Engineering with Computers (1999) 15: 12–36

Engineering with Computers

Object-Oriented Symbolic Derivation and Automatic Programming of Finite Elements in Mechanics

D. Eyheramendy[1,2] and Th. Zimmermann[1]

[1]Laboratory of Structural and Continuum Mechanics (LSC), Swiss Federal Institute of Technology of Lausanne (EPFL), Lausanne, Switzerland; [2]CDCSP/ISTIL Université Claude Bernard Lyon 1, Villeurbanne, France

Abstract. *Symbolic approaches to assist in the development of finite element formulations have been used since the late 1970s. Today, symbolic mathematical software such as Mathematica, Maple, etc., has proved to be helpful when testing formulations. In earlier work, the authors introduced a new way of integrating naturally symbolic concepts in numerical finite element codes, taking advantage of an object-oriented code organization. In this paper, we wish to prove on practical examples that the proposed approach is very attractive and promising today, leading to an alternative way of conceiving finite element codes. After presenting a state-of-the-art of symbolic approaches for finite element developments, we first give a practical application of symbolic developments (for discontinuous space-time formulations), and then examples of Computer Aided Software Engineering tools that can be introduced into such a finite element environment.*

Keywords. Finite elements; Object-oriented programming; Symbolic approaches

1. Introduction

New technologies in computer science applied to mechanical computations open the way to alternative approaches for the solution of mechanical problems. The usual approach consists of performing a theoretical study of the given problem, which normally leads to tedious procedures carried out by hand and followed by a computer model implementation. Nowadays, these successive operations can be performed more efficiently through the use of *high level software tools*, as shown in Fritzson and Fritzson [1]. These tools can be grouped into three main categories. The first corresponds to the last generation of high level programming tools, which can be decomposed into two main classes: the classical procedural languages (Fortran 77 and 90, Pascal, etc.) and the object-oriented languages (C++, Smalltalk, Java, etc.). The second group includes algebraic software systems such as Maple, Mathematica, Matlab, etc. As shown in Fritzson and Fritzson [1], it is worth having a third type of approach, hybrid, and dedicated to general mechanical analysis, which means an approach based on mixed symbolic-numerical tools. The objective of this paper is to introduce an example of the development of a finite element formulation in such an environment, based on high level programming languages, capable of both manipulating algebraic equations and performing efficient numerical computations. It must be widely open to all types of future extensions, such as the application to new finite element formulations or alternative numerical schemes.

In Section 2, an attempt is made to classify different kinds of implementations of symbolic concepts for the finite element method. It is followed by a description of the main concepts of a new unifying environment called FEM_Theory, mixing symbolic and numerical features to support and speed up the development of finite element code. Earlier published work is extended in Section 3 to an example of the development of space-time formulations based on a one-dimensional linear advection equation; and in Section 4, we show that high level tools to assist the code developer can be conceived very easily in such an environment. This is illustrated through the examples of writing consistency control and dimensional analysis.

Correspondence and offprint requests to: Dr Th. Zimmermann, LSC, Swiss Federal Institute of Technology of Lausanne, 1015 Lausanne, Switzerland. E-mail: lsc@dgc.epfl.ch

2. Overview of the Use of Mathematical Tools and Object-Oriented Programming for Finite Elements

2.1. Use of Algebraic Computation Tools for Finite Elements

The use of algebraic manipulations software has always been a point of interest for finite element development. The first related works date from the beginning of the development of the finite element method in the 1970s. Among these, Luft et al. [2] describe a methodology to automatically generate finite element matrices based on the characteristics of the new element; the approach is restricted, however, to a finite number of problems: plane strain, bending and shallow shells. Since then, a lot of people have used algebraic computation capabilities to assist finite element solution procedures. Similar works, organized in three main categories, are mentioned here. First, some authors apply symbolic computer systems directly to finite element analysis, mixing both analytical and numerical approaches. The second category groups all the works whose main objective is to improve the efficiency of numerical computations in classical finite element codes. Finally, some authors aim at accelerating finite element code development using either existing tools or generating them.

2.1.1. Semi-analytical/numerical Approaches

In this type of approach, a classical finite element approach is programmed within a symbolic software package. Some variables are kept as symbolic parameters, and thus their influence on the computations can be evaluated. Two typical examples are given here.

In Choi and Nomura [3], an application to 2D elasticity is developed within the symbolic algebra software Mathematica. The displacement fields for a 2D body subject to linear temperature distribution is obtained in a semi-analytical form. Two tests are performed: a homogeneous elastic body with rectangular shape; and a body containing a circular inhomogeneity. This method renders possible the automation of otherwise tedious code writing, and can be useful for sensitivity analysis because all relevant parameters remain in symbolic form. In Ioakimidis [4], the software tool Mathematica is used for the solution of two simple elasticity problems by the finite element method. The principle of the approach consists of keeping a parameter in the symbolic form of the finite element matrices, and using Taylor series expansion for approximations. Thus, the objective is to try to optimize the parameter of the computation. It is applied [4] to a square plane isotropic elastic medium under symmetric loads, divided in eight triangular elements. The problem is solved by means of a Gauss–Seidel method, which makes it possible to study the influence of Poisson's ratio. The second example consists of the bending analysis of a rectangular isotropic elastic plate with simply supported edges and loaded with a uniformly distributed perpendicular load. Here the influence of the ratio of the dimensions is studied.

This semi-analytical/numerical environment should obviously provide a convenient framework for the optimization of parameters, through the use of finite element techniques for the computation. But at the current state of development in software and hardware, this can only be applied to small problems. Moreover, extension to alternative linear and *a fortiori* nonlinear problems seems difficult.

2.1.2. Enhancing Finite Element Code Performance

Another current use of mathematical software tools consists of performing some preliminary computations in order to enhance the efficiency of the finite element code.

In Yang [5], expressions for linear isotropic materials in statics in 2D and 3D are evaluated algebraically, and integration of the stiffness matrix and external forces is performed. Thus the integration scheme is optimized before the code is written.

In Silvester and Chamlian [6], the analytical integration scheme is also optimized, through the use of Maple, and the Fortran finite element code is generated directly by means of a Maple functionality. The code is then applied to solve finite element problems in magnetics. An approach with similar purposes is developed by Yagawa et al. [7]. Here REDUCE and Macsyma are employed to optimize a 2D 4-node isoparametric element for elastic analysis and to generate the corresponding code. In Bardel [8], an application of symbolic computing to the hierarchical FEM is shown; in this method, the degree p of the approximating polynomial functions tends to infinity, which addresses the problem of the accurate computation of the integrals for large values of p. The approach chosen in this paper is to evaluate them in a symbolic way

by means of the package REDUCE. This is illustrated on 2D elasticity, and the Fortran code is produced automatically, by means of a REDUCE function.

Two important features of using existing mathematical packages are the following. On the one hand, it is possible to use the power and flexibility of these environments to optimize the expressions needed to evaluate finite element matrices. On the other hand, the numerical code can be generated directly within the environment; the advantage is that the code which is generated automatically does not need any debugging.

2.1.3. Speeding up Finite Element Code Development

The derivation of finite element matrices generally involves tedious mathematical computations. The idea is to reduce the time spent on these manipulations through the use of a symbolic mathematical environment to determine the matrices of the finite element method, and eventually introduce the final elemental forms automatically into an existing numerical code (written in Fortran for all the examples of this section). This leads to a systematic development of a finite element code for a given formulation. Some of the works presented below propose programs which directly generate the correct matrices. They are fed with various input parameters, such as number of nodes or number of degrees of freedom. Some other works use classical mathematical software to perform the derivations.

An illustration of the first approach is given by Gunderson and Cetiner [9]. This paper presents the main features needed to develop finite element stiffness matrices with a computer. An illustration is made for the development of a third order triangular plate bending element. This makes it possible to test, at low cost, new elements for solving a given practical problem. References [10] and [11] are based on the same approach. Applications are shown on a cylindrical shell and on the analysis of a curved beam element [9], whereas in [10], simple examples are shown but the method is applied to space-time elements. In Luft et al. [2], the use of algebraic software is suggested to manipulate polynomials and perform numerical integration for finite element development. This methodology includes the choice of parameters, such as the number of nodes, number of degrees of freedom per node for each variable, expansion of the polynomial for displacements, geometric and material properties. The user then keeps the main features of a finite model under his control in order to obtain the correct matrix forms. Many authors have followed the same approach. In Barbier [12], the mathematical package REDUCE is used to automatically produce elemental mass and stiffness matrices by means of Hermite polynomials, and then generate the corresponding Fortran code for a conventional finite element code. In the same way, in Korncoff and Fenves [13], symbolic generation of a finite element stiffness matrix is achieved. Here, the authors have taken advantage of the user-friendly capabilities of MACSYMA: a library option gives access to a set of pre-defined matrices' shapes for material properties in linear elasticity. In Noor and Anderson [14], the potential of using symbolic manipulations in the development of nonlinear finite elements is shown. This is the only work that was found relating to the study of nonlinear problems. The development of nonlinear finite elements goes through three steps: the generation of the algebraic expressions for the stiffness coefficients of nonlinear analysis; the generation of the corresponding Fortran code for numerical evaluation of stiffness coefficients; and the checking of the consistency of the Fortran code generated by comparing it to the Fortran statements for the arrays of coefficients given in the MACSYMA format. Two examples illustrate the approach. A displacement formulation for a 2D shear-flexible, doubly-curved deep shell element, and a mixed formulation for the same model with discontinuous stress-resultant fields at inter-element boundaries. In Cameron [15], the algebraic software Maple is used for multivariate polynomials computation for finite element models. Polynomials and their derivatives are computed through the use of Horner's method, and efficient C and Fortran codes are produced. In Leff and Yun [16], a system for the generation of the global stiffness matrix is described. An input file for a specific problem is created for a system called SFEAS (Symbolic Finite Element Analysis System), which generates a file in the symbolic mathematical language REDUCE. The result is run and a Fortran code is produced, and then integrated into the equation solving system. The code produced here is much more efficient than NASTRAN, but the preprocessing phase which includes running the REDUCE system is slow. In Wong [17], a Lisp-based system to derive formulas for the finite element method and to generate Fortran code directly is described. Efficient techniques for code generation are employed, such as automatic labeling of expressions and exploitation of symmetries in expressions. It is the only reference in which the problem of automatic programming is

clearly addressed. The package is written in Lisp and runs with MACSYMA. The input can be given by the user interactively or introduced via a script file. The different entities needed for finite element formulations can be generated, for example *B*-matrix (see [77, p. 87]), Jacobian matrix, stiffness matrix, etc. The accent is put on the optimization of code generation, aiming at getting an efficient numerical code. These last two examples are probably among the best systems which were developed. Many other similar applications can be found in [18] and [19], and the references therein.

The examples given in this section show the usefulness of high level tools in the development of finite elements, and this analysis draws the main lines of the concepts needed for a general purpose environment dedicated to the finite element method. The proposed approaches demonstrate the potential of symbolic software tools for enhancing FE techniques in a computerized environment; on the one hand, the domain of application is wide, and on the other, various solution schemes exist. They show that, in order to get a general purpose system for fast prototyping of finite elements, several ingredients are necessary: a natural and user-friendly description of the problem, an efficient symbolic computation tool and, finally, an efficient link between the symbolic tool and the numerical finite element code. All these systems need a preliminary analysis, usually performed manually, before the development can be passed over to the computer algebra software, and suffer from a lack of generalization capabilities. In fact, all these systems have their drawbacks. The first and most important one is that all the systems still need a preliminary analysis performed manually, which can be rather tedious. The second one is the necessity to use multiple systems which *a fortiori* require that the developer should know each of them. For example, in [6–8], derivations of finite element matrices are obtained through an algebraic system (Maple, REDUCE, Macsyma); the elemental forms are then introduced into a classical finite element code by means of a classical programming language (in all these cases Fortran). Consequently, the user has to make the symbolic computations in one environment and the numerical computations in another, with the necessity for him to be able to evolve in two different programming environments, and to learn both an algebraic software language and a classical programming language. The third drawback has to do with the computerized symbolic manipulations. Each of these systems has been developed to optimize specific features of the finite element approach; for example, in Yagawa et al. [7], numerical integration is optimized, whereas in Cameron [15] it is the accuracy of the computation of polynomials that is optimized. This means that all these systems are specialized for some specific tasks. In fact, the extension of these tools or approaches to new finite element problems can become a tremendously time-consuming task, and can lead to impossibilities in complex nonlinear approaches.

The use of object-oriented techniques makes it possible to overcome these difficulties, while keeping the main advantages of the symbolic approaches.

2.2. Object-Oriented Finite Element Programming

Several difficult steps precede the actual development of FE software, which represents only the last step in the process of developing simulation tools. At the very beginning lies a given physical problem, which is generally modelled by a set of partial differential equations. At this stage, assumptions are made on the geometry, the kinematics, the loading, etc. A finite element strategy is then applied to the mathematical model. This results in general in a few pages describing the algorithms and the matrix form of the problem, expressed in a rather simple mathematical language. Traditional approaches lead to the elaboration of the corresponding computational tool, which is usually quite different from the original mathematical form. The problem of both the architecture of the software and the language used in this development is a crucial point evoked, for example, in [20,21]; to summarize, it is necessary in some sense to get closer to the natural mathematical or mechanical language. Thus, the coupling between conventional procedural approaches (the most popular is Fortran) and the developing of high level data abstraction concepts with simple and natural programming rules, leads to a new generation of FE codes [22,20].

A new approach for the FE code organization, advocated in [23,24], promotes object-oriented programming. This approach naturally encompasses concepts for a high-level architecture, and evolution towards more natural mathematical languages. For the first time [24,25], object-oriented programming has been proposed as a general methodology for finite element implementation. Both implementation examples use a Lisp-based system. One of the key points of the method to get better structured programs is the very high level data abstraction capabilities of the approach. In Rehak and Baugh [24],

objects of matrix type appear, and in Miller [25], structural objects such as node, degree of freedom, and element are described. The latter is completed in Miller [26], where object-oriented languages are discussed. In Fenves [27], the modularity and reusability of object-oriented finite element codes are put forward, and the efficiency in the design and the implementation of FE is emphasized. The same conclusions are drawn by Forde et al. [28]. Here an interesting comparison is performed between a classical FE code (a C program) and an equivalent object-oriented version (a mixed C–Object Pascal program). The size of the OO code is smaller, mainly due to the use of both hierarchical organization and inheritance. Similar remarks can be found in many papers: [29–44]. In Zimmermann et al. [45–47], a complete OO environment for linear FE analysis is thoroughly discussed. A new concept is introduced here as a programming rule, 'the non-anticipation rule'. By never anticipating the state of the object when sending it a message, the code becomes much more robust. The extension of the ideas to nonlinear analysis can be found in Dubois-Pélerin and Pegon [48,49], with additional interesting concepts such as 'unassembled matrix', which seems to allow a more flexible implementation of solution schemes by means of alternative storage. A complementary approach to that proposed by Dubois-Pélerin and Zimmermann [47] is proposed by Menétrey and Zimmermann [50] for nonlinear constitutive laws, here J_2 plasticity. Accordingly, in References [51,52], an advanced description of the object 'material' is given. The integration of complex constitutive laws in a C++ object-oriented FE code is made easier and more flexible through the use of C++ programming rules, permitting dynamic binding and linking of the code. Since then, the object-oriented paradigm has been used in many fields of computational mechanics: in parallel implementations of the FE code [53–55], in rapid dynamics [56,57], in multi-domain analysis for metal cutting, mould filling, in composite material forming [58,59], and in fracture mechanics [60]. This list is of course non-exhaustive, but shows that these ideas are now widespread in the computational mechanics community. In most of these works, it has been shown that the implementation more closely resembles the mathematical developments. Roughly speaking, the algorithms are easier to describe, and the definition of basic mathematical entities is natural. The object-oriented paradigm has been shown to be the most appropriate to easily describe complex phenomena, but this description is usually limited to the elemental forms and their management within complex solution algorithms.

2.3. Object-Oriented Hybrid Symbolic-Numerical Approach for Finite Element Analysis

2.3.1. Basis of a Hybrid Symbolic-Numerical Approach for Finite Element Formulations

Taking into account the features developed in the works on symbolic derivations reported above, and on object-oriented finite element approaches, the idea is now to develop a system dedicated to fast prototyping of finite element formulations.

First, the need to deal with a large range of problems leads to the creation of an environment capable of managing all the concepts needed to mathematically describe both the physical problem (e.g. differential equations) and the elaboration of the finite element formulation (e.g. variational formulations, integration by parts, weak forms, finite element approximations, etc.). A second important feature is the necessity to keep a traditional numerical code, because of its efficiency. This can be justified, for example, for the following reason: complex geometric domains are necessary to test finite element formulations; somehow, tests have to be made on real life problems. The natural integration of both a numerical finite element environment and features for symbolic manipulations can easily be achieved through the object-oriented paradigm. Nowadays, in the category of high level languages, object-oriented programming is getting more and more attention in computational mechanics, as shown above. In the particular context of finite element software development, this type of approach leads to better structured codes for which maintenance and extendibility are facilitated. These are the capabilities of the approach to represent complex systems which lead us to select it. The result should be a global environment in which the numerician is able to move naturally, always using a language close to his natural one. This work can be seen as an extension of previous ideas developed for object-oriented concepts applied to finite elements [45–47] to the symbolic derivation of the finite elements formulations; it may be seen as a new way of programming finite elements. The link between the numerical world and the symbolic world leads to the development of object-oriented concepts for the automatic programming of symbolic elemental matrix forms derived from finite element formulations. The new environment for symbolic derivation is called FEM_Theory [61,62].

2.3.2. Fast Overview of the Object-Oriented Environment FEM_Theory

Symbolic concepts have been introduced into an object-oriented environment capable of representing the different steps of the derivation of numerical modeling [30,61–63]. The main classes of the environment are recalled here.

- Class **Term** represents the smallest entity manipulated here, the term. Its instances know their name, their indices, their derivation indices and time derivation indices. They are merely character strings, but they are capable of analyzing themselves, and define whether the field is scalar or tensorial; they can also identify if and which type of derivation operator is applied to them. Finally, they are capable of discretizing themselves.
- Class **Expression** has a variety of behavior. First, symbolic manipulations like addition, multiplication, inversion, derivation in local and global frames, identification of specific operators (like the divergence), distribution and substitution. Secondly, discretization of terms and, finally, code generation. Class **Integral** implements the same mathematical tools as **Expression**. The main difference consists in the application of the linearity property to expand the integral. This class has a specific discretization scheme and code generation process.
- Class **Functional** also corresponds to an expression, but with integrals as components. Most of the behavior is inherited from the super class **Expression**. However, it implements specific discretization and code generation. A detailed description of classes is beyond the scope of this article; concepts are described by Zimmermann and Eyheramendy [61], a complete description of the environment is given in Eyheramendy and Zimmermann [62], and automatic programming principles are given by Eyheramendy and Zimmermann [63]. In the following section, a practical example of a general development of a numerical formulation in the new environment is illustrated on an advective problem.

3. An Example of Development to Solve Advective Systems

3.1. Discontinuous Space-Time Formulations

In this section, the aim is to provide the environment with features for computational fluid dynamics. In this context, faced with the computation of deforming domains leads to a crucial strategic choice. This problem can be solved by adding a new unknown and a new equation to handle the interface (e.g. see [64–66] and references therein); but without using additional unknowns, the formulation to be used somehow needs to embed Lagrangian ingredients. The first possibility would be to use a fully Lagrangian formulation; large and sometimes unnecessary mesh distortions are one of the drawbacks of the method. An alternative approach is to use formulations mixing Lagrangian and Eulerian concepts. One of the most widely used is the Arbitrary Lagrangian Eulerian approach, widespread in the finite element community [67,68]. Discontinuous in time space-time formulations, initially used on a fixed mesh for accuracy purposes (e.g. see [69] for elastodynamics), were used with moving meshes first in [70–73]. The great interest of the formulation is its simplicity and its flexibility, i.e. its capability to allow moving meshes (driven or not). This method has also been used in large-scale flow simulations (e.g. see [74,75] and references therein).

The purpose of this section is to show how natural and easy it is to introduce the concepts needed to handle this kind of formulation into the FEM_Theory environment.

3.2. Integration of Discontinuous Space-Time Formulations Concepts in FEM Theory

3.2.1. The Objects Needed for Discontinuous Space-Time Formulations

The best way to illustrate the new approach we want to introduce in the environment is to isolate the new concept by means of a simple formulation. The new objects and behavior can then be deduced, and a new class can be described.

A discontinuous space-time formulation for a linear one-dimensional advective equation. In this section, the formulation is presented on the resolution of a simple linear one-dimensional advective equation, which is discussed at length by Shakib [76]. The purpose here is to introduce symbolic object-oriented concepts to manage the kind of space-time formulations described in [71–73] in FEM_Theory. The formulation is recalled first.

The strong form of the problem is given as follows: Find $u(x,t)$ with appropriate continuity conditions on $\Omega = [0,1]$ for $0 \le t \le t_f$ such that

$$u_{,t} + Au_{,x} = 0 \text{ on } \Omega$$

with boundary conditions $u(0,t) = \bar{u}$, $u(1,t) = 0$, and

initial conditions $u(x,0) = u_0$, where A is the advection constant.

The variational formulation is written on the space-time domain Q_n, on a space-time slab bounded by t_n and t_{n+1}, as illustrated in Fig. 1 (see [74] for more details about notations). Define the approximation spaces for solutions u and weighting functions w:

$$(\mathrm{S}^h)_n = \{u^h \in [H^1(Q_n)]^h | u^h = \bar{u} \quad \text{on } (P_n)_{\bar{u}}\}$$

$$(\mathrm{V}^h)_n = \{w^h \in [H^1(Q_n)]^h | w^h = 0 \quad \text{on } (P_n)_{\bar{u}}\}$$

The approximation of the variational form is: For each time slab $[t_n,t_{n+1}]$, find $u^h \in (\mathrm{S}^h)_n$ such that $\forall w^h \in (\mathrm{V}^h)_n$:

$$\int_{Q_n} (u^h_{,t} + u^h u^h_{,x}) w^h dq + \sum_{e=1}^{n_{el}} \int_{Q_{ne}} (u^h_{,t} + u^h u^h_{,x})$$

$$\tau(w^h_{,t} + u^h w^h_{,x}) dq + \int_{\Omega_n} [[u^h]] (w^h)^+_n dv = 0$$

The design of the stabilization parameter proposed by Shakib [76] is

$$\tau = \left(\left(\frac{2}{\Delta t} \right)^2 + \left(\frac{2|u|}{h} \right)^2 \right)^{-\frac{1}{2}}$$

where $\Delta t = t_{n+1} - t_n$ and h is the mesh parameter (spatial length of the element in the current case), and where $[[u^h]] = (u^h)^+_n - (u^h)^-_n$ is called the 'jump term', corresponding to the following definition: $(u^h)^\pm_n = \lim_{\varepsilon \to 0} u^h(t_n \pm \varepsilon)$.

The first integral of the formulation corresponds to the classical Galerkin formulation written on domain Q_n; the second one is the Galerkin Least-Squares term, added for stabilization purposes; the last one allows weak enforcement of the continuity of the solution u over the global domain. Theoretical details about this formulation can be found in Shakib [76].

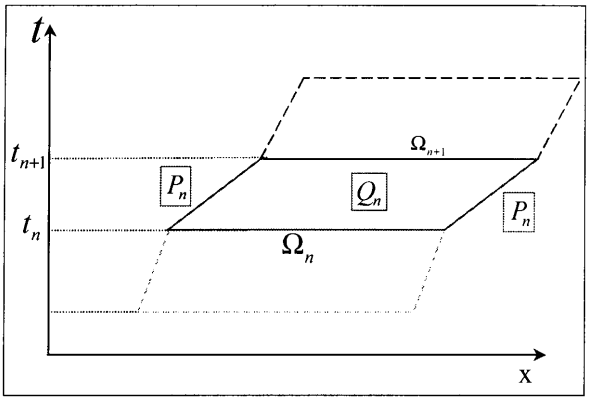

Fig. 1. Description of the space-time domain.

The objects for the discontinuous space-time formulation. The first two terms of the formulation on the space-time domain can be directly introduced in the FEM_Theory environment. In the sense of the finite element method, the time can be considered as an additional coordinate. So, the numerical treatment is obvious in the symbolic environment. But a new concept is needed to represent the third term, i.e. the 'jump term' $\int_{\Omega_n} [[u^h]] (w^h)^+_n dv$. Part of it is known, i.e. $(u^h)^-_n$ is computed at the previous time slab; and part of it is the current unknown, $(u^h)^+_n$. From the point of view of the finite element method, the formulation leads to the solution of a linear system at each time slab of the form $Kd = f$, where d is the vector of the nodal unknowns. The elemental contributions coming from the first two terms are obvious if classical finite elements are used. It is worth describing the elemental contributions due to the 'jump term'. They can be expressed by means of notations introduced by Hughes [77] as follows, on an element illustrated in Fig. 2.

$$K^e_{jump} = \int_{\Omega^e_n} N^t N \, d\Omega \quad \text{and} \quad f^e_{jump} = K^e_{jump} d^-$$

where N is the classical matrix of shape functions of [77], and d^- is the vector of nodal unknowns computed on the previous time slab. The integration is done on the space domain in the initial configuration (at t_n), i.e. on the surface Ω^e_n, as seen in Fig. 2. This shows that the FEM_Theory environment is capable of building elemental matrices such as K^e_{jump}; the new concepts to add here are those to manage and interpret the 'jump term', i.e. mainly concepts linked to the numerical integration scheme, and to the automatic implementation into the numerical code.

The idea is then to introduce a new object to represent the 'jump term' in the variational formulation, and to enrich the existing objects to handle this new object, particularly for automatic integration into the numerical code. To make the implementation easier, one can note the following:

$$\int_{\Omega_n} [[u^h]] (w^h)^+_n dv = \left[\left[\int_{\Omega_n} u^h (w^h)^+_n dv \right] \right]$$

This allows a treatment of the 'jump' at a higher level than inside the integral. The new object, an instance of a class called **JUMP_TERM,** is represented by the double bracket notation. It is natural to manipulate it as a special term in the formulation.

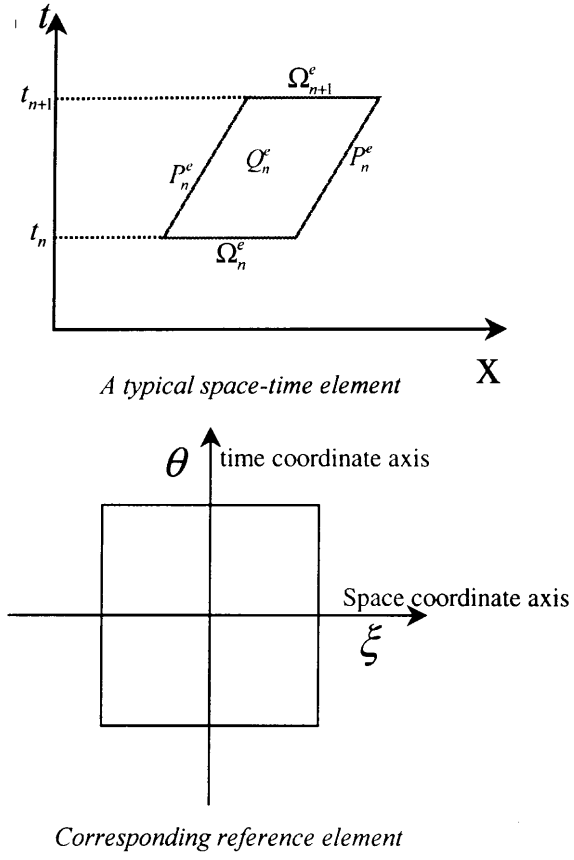

Fig. 2. An example of space-time element for a one-dimensional space.

The structure of the object appears in Fig. 3. The object has as the only piece of data its variable, which is in this example an integral. Most of the tasks are decentralized to the attribute variable; the algebraic manipulation methods are inherited from class **Term**. This object is a specialization of an object term. Additional tasks have to be added to other objects such as products for generating the code with the selector of methods needed for 'jump term' in the numerical code. Note that the space domain of the integral Ω^e_n has to be recognized; the name of the domain used is <<Sp>>, for spatial domain. A generalization of the generation of the code is needed here. Finally, the variational formulation (successively classes **IntEquation, DiscretizedEquation, System**) is now given an attribute to characterize the type of the formulation, i.e. either 'Semi-discrete approach' or 'Space-time approach'. In the latter, time is considered as a simple coordinate like a space coordinate.

The structure of the class JUMP_TERM is summarized in Table 1 (see elsewhere [62] for similar descriptions).

Class **JUMP_TERM** has one attribute called **variable**. This class is used either for the continuous problem or for the discrete problem; the simplicity of this structure avoids distributing tasks among two classes. For the continuous problem, **variable** can *a priori* be of any class (**Term, Integral, Expression**, etc.). Only class **Integral** is used here. For the discrete problem, **variable** is an instance of **DiscretizedExpression**. The consequence of this is that both tasks, for continuous and discrete problems, lie in the same structure.

Tasks linked to the integration into the hierarchical parent tree organization [62] and to algebraic manipulations are inherited from superclasses. The tasks may be decomposed into four groups. Most of the methods are in fact a specialization of existing methods of the class **Term**.

Most of the tasks are decentralized to the attribute **variable**. The only method which is specific to the class is *findMatrixCorrespondingToUnknown: discretVar*; it returns, in the case of the discrete form, the elemental contribution corresponding to the nodal unknown vector of the 'jump term'.

The analysis tasks are, as before, specializations of the class **Term**. This behavior is part of a group of tasks called manipulations.

The discretization procedure is decentralized to

Class **JUMP_TERM**

Example : $[[\int_{\Omega_n} u^h (w^h)^+_n dv]]$

main attributes :

variable : $\int_{\Omega_n} u^h (w^h)^+_n dv$ (here an integral)

main tasks :

manipulations in the variational form such as addition, product, … (can be inherited from class **Term**)

generation of code

Fig. 3. Typical instance of **JUMP_TERM**.

Table 1. Class **JUMP_TERM**

Class JUMP_TERM
Inherits from: Term, StructureWithDimension, FEMTheoryMathematicalStructures, FEMTheory, Object

Inherited tasks	Inherited attributes	Inherited methods
1) access to data of the hierarchic parent	hierarchicParent	getDiscretizationInfosForTerm: term getListOfTerms giveHierarchicParent giveSpaceDimension knowsAsUnknow: term (from **FEMTheoryMathematicalStructures**)
2) algebraic manipulations		+, *,… (from **Term**)
Tasks	**Attributes**	**Methods**
1) manipulation	– variable	asFunctional comesFromSurfaceLoad deriveWithRespectToVariable: i findAllUnknowns findMatrixCorrespondingToUnknown: discretVar get0KAFUnknownMatrix getDirectionalDerivative getJumpTermCorrespondingToUnknown: discretVar printString replaceYourselfUsingDictionary: dict transpose variable: aTerm
2) analysis		isBodyMatrix isSurfaceMatrix isZero
3) discretization		getDiscretizedForm
4) creation of code		createCPlusLoadMethodsIn: path forTimeDependentElement: elementName

the attribute variable, and the scheme described by Zimmermann and Eyheramendy [61,62] to build elemental contributions doesn't need any special implementation.

Code generation here implies the creation of the elemental contributions on the left- and right-hand sides of the discrete linear system. Both operations are initiated at this stage, but practical tasks are decentralized to the discrete form stored in the attribute **variable**.

The FEM_Theory environment is enriched with new graphical features. The first is the notation used for the 'jump term'. The double bracket notation introduced here can be directly integrated into FEM_Theory. As an example, the term $[[\int_{\Omega_n} u(w)_n^+ dv]]$ is represented as <<[[INT{UW//Sp}]]>> (see the next section). Note that here the term corresponding to the continuous problem is shown.

A new prompter window is added in order to ask the user which type of formulation he wishes to derive (see Fig. 4); this only influences the choice of the time coordinate to be taken into account for the space-time formulation as a simple coordinate axis. An example of space-time formulation derivation follows.

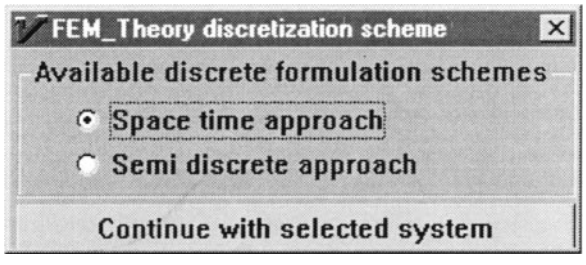

Fig. 4. Prompter to define the type of finite element formulation.

3.3. The Linear One-Dimensional Advection Equation in FEM_Theory

3.3.1. Derivation of the Linear 1D Advection Equation

The space-time formulation for the linear one-dimensional advection equation, derived in FEM_Theory is shown in Fig. 5 (the notations are close to those of the previous section). As similar derivations have already been shown in previous sections, only a brief line by line description is made here. On line 1 the original Galerkin formulation on the whole space-time domain is posted. On line 2, the 'jump term' is added; the formulation is written on a time slab; it is a Galerkin continuous in space and discontinuous in time form. On line 4, the formulation is approximated. The reference element chosen here is a four-node element for <<U>> and <<W>>. Stabilization Galerkin least-squares terms are added on line 5 (to the approximated formulation). The arbitrariness of the weighting function is invoked, and the resulting system of discrete equations appears on line 6. The equation is then transposed, in line 7, and an obvious change of notation is made. Bilinear interpolation for <<U>> and <<W>>, and a two-by-two Gauss integration rule, are chosen for all elemental contributions. The code is then generated.

Fig. 5. Derivation of a space-time formulation for a 1D advection equation.

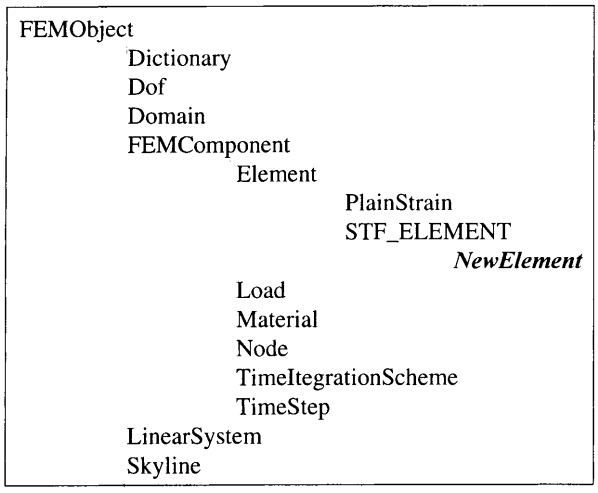

Fig. 6. Partial view of the hierarchy of FEM_Object for space-time formulation.

3.3.2. Code Generated Automatically in FEM_Object

As described in Eyheramendy and Zimmermann [63], a new class is generated. As the numerical code FEM_Object has been enhanced to support deforming domains with space-time formulations [30], the new element **NewElement** is now added as a subclass of a class called **STF_ELEMENT**, which contains special features for moving domains, as shown in the hierarchy of FEM_Object in Fig. 6. Note that on the list of methods given in Fig. 7, a new method to compute the 'jump term' in the load vector of the 'right-hand side' in the FEM_Object appears. The contribution to the left-hand side of

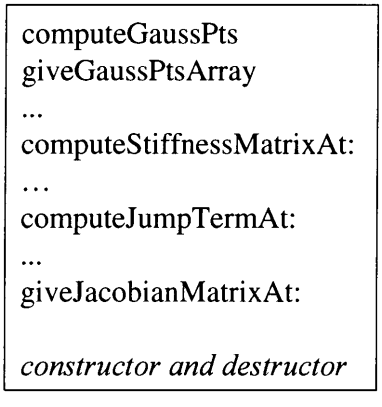

Fig. 7. Partial view of the methods added for space-time formulation in FEM_Object.

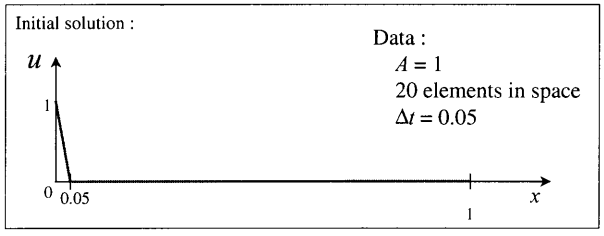

Fig. 8. Initial solution for u of the numerical test of the linear advection equation.

the 'jump term' is automatically integrated into the stiffness matrix.

3.3.3. Numerical Tests

The test done uses as an initial condition a discontinuity over one element, as shown in Fig. 8. The space is decomposed into 20 elements; the height of the time slab is $\Delta t = 0.05s$. The boundary conditions are $u(0) = 1$ and $u(1) = 0$. The first test is done with a fixed mesh. The results are shown in Fig. 9, with label STF (space-time formulation) for various values of the stabilization parameter at $t = 0.5$. These results show that for appropriate values of stabilization parameters, the oscillations before and after the discontinuity are attenuated, and that the discontinuity is caught correctly. A second derivation has been done with a semi-discrete approach, with an equivalent stabilization scheme. The results are also reported in Fig. 9. The numerical integration scheme in time used for solving the problem in the FEM_Object is a generalized trapezoidal rule presented by Hughes [77, Chap. 8]. The results are posted for various values of the stabilization parameter. The first remark that can be made is that the semi-discrete formulation cannot catch the sharp discontinuity as the space-time formulation does. Adding stabilization rapidly adds too much diffusivity. But the position of the discontinuity can also be caught. In conclusion, the space-time formulation seems to give better results in capturing sharp discontinuities.

Finally, a mesh moving procedure is introduced into the numerical computation. The mesh is moved with the advection velocity ($A = 1$). The results are shown in Fig. 10. It shows that the local advective effects disappear; in fact, the exact solution is obtained. This feature can be interesting whenever sharp discontinuities have to be caught.

3.4. Towards the Numerical Solution of Navier–Stokes Equation

In this section, the general framework to introduce symbolic concepts for space-time formulations which are discontinuous in time is created. This is illustrated through a simple one-dimensional equation. Obviously, all the features created here can be used for two- or three-dimensional problems, and even for nonlinear problems. An application of space-time formulations, discontinuous in time, to non-linear problems, including the solution of the Navier–Stokes equations, is discussed by Eyheramendy [30].

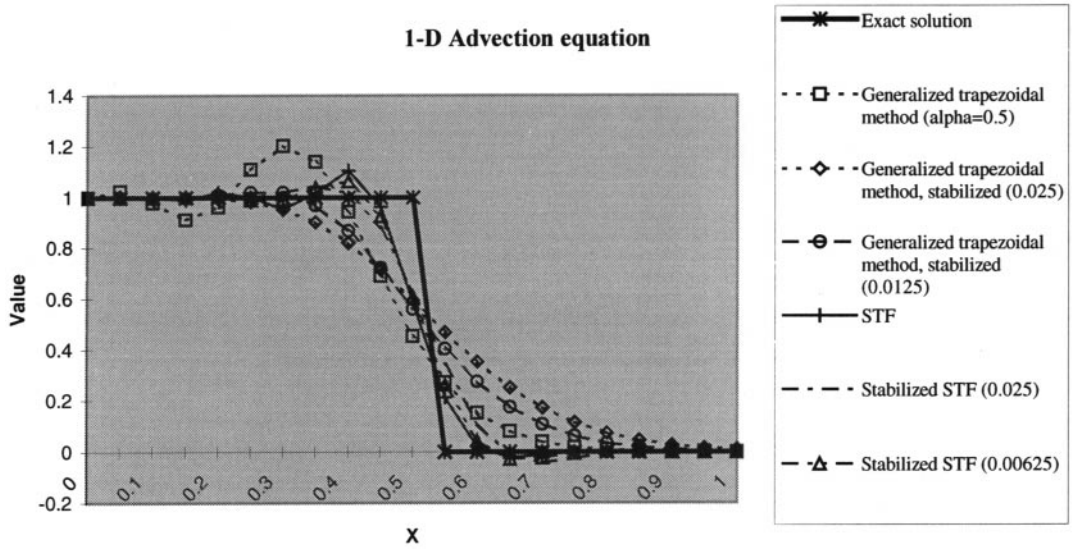

Fig. 9. Numerical test of the linear advection equation at $t = 10\ \Delta t$.

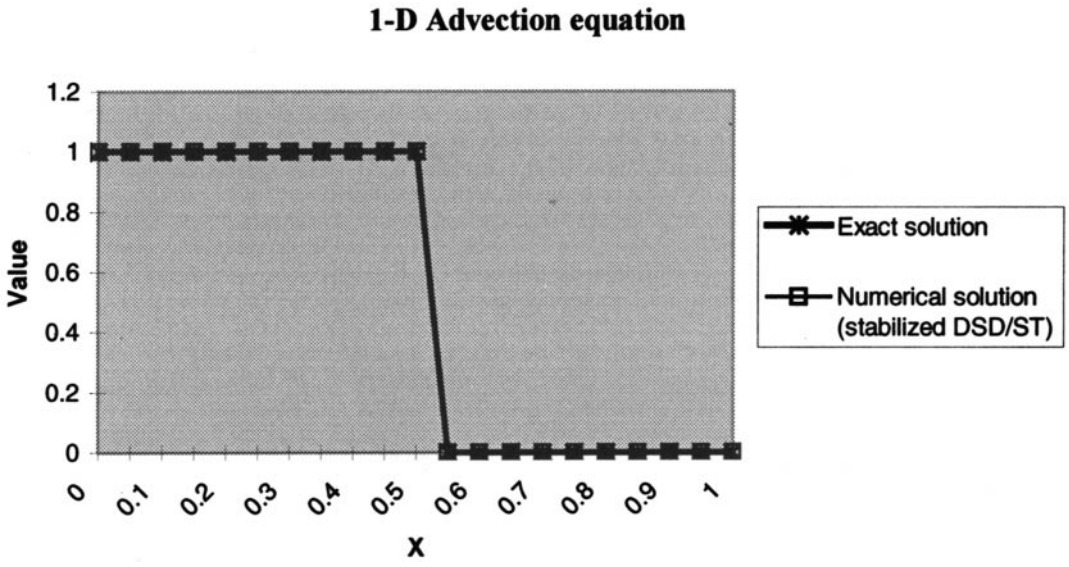

Fig. 10. Numerical solution for moving domain at $t = 10\ \Delta t$.

4. Computer Aided Software Engineering for Finite Element Developments

4.1. High Level Concepts of CASE Tools for Finite Element Developments

In this section, we wish to introduce several tracks to easily implement CASE tools to develop finite element formulations. The first is a tool to check the dimensional consistency in the symbolic environment. This could, of course, be extended to the numerical part of the code. The second consists in ensuring writing consistency of formulations, using index notation.

4.2. Dimensional Analysis in an Object-Oriented Environment for Finite Elements

4.2.1. A Brief Analysis of Dimensional Analysis
The point of departure is here the international system of units (ISO). Take the example of the French norm [AFNOR: NF X 02-051] or the Swiss norm [SNV 02121100]. From the normalization (see Table 3), it can be deduced that each magnitude has a unit that can be expressed by means of the seven basic units shown in Table 2. The definition of the unit can be completed by the use of a factor and a prefix, e.g. $1\ \text{ft} = 0.3048\ \text{m}$, where the factor is 0.3048, and $1\ \text{kN} = 10^3\text{N}$ (prefix k). All the units can be expressed in this way. The aim of this part is to build structures to represent the units, capable of conversion and analysis.

4.2.2. The Objects for Dimensional Analysis
In FEM_Theory, the unit, the basic object to be associated with a magnitude, and the behavior for dimensional analysis are inherited from the class **StructureWithDimension** (see [61,62]), e.g. for terms, expressions, integrals, etc. This object is illustrated in Fig. 11 through the example of Newton. The object has a name, a dimension and its definition can be completed by a prefix and a factor of conversion. For the sake of simplicity, the factor and prefix are not taken into account. The unit can have access to a database, where it could find all the data needed for conversion (similar to Table 3). At least, the unit can be associated with an object. So, class **Unit** is defined.

The main component of the object unit is its dimension (instance of class **Dimension**). The goal is to build a structure capable of giving a representation of the dimension based on Table 2. This is illustrated in Fig. 12. The idea is to use an existing structure of Smalltalk, the dictionary [78–80]. The key to getting access to the data stored in the dictionary is a symbol corresponding to the name of the magnitude. The data stored at the corresponding key is an integer, with a sign which gives the power of the basic unit. In Fig. 12, the power corresponding to the length (l) is 1, to the mass (m) is 1, to the time (t) is −2; the others are 0. The result for Newton is kg.m.s^{-2}. The main tasks of

Table 2. Basic units of the international system

Magnitude		Unit basis	
name	symbol	name	symbol
Length	l	meter	m
Mass	m	kilogram	kg
Time	t	second	s
Intensity of current	I	ampere	A
Thermodynamic temperature	T	kelvin	K
Quantity of material	n	mole	mol
Luminosity intensity	I_v	cadela	cd

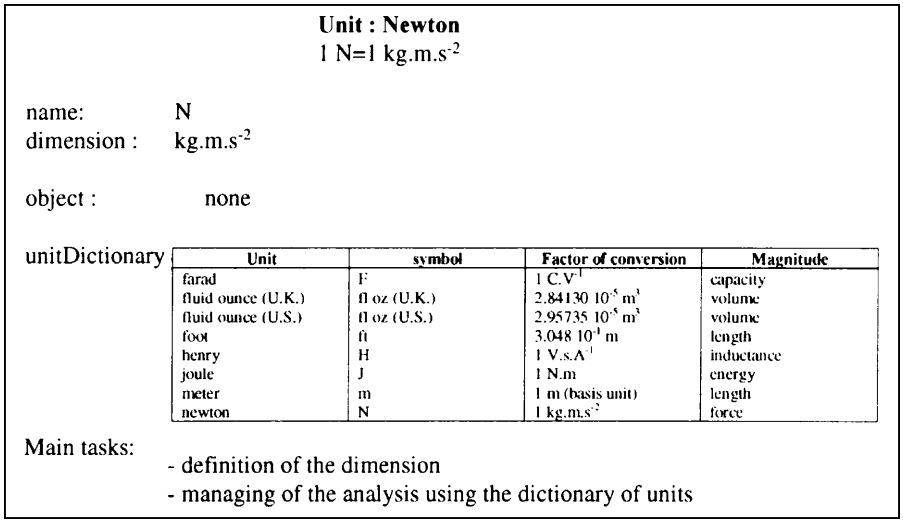

Fig. 11. Definition of the object unit in the example of Newton.

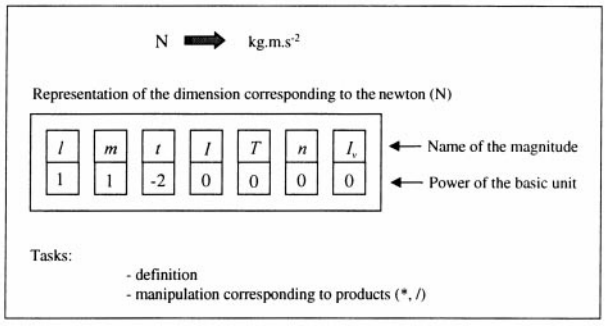

Fig. 12. Example of dimension.

this object are to define itself, by asking information from the user for example, and to combine dimensions in products. It is best to illustrate this through an example. Consider the product $P = m \cdot g$, where m is the mass expressed in kg (kilogram), g the acceleration of gravity expressed in m.s^{-2}, and P the weight. The result P is then expressed in kg.m.s^{-2}. This is found as follows. Represent the dimension of a magnitude by the notation [ ]. So, the dimension of P, $[P]$, is obtained by multiplying the dimension of m by the one of g: $[P] = [m]*[g]$. The 'product' which makes it possible to obtain this dimension is sketched in Fig. 13. The final dimension of P is obtained by simply adding the indices corresponding to the basic units. Thus, this object has the possibility to be multiplied or divided by another object dimension; so a basic algebra is defined at the level of the object **Dimension**.

The last object needed for dimensional analysis is a dictionary to store the units that can be seen, for example, in Table 3. A simple dictionary object in Smalltalk could be used here, but a specialization scheme is needed to look up units in the dictionary. This object is an instance of **FEMTheoryGeneralDictionaryOfUnits**.

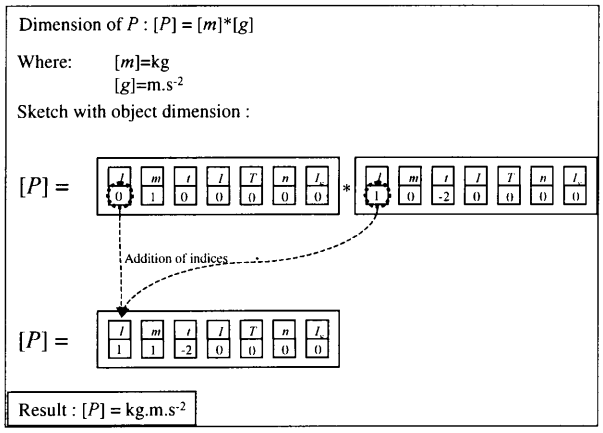

Fig. 13. Sketch for the dimension of $P = m \cdot g$.

4.2.3. The Classes

*Class **Dimension***. Class **Dimension**, presented in Table 4, is a dictionary of size 7, which corresponds to the number of basic units, i.e. length (symbol *l*), mass (symbol *m*), time (symbol *t*), intensity of current (symbol *I*), thermodynamic temperature (symbol *T*), material quantity (symbol *n*) and luminosity intensity (symbol *Iv*) – see AFNOR X02-051 or SNV 012100 for more details. Each of these symbols gives access in the dictionary to an integer which represents the power of the corresponding basic unit. The behavior of the classes consists, first, in defining the dimension and, second, in performing basic algebra manipulations on it.

*Class **Unit***. The class **Unit** (see Table 5) has five attributes. The first four make it possible to define the unit: the attribute **dimension** which is an instance of **Dimension** is completed by the attribute **name**, instance of **Symbol**. The unit can be the unit of a data, a term, a product, etc.; it is the attribute **object**. At last, the unit may need information to complete its definition from data stored in the dictionary of units, attribute **unitsDictionary**.

The first part of the behavior is linked to the complete definition of the unit, i.e. its symbol, attribute **name**, its dimension, and perhaps an object (term, expression, etc.) with which the unit is associated. The second part of the behavior is the management of the data contained in the dictionary of units.

Remark 1: The definition of the dimension by the user is decentralized to the attribute *dimension* itself (access to the prompter of Fig. 17).

Remark 2: To give a complete definition of all types of units two attributes could be added here. The first one is needed to represent the prefix of the unit (see NF 02-051), e.g. prefix ‘k’ for ‘kilo’ corresponding to 10^3. This new attribute *prefix* could be an instance of a new class **Prefix** similar to class **Unit**, but managing the prefixes. A second attribute, call it *factor*, a float instance of **Float**, is needed to achieve conversions between the units of the international system and the others (e.g. darcy, gallon, foot, etc.).

*Class **FEMTheoryGeneralDictionaryOfUnits***. This class, presented in Table 6, is used only to store the different units that can be used in FEM_Theory, and behaves as a classical Smalltalk dictionary [80,83]. Only one instance appears during execution; this instance is stored on disk, and recovered whenever needed, using the tool class **ObjectFiler** [79].

The key used to store the objects of type **Unit** is a symbol that is the name of the unit. For example the unit ‘Joule’ corresponding to work or energy, has as symbol *J* and dimension $kg.m^2.s^{-2}$. All the behavior of the class is inherited from **Dictionary**. Only one special method is added to get a unit from the definition of its dimension. This method allows us to make a loop on the values of the dictionary in order to get the key, i.e. from the definition of the dimension $kg.m^2.s^{-2}$, to get the unit *J*.

Table 3. Example of dimension of units (from NF X 02-051)

Unit	symbol	Factor of conversion	Magnitude
farad	F	1 $C.V^{-1}$	capacity
fluid ounce (U.K.)	fl oz (U.K.)	2.84130 10^{-5} m^3	volume
fluid ounce (U.S.)	fl oz (U.S.)	2.95735 10^{-5} m^3	volume
foot	ft	3.048 10^{-1} m	length
henry	H	1 $V.s.A^{-1}$	inductance
joule	J	1 N.m	energy
meter	m	1 m (basis unit)	length
newton	N	1 $kg.m.s^{-2}$	force

Table 4. Class **Dimension**

Class Dimension Inherits from: Dictionary, FEMTheoryDictionaries,…, Object		
Inherited tasks	Inherited attributes	Inherited methods
–	–	– *all the methods for dictionaries*
Tasks	Attributes	Methods
1) Definition	–	answerYourselfFor: an Obj asArray atAllPut: anInteger define defineFor: obj getBasicUnits giveBasicUnitsArray isDefined isNotDefined
2) Algebra		* aDimension / aDimension inverse power: anInt = aDict

Table 5. Class **Unit**

Class Unit Inherits from: FEMTheory, Object		
Inherited tasks	Inherited attributes	Inherited methods
–	–	–
Tasks	Attributes	Methods
1) Definition	– dimension – name – object	define defineDimension dimension: aDim isDefined isUnit name: aSymbol object: anObj prefix: aSymbol
2) Manipulations	– unitDictionary	getBasicUnits getDimension giveDimension giveName giveObject givePrefix initUnitDictionary

*Class **StructureWithDimension** and subclasses.*
The class **StructureWithDimension** (see Table 7) regroups the behavior common to subclasses needed for representing the variational formulation (see the general hierarchy of classes of FEM_Theory, Section 2). The only attribute of the class is called a **unit** and becomes an instance of a class **Unit**. The only class level behavior is linked to the man-

Table 6. Class **FEMTheoryGeneralDictionaryOfUnits**

Class FEMTheoryGeneralDictionaryOfUnits Inherits from: FEMTheoryDictionaries, Dictionary,…, Object		
Inherited tasks	Inherited attributes	Inherited methods
–	–	–
Tasks	Attributes	Methods
Manipulations		findUnitOfDimension: aDim

Table 7. Class **StructureWithDimension**

Class StructureWithDimension Inherits from: FEMTheory, FEMTheoryMathematicalStructures, Object		
Inherited tasks	Inherited attributes	Inherited methods
–	– hierarchicParent	– "management of the attribute *hierarchicParent*"
Tasks	Attributes	Methods
Managing of the unit	unit	addDimensionCharacteristicsTo: col forObject: an Obj deduce Dimension findDimensionBackwards giveUnit updateDimensionForTerm: aTerm

agement of the unit, i.e. its definition and the procedure to check the consistency of units in a variational formulation. This scheme is described in the next section.

4.2.4. Strategy for Dimensional Analysis in FEM_Theory

The scheme for dimensional analysis is based on the data structure presented in Section 2. The simple algorithm described here involves the class **StructureWithDimension** and subclasses (**Term, Expression, IntEquation**, etc.). The problem is to deduce the dimension of an object within a complex expression, just by giving the dimension of some terms. The purpose is not to give a general algorithm for the problem, rather to give an overview of the possibilities of such a tool. The principle of the algorithm is sketched in Fig. 14 on the equation $\int_\Omega(\sigma_{ij,j} + f_i)w_i dv = 0$ taken from the example of linear elasticity [81]. This scheme is described to find the dimension of the object integral $\int_\Omega(\sigma_{ij,j} + f_i)w_i dv$, but could be applied to any objects:

- on the screen, the object integral is selected;
- the tool '*Find Dimension For Term'* is selected and applied; consequently, the message *deduceDimensionSelection: integralString* is sent to the object **IntEquation**;
- the message goes down the roots of the tree following the dotted arrows in Fig. 14, until the selected integral is recognized (highlighted in gray in the figure);
- when the integral is found, the message *deduceDimension* is sent to the integral itself;
- in the method *deduceDimension*, a first search is made while descending the roots, sending successive messages *findDimensionForward*, shown by plain arrows in the figure; the goal is to try to deduce the dimension of an object, just from the dimensions of the objects composing it; this scheme is successful when one branch of a root at a sum level is completely defined;
- in the method *deduceDimension*, if the dimension is not found with a process descending the roots, an ascending process is started by the message *findDimensionBackward* (dashed line in the figure), which has the task of sending the message

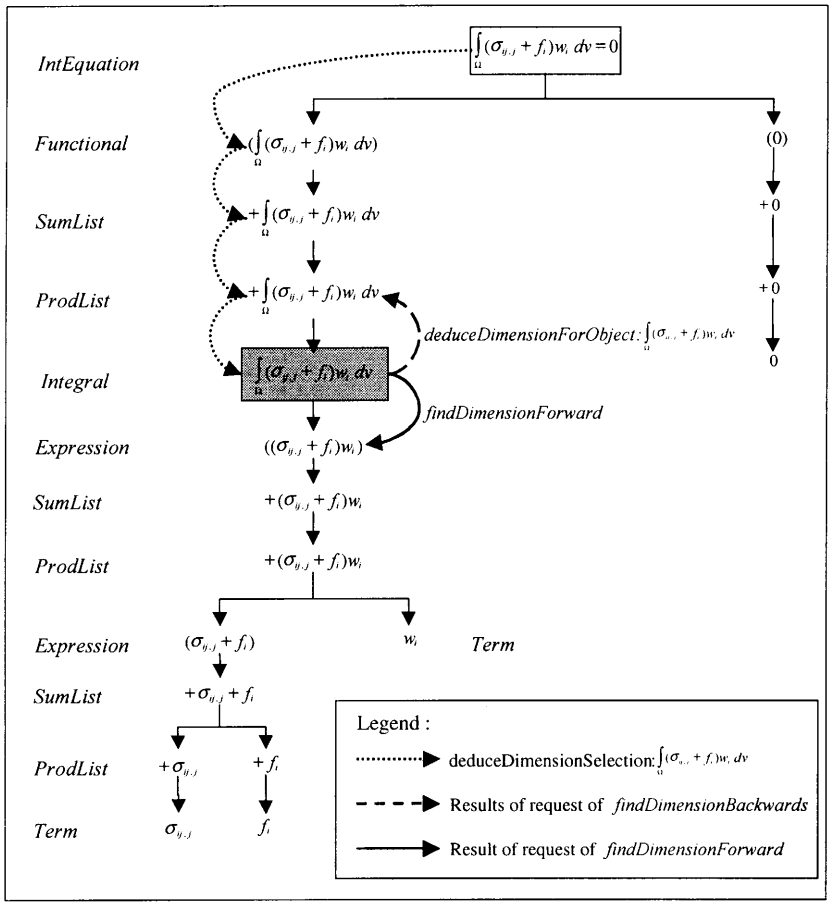

Fig. 14. Sketch illustrating the dimensional analysis strategy.

deduceDimensionForObject: integral; in this method either message *findDimensionForward* or *findDimensionBackward*, or both, are sent to try to deduce the dimension at the current node (recursive message passing);

- so, the messages go down and up at each node, i.e. each object composing the tree of the equation; the process stops either when the dimension asked by the user is found, or when all the nodes of the tree have been tested.

The methods enumerated here will be implemented differently for each object, but the scheme presented for the object 'integral' is the same for all objects. Notice that this scheme cannot solve all the situations. It is based on the assumption that each node can be solved locally. This is true for the most common situations, but the scheme fails when reasoning concepts at a global level become necessary.

4.2.5. The Graphical Environment for Dimensional Analysis

The object presented in the above sections can be visualized within the graphical environment presented and discussed by Eyheramendy and Zimmermann [82]. In the graphical environment of FEM_Theory, a push-button is added (see Fig. 15), which launches an editor that can be seen in Fig. 16. In this editor, the units contained in the dictionary,

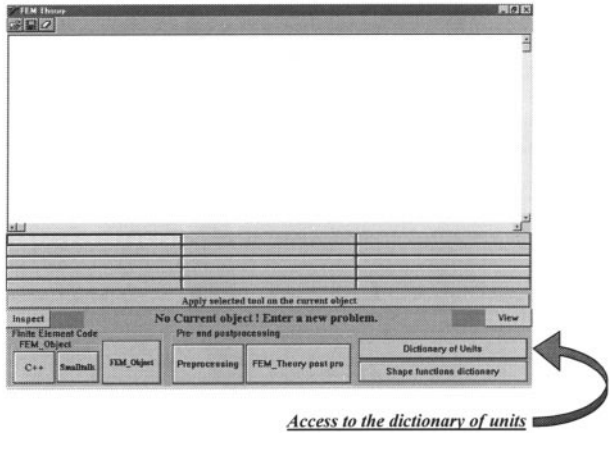

Fig. 15. Main window of FEM_Theory with the management of the dictionary of units.

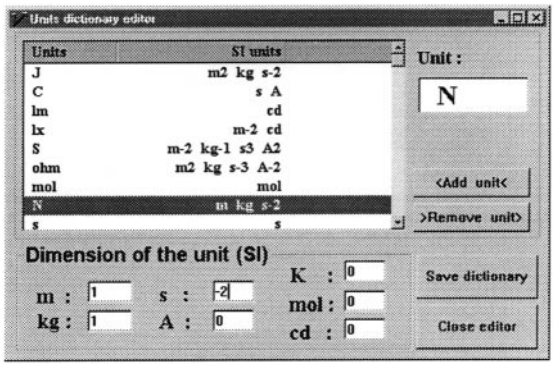

Fig. 16. Units dictionary editor.

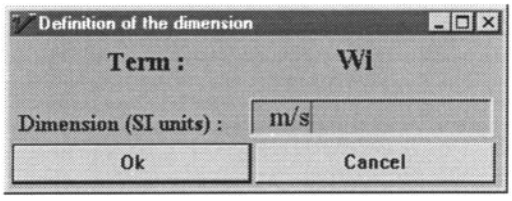

Fig. 17. Prompter to define a dimension.

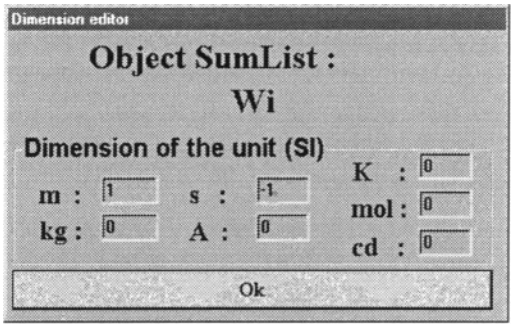

Fig. 18. Prompter to visualize a dimension.

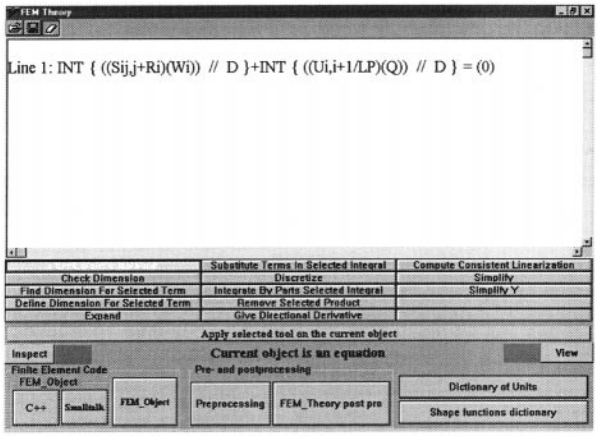

Fig. 19. Dimensional analysis of a penalty formulation for 2D Stokes problem.

instance of **FEMTheoryGeneralDictionaryOfUnits** and stored on disk in a file named 'fem.dct', can be viewed and new units can be added. The units are described in this editor by their name, and their dimension is given, e.g. Newton (symbol N) is highlighted and its dimension is $kg.m.s^{-2}$. During a derivation, the dimension of a term can be defined by the user; this is done by means of the editor in Fig. 17. The dimension of every object can also be visualized through the use of the prompter in Fig. 18. The list of tools for the instances of class **IntEquation** is enriched with new tools: '*Define Dimension For Selected Term*', '*Find Dimension For Selected Term*', '*Check Dimension*'.

The dimension can be defined in terms of any units, including derived units; the conversion of units is obvious in this framework.

4.2.6. A Simple Illustration of Dimensional Analysis in FEM_Theory

The goal of this section is to give a trivial example of the usefulness of a dimensional analysis scheme in the symbolic environment. Take the example of the penalty formulation for Stokes flow presented in Eyheramendy and Zimmermann [63, Section 3]. The formulation is posted onto the screen of FEM_Theory in Fig. 19, line 1. The problem and the notations are defined in Eyheramendy and Zimmermann [63, Section 3]. Let us define in the formulation on line 1 the dimensions of the terms that are obvious. This is done by selecting the term on screen line 1, and applying to it the new tool '*Define Dimension For Selected Term*'. This tool gives access to the prompter shown in Fig. 20. Here are the definitions of the following terms (the notation 'bracket' [X] means 'dimension of X'):

- the weighting velocity w_i: $[w_i] = m.s^{-1}$
- the pressure P: $[P] = N.m^{-2}$
- the weighting pressure Q: $[Q] = N.m^{-2}$
- the body loads (dimension given through the use of its expression, i.e. the product between the density and the acceleration of gravity) R: $[R] = [\rho] \cdot [g] = (kg.m^{-3}) \cdot (m.s^{-2})$.

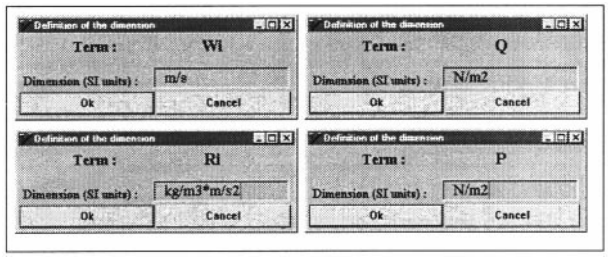

Fig. 20. Definition of the dimensions for selected terms.

The dimension of all the entities composing this equation are now defined, and their dimension can be retrieved through the tool '*Find Dimension For Selected Term*'. The result is posted in prompters, such as those in Fig. 21, e.g. the dimension of the term $1/\lambda$ is

$$\left[\frac{1}{\lambda}\right] = m.kg^{-1}.s$$

The dimension of the various objects of the equation are shown in Fig. 21.

4.2.7. Dimensional Control in Finite Elements

The dimensional analysis process has been applied here in the context of the symbolic development of finite elements for trivial purposes. This ensues from the wish to develop concepts for finite elements with a high level of abstraction in the finite element derivation. The next step would be to use all theoretical concepts developed and used during the symbolic derivation in the numerical computation. The control of the data introduced for a computation by the user is a crucial problem in numerical computation. The proposed approach could be extended to solving this problem. First, the tree structures proposed in the previous sections can be used in any context, i.e. not only in a symbolic environment. Any type of structure can be given a characteristic 'unit'. From there, the control of dimensions could easily be done, even during a numerical computation, through the use of a similar approach to that presented in the previous section. A second extension would be to pass information about dimensional analysis from the symbolic environment to the numerical one, in which it could be used to check dimensions.

4.3. Checking Index Writing Consistency

4.3.1. Goal

In FEM_Theory, the writing of the formulation is based on index notation. This notation is used for its general aspect, but mistakes in the notation are easy to make, and can have disastrous consequences on the discretization process. The idea is to introduce a checking process for consistency of the writing. This new tool does not need any new object, only an enhancement of the classes involved in the representation of the variational formulation of the continuum problem. The process is described next.

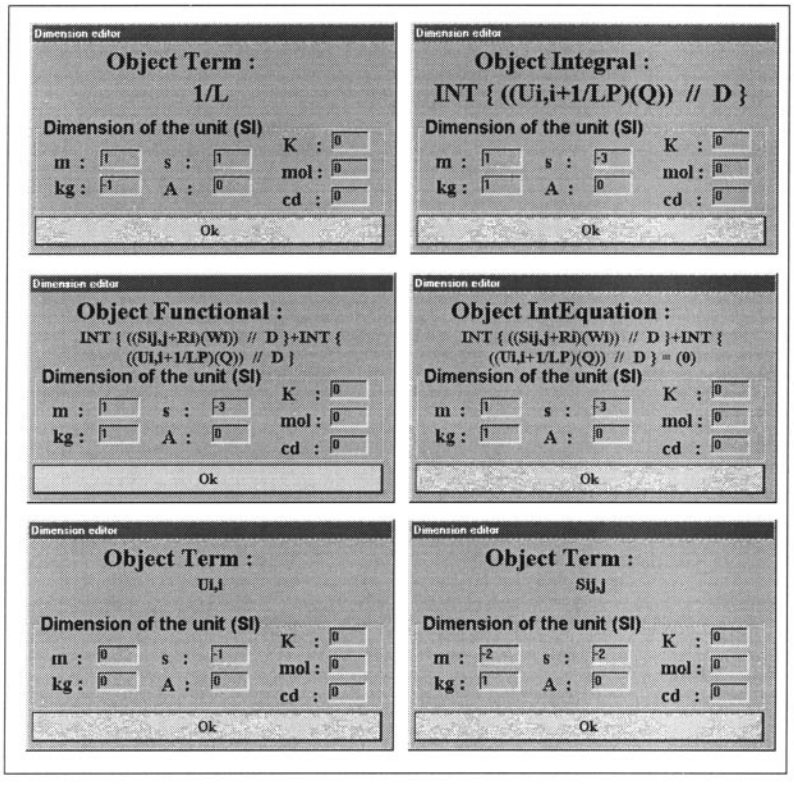

Fig. 21. Dimensional analysis of various objects.

```
checkIndicialNotation
   "Check if the indicial notation of the receiver is correct"
      | reply str |                                  Definition of the local variables

   reply := (self at:1) checkIndicialNotation.      Initialization of the string with reference to the first product
   self do:[:p|                                     Loop over the products composing the sum.
      str:= p checkIndicialNotation.                (a) Ask its products to check their notation (result stored in str)
      (reply isAnAnagramOf: str)                    (b) Check if the notation of the current product is the same as the
                                                        one of the reference
         ifFalse:[
         FEMTheoryMessage openOnMessage:            ... if not answer an error message
         ('Error in the index notation of :',(self printString)).
         ^nil
         ].
   ].

   ^reply                                           (c) Returns the string representing the indices of the sum,
                                                        which is the same as each product component
```

Fig. 22. Method *checkIndicialNotation* in class **SumList**.

4.3.2. Implementation of Writing Analysis

Contrary to the dimensional analysis process described in the previous section, the checking of the writing can be made at the local level, i.e. at the level of each object (see all the objects involved in the process in Fig. 24). Thus, each object is able to recover the contracted indices characterizing itself. The implementation ensues naturally. Each object has a method called *checkIndicialNotation*, which returns a string representing the indices of the receiver after contracting, e.g. the object $\sigma_{ij,j}$ returns the string '*i*' which is the contraction of the indices '*ijj*' (rules for classical index notation). For all the objects, the structure of the method is the same:

(a) ask the objects composing it to check their index notation (message *checkIndicialNotation*); they return a string representing the contracted indices;
(b) check the coherence of the indices at its level if necessary;
(c) return the string representing the indices (contracted).

Two examples of implementation of this method are given in Figs 22 and 23, for objects integral (class **Integral**) and sum (class **SumList**); they respect the three points given previously. The message for checking the notation goes down the tree, as illustrated in Fig. 24. The process ends when each node of the tree has made this check.

4.3.3. Example of Analysis

An illustration of the use of this scheme in FEM_Theory is shown in Fig. 25; on line 1, the penalty formulation for Stokes of the previous section [63] is posted. In the integral selected on line 1 (highlighted object on the screen), the prompter of Fig. 26 allows the replacement of the term $\sigma_{ij,j}$ by the expression $C_{ijkl}\varepsilon_{kl}(u)$, and instead of $C_{ijkl}\varepsilon_{kl,j}(u)$ as it should be. But an error is introduced in the prompter (the index 'j' is left out). Then, the prompter in Fig. 27 appears, indicating that the expressions introduced are not correct.

5. Conclusion

This paper illustrates the fast and natural extendibility capabilities of object-oriented symbolic environments for finite elements. With respect to existing symbolic finite element approaches, a new track to develop finite element software is proposed here. In Section 3, we showed that the addition to the environment of a new formulation, like the discontinuous space-time formulation, requires only

```
checkIndicialNotation
   "Check if the notation of the receiver is coherent"

   ^ self giveIntegrand checkIndicialNotation      (a) Asks its integrand to check its notation and (c) returns
                                                       the contracted indices.
                                                   Note that no part (b) for coherence control is needed here
```

Fig. 23. Method *checkIndicialNotation* in class **Integral**.

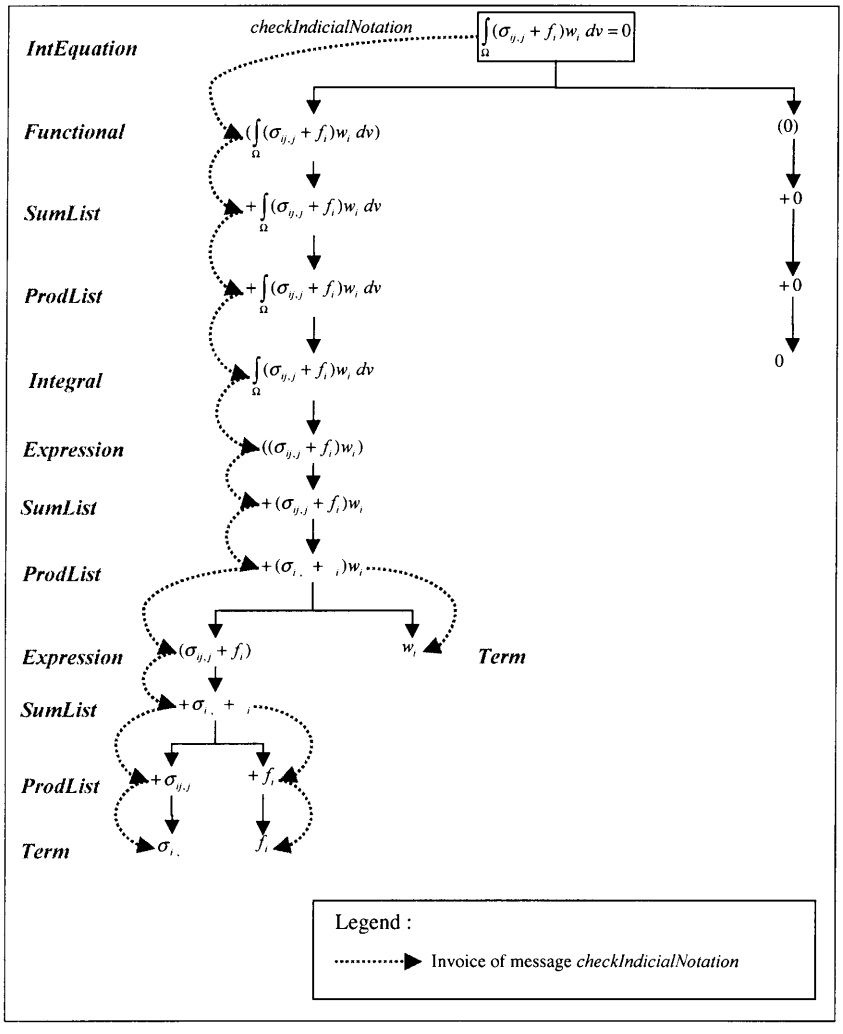

Fig. 24. Sketch for the checking of index notation.

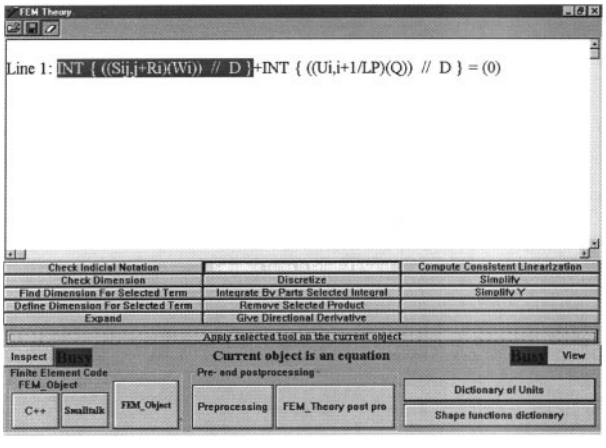

Fig. 25. Illustration of the writing consistency on a penalty formulation of Stokes' problem.

minor extensions; the only extensions needed are, first, the generalization of the scheme to handle spatial differential operators correctly, secondly, the introduction of new matrix forms for the spatial differential operators, and thirdly, the enrichment of the shape functions database. These changes are taken into account at a high level of abstraction, close to the mathematical formulation. Moreover, this extension, illustrated here on a one-dimensional advective equation, is also valid for any other kind of equation (e.g. Navier–Stokes). In Section 4, we showed that CASE tools can be integrated very easily into such an environment, to help the user in conducting his derivations.

In fact, a global computerized framework for finite element development has been created, based on a hybrid symbolic/numerical approach. The application of the object-oriented paradigm is considered

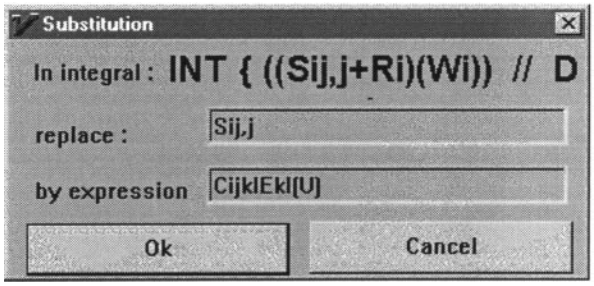

Fig. 26. Prompter for the replacement of an expression with a notation error.

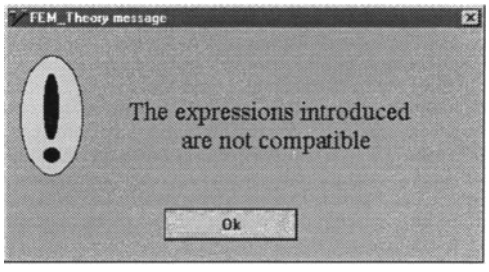

Fig. 27. Notification of the error in the notation.

crucial for this kind of approach, allowing a fast and simple introduction of high level concepts. The approach has already been tested on various mechanical problems including nonlinear ones: heat diffusion [84], linear elasticity in statics [88] and dynamics [81], a trivial beam [61,63], Stokes flow in the incompressible limit [63,82], and incompressible Navier–Stokes [30,83,86]. Various finite element formulations were used on these problems: like classical Galerkin formulations [61–63,81], Galerkin least-squares formulations [63,89], and Galerkin space-time formulations [30]. These developments demonstrate the broadness of the approach, capable of dealing with problems in solid, structural and fluid mechanics. The major drawback remains at present the relative lack of numerical efficiency of automatically generated code.

The ideas developed in this paper represent, in the authors' opinion, an important step towards a general environment for easy development of computerized solution schemes for mechanical problems. Nevertheless, the present environment is still limited, at this stage, to the introduction of finite element matrices. Extension to algorithmic descriptions of finite element formulations should allow the introduction of new solution schemes, e.g. new time integration schemes, strategies for updating variables, strategies for updating meshes or remeshing, constitutive modeling, etc. This is a particularly crucial point for nonlinear finite element analysis of, for example, coupled systems. Extensions to strategies such as parallelism, indeed an important ingredient, especially for high performance computations, would be natural. Mixing symbolic and numerical concepts opens new doors for the development of scientific software.

Acknowledgements

The financial support of the first author by the Swiss National Science Foundation under grant 20-45697.95 is acknowledged.

References

1. Fritzson, P; Fritzson, D. (1992) The need for high-level programming support in scientific computing applied to mechanical analysis. Computers & Structures, 45, 387–395
2. Luft, RW; Roesset, JM; Connor, JJ. (1971) Automatic generation of finite element matrices. J. Struct. Div., Proceedings of ASCE, January, 349–361
3. Choi, DK; Nomura, S. (1992) Application of symbolic computation to two-dimensional elasticity. Computers & Structures, 43, 645–649
4. Ioakimidis, NI. (1993) Elementary applications of MATHEMETICA to the solution of elasticity problems by the finite element method. Comput. Methods Appl. Mech. Engrg., 102, 29–40
5. Yang, CY. (1994) An algebraic-expressed finite element model for symbolic computation. Computers & Structures, 52(5), 1069–1077
6. Silvester, PP; Chamlian, SV. (1994) Symbolic generation of finite elements for skin-effect integral equations. IEEE Trans. Magnetics, 30(5), 3594–3597
7. Yagawa, G; Ye, GW; Yoshimura, S. (1990) A numerical integration scheme for finite element method based on symbolic manipulation. Int. J. Numer. Methods Engrg., 29, 1539–1549
8. Bardel, NS. (1989) The application of symbolic computing to the hierarchical finite element method. Int. J. Numer. Methods Engrg., 28, 1181–1204
9. Gunderson, RH; Cetiner, A. (1971) Element stiffness matrix generator. J. Struct. Div., Proceedings of ASCE, January, 363–375
10. Hoa, SV; Sankar, S. (1980) A computer program for automatic generation of stiffness and mass matrices in finite-element analysis. Computers & Structures, 11, 147–161
11. Cecchi, MM; Lami, C. (1977) Automatic generation of stiffness matrices for finite element analysis. Int. J. Numer. Methods Engrg., 11, 396–400
12. Barbier, C. (1992) Automatic generation of bending element matrices for finite element method using REDUCE. Engineering Computations, 9, 477–494
13. Korncoff, AR; Fenves, SJ. (1979) Symbolic generation of finite element stiffness matrices. Computers & Structures, 10, 119–124
14. Noor, AK; Andersen, CM. (1981) Computerized sym-

bolic manipulation in nonlinear finite element analysis. Computers & Structures, 13, 379–403
15. Cameron, F. (1997) Automatic generation of efficient routines for evaluating multivariate polynomials arising in finite element computations. Adv. in Engr. Soft., 28, 239–245
16. Leff, L; Yun, YY. (1991) The symbolic finite element analysis system. Computers & Structures, 41, 227–231
17. Wang, PS. (1986) FINGER: A symbolic system for automatic generation of numerical programs in finite element analysis. J. Symbolic Computation, 2, 305–316
18. Noor, AK; Andersen, CM. (1979) Computerized symbolic manipulation in structural mechanics-progress and potential. Computers & Structures, 10, 95–118
19. Noor, AK; Elishakoff, I; Hulbert, G. (1990) Symbolic Computations and their impact on mechanics. Winter Annual Meeting of the American Society of Mechanical Engineers, Dallas, Texas, November 25–30, PVP, 205
20. Breitkopf, P; Touzot, G. (1997) Architecture des logiciels et langages de modélisation. La Revue Européenne des éléments finis, 1(3), 333–368
21. Chambon, R; Thomas, JB. (1988) Langages pour le calcul des structures. Pluralis, 2, 261–271
22. Verpaux, P; Charras, T; Millard, A. (1988) CASTEM 2000: une approche moderne du calcul des structures. Pluralis, 2, 261–271
23. Collain, E; Fouet, JM; Regnier, G. (1988) Pour un calcul de structures orienté objets. Pluralis, 2, 371–390
24. Rehak, DR; Baugh Jr, JW. (1989) Alternative Programming Techniques for Finite Element Programming Development. Proceedings IABSE Colloquium on Expert Systems in Civil Engineerings, Bergamo, Italy, IABSE
25. Miller, GR. (1988) A LISP-based object-oriented approach to structural analysis. Engr. with Comp., 4, 197–203
26. Miller, GR. (1991) An object-oriented approach to structural analysis and design. Computers & Structures, 40(1), 75–82
27. Fenves, GL. (1990) Object-oriented programming for engineering software development. Engr. with Comp., 6, 1–15
28. Forde, BWR; Foschi, RO; Stiemer, SF. (1990) Object-oriented finite element analysis. Computers & Structures, 34, 355–374
29. Filho, JSRA; Devloo, PRB. (1991) Object-oriented programming in scientific computations: The beginning of a new area. Eng. Computations, 8, 81–87
30. Eyheramendy, D. (1997) Object-oriented finite element programming: Symbolic derivations and automatic programming. PhD thesis report 1752, Swiss Federal Institute of Technology
31. Lucas, D; Dressler, B; Aubry, D. (1992) Object-oriented finite element programming using the ADA language. In Hirsh, C. *et al*. (eds.), Numerical Methods in Engineering '92, 591–598
32. Dubois-Pèlerin, Y; Bomme, P; Zimmermann, Th. (1991) Object-oriented finite element programming concepts. Proceedings of European Conference on new Advances in Computational Structural Mechanics. Elsevier, pp 95–101
33. Baugh, JW; Rehak, DR. (1992) Data abstraction in engineering software development. Comp. Civ. Engr., 6, 282–299
34. Devloo, PRB; Magalhaes, CA; Noel, AT. (1992) On the implementation of the p-adaptive finite element method using the object oriented programming philosofy. Numerical Methods in Engineering and Applied Sciences, Part 1, CIMNE, Barcelona
35. Devloo, PRB. (1992) An object oriented approach to finite element programming (Phase I): a system independent windowing environment for developing interactive scientific programs. Advances in Engineering Software, 14, 41–46
36. Ross, JT; Morrow, JP; Wagner, LR; Luger, GF. (1992) Two paradigms for OOP models for scientific applications. Proceedings of 8th Conf. held in conjunction with AEC Systems 92, Dallas, TX, ASCE, pp 535–542
37. Ross, JT; Wagner, LR; Luger, GF. (1992) Object-oriented programming for scientific codes. I: Thoughts and concepts. Comp. Civ. Engr., 6, 480–496
38. Ross, JT; Wagner, LR; Luger, GF. (1992) Object-oriented programming for scientific codes. II: Examples in C++. Comp. Civ. Engr., 6, 480–496
39. Mackie, RI. (1992) Object-oriented programming of the finite element method. Int. J. Num. Meth. Engr., 35, 425–436
40. Scholz, SP. (1992) Elements of an object-oriented FEM ++ program in C++. Comp. and Struct., 43, 517–529
41. Nielsen, LO. (1994) A C++ class library for FEM special purpose software. Internal report, Department of Structural Engineering, Technical University of Denmark, Serie R vol. 308
42. Zeglinski, GW; Han, RPS. (1994) Object oriented matrix classes for use in a finite element code using C++. Int. J. Num. Meth. Engr., 37, 3921–3937
43. Drolet, J. (1996) Towards a cross-platform finite element application framework: A toll to simplify finite element simulations. Proceedings of 1st Structural Specialty Conference, Edmonton, Canada
44. Mackie, RI. (1997) Using objects to handle complexity in finite element software. Eng. with Computers, 13, 99–111
45. Zimmermann, Th; Dubois-Pèlerin, Y; Bomme, P. (1992) Object-oriented finite element programming: I. Governing principles. Comput. Methods Appl. Mech. Engrg., 98, 291–303
46. Dubois-Pèlerin, Y; Zimmermann, Th; Bomme, P. (1992) Object-oriented finite element programming: II. A prototype program in Smalltalk. Comput. Methods Appl. Mech. Engrg., 98, 361–397
47. Dubois-Pèlerin, Y; Zimmermann, Th. (1993) Object-oriented finite element programming: III. An efficient implementation in C++. Comput. Methods Appl. Mech. Engrg., 108, 165–183
48. Dubois-Pélerin, Y; Pegon, P. Object-Oriented programming in nonlinear finite element analysis. Computers & Structures (submitted)
49. Dubois-Pélerin, YD; Pegon, P. (1997) Improving modularity in object-oriented finite element programming. Commun. Numer. Methods. Engin., 13, 193–198
50. Menétrey, Ph; Zimmermann, Th. (1993) Object-oriented non-linear finite element analysis: application to J2 plasticity. Computers & Structures, 49(5), 767–777
51. Besson, J; Foerch, R. (1997) Large scale object-ori-

ented finite element code design. Comput. Methods Appl. Mech. Engrg., 142, 165–187
52. Foerch, R. (1996) Un environnement orienté objet pour la modélisation numérique des matériaux en calcul des structures. PhD thesis report, Ecole Nationale Supérieure des Mines de Paris
53. Angus, IG. (1992) Parallelism, object-oriented programming methods, portable software and C++. Proceedings of 8th Conf. held in Conjunction with AEC Systems 92, Dallas, TX, ASCE, pp. 506–513
54. Buffat, M; Yudiana, I; Leribault, C. (1992) Parallel simulation of turbulent compressible flows with unstructured domain partitioning. Performance on T3D and SP2 using OOP. In Schiano, A; Ecer, JP; Satofuka, N. Parallel computational fluid dynamics: Algorithm and results Advanced Computers, Elsevier, pp 76–83
55. Hsieh SH; Sotelino, ED. (1997) A message-passing class library C++ for portable parallel programming, Eng. with Computers, 13, 20–34
56. Potapov, S; Jacquart, G. (1997) Un algorithme ALE de dynamique rapide basé sur une approche mixte éléments finis-volumes finis. Actes du 3ième Colloque National en Calcul des Structures de Giens, Hermès, pp 509–514
57. Potapov, S. (1997) Un algorithme ALE de dynamique rapide basé sur une approche mixte Eléments finis-Volumes finis. Implémentation en langage orienté objet C++, PhD thesis report, Ecole Centrale Paris.
58. Walterthum, L. (1996) Programmation orientée objet et calcul par éléments finis. Application à la conception d'un logiciel de simulation en mise en forme des matériaux. PhD thesis report, Université de Franche-Comté
59. Gelin, JC; Walterthum, L. (1995) Conception d'un logiciel orienté-objets pour la simulation de processus de formage. Actes du 2nd Colloque national en calcul des structures, Giens, Hermès, pp 552–558
60. Kawata, H; Yoshimura, S; Yagawa, G; Kawai, H. (1995) Object-oriented system for evaluation of fracture mechanics-Parameters of linear and nonlinear 3D cracks. Proceedings of IECS 95, vol. 1, pp 39–44
61. Zimmermann, Th; Eyheramendy, D. (1996) Object-oriented finite elements: I. Principles of symbolic derivations and automatic programming. Comput. Methods Appl. Mech. Engrg., 132, 277–304
62. Eyheramendy, D; Zimmermann, Th. (1996) Object-oriented finite elements: II. A symbolic environment for automatic programming. Comput. Methods Appl. Mech. Engrg., 132, 259–276
63. Eyheramendy, D; Zimmermann, Th. (1998) Object-oriented finite elements: III. Theory and application of automatic programming. Comput. Methods Appl. Mech. Engrg, 154, 41–68
64. Codina, R; Schäfer, U; Oñate, E. (1994) Mould filling simulation using finite elements. Int. J. Num. Heat Fluid Flow, 4, 291–310
65. Sussman, M; Smereka, P; Osher, S. (1994) A level set approach for computing solutions to incompressible two-phase flow. J. Comp. Phy., 114, 146–159
66. Tezduyar, TE; Aliabadi, S; Behr, M. (1997) Enhanced-discretization interface-capturing technique. AHPCRC-University of Minnesota, Preprint 97-019
67. Hughes, TJR; Liu, WK; Zimmermann, Th. (1981) Lagragian–Eulerian finite element formulation for incompressible viscous flows. Comput. Methods Appl. Mech. Engrg., 29, 329–349
68. Huerta, A; Liu, WK. (1988) Viscous flow wih large free surface motion. Comput. Methods Appl. Mech. Engrg., 69, 277–324
69. Hughes, TJR; Hulbert, GM. (1988) Space-time finite element methods for elastodynamics: formulations and error estimates. Comput. Methods Appl. Mech. Engrg., 66, 339–363
70. Hansbo, P. (1992) The characteristic streamline diffusion method for convection-diffusion problems. Comput. Methods Appl. Mech. Engrg., 96, 239–253
71. Hansbo, P. (1992) The characteristic streamline diffusion method for the time-dependent incompressible Navier–Stokes equations. Comput. Methods Appl. Mech. Engrg., 96, 239–253
72. Tezduyar, TE; Behr, M; Liou, J. (1992) A new strategy for finite element computations involving moving boundaries and interfaces – The deforming-spatial-domain/space-time procedure: I. The concept and the preliminary numerical tests. Comput. Methods Appl. Mech. Engrg., 94, 339–351
73. Tezduyar, TE; Behr, M; Mittal, S; Liou, J. (1992) A new strategy for finite element computations involving moving boundaries and interfaces – The deforming-spatial-domain/space-time procedure: II. Computation of free-surface flows, two liquid flows, and flows with drifting cylinders. Comput. Methods Appl. Mech. Engrg., 94, 353–371
74. Behr, M; Tezduyar, TE. (1994) Finite element solution strategies for large-scale flow simulation. Comput. Methods Appl. Mech. Engrg., 112, 3–24
75. Masud, A; Hughes, TJR. (1997) A space-time Galerkin/Least-squares finite element of the Navier–Stokes equations for moving domain problems. Comput. Methods Appl. Mech. Engrg., 146, 91–126
76. Shakib, F. (1988) Finite Element analysis of the compressible Euler and Navier–Stokes equations. PhD thesis report, Stanford University
77. Hughes, TJR. (1987) The Finite Element Method, Prentice-Hall
78. VisualSmalltalk Enterprise – 32 Bit Pure Object-Oriented Programming System (1995) User's guide, ParkPlace Digitalk
79. VisualSmalltalk Enterprise – 32 Bit Pure Object-Oriented Programming System (1995) Language Reference, Parkplace Digitalk
80. VisualSmalltalk Enterprise – 32 Bit Pure Object-Oriented Programming System (1995) Encyclopedia of classes for Win32, ParkPlace Digitalk
81. Eyheramendy, D; Zimmermann, Th. (1996) Object-oriented finite element Programming: an interactive environment for symbolic derivations. Application to an Initial Boundary Value Problem. Advances in Engineering Software, 27, 3–10
82. Eyheramendy, D; Zimmermann, Th. (1998) Fonctionnalité d'un environnement orienté objet pour le développement de code éléments finis. La Revue Européenne des Elements Finis, 7, No 1–3
83. Eyheramendy, D; Zimmermann, Th. (to appear). Object-oriented finite elements: IV. Symbolic derivation and automatic programming of non-linear prob-

lems. Comput. Methods Appl. Mech. Engrg. (submitted)
84. Eyheramendy, D; Zimmermann, Th. (1995) Programmation orientée objet appliquée à la méthode des éléments finis: dérivations symboliques, programmation automatique. Revue Européenne des éléments finis, 4, 327–360
85. Eyheramendy, D; Zimmermann, Th. (1998) Intégration d'une approche variationnelle pour la méthode des éléments finis dans un environnement orienté objet: Application à un problème de convection non-linéaire. La Revue Européenne des éléments finis, 7, No 5
86. Zimmermann, Th; Eyheramendy, D. (1995) Symbolic object-oriented Finite Element programming – Application to incompressible viscous flow. Proceedings of IECS 95 Hawaii, vol. 1, pp 21–26
87. Zimmermann, Th; Eyheramendy, D; Bomme, P; Commend, S; Arruda, RS. (1997) Object-oriented finite element programming: Languages, Symbolic derivations, Reasoning capabilities. Proceedings of NAFEMS 97, Stuttgart, vol. 1, pp 652–663
88. Eyheramendy, D; Zimmermann, Th. (1997) Dérivations symboliques pour code éléments finis – Application à un problème d'élasticité. Actes du 3ième Colloque national en calcul des structures de Giens, vol. 2, pp 553–558
89. Zimmermann, Th; Eyheramendy, D; Bomme, P. Object-oriented finite element programming – From governing principles to automatic coding. In Onate, E; Idelsohn, SR (eds), CIMNE, (Proceedings of the Word Congress on Computational Mechanics, Buenos Aires, Argentina)