stepN].
nodeDictionary do: [:aNode |
    aNode assembleNodalLoadsAt: stepN].

terminate: stepN
"perform all operations for terminating time step stepN."

"Output"
stepN printNumber.
nodeDictionary do: [:node |
    node printOutputAt: stepN].
elementDictionary do: [:element |
    element printOutputAt: stepN].

"Update"
nodeDictionary do: [:node |
    node updateYourself].
elementDictionary do: [:element |
    element updateYourself].
timeStepDictionary at: 'previous'
    put:(timeStepDictionary at: 'current').
timeStepDictionary at: 'current'
    put: nil.
linearSystem updateYourself.
```

2.3. Encapsulation of state

Strict encapsulation avoids global parameter management; it should, however, be applied to the encapsulation of space and state, meaning that one should never anticipate the state of an object before sending a message to it; and, of course, the object should be capable of returning a meaningful response in all cases.

A typical situation is illustrated in Fig. 5 in which method giveStiffnessMatrix of class Element is shown, which will return the stiffness if it exists and build it if not. After several years of experimenting with object-oriented programming of the finite element method, “state encapsulation” (initially described in refs [14–16] as “Non anticipation”) appears to be an essential means of achieving code robustness.

2.4. Description of a FE program in Smalltalk

The set of classes forming the hierarchy of FEMObject is illustrated in Fig. 1. Although the illustration corresponds to an implementation in Smalltalk, the hierarchy is language independent. The selection of

```smalltalk
Class Newmark
inherits from: TimeIntegrationScheme, FEMObject, Object

Inherited tasks Inherited attributes Inherited methods

creation number giveNumber
       domain giveDomain

Tasks Attributes Methods

1 ) figuring out if the left-hand side of the linear system must be recalculated or not
   requiresNewLhsAt

2 ) managing algorithmic parameters
   beta giveBeta
gamma giveGamma

Fig. 4. Tasks list of class Newmark.

Fig. 5. Example of encapsulation of state.
```
classes for the implementation of the finite element method is certainly not unique. The best choice will be the one which achieves true encapsulation, useful inheritance, appropriate polymorphism and, as a corollary, the best reusability. Some redoing may sometimes be necessary in order to achieve this goal.

3. User interface

The user interface has three tasks: the steering of the program, the input of geometric and physical data, and graphic pre- and post-processing.

3.1. Steering of the program

In a Smalltalk implementation the user sends his messages to the system from the Transcript window (Fig. 2). This message passing is indeed similar to a macrocommand language.

Smalltalk (being an interpreted language) will activate only the parts of the class hierarchy which are needed for the problem being solved and the macrocommands can be activated interactively or in a sequence of operations. This same approach can also be implemented in a compiled language like C++. As a result, partial compilation of the environment will be possible and macrocommands appear as a natural extension of the object-oriented approach.

3.2. Interactive graphics

Geometric data are by definition associated with viewing. It seems natural to associate with each class of objects a method which projects the object on the screen and then to be able to manipulate it. A possible way to handle this situation is provided by the VO principle.

The implementation of the VO (Views, Owner) principle is illustrated in Fig. 6. In such an implementation the view cumulates the roles of the view itself and of the controller of the user’s actions, entered on the keyboard or with the mouse. This approach perfectly fits the object-oriented logic as each object will be able to project itself on the screen and to display its methods in the menu-bars, providing direct access for the user, e.g. through mouse clicks, and separate control possibilities for each object (Fig. 7). The user friendliness and the efficiency of this approach is illustrated below.

Fig. 6. The VO principle.

Fig. 7. The preprocessing environment.
in an example of mesh generation for a slope stability problem.

3.2.1. Example: preprocessing of the mesh for a slope stability

A layout of the problem is given in Fig. 8. Fig. 9(a)–(g) describes the preprocessing procedure.

The whole procedure takes 5 min to generate this coarse mesh. Automatic mesh refinement will then produce the refined mesh needed for the analysis. It is worth recalling that the user has direct access to the objects shown on the screen and that no additional toolbox is involved.

3.3. Physical data

Physical data are associated with geometric data and do not, in general, require any special treatment. It will be sufficient, most of the time, to point to the object and to fill out the empty slots in a template which is automatically displayed.

An additional aspect of physical data concerns units and the need for the enforcement of coherence of units. Dimensional analysis can be used throughout the class hierarchy to check data coherence. The same is true for automatic coding to check the dimensional coherence of the equations. This aspect will be discussed in Section 5.

4. Automatic programming

The desire to extend the previous approach to automatic finite element programming seems natural. This requires the development of a class hierarchy dedicated
Fig. 9. continued
(d) Z_PREPRO - Graphic Mesh Generation

(view global mesh)

(e) Z_PREPRO - Graphic Mesh Generation

(define the selected quad’s material (or function) = 2)

Fig. 9, continued
Fig. 9. continued
to symbolic manipulations. This new class hierarchy will allow the user to develop a new finite element formulation starting from a differential statement of an initial-boundary-value-problem, without paying attention to the details of the implementation.

The proposed approach is best illustrated in an example: the case of transient heat diffusion is examined here. The strong statement of the initial-boundary-value-problem is given as:

Find \(T\) on \(\Omega \times [0, T]\) such that

\[
\rho CT, + q, = f \quad \text{on } \Omega \times [0, T]
\]

\(T = g\) on \(\Gamma_g \times [0, T]\)

\(-q_n = h\) on \(\Gamma_h \times [0, T]\)

\(q_i = -K_0 T_j\) on \(\Omega \times [0, T]\)

\(T(x; 0) = T_0(x)\) \(x \in \Omega\)

where \(\Omega\) defines the domain with boundaries \((\Gamma = \Gamma_g \cup \Gamma_h)\), \(T\) is the solution field of temperatures, \(\rho\) and \(C\) are constants, \(g\) and \(h\) are given functions of space and time.

A variational statement of the problem is:

Find \(T \in S\) such that \(\forall w \in V\) one has

\[
\int_{\Omega} \rho CT,w dv + \int_{\Omega} q,w dv \int_{\Omega} f w dv
\]

with \(\int_{\Omega} pCT(0)w dv\)

with \(S\) the space of solution fields:

\(S = \{T(x; t) | x \in \Omega \text{ and } t \in [0, T] \text{ in } T(x; t) = g(x; t)\text{ for } x \in \Gamma_g \text{ and } T \text{ regular}\}\)

and \(V\) the space of virtual fields:

\(V = \{T(x; t) | x \in \Omega \text{ and } t \in [0, T] \text{ in } T(x; t) = 0\text{ for } x \in \Gamma_g \text{ and } T \text{ regular}\}\)

The introduction of appropriate discrete approximations of the solution will lead to the final matrix form which can be written:

\[MT + KT = Q\]
with \( \mathbf{M} \) the matrix of capacities, \( \mathbf{K} \) the matrix of conductivities, \( \mathbf{Q} \) the heat source vector, \( \mathbf{T} \) the vector of nodal temperatures, \( \frac{\partial \mathbf{T}}{\partial t} \) the time derivative of \( T \).

It is proposed, here, to perform the necessary manipulations in a symbolic object-oriented environment which will allow the user to perform, on the computer, more or less the same derivations as he would perform otherwise by hand. The sequence of operations can be followed by line in Fig. 10.

Line 1 corresponds to the differential equation weighted by \( w \) and integrated in the domain \( \Omega \), \( T \) is the temperature, \( w \) the virtual temperature; with the following notations: \( \rho = D \) is the density, \( C \) the capacity, \( f = R \) the volumic source term. Several classes of symbolic objects appear already at this level: class Term with instances ‘D’, ‘C’, ‘\( T, t \)’, ‘\( Si, i \)’, ‘W’, ‘R’, class Expression, with instance (‘\( \text{DCT}, t + \text{Si}, j \)’) where ‘\( Si \)’ is the heat flux vector etc., class Integral (‘\( \text{INT}\{\} \)’), class IntEquation (≡Line 1).

Line 2 results from an expansion of Line 1.

Line 3 ‘\( \text{INT}\{(\text{WSi}, i)/D\} \)’ in line 2 is integrated by parts resulting in two new integrals, ‘\( \text{INT}\{(\text{NiWSi})/dD\} \) – \( \text{INT}\{(\text{W}, i Si)/D\} \)’ (where ‘\( Ni \)’ is a unit outward normal vector component), the first one expressed over the domain boundary.

Line 4: the natural boundary condition is introduced into the surface integral of Line 3.

Line 5: the constitutive equation is substituted into the third integral in Line 4.

Line 6: Line 5 is expanded.

Line 7: discretization of Line 6 is achieved by using various user inputs activated by the ‘Discretize’ push button. ‘Nd’ discretizes ‘\( T \)’, ‘\( N*\text{d}^* \)’ and ‘\( W \)’.

Line 8: arbitrariness of the virtual field has been invoked to eliminate \( \text{d}^* \).

Line 9: through transposition and the explicit introduction of the shape functions, the final form is obtained.

Code generation results in adding a new element to the class hierarchy of FEM_Object; this is done by activating the Smalltalk (respectively C++) push button and will not be described in detail here.

The class hierarchy developed for the symbolic manipulations is illustrated in Fig. 11. It sits next to the class hierarchy already developed for numerical finite element analysis, FEM_Object, within the same environment. The symbolic environment consists of the classes already identified in the previous example, plus a few additions to the preexisting environment mainly in subclasses of class Collection.

A complete description of the environment is beyond the scope of this paper; the interested reader can find additional information and an example of application of the newly created element in Ref. [20].

5. Object intelligence

A further development of the object-oriented approach applied to finite element method consists in the introduction of reasoning processes which can be activated at the object level or alternatively at the level of a group of objects. The usefulness of the approach is illustrated in two examples.

5.1. Example 1

Given \( E \) and \( \nu \), the elastic modulus and Poisson’s ratio define the \( \lambda, \mu \) Lamé’s coefficients which are required for the material model. A set of rules is assumed to exist which define each elastic coefficient as a function of a pair of others. The form of such rules will be:

\[ E = E(\mu, \lambda). \]

If \( \mu \) known and \( \lambda \) known Then \( E = E(\mu, \lambda). \)
The complete set of rules is then:

\[
\begin{align*}
R1 : (v, \lambda) & \rightarrow E \\
R2 : (v, \mu) & \rightarrow E \\
R3 : (\mu, v) & \rightarrow E
\end{align*}
\]

The arrows (\(\rightarrow\)) indicate inferences. A forward chaining inference engine coupled with class ElasticMaterial will suffice to manage the problem of returning the requested pair of elastic parameters given any other pair.

5.2. Example 2

Given the equation of motion with partial definition of the units, define the others and check compatibility.

\[
\sigma_{ij,j} + f_i = \rho u_{i,t}
\]

Here, again a set of rules can be stated which will define the unit of \(\sigma_{ij,j}\) given the ones of \(\sigma_{ij}\). With notation \([\sigma_{ij}]\) for the unit of \(\sigma_{ij}\); \([\sigma_{ij,j}]\) and \([f_i]\), etc.

This time, however, the reasoning includes more than one object and may require backward chaining; for instance, to prove \([f_i] = [N/m^3]\) given partial information on the other terms.

5.3. Seamless integration

Lots of work has been done to integrate knowledge in terms of rules in object-oriented applications [21–25]. Most of the time, the rules are considered as an extension of an existing language: C++ for [21–23]; or as an extension of an existing environment such as Smalltalk [24], or Prolog [25]. However, the proposed approach is original because it merges object-oriented applications with propositional rule-based expert systems. It means that this approach is generic and does not depend on a specific language, even though it was developed in C++.

5.3.1. Intelligent objects

The proposed intelligent objects are designed as shown in Fig. 12. Each object consists of three parts:

- an object of the specific application described by methods;
- a knowledge base, i.e. a set of rules representing the available knowledge for this object; and finally
- a brain to infer on the knowledge base. Its behavior essentially describes the available reasoning processes. Inferences conducted by the brain acquire knowledge which is shared by other objects through a common device called a blackboard [26].

5.3.2. Global architecture

Designing such intelligent objects leads to the global organization shown in Fig. 12, which represents the architecture of an intelligent object-oriented application. The original application is implemented by its own hierarchy which is totally or partially reproduced at the level of intelligence in order to automatically organize the brain objects and the knowledge world as well. To duplicate the hierarchy of the application, each of its classes must be capable of self-identification, this capability requires the metaclass concept.

The amount of duplication of the application hierarchy depends on the available amount of knowledge. If at least one object of each class of the application hierarchy is described in one rule, the duplication will be total, otherwise it will only be partial.

5.3.3. Knowledge representation

Rules are methods acting on objects, therefore rules are encapsulated either into classes or into specific objects. In the former case, rules are related to all instances of the class, in the latter case they customize a particular instance. This is a convenient way to handle exceptional situations.

The relationship between classes and instances existing in the object-oriented application is mapped into the expert system environment, as Fig. 13 illustrates. A direct consequence is that rules associated with an intelligent object are of three kinds:
• rules inherited from the superclass of the object class;
• rules associated with its own class;
• its own rules.

As an example, consider the instance of class Term named ‘u’ in Fig. 13, it will be able to activate both rules R1 inherited from class Term and R2. Furthermore, rules may be overloaded in subclasses to specialize the behavior of objects as polymorphism does for the usual methods of a class.

The proposed approach integrates rules in objects, on which inheritance and polymorphism are reproduced; this demonstrates a seamless integration of rules as methods in object-oriented applications.

5.4. Reasoning modes

This section presents the reasoning modes implemented in intelligent objects. They represent the capability of a brain object to use its associated knowledge; in other words it describes the brain’s intelligent forms of behavior.

5.4.1. Generic scheme

The design of intelligent objects enables a brain to keep control of the knowledge processing by creating efficient reasoning mechanisms working on a limited number of rules. These reasoning mechanisms are based on usual backward and forward chaining, which are not considered as global procedures as usual, but as local and temporary objects which die when the process ends. Fig. 14 shows the generic implementation scheme that is activated to run a reasoning mechanism on a given knowledge base.

5.4.2. Deduction and definition processes

Two basic reasoning modes are implemented and explained here: deduction and definition processes. In fact, they correspond to the two well-known search strategies: forward chaining and backward chaining respectively. Other reasoning modes dedicated to specific needs may be created, based on these basic algorithms [27].

The deduction process is implemented by forward chaining working on the ruleset of an intelligent object. It is activated when new data come to its knowledge. This process tries to deduce data by firing as many rules as possible. Acquired knowledge may refine the object definition or be written on the blackboard to be used later on.

Rule R2 is an example of a deduction rule, where name and dimension may both be defined as attributes of an object of class Term. The second case is illustrated by the following rule:

\[ R3: \text{If } \text{[u]} \text{ is meter then } [u_{ij}] \text{ is [ ] (non-dimensional)} \]

in which the ‘Then’ statement concludes on the dimension of \( u_{ij} \). Its activation generates knowledge about the specific object of class Term named ‘\( u_{ij} \)’, which will stay on the blackboard until it is used by another object. This delayed use is automatically managed by the demanding object itself.

The definition process is implemented by backward chaining on the ruleset of an intelligent object, and it requires knowledge of the goal to achieve. This research strategy selects the conclusive rules for the goal; if their premises are not verified, they are considered as subgoals to be proven each in turn. When

---

**Fig. 13. Seamless integration scheme.**

**Fig. 14. Implementation scheme of runReasoningProcess.**
the process fails, due to missing information, external assistance is needed. Three types of help can be distinguished, which are:

- consultation of the blackboard contents;
- broadcasting the request to another object;
- addressing the question to the user.

As an illustration, consider the message getDimension() sent to an object to obtain its dimension. Its implementation is given in Fig. 15, and can be considered as a generalization of the state encapsulation presented in Section 2.

The two reasoning modes will automatically be coupled, because when a backward chaining is being processed, new information becomes available either from the user, or from the Blackboard object. This new data feeds the object, and leads to the activation of forward reasoning in order to take advantage of this newly acquired knowledge. This reasoning mode is sometimes called ‘opportunistic reasoning’ [28].

### 5.5. Implementation aspects

The intelligence within objects is implemented through the hierarchy shown in Fig. 16 representing an object-oriented rule based expert system. The classes written in italics are abstract classes, from which no objects are instantiated, and which essentially serve to regroup types of behavior common to subclasses. Besides these classes, the hierarchy distinguishes six group of classes that were generated by an object-oriented analysis of the global system.

Briefly speaking, the black dots (●) show the classes which are the basic components of an expert system i.e. class ExpertSystem, class KnowledgeBase and class ReasoningProcess. The open circles (○) correspond to the classes of objects which are identified in the rule definition, they are class Rule, class Fact, class ExpertAttribute, class ExpertClass, class Value and its subclasses, class Function and class ExternalFunction. Objects corresponding to these classes are instantiated when rules are analyzed from a file by a Parser object. The left arrows (→) mark the classes dedicated to knowledge processing, i.e. class Blackboard, class SimpleForwardReasoning, class SimpleBackwardReasoning and its subclasses. The first one creates objects which are necessary to perform a deduction process. The second one instantiates objects needed by the definition process, and subclasses of the class SimpleBackward-Reasoning differ from the behavior adopted to access the missing information. Further information on these subclasses can be found in Ref. [27]. Finally the stars (☆) denote the utility classes necessary to manage properly the object within the expert system, these classes are: class LinkedList, class Parser, class Stack class ValuesManager and its subclasses.

Referring to Fig. 13, the present implementation identifies R1 and R2 as Rule objects associated with ExpertClass object named ‘Term’, and ExpertAttribute object named ‘u’, respectively. Premises and conclusions of rules i.e. ‘name is u’ and ‘dimension is [m]’ are instances of class Fact and they are stored in LinkedList objects. Each Fact object consists of an ExpertAttribute object and a Value object, for example ‘dimension’ is an instance of class ExpertAttribute, and ‘[m]’ is an instance of class Value. KnowledgeBase objects encapsulate these components, e.g. rules, facts, expert attributes, values and so on. A knowledge base does not directly access them, this task is delegated to its own KnowledgeComponentManager object.
5.6. Example

The procedure is best illustrated with an example; dimensional analysis is taken again for further illustration. To make things simple assume we are given:

\[ [u] = m \]  
\[ [(\cdot)_i] = m^{-1} \]  
\[ [(\cdot)_i] = s^{-1} \]  
\[ [E_{ijkl}] = N/m^2 \]

Then, the rules below follow:

1. If the dimension of the LHS (left-hand-side) of the equation is known then the dimension of the equation is the dimension of the LHS.
2. If the dimension of the RHS (right-hand-side) of the equation is known then the dimension of the equation is the dimension of the RHS.
3. If all terms have known dimensions and the dimension of the product is unknown then the dimension of the product is the product of dimensions.
4. If the term is derived [()] then the dimension of the term is the dimension of its primitive multiplied by \([s^{-2}]\).
5. If all terms have known dimensions and the dimension of the product is unknown then the dimension of the product is the product of dimensions.
6. If one term has an unknown dimension and the dimension of the product is known then the dimension of the term can be computed.
7. If the term is named ‘’ then the dimension of the term is the dimension of its primitive multiplied by \([m^{-1}]\).
8. If the term is time derived twice [()] then the dimension of the term is the dimension of its primitive multiplied by \([s^{-2}]\).
9. If the term is named ‘’ then the dimension of the term is the dimension of its primitive multiplied by \([m^{-1}]\).
10. If the term is named ‘’ then the dimension of the term is the dimension of its primitive multiplied by \([m^{-1}]\).

The dimension of the primitive represents the dimension of the underived term, i.e. the dimension of the primitive of \(u_{i,j}\) is the dimension of \(u\).

This knowledge base is complemented by two rules dedicated to specific instances of class Term, i.e. ‘’ and ‘’ respectively. We assume here that the following, very specific, rules result from the initial-boundary-value-problem definition.

R9: If the term is named ‘’ then the dimension of the term is \([m]\) and \([u_{i,j}]\) is non-dimensional, i.e. \([\ldots]\).
R10: If the term is named ‘’ then the dimension of the term is \([N/m^2]\).

Fig. 17 shows hierarchical definitions of the objects of class Equation, which will facilitate the description of the reasoning process that leads to the determination of the dimension of all terms. The arrows can be interpreted as a ‘part-of’ relationship, and the dashed lines connect the objects of class Term, the

---

**Fig. 17. Hierarchical definition of the problem.**
primitive of which is the Term instance ‘$u$’. The rules associated with the objects are indicated in parentheses.

Notice that the knowledge bases consist of rules that define how the dimension of an object of a class is determined, and also how to take advantage of this new information. The first category of rules will be used in a backward chaining. Each of the rules delega tes the research of the information to its components through its premises. The Equation instance requires the dimension of its LHS, for example a Sum object, which in turn requires the access to the dimension of its Term instances. As for the forward chaining, its aim is to propagate as much as possible the acquired knowledge. If the dimension of the equation is known, it will be forwarded to its LHS, and RHS, which in turn will broadcast the information to their components.

The resolution process is illustrated in Fig. 18, in which the rules used by the backward chaining are represented in plain lines, the ones activated by the forward chaining in dashed lines, and dotted lines indicate messages sent by one object to another.

First of all, let us assume that the Term instance ‘$u$’ receives information that induces a forward chaining on its knowledge base i.e. R7, R8 and R9. R9 has verified premises, therefore it can be fired, its conclusion defines $[u] = [m]$, and $[u_{ij}] = [\cdot]$. A part of the information is not relative to the object definition, i.e. $[u_{ij}]$, it is stored on the blackboard.

Fig. 18 shows that the process starts by sending the callDimension() message to equation (vi). With respect to the implementation scheme, the method instantiates a backward chaining on R1 and R1b. R1 premises broadcast the getDimension() to the Sum instance. As the information is unavailable, the message is sent to $\sigma_{ij}$ through the premises of R3. Again as $[\sigma_{ij}]$ is unknown, a new backward chaining is performed on R7 and R8. The premises of R8 are satisfied, its conclusion can be applied, but it requires $[\sigma_{ij}]$, i.e. the dimension of the primitive of $\sigma_{ij}$. The Term instance $\sigma_{ij}$ is identified in the getDimensionOfPrimitive(); as this object belongs to the equation (v), the process skips to its hierarchy. $\sigma_{ij}$ does not know its dimension, and it tries to determine it by sending the callDimension() message to its ‘hierarchical parent’, i.e. equation (v). As the premises of R1b of the class Equation and the ones of R5 of the class Product are successively verified, the request is finally sent to $E_{ijkl}$. Here, among the available knowledge, R7, R8 and R10 has verified premises, and leads to the conclusion that $[E_{ijkl}] = [N/m^2]$. The other term of the product, i.e. $E_{ijkl,kl}$ is also analyzed in order to return its dimension. The information is available from the blackboard i.e. $[u_{j}] = [\cdot]$. $E_{ijkl,kl}$ can now be computed with R5 and leads to $[N/m^2]$, as well as the dimensions of the equation (v) and $[\sigma_{ij}]$; both are equal to $[N/m^2]$. Then $[\sigma_{ij}]$ becomes known with R8 $[N/m^2]$, and the Sum instance is therefore determined by R3 to be also equal to $[N/m^2]$. A forward chaining is activated on R4 to deduce $[f] = [N/m^2]$, and finally equation (vi) is totally determined, as well as the product. The product receives its dimension $[N/m^2]$ and induces a forward chaining from which it computes $[u_{ij}t]$ to determine $[p]$. The process to access $[u_{ij}t]$ is identical with the one developed for $[u_{ij}]$, and leads to $[m/s^2]$. Finally $[p]$ is found to be equal to $[N/m^4 s^2]$.

6. Conclusion

This paper reflects an attempt to develop object-oriented techniques aiming at a user-friendly environment for finite element developers. The environment combines symbolic and numeric capabilities, includes reasoning and some advanced graphical tools. Most capabilities were tested under both Smalltalk and C++. The idea is to build an environment which favors creativity by offering the possibility to conceive programs and test new ideas in a matter of minutes. At the present stage of development, both the feasibility and the usefulness of the proposed approach look encouraging.

References


