Object-oriented programming and multibody systems

Implementation of a dedicated finite element code

Valérie Kromer* — François Dufossé — Michel Gueury***

** ERIN-ESSTIN, Université Henri Poincaré, Nancy 1*
2, rue Jean Lamour
F-54519 Vandœuvre-lès-Nancy
{kromer, gueury}@esstin.uhp-nancy.fr

*** SA VALUTEC, Université de Valenciennes*
Le Mont Houy- C3T, BP 14
F-59314 Valenciennes cedex 9
Francois.Dufosse@univ-valenciennes.fr

ABSTRACT. *This paper contains a description of the design of a finite element program dedicated to multibody systems analysis that implements the concepts of object-oriented programming. The principal feature of the mechanical formalism used in this work is to provide a unified framework for both rigid and flexible bodies. We will show that the object-oriented programming greatly simplifies the implementation of other formalisms concerning polyarticulated systems, thus conferring high flexibility and adaptability to the developed software.*

RÉSUMÉ. *Cet article décrit l'architecture orientée objet d'un code de calcul par éléments finis dédié à l'analyse des systèmes multicorps. Le formalisme mécanique mis en œuvre est caractérisé par le traitement unifié des segments rigides et flexibles. Il est montré que la programmation orientée objet simplifie grandement l'introduction d'autres formalismes relatifs aux systèmes polyarticulés, conférant ainsi une flexibilité et une adaptabilité accrues au logiciel développé.*

KEYWORDS: *object-oriented programming, multibody systems, C++, finite element.*

MOTS-CLÉS : *programmation orientée objet, systèmes multicorps, C++, éléments finis.*

Revue européenne des éléments finis. Volume 12 – n° 7-8/2003, pages 933 to 962

1. Introduction

Object-oriented programming is currently seen as the most promising way of designing a new application. It leads to better structured codes and facilitates the development, the maintainability and the expandability of such codes. The object-oriented programming was proposed as a general methodology for finite element method implementation for the first time in (Miller, 1988). Over the past decade, it has been successfully applied to various domains of interest in finite element developments: constitutive law modeling (Besson *et al.*, 1998), parallel finite element applications (Breitkopf *et al.*, 1998), coupled problems (Klapka *et al.*, 1998), nonlinear analysis (Dubois-Pèlerin *et al.*, 1998), symbolic computation (Eyheramendy, 2000), and finite element analysis program architecture (Mackie, 1991, Scholz, 1992, Remy *et al.*, 1992, Besson *et al.*, 1998), among others. The recent article of Mackerle (Mackerle, 2000) gives a bibliography with more than 150 references covering the period 1996-1999.

However, little effort has been made to implement object-oriented programming in multibody systems analysis. As a matter of fact, like many other engineering applications, multibody systems analysis codes (ADAMS (Ryan, 1990), DADS (Nikravesh, 1982)) are written in Fortran. Therefore, the main objective of this work is to describe one approach to the design and implementation of a multibody systems analysis code using an object-oriented architecture.

The principal features of object-oriented programming are summarized in the second section of the paper, which also gives a state of the art of object-oriented programming in numerical methods. Is is shown that the structure of multibody systems present similarities with object-oriented concepts, and, consecutively lend themselves very well to object-oriented programming techniques. The architecture of the computational engine of the software is presented in the third part of the paper, with the global description of the computational engine and the description of the most important basic classes. Emphasis is placed on the adequacy between object-oriented programming and the finite element method within a given formalism and given hypotheses used for the treatment of multibody systems. It is shown that object-oriented programming greatly simplifies the choice and the implementation of other formalisms concerning polyarticulated systems, thus conferring high flexibility and adaptability to the developed software. An example of possible code extension is given to illustrate extensibility and reusability.

2. Adequation between the object-oriented programming and multibody systems analysis

2.1. *Object-oriented programming in scientific programming*

2.1.1. *Object-oriented concepts*

Procedural languages like C, Pascal and Fortran consider data as passive and memory occupying elements, which can be manipulated only by functions and procedures. In contrast, the object-oriented programming comes from the idea that tools (methods) must be associated with the information (data or attributes) they manage. The key concepts of object-oriented programming (abstraction, encapsulation, inheritance, polymorphism) can be found in many computer journals and language user guides. Thus we will only present briefly those concepts.

The *abstraction* of the data type is realized by the means of a *class*. A class incorporates the definition of the structure as well as the operations on the abstract data type. An *object* is an *instance* of the class, which means that it is the material realisation of the abstract data type defined by the class.

The *encapsulation* designates the independent self-containment of classes and their methods and attributes. The class members can be declared as *public*, *protected*, or *private*. Usually, for safety reasons, the attributes are declared as private and can be reached through public methods, defining the *interface* of the objects. This full modularity both greatly enhances introducing modifications in the development phase of the software, as well as performing on-going maintenance.

The *inheritance* concept is used to define object hierarchies. An object can have many children (instances of a *subclass*) which inherit its data and member functions. It can also have one or several parents (single or multiple inheritance). A subclass is usually of a special type and has additional methods and attributes members relating to it.

One of the most important characteristic of object-oriented languages is also the possibility of defining abstract objects using virtual member functions. This characteristic enables the same function to respond differently when performed on objects from different classes. Abstract objects allow the writing of generic algorithms and the easy extension of the existing code. This capability of object-oriented applications to interpret the same request differently depending on the object being processed is called *polymorphism*.

In addition to the previous object-oriented concepts, some other features can also greatly ease programming. For example, some object-oriented languages allow the *overloading* of functions and operators such as =, +, +=, <, etc. Moreover, they give the ability to use the *templates mechanism*, which allows the definition of the concepts (methods, algorithms) independently of the type of object being used. This mechanism is widely used, for instance, to manipulate generic arrays. Templates

greatly ease and improve the use of objects while requesting few lines of code. They not only increase legibility without losing efficiency, but also allow the methods to be reused without considering how data are implemented.

2.1.2. *Object-oriented languages*

Fortran 77, C and Fortran 90 are clearly not object-oriented languages, although it is sometimes possible to create structures that simulate the behavior of objects, by storing function pointers in structures. However, those languages have limitations, since, for example, they do not support inheritance or dynamic polymorphism.

The first object-oriented programming language, Smalltalk, was developed at the Xerox Palo Alto Research Center in the beginning of the 1980s (Goldberg *et al.*, 1983). Although it is considered as a pure object-oriented language, it is not used for real numerical applications because of efficiency reasons (Scholz, 1992). Other languages with object representations are available, such as Ada 95, Pascal or Eiffel. Java has recently been introduced (Arnold *et al.*, 1998) as a platform independent language. Despite its C++ like syntax, it supports neither multiple inheritance nor templates nor operator overloading. We selected C++, which seems to be, currently, the language providing numerical efficiency, portability, flexibility and is easy to use.

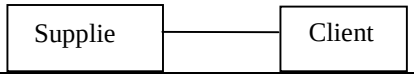
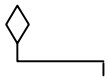
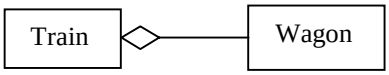
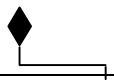
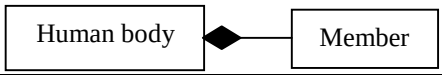
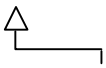
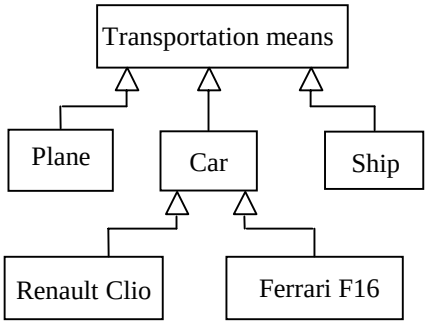
Relationship	Symbol	Example
Association		
Aggregation		
Composition		
Inheritance		

Table 1. *Association, aggregation, composition and inheritance relationships*

2.1.3. *Object-oriented methodologies*

As the complexity of the softwares increases, it becomes necessary to employ good analysis tools. Since the 1990s, new techniques in modeling and analysis have been brought by object-oriented programming. These are HOOD (Lai, 1991), BOOCH (Booch, 1993), OMT (Rumbaugh, 1991), among others. Today, UML (Unified Modeling Language) (Rumbaugh *et al.*, 1999) appears to be the technique which is most commonly used. UML fuses the concepts introduced by the previously mentioned techniques, which results in a standardized modeling language, with a set of symbols and rules describing the relationships between the symbols. As an example, Table 1 illustrates the association, aggregation, composition and inheritance relationships, which will be used in this article. UML is used in section 3 of this article.

2.2. *Object-oriented programming and multibody systems*

Little effort has been made to implement object-oriented programming in multibody systems analysis. However, if we examine the structure of multibody systems, we come to the conclusion that they present similarities with object-oriented concepts, and consecutively lend themselves very well to object-oriented programming techniques (Kunz, 1998, Tisell *et al.*, 2000).

For instance, at its highest level of abstraction, the architecture of a multibody system can be thought of as consisting of four basic objects: bodies, constraints between bodies, loads and motions.

Encapsulation is an ideal concept for multibody systems, as its implementation concentrates the attributes and methods associated with an object such that access is permitted only through well-defined interfaces. This full modularity both greatly enhances introducing modifications in the development phase of the software, as well as performing on-going maintenance.

One can illustrate the polymorphism concept by considering an arbitrary constraint connecting two (or more) arbitrary bodies. The implementation details of the specific bodies and joints involved in the connection are hidden by abstraction in the joint and body objects. Through polymorphism, it is possible to describe any constraint between two bodies without regard to joint type (revolute joint, translational joint, among others) and body type (rigid, flexible).

The application of object-oriented programming to multibody systems analysis is still a relatively unexplored area, although there has been some work on the topic. In 1993, Otter (Otter *et al.*, 1993) presented a Dymola class library for the symbolic generation of the equations of motion of rigid multibody systems in tree-structure. For closed-loops, it is necessary to use specific types of joints that “cut” the loops into a tree-structure. In 1998, Kunz (Kunz, 1998) described a multibody systems analysis computer program designed according to object-oriented principles, with

the purpose of integrating flexible and generic bodies into the architecture in a unified manner. This integration is performed by transforming the equations of motion into minimum coordinate set form using the velocity transformation. No indication is given on the treatment of the flexible bodies, which seem to have been always assumed to be linear elastic. For the treatment of closed-loops, the solution proposed is of the same kind as the method used by Otter: one of the joints in the loop is cut, thus reducing the closed-loop to an open-loop. It becomes necessary to distinguish dependent and independent coordinates resulting from the cut, in order to obtain equations of motion in the same form as the open-loop equations of motion. More recently, Tisell (Tisell *et al.*, 2000) presented a Mechamos prototype of a multibody system analysis tool, based on an object-relational database management system. The analysis is based on symbolic formulation of Kane's equations of motion, for multibody systems consisting of rigid bodies only. In this work, the authors insist on the fact that multibody systems analysis is only taken as an example of one activity among the whole range of engineering activities that takes place within the entire engineering information system which should support the complete engineering process during the entire life-cycle of a product. Indeed, the actual purpose of the authors is not the description of a multibody systems analysis tool based on object-oriented principles, but rather to show how database technology is a key technology for solving the problem of managing engineering data in an engineering information system environment.

The original work presented in this paper is aimed at describing the object-oriented architecture of a multibody systems analysis code, which is able to integrate flexible and rigid bodies in a unified framework, and to deal with both open-loop and closed-loop systems in a systematic way, without the need of an artificial cut of one of the joints. Moreover, the flexible bodies are not supposed to support only small displacements and small strains, but can undergo large rotations and large strains (Dufossé *et al.*, 2000, Dufossé, 2001). The computational model of the software has been developed in Borland C++ Builder. It consists of an arbitrary collection of rigid and flexible bodies, connected together by a variety of joints, loaded by concentrated or distributed forces and moments, that can be subjected to specified or constrained motions. As mentioned in section 2.1.2, the computational engine is written in C++ ANSI, independently of the computational model and the compiler.

In this paper, we will only focus on the computational engine of the software. Although the current architecture is based on given hypotheses and formalism, which will be presented in the next section, its originality lies within the fact that it has been conceived in order to provide a flexible and extensive set of objects that facilitate multibody systems analysis and which can be adapted to meet future developments.

3. Computational engine architecture

3.1. *Global description of the computational engine*

The global architecture is organized around several basic classes associated with the classical steps of a multibody finite element analysis (geometry definition, meshing, interactions definition, finite element formulation, behaviour law, joints definition, solving algorithms). The description of these classes is presented in the next sections. Since these are intended to be high-level descriptions of the code architecture, the included code fragments do not show all of the member variables and member functions that exist in the actual code.

The management of the different stages of the analysis is performed by the major class **Domain**, which represents the heart of the software architecture (Figure 1). It has been created to represent an elementary problem (corresponding to a specific multibody system analysis). The complete description of the class **Domain** is given in section 3.3.

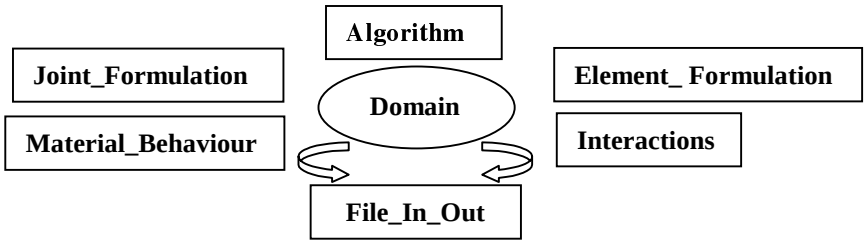


Figure 1. *Global architecture of the software*

3.2. *Description of the basic classes*

3.2.1. *Common tools*

Some utility classes have been developed to manage mathematical tools, variable size of objects, stacks, buffers, pools for memory management, read and write protection on objects and execution time measurement, among others (for example, classes **Vect**, **TabVect**, **Matrix**, **SMatrix**, **DMatrix**, **List**). The implementation of these classes is based on the templates mechanism.

3.2.2. *Finite element formulation*

3.2.2.1. General organization

The **Element** and **Node** classes are traditional classes used for the representation of finite elements. The class **Element_Formulation** (Figure 2) is the abstract master

class of the finite elements library. It provides virtual methods of computation of the different finite element arrays and tables (Figure 3) (as, for example, the classical mass and rigidity matrices, the strain increments vector, the non-linear acceleration vector, the internal forces vector). The effective computation depends on the chosen formalism and may vary significantly. In these conditions, the object-oriented programming eases the implementation of new formalisms, without requiring a complete redefinition of the software's architecture. In order to give a better understanding of the general organization, the formalism implemented in the current code is explained briefly in the next section.

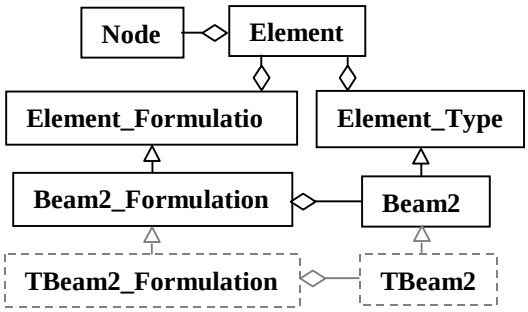


Figure 2. *Classes associated with the finite element modelization*

```
Class Element_Formulation
{
private :
    int label_ ;
    int nb_eq_ ;
    int nb_dof_ ;
    Element *el_ ;
    Domain *domain_ ;
public :
  //constructors
    Element_Formulation();
    Element_Formulation(Element *element, Domain *d) ;
    Element_Formulation(const Element_Formulation &ef) ;
  //virtual methods
    virtual Matrix<double> * Calculate_Mass_Matrix()= 0 ;
    virtual Matrix <double> * Calculate_Strain_Operator()= 0 ;
    virtual Vect<double> * Calculate_Strain_Increments()= 0 ;
    virtual Vect<double> * Calculate_NL_Acceleration()= 0 ;
} ;
```

Figure 3. Element_Formulation *class definition*

The abstract class **Element_Type** contains virtual methods for the calculation of the element's shape function and its derivatives. It possesses an attribute which is a pointer to an **Element** object. From **Element_Type**, subclasses corresponding to a specific finite element are derived (for example, **Beam2** for a two-nodes beam element).

The class **Beam2_Formulation** is derived from **Element_Formulation.** One of its attributes is a pointer to the desired finite element, **Beam2**, for example. This class corresponds to a specific finite element formulation, and allows the redefinition of the methods according to the type of the element. For instance, as will be shown in the next section, the formalism implemented in the present code corresponds to spatial beam elements and requires the calculation of curvatures and curvature increments of the beams. As a consequence, besides the methods already defined in its mother class **Element_Formulation,** the class **Beam2_Formulation** also contains specific methods such as *Calculate_Curvatures()*, *Calculate_Curvatures_Increments()* (Figure 4).

```
Class Beam2_Formulation : virtual public Element_Formulation
{
private :
    Beam2 *beam_ ;
public :
  //constructors
    Beam2_Formulation();
    Beam2_Formulation(Element *element, Domain *d) ;
    Beam2_Formulation(const Beam2_Formulation &bf) ;
  //destructor
    ~Beam2_Formulation() ;
  //definition of the virtual methods
    Matrix<double> * Calculate_Mass_Matrix()= 0 ;
    Matrix<double> * Calculate_Strain_Operator()= 0 ;
    Vect<double> * Calculate_Strain_Increments()= 0 ;
    Vect<double> * Calculate_NL_Acceleration()= 0 ;

  //specific methods
    Beam2 *Type_Element() {return beam_ ;}
    Vect<double> * Calculate_Curvatures()= 0 ;
    Vect<double> * Calculate_Membrane_Strain_Increments()= 0
    Vect<double> * Calculate_Curvature_Strain_Increments()= 0 ;
    Vect<double> * Calculate_Curvature_Increments()= 0 ;
} ;
```

Figure 4. Beam2_Formulation *class definition*

3.2.2.2. Chosen formalism

The formalism implemented in the current code is based on specific kinematic assumptions, for the description of the dynamics of a flexible beam. Two reference frames are used, an inertial reference frame (I) for the description of the translational motion and a body-fixed frame (B) attached to the cross-section for the rotary motion (Park *et al.*, 1991, Downer *et al.*, 1992).

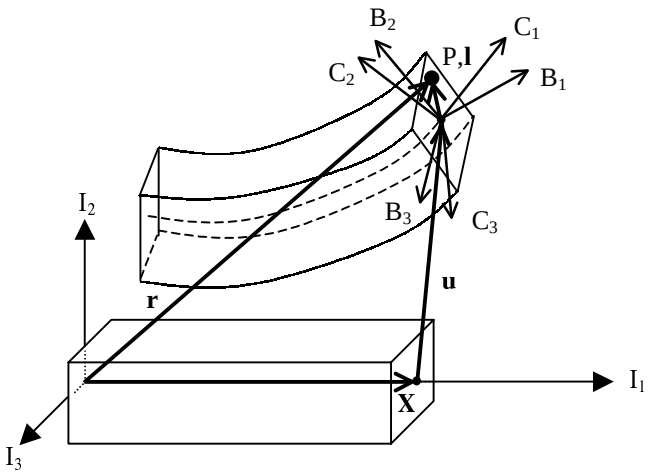


Figure 5. *Spatial beam kinematics*

The location from the inertial origin of an arbitrary point P on a beam (Figure 5) is represented by the following position vector:

$$\{r\} = \{X^I\} + \{u^I\} + \{l^B\} \qquad [1]$$

where $\{X\}$ is the position vector of a point of the original neutral axis, $\{u\}$ is the total translational displacement vector of the neutral axis, $\{l\}$ is a vector connecting the beam neutral axis to the material point P located on the deformed beam cross-section. The notation $\_^I$ or $\_^B$ in [1] indicates that the quantity is expressed with respect to the frame (I) or (B), respectively.

The velocity, acceleration and virtual displacement vector of the material point P are given by:

$$\{\dot{r}\} = \{\dot{u}^I\} + [\tilde{\omega}]^B \{l^B\}$$

$$\{\ddot{r}\} = \{\ddot{u}^I\} + [\tilde{\dot{\omega}}]^B\{l^B\} + [\tilde{\omega}]^B[\tilde{\omega}]^B\{l^B\}$$

$$\{\delta r\} = \{\delta u^I\} + [\delta\tilde{\alpha}]^B\{l^B\}$$

where $[\tilde{\omega}]$ is the skew-symmetric angular velocity tensor and $[\delta\tilde{\alpha}]$ the skew-symmetric virtual rotational tensor of the body-fixed basis, respectively.

The motion due to rigid motion is not distinguished from that due to the deformations. Moreover, the translational inertia is completely decoupled from the rotary inertia. The advantage to this is that the beam inertia is identical in form to that of rigid body dynamics. As a consequence, the same formalism can be used for mechanisms containing rigid elements as well as deformable elements.

The principle of virtual work for a given set of particles in a continuum occupying a domain Ω with a surface $\partial\Omega$ is stated as:

$$\int_\Omega \delta r_i \rho \ddot{r}_i dV + \int_\Omega \sigma_{ij}^I \frac{\partial \delta r_i}{\partial x_j} dV = \int_\Omega \delta r_i f_i dV + \int_{\partial\Omega} \delta r_i t_i dS \qquad [2]$$

where x_i represent the inertial coordinates of a particle, δr_i a kinematically admissible virtual displacement, $\ddot{r}_i$ the acceleration, f_i the external force per unit mass, t_i the stress vector acting on a surface with outward normal components n_i, σ_{ij}^I the inertial components of the Cauchy stress tensor, and ρ the mass density.

The key point to the current formulation is the calculation of the internal force operator, due to member flexibility. It is defined by:

$$\delta\Im_I = \int_\Omega \sigma_{ij}^I \frac{\partial \delta r_i}{\partial x_j} dV$$

In order to provide conceptual and computational simplifications for the stress representation, we use the technique proposed by Downer (Downer *et al.*, 1992): a convected frame (C) is introduced as a reference for the Cauchy stress and the conjugated strain. This frame is constant on the element level, one of its axes lies tangent to the deformed beam neutral axis (Figure 5). This frame does not coincide with the body frame (B), the relative difference between (B) and (C) models the effects of torsion deformations and transverse shear. Let (ξ, ψ, ζ) denote the coordinates of the convected reference frame (C). The internal force operator becomes:

$$\delta\Im_I = \int_\Omega \sigma_{ij}^C \delta\varepsilon_{ij}^C dV = \int_\Omega \left\langle \sigma_{\xi\xi}, 2\sigma_{\xi\psi}, 2\sigma_{\xi\zeta} \right\rangle \begin{Bmatrix} \delta\varepsilon_{\xi\xi} \\ \delta\varepsilon_{\xi\psi} \\ \delta\varepsilon_{\xi\zeta} \end{Bmatrix} dV$$

The vector of virtual strain can be rewritten as:

$$\begin{Bmatrix} \delta\varepsilon_{\xi\xi} \\ 2\delta\delta_{\xi\psi} \\ 2\delta\delta_{\xi\zeta} \end{Bmatrix} = \{\delta\gamma\} + \left[\tilde{I}^C\right]^T \{\delta\kappa\}$$

where $\{\delta\gamma\}$ represents the membrane and two transverse shear strains, and $\{\delta\kappa\}$ represent the torsion and two bending strains.

By integrating over the area coordinates of a symmetric cross-section, the virtual work of the internal forces is:

$$\delta\Im_I = \int_\xi \left(\langle\delta\gamma\rangle \{N_\gamma\} + \langle\delta\kappa\rangle \{M_\kappa\} \right) d\xi = \int_\xi \langle \delta u, \delta\alpha \rangle [L]^T \begin{Bmatrix} N_\gamma \\ M_\kappa \end{Bmatrix} d\xi$$

In the above expression, $\{N_\gamma\}$ represents the axial and transverse shear forces per unit length, $\{M_\kappa\}$ represents the torsion and bending moments per unit length:

$$\{N_\gamma\} = \int_S \{\sigma\} dS \qquad \{M_\kappa\} = \int_S \left[\tilde{I}^C\right]^T \{\sigma\} dS$$

$[L]$ is the strain operator such as:

$$\begin{Bmatrix} \delta\gamma \\ \delta\kappa \end{Bmatrix} = [L] \begin{Bmatrix} \delta u \\ \delta\alpha \end{Bmatrix}_2$$

For the finite element discretization, the displacement field along the beam is approximated by:

$$\{u\} = \sum_{i=1}^{n} N_i \{u_i\}$$

where n is the number of nodes per element, $\{u_i\}$ is the vector of the degrees of freedom at the element nodes, and N_i are the linear shape functions.

By the same way, the virtual rotations $\{\delta\alpha\}$, the angular velocities $\{\delta\omega\}$ and the angular accelerations $\{\delta\dot{\omega}\}$ are approximated as follows:

$$\{\delta\alpha\}=\sum_{i=1}^{n} N_i \{\delta\alpha_i\} \quad \{\delta\omega\}=\sum_{i=1}^{n} N_i \{\delta\omega_i\} \qquad \{\delta\dot{\omega}\}=\sum_{i=1}^{n} N_i \{\delta\dot{\omega}_i\}$$

These approximations applied to the variational form [2] lead to the final discrete equations of motion of a flexible beam element:

$$\begin{bmatrix} m & 0 \\ 0 & J \end{bmatrix} \begin{Bmatrix} \ddot{u} \\ \dot{\omega} \end{Bmatrix} + \begin{Bmatrix} 0 \\ D \end{Bmatrix} + \begin{Bmatrix} S^I \\ S^B \end{Bmatrix} = \begin{Bmatrix} F^I \\ F^B \end{Bmatrix} \qquad [3]$$

where [m] and [J] represent the mass and inertia matrices; $\{\ddot{u}\}$ and $\{\dot{\omega}\}$ represent the nodal accelerations vectors; $\{D\}$ represents the non-linear acceleration, $\{S^{I,B}\}$ and $\{F^{I,B}\}$ represent the internal and external force vectors partitioned into translational and rotational parts, respectively. These equations can be specialized to the case of static equilibrium as $\{S\}=\{F\}$. The equations of motion [3] can also represent a rigid body by setting the internal force vector $\{S\}$ to zero.

As one can see, the unconstrained equations of an arbitrary configuration of flexible beams and rigid bodies are written in terms of one set of kinematical coordinates denoting both the nodal coordinates of the flexible members and the physical coordinates of the rigid bodies.

The formalism presented above has been implemented in the current software. However, it is possible to implement other formalisms without requiring a complete redefinition of the software's architecture.

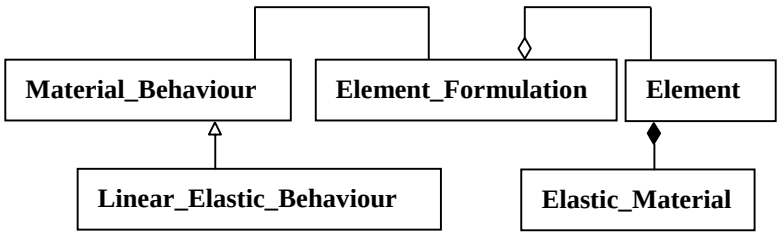


Figure 6. *Classes associated with material behaviour*

3.2.3. *Behaviour law*

In most object-oriented calculation software, the material behaviour is included in the finite element formulation. In our code architecture, a separated class **Material_Behaviour** has been created which contains the virtual methods for the calculation of the stresses according to a given material behaviour law (Figure 6).

The current formalism introduces a rate-type constitutive law that relates the instantaneous rate of stress to the instantaneous rate of deformation, according to:

$$\dot{\sigma}_{ij}^{C} = C_{ijkl}\dot{\varepsilon}_{kl}^{C}$$

where $\dot{\sigma}_{ij}^{C}$ and $\dot{\varepsilon}_{ij}^{C}$ are the convected frame stress and strain rates, respectively, C_{ijkl} is the material response tensor.

This law is obtained from the objective Truesdell rate equation, transformed from the inertial basis to the convected basis. The stress update procedure is obtained from the time integration of the rate equation:

$$^{n+1}\sigma_{ij}^{C} = {}^{n}\sigma_{ij}^{C} + \int_{t^n}^{t^{n+1}} C_{ijkl}\dot{\varepsilon}_{kl}^{C}\, dt = {}^{n}\sigma_{ij}^{C} + C_{ijkl}\Delta\varepsilon_{kl}^{C}$$

The resultant forces can be obtained by a simple additive procedure:

$$^{n+1}\begin{Bmatrix} N_\gamma \\ M_\kappa \end{Bmatrix} = {}^{n}\begin{Bmatrix} N_\gamma \\ M_\kappa \end{Bmatrix} + \begin{Bmatrix} \Delta N_\gamma \\ \Delta M_\kappa \end{Bmatrix}$$

For a linear elastic material, the resultant stress increments are deduced from:

$$\{\Delta N_\gamma\} = \begin{bmatrix} ES & 0 & 0 \\ 0 & GS & 0 \\ 0 & 0 & GS \end{bmatrix}\{\Delta\gamma\} \qquad \{\Delta M_\kappa\} = \begin{bmatrix} GJ & 0 & 0 \\ 0 & EI_2 & 0 \\ 0 & 0 & EI_3 \end{bmatrix}\{\Delta\kappa\}$$

where E and G are the Young and Coulomb modulus, S the cross-sectional area, J the torsional inertia, I_2 and I_3 the inertia momentum.

The methods for a linear elastic material (*Calculate_Tangent_Operator()*, *Calculate_Stress_Increments()*, *Calculate_Resultant_Stress_Increments()*) are defined in the class **Linear_Elastic_Behaviour** derived from **Material_Behaviour** (Figure 7, Figure 8).

```
Class Material_Behaviour
{
private :
    int code_ ;
    double time_ ;
    Element *el_ ;
    Domain *domain_ ;
    Element_Formulation *f_ ;
public :  //examples of specific methods
  //constructors
    Material_Behaviour(Element *element, Element_Formulation *f,
    Domaine *d);
  //virtual methods
    virtual Matrix<double> * Calculate_Tangent_Operator()= 0 ;
    virtual Vect<double> * Calculate_Stress_Increments()= 0 ;
    virtual Vect<double> * Calculate_Resultant_Stress_Increments()= 0 ;
} ;
```

Figure 7. Material_Behaviour *class definition*

```
Vect<double>* Linear_Elastic_Behaviour::
Calculate_Resultant_Stress_Increments ()
{
    Vect<double> *inc_N_M;
    inc_N_M=new Vect<double>(6);
    Matrix<double> *C;
    C=this->Calculate_Tangent_Operator();
    Vect<double> *strain_inc;
    strain_inc=f_-> Calculate_Strain_Increments ();
    (*inc_N_M)=(*C)*(*strain_inc);
    delete C; delete strain_inc;
    return inc_N_M;
};
```

Figure 8. *Example of a method of the class* **Linear_Elastic_Behaviour**

3.2.4. *Parametrization of the rotations*

Various methods have been proposed for the parametrization of large rotations (Euler angles, Bryant angles, rotational vector, Rodrigues parameters, Euler

parameters). An overview of these methods can be found in (Cardona *et al.*, 1988) (Cardona, 1989).

An adequate representation of the finite rotations is required for the modelization of 3D multibody systems, as it is used in several steps of the formalism (calculation of the internal forces, updating of the rotational orientation, among others). Details concerning those different steps associated with the corresponding treatment of the finite rotations can be found in (Downer, 1990).

```
Class Rotation
{
  private :
    int nb_param_ ;
    Matrix<double> *R_ ;
    Vect<double> *param_ ;
  public :
    virtual Matrix<double> * GiveMatrRotFrom(Vect<double> &q)=0 ;
    virtual Vect<double> * GiveVectParamRotFrom(Matrix<double> &R)=0 ;
} ;
```

Figure 9. Rotation *class definition*

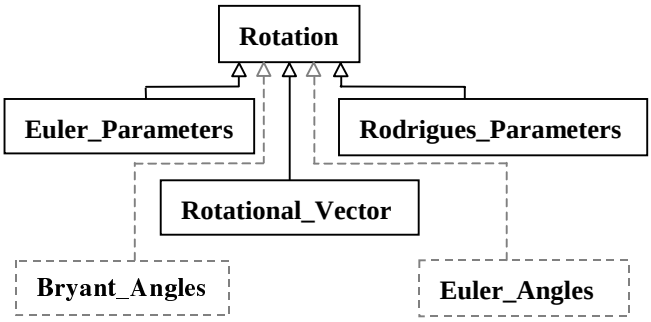


Figure 10. *Derived classes from the* **Rotation** *virtual class*

The abstract class **Rotation** has been developed in order to manage the information for the different parametrizations of the finite rotations. It contains several virtual methods, as for example the method *GiveMatrRotFrom()*, which calculates the rotation matrix from the parametrization system, or the method *GiveVectParamRotFrom()*, which gives the rotation parameters from the rotation matrix (Figure 9). For instance, given any finite rotation uniquely represented with a rotation angle θ and a rotation axis $\{n\}$ (Euler-Chasles theorem), the rotation matrix describing the rotation can be expressed as a function of the Euler parameters as:

$$[R] = [I] + 2q_0[\tilde{q}]^T + 2[\tilde{q}][\tilde{q}] \qquad [4]$$

where the four Euler parameters are defined by:

$$q_0 = \cos\frac{\theta}{2} \qquad \{q\} = \sin\frac{\theta}{2}\{n\}$$

and are subject to the constraint equation:

$$q_0^2 + \langle q \rangle \{q\} = 1$$

The classes corresponding to a specific parametrization of the rotations are derived from the **Rotation** base class (Figure 10). For instance, in addition to the method *GiveMatrRotFrom(),* which has been illustrated in the case of the Euler parameters by equation [4], the **Euler_Parameters** class contains a method for the calculation of an average rotation matrix and a method defining the multiplication of two rotation matrices with the quaternions law.

3.2.5. *Joint formulation*

As the **Element** class, the **Joint** class has been created for the modelization of the joints between bodies (Figure 11).

```
Class Joint
{
  private :
    Node **node_;
    int nb_nodes_,label_,code_ ;
    Matrix<double> *B_ ; //constraint Jacobian matrix
  public :  //examples of specific methods
    int NodeLabel(int n) {return node_[n-1]->Label() ;}
    Node *PtrNode(int i) {return node_[i-1] ;}
    ….
} ;
```

Figure 11. Joint *class definition*

The abstract class **Joint_Formulation** allows the treatment of the joints according to different formalisms (Figure 12). Currently, the only available class is the derived class **Jacobian**, for the calculation of the constraint Jacobian matrices required in the Lagrange multipliers technique that we have chosen to use to couple the algebraic constraint equations with the differential equations of motion of the

assembled mechanism (Cardona *et al.*, 1991) (Ibrahimbegovic *et al.*, 2000). The incorporation of the constraints via the Lagrange multiplier technique is straightforward, as the inertially-based degrees of freedom of the beam components, which embody both the rigid and deformation motions, are kinematically of the same sense as the physical coordinates of rigid body components.

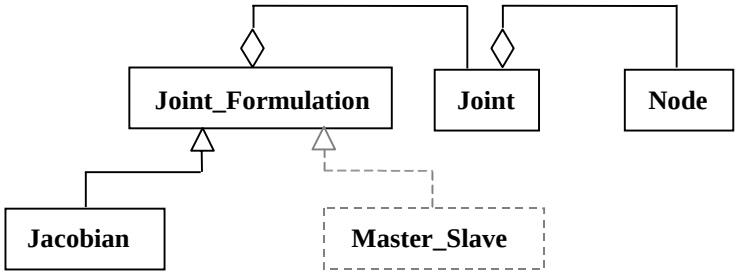


Figure 12. *Classes associated with constraints modelization*

Two types of constraint conditions exist: holonomic or configuration constraints and nonholonomic or motion constraints.

Holonomic constraints are formulated as implicit functions of the displacement coordinates and eventually of time. They express a restriction on the number of degrees of freedom and therefore, on the set of possible configurations of the system. A set of algebraic equations representing holonomic constraint conditions between the displacement coordinates are written as:

$$\left\{\Theta^{H}\left(\{u\},t\right)\right\}=\{0\}$$

The differential of these constraints is given as:

$$\left\{\delta\Theta^{H}\right\}=\left[\frac{\partial\Theta^{H}}{\partial u}\right]\{\delta u\}=\left[B^{H}\right]\{\delta u\}=\{0\} \qquad [5]$$

Nonholomic constraints conditions can only be expressed as a relation between the differentials of the coordinates and not as a finite relation between the coordinates themselves. Constraints of this kind are such that they cannot be integrated to be transformed to the holonomic type. They bring a restriction on the behaviour of the system, but not on the set of possible configurations. A set of non-integrable equations concerning nonholonomic constraints between the virtual displacements and rotations are written as:

$$\left\{\delta\Theta^{NH}\left(\{u\},\{\delta u\},[R],\{\delta\alpha\}\right)\right\}=\left[B^{NH}\right]\begin{Bmatrix}\delta u\\ \delta\alpha\end{Bmatrix}=\{0\} \qquad [6]$$

Another distinction is made between constraints involving equalities, which are called bilateral constraints, and constraints involving inequalities, named unilateral constraints. Holonomic constraints are only of the bilateral type. Unilateral constraints are usually considered of the nonholonomic type.

Given the constraints [5] and [6], the virtual work expression [2] for the unconstrained system is modified by the inclusion of the virtual work which enforces the constraints through the Lagrange multiplier technique as:

$$\int_{\Omega} \delta r_i \rho \ddot{r}_i dV + \int_{\Omega} \sigma_{ij}^{I} \frac{\partial \delta r_i}{\partial x_j} dV + \lambda_i^{H} \delta \Theta_i^{H} + \lambda_i^{NH} \delta \Theta_i^{NH} = \int_{\Omega} \delta r_i f_i dV + \int_{\partial\Omega} \delta r_i t_i dS$$

The equations of motion for constrained flexible multibody systems are written as follows:

$$\begin{bmatrix} m & 0 \\ 0 & J \end{bmatrix} \begin{Bmatrix} \ddot{u} \\ \dot{\omega} \end{Bmatrix} + [B]^T \{\lambda\} = \begin{Bmatrix} Q^u \\ Q^\omega \end{Bmatrix} \qquad [7]$$

In order to alleviate the equations, the same notation has been used (in [7] or in [3]) to represent elementary or assembled matrices and vectors. The right-hand side vector of [7] contains the remaining force-type terms as:

$$\{Q\} = \begin{Bmatrix} Q^u \\ Q^\omega \end{Bmatrix} = \begin{Bmatrix} F^I \\ F^B \end{Bmatrix} - \begin{Bmatrix} S^I \\ S^B \end{Bmatrix} - \begin{Bmatrix} 0 \\ D \end{Bmatrix} \qquad [8]$$

In equation [7], the notation contains both the holonomic and non-holonomic constraints in the constraint force vector $[B]^T \{\lambda\}$ as:

$$[B] = \begin{bmatrix} B^H \\ B^{NH} \end{bmatrix} \qquad \{\lambda\} = \begin{Bmatrix} \lambda^H \\ \lambda^{NH} \end{Bmatrix}$$

The [B] matrix is called the constraint Jacobian matrix. It is deduced from the kinematic relationship between the bodies of the system. These relationships can be imposed on the nodal degrees of freedom of two separated flexible beams, or between a beam nodal degree of freedom and a rigid body. Classical Jacobian matrices modeling standard joints (universal, revolute, spherical, translational joints) are described in (Chiou, 1990). Specific joints (rigid or flexible wheel element) describing contact conditions (with bi- or unilateral contact, with or without friction and slipping) are described by Cardona (Cardona, 1989). Thus, given the Jacobian matrices for the joints and beam connections, the equations [7] can be employed in a

systematic manner in order to represent an arbitrary assemblage of articulated flexible and rigid components.

The constraint Jacobian matrices calculated by methods of the class **Jacobian** are stored as attributes of the class **Joint** (Figure 9). This organization is not the only, or perhaps not even the best, organization for classes associated with the constraints modelization. For example, one could argue that the calculation of the Jacobian matrices could be instead performed directly inside the **Joint** class, and that there is no specific need for the abstract class **Joint_Formulation.** Nevertheless, in our sense, the current organization favors the modularization and the understanding of the software. Also, although not currently implemented, other procedures of treatment for the constraint equations (Master/Slave methods, among others) can be introduced in a straighforward way, by a simple derivation of **Joint_Formulation**.

3.2.6. *External interactions*

The word « interaction » designates a relationship of the studied domain with the outside, *i.e*, a boundary condition or an applied load. We define an interaction at the element level, by specifying the node where the interaction is applied (in the case of a boundary condition or a punctual action), the direction of the interaction and its time evolution.

The class **Time_Function** has been created to manipulate any function depending on the time. At a given time, the value of the function is given by the method *Calculate_Function().* In order to represent any interaction, the class **Interaction** has been implemented (Figure 13). At a given time, the method *Calculate_Force()* calculates the components of the vectors of forces and moments applied to the element. The applied loads and the boundary conditions are stored as attributes of the class **Domain**, by the use of the template class **List** (Figure 14).

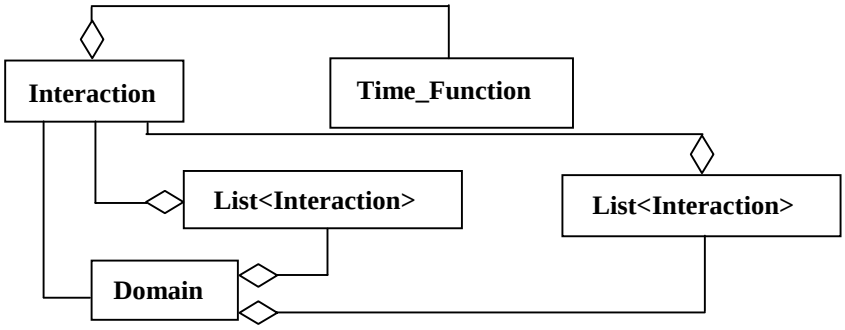


Figure 13. *Classes associated with the interactions (boundary conditions and loading)*

```
List<Interaction> *boundary_conditions_list_ ;
List<Interaction> *loading_list_ ;
//return load number i
Interaction *PtrLoad(int i)
{return (Interaction *)loading_list_->Find(i) ;}
//return boundary condition number i
Interaction *PtrBound(int i)
{return (Interaction *)boundary_conditions_list_->Find(i) ;}
```

Figure 14. *Use of* **Interaction** *instances in the class* **Domain**

3.2.7. *Computation procedure*

The different algorithms used for the resolution of multibody systems equations [19] are defined in classes derived from the abstract class **Algorithm** (Figure 15).

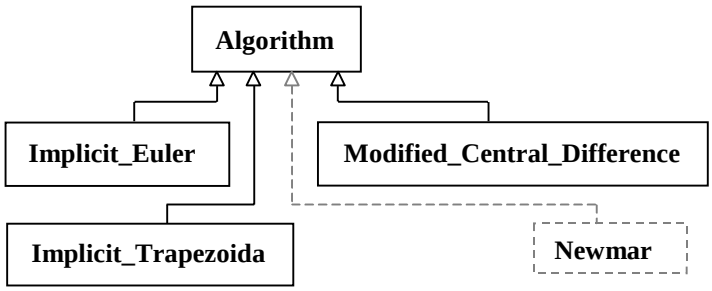


Figure 15. *Abstract class* **Algorithm**

In order to introduce an attractive modularity in the solution procedures, which favors the object-oriented programming, the integration of the generalized coordinates (translational coordinates $\{u\}$ and angular velocities $\{\omega\}$) is performed separately from the calculation of the constraint forces (Lagrange multipliers $\{\lambda\}$) in the current computational engine (Park *et al.*, 1988, Park *et al.*, 1990), which also includes a proper treatment of three-dimensional finite rotations (Ibrahimbegovic, 1997), as well as a method to satisfy the kinematic constraints conditions.

Some indications concerning the chosen techniques are given below. Nevertheless, as the class **Algorithm** already contains methods and attributes which are common to the different algorithms, the implementation of new algorithms can be achieved easily.

3.2.7.1. Integration of the generalized coordinates

The generalized coordinates are calculated with a particular explicit integration procedure, called « two-stage staggered algorithm », first developed by Park (Park *et al.*, 1990),. The algorithm is based on an interlaced application of the central difference algorithm such that the generalized coordinates are advanced one-half time step at a time, as follows:

$$^{n+1/2}\{\dot{u}\} = {}^{n-1/2}\{\dot{u}\} + h^n\{\ddot{u}\}$$

$$^{n+1/2}\{u\} = {}^{n-1/2}\{u\} + h^n\{\dot{u}\}$$

$$^{n+1/2}\{\omega\} = {}^{n-1/2}\{\omega\} + h^n\{\dot{\omega}\}$$

The stability analysis of this algorithm can be found in (Downer, 1990). Unconditional stability in time integration schemes can be obtained with implicit algorithms, such as Newmark, but increases the complexity of the computation scheme.

3.2.7.2. Updating the rotational orientation

The rotational orientation parameters are not directly integrable from the angular velocity vector. As a consequence, given the angular velocity, a procedure must be developed to update the configuration orientation. The Euler parameters representation has been chosen in this work, because of its algebraic nature and especially because it does not possess any singularity limitation. As such, Euler parameters appear to be the only set that allows the treatment of rotations of arbitrary magnitude without resorting to any special precautions.

Given the rotation matrix $[R]$ describing the orientation of the body-fixed reference frame (B) with respect to the inertial reference frame (I), the angular velocity tensor is obtained by:

$$[\tilde{\omega}] = -[\dot{R}][R]^T = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \qquad [9]$$

where ω_i are the components of the angular velocity vector $\{\omega\}$.

Given [9] and [4], the angular velocity vector can be expressed in terms of the Euler parameters and their time derivatives as:

$$\begin{Bmatrix} 0 \\ \omega \end{Bmatrix} = 2\begin{bmatrix} q_0 & \langle q \rangle \\ -\{q\} & q_0[I] - [\tilde{q}] \end{bmatrix} \begin{Bmatrix} \dot{q}_0 \\ \dot{q} \end{Bmatrix} \qquad [10]$$

By inverting [10], the Euler parameters derivatives are obtained in terms of the body frame angular velocity components:

$$\begin{Bmatrix} \dot{q}_0 \\ \dot{q} \end{Bmatrix} = \frac{1}{2}\begin{bmatrix} 0 & -\langle\omega\rangle \\ \{\omega\} & [\tilde{\omega}]^T \end{bmatrix}\begin{Bmatrix} q_0 \\ q \end{Bmatrix} = [A(\omega)]\{q\} \qquad [11]$$

The configuration orientation is deduced from a time discretization of [11]. The Euler parameters derivatives are subject to the following constraint conditions:

$$\dot{q}_0 q_0 + \langle\dot{q}\rangle\{q\} = 0 \qquad [12]$$

Among several possible schemes, the implicit trapezoidal formula gives an approximation which satisfies [12] in the discrete sense:

$$\frac{1}{h}\left({}^{n+1}\{q\} - {}^{n}\{q\}\right) = \left[A\left({}^{n+1/2}\omega\right)\right]\frac{1}{2}\left({}^{n+1}\{q\} + {}^{n}\{q\}\right)$$

where h is the time step.

The discrete orientation update is then given by:

$${}^{n+1}\{q\} = \frac{1}{D}\left([I] + \frac{h}{2}\left[A\left({}^{n+1/2}\{\omega\}\right)\right]\right)\left([I] + \frac{h}{2}\left[A\left({}^{n+1/2}\{\omega\}\right)\right]\right){}^{n}\{q\}$$

where $D = 1 + \frac{h^2}{4}\|\omega\|^2$ and the notation $\|x\|$ designates the euclidian norm.

3.2.7.3. Lagrange multipliers treatment

The Lagrange multipliers are solved by the use of a stabilized constraint force procedure. The kinematic constraint conditions are replaced by the following penalty equations:

$$\{\dot{\lambda}\} = \frac{1}{\varepsilon}[B]\begin{Bmatrix} \dot{u} \\ \omega \end{Bmatrix} = \frac{1}{\varepsilon}[B]\{\dot{d}\} \qquad [13]$$

where ε is a penalty coefficient.

In order to obtain a numerical solution to the companion differential equation [13], the constrained equations of motion [7] are first integrated using the implicit forward Euler formula as:

$$^{n+1}\{\dot{d}\} = h\,^{n+1}\{\ddot{d}\} + {}^{n}\{\dot{d}\}$$

The stabilized differential equation for the Lagrange multipliers is obtained by substituting the result of the integration of [7] into [13] as:

$$\varepsilon\,^{n+1}\{\dot{\lambda}\} + h[B][M]^{-1}[B]^{T\,n+1}\{\lambda\} = h[B][M]^{-1\,n+1}\{Q\} + [B]\,^{n}\{\dot{d}\}$$

where $[M]$ represents the mass and inertia matrix of [7].

The forward Euler formula is applied to obtain a discrete update of the Lagrange multipliers, as follows:

$$\left(\varepsilon[I] + h^2[B][M]^{-1}[B]^T\right)^{n+1}\{\lambda\} = \varepsilon^{n}\{\lambda\} + h^2[B][M]^{-1\,n+1}\{Q\} + h[B]^{n}\{\dot{d}\} \quad [14]$$

As one can see, the updating of the generalized coordinates, of the rotational orientation and of the Lagrange multipliers are performed separately. For each time step, the generalized coordinates are first calculated and are used to update the right-hand side vector $\{Q\}$. This vector is put into the Lagrange multiplier update module. Finally, the right-hand side vector $\{Q\}$ is corrected with the current constraint force vector $[B]^T\{\lambda\}$. The procedure is then advanced to the next time step.

These separated treatments lead to an attractive modular software implementation, which favors the object-oriented programming and eases the introduction of new algorithms and new formalisms in the future. The next section is dedicated to the description of the class **Domain,** which manages the different stages of the analysis.

3.3. *Description of the class* Domain

The class **Domain** has already been evoked in the general description of the software architecture. It is used for the representation of an elementary problem and drives the different stages of the analysis. The summarized description of **Domain** is given in Figure 16.

The analysis is performed through the message S*olve()* of the class **Domain** (Figure 17). This message first generates the lists of nodes, elements, joints and interactions from the files created by the preprocessor. Three initialization methods are then activated and the iterative resolution is initiated. At the end of the resolution, *Terminate()* is activated in order to free the memory occupied by the initializations.

```
Class Domain
{
  private : //examples of specific private attributes
    List<Element> *element_list_ ;
    List<Node> *node_list_ ;
    List<Joint> *joint_list_ ;
    List<Interaction> *boundary_conditions_list_ ;
    List<Interaction> *loading_list_ ;
    Matrix<double> *global_mass_matrix_;
    Vect<double> *global_NL_acceleration_ ;
    Element elem_ ;
  protected : //examples of specific protected attributes
    int nb_node_,nb_element_,nb_joint_,nb_eq_ ;
  public :  //examples of specific public methods
    void GetElement_Domain() ;
    void GetNode_Domain() ;
    void Initialization() ;
    void Give_Init_Conditions() ;
    void InitRotation_Euler_Param() ;
    int NbEq() {return nb_eq_ ;}
    void TopElement() {element_list_->Top() ;}
    Element *GetElement() {return element_list_->Get() ;}
    void Solve() ;
    void Solve_at(const double &time) ;
    void Terminate() ;
};
```

Figure 16. Domain *class definition*

At each time step, the calculation is performed by the method *Solve_at()*. The Figure 18 shows the operations which follow the activation of *Solve_at()*. Summarized explanations of these operations are given as inline comment remarks.

From Figures 17 and 18, it can be seen that the implementation of the *solve()* and *solve_at()* methods of **Domain** does not depend on the chosen formalisms and solving algorithms. The introduction of new formalisms or new solving strategies can be achieved easily, with small changes of some existing classes or the definition of new classes. In any case, it does not require the complete redefinition of the software architecture.

```
void Domain::Solve()
{
  this->GetElement_Domain() ;
  this->GetNode_Domain() ;
  ….
  //initialization of the different matrices and vectors
  this->Initialization() ;
  //initial conditions
  this->Give_Init_Conditions() ;
  //initialization of the Euler parameters and the rotation matrices
  this-> InitRotation_Euler_Param() ;
  double time=0. ;
  //max_time and deltat are defined by the user
  int nb_step=(int)(max_time/(deltat/2.)) ;
  for (int step=1 ;step<=nb_step ;step++)
  {
      time+=deltat/2. ;
      this->Solve_at(time) ;
  }
  this->Terminate() ;
}
```

Figure 17. *Solve() method of the class* **Domain**

3.4. *Example of code extension*

One example of code extension concerns the deployment dynamics of beam structures. The simulation of extrusion of beam-like structures can be interesting to study various problems such as tethers, space assembly, hot rolling and fluid jets. As the beam extends from the guide, the spatial volume which deforms changes.

To model this effect, one solution is to discretize the beam by an equal number of finite elements growing in length (Downer, 1990). This moving grid approach can readily be adopted in the beam formulation presented in 3.2.2.2. The grid deformation is accounted for in a proper manner by making the finite element basis functions implicit functions of time (space-time discretization). Due to the changing spatial reference for the dynamic variables, the inertia operator acquires terms representing the convective rate of change of the variable in addition to those representing the local rate of change of the variable. The internal force formulation presented in 3.2.2.2 and the computation procedure exposed in 3.2.7 remain.

The introduction of this new functionality into the current code architecture could be performed by deriving a class **TBeam2** from **Beam2**, which would modelize a **Beam2** element growing in length. Also, it would be necessary to derive a new class **TBeam2_Formulation** from **Beam2_Formulation**, in order to integrate the specific convective terms appearing in the inertia operator (Figure 2).

```
void Domain::Solve_at(const double &time)
{
  // updating scheme for the generalized coordinates
  this->Algo() ;
  // updating the Euler parameters and the rotation matrices
  this->Update_Rotational_Orientation(time) ;
  global_mass_matrix_=new Matrix<double>(this->NbEq());
  global_NL_acceleration_=new Vect<double>(this->NbEq());
  ….
  for (this->TopElement() ;elem_=this->GetElement()) ;)
  // iteration on all elements
  {
    // choice of the finite element formulation
    Beam2_Formulation eq(elem_,this,time) ;
    Matrix<double> *mass_elem ;
    // calculation of the elementary mass matrix
    mass_elem=eq.Calculate_Mass_Matrix() ;
    // localization table of the element
    int *loc ;
    loc=elem_Loce() ;
    // calculation of the global mass matrix
    this->Assemble(global_mass_matrix_,mass_elem,loc) ;
    delete mass_elem ;
    ….
  }
  // calculation of the external force vector from loading_list_
  this->CreateFext(time);
  // definition of the right-hand side vector Q (equation [8])
  this->Create_System_one(time) ;
  // calculation of the Lagrange multipliers λ according to
   // the chosen algorithm (for example, Euler formula [14])
  this->Solve_Lagrange_Multipliers(time) ;
  // correction of the right-hand side vector Q with the current
  // constraint force vector [B]^T λ
  this->Create_System_two(time) ;
  //calculation of the nodal accelerations vectors ü and ω̇ (eq. [7])
  this->Solve_Generalized_Coordinates(time) ;
  // saves the current solution and frees memory
  this->Terminate(time) ;
}
```

Figure 18. *Solve_at() method of the class* **Domain**

4. Conclusions

A finite element calculation software may advance in different directions. In particular, for software dealing with multibody systems, many formalisms and hypotheses can evolve and combine with each other, as for example, the choice of the finite elements, the choice of the referential frame, the definition of physical or material parameters, the choice of the parameters for the representation of the finite rotations, the choice of the solving algorithm, the formalism and the treatment of the joints, the flexibility or the rigidity of the bodies, among others.

In this paper, the architecture of the computational engine of a new finite element software for the simulation of flexible mechanisms has been presented. The program has been designed according to object-oriented principles, which allow us to simplify the architecture of the program. Although the current architecture is based on given hypotheses and formalism, its originality lies within the fact that it has been conceived in order to provide a flexible and extensive set of objects that facilitate multibody systems analysis and which can be adapted to meet future developments. The introduction of new formalisms or new solving strategies can be achieved easily, with small changes of some existing classes or the definition of new classes. In any case, it does not require the complete redefinition of the software architecture.

5. References

Arnold K., Gosling J., *The Java programming language*, Reading, Addison-Wesley, 1998.

Besson J., Foerch R., "Object-oriented programming applied to the finite element method. Part I. General concepts", *Revue européenne des éléments finis*, vol. 7, n° 5, 1998, p. 535-566.

Besson J., Leriche R., Foerch R., Cailletaud G., "Object-oriented programming applied to the finite element method. Part II. Application to material behaviors", *Revue européenne des éléments finis*, vol. 7, n° 5, 1998, p. 567-588.

Booch G., *Object oriented analysis and design with applications*, Redwood City, Benjamin Cummings, 1993.

Breitkopf P., Escaig Y., "Object-oriented approach and distributed finite element simulations", *Revue européenne des éléments finis*, vol. 7, n° 5, 1998, p. 609-626.

Cardona A., Géradin M., "A beam finite element non-linear theory with finite rotations", *International Journal for Numerical Methods in Engineering*, vol. 26, 1988, p. 2403-2438.

Cardona A., An integrated approach to mechanism analysis, Ph.D. Thesis, University of Liège, 1989.

Cardona A., Géradin M., Doan D.B., "Rigid and flexible joint modelling in multibody dynamics using finite elements", *Computer Methods in Applied Mechanics and Engineering*, vol. 89, 1991, p. 395-418.

Chiou J.C., Constraint treatment techniques and parallel algorithms for multibody dynamic analysis, Ph.D. Thesis, University of Colorado, 1990.

Downer J.D., A computational procedure for the dynamics of flexible beams within multibody systems, Ph.D. Thesis, University of Colorado, 1990.

Downer J.D., Park K.C., Chiou J.C., "Dynamics of flexible beams for multibody systems: a computational procedure", *Computer Methods in Applied Mechanics and Engineering*, vol. 96, 1992, p. 373-408.

Dubois-Pèlerin Y., Pegon P., "Object-oriented programming in nonlinear finite element analysis", *Computers and Structures*, vol. 67, 1998, p. 225-241.

Dufossé F., Approche orientée objet appliquée à la conception d'un logiciel dédié à l'analyse des systèmes multicorps, Ph.D. Thesis, Université Henri Poincaré, Nancy 1, 2001.

Dufossé F., Kromer V., Mikolajczak A., Gueury M., "Simulation of 3D polyarticulated mechanisms through object-oriented approach", *in: N. Mastorakis, ed., Problems in Modern Applied Mathematics, Mathematics and Computers in Science and Engineering*, 2000, p. 84-89.

Eyheramendy D., "An object-oriented hybrid symbolic/numerical approach for the development of finite element codes", *Finite Elements in Analysis and Design*,vol. 36, 2000, p. 315-334.

Goldberg A., Robson D., *Smalltalk-80: The language and the implementation*, Reading, MA, Addison-Wesley, 1983.

Ibrahimbegovic A., "On the choice of finite rotation parameters", *Computer Methods in Applied Mechanics and Engineering*, vol. 149, 1997, p. 49-71.

Ibrahimbegovic A., Mamouri S., "On rigid components and joint constraints in nonlinear dynamics of flexible multibody systems employing 3D geometrically exact beam model", *Computer Methods in Applied Mechanics and Engineering*, vol. 188, 2000, p. 805-831.

Klapka I., Cardona A., Géradin M., "An object-oriented implementation of the finite element method for coupled problems", *Revue européenne des éléments finis*,vol. 7, n° 5, 1998, p. 469-504.

Kunz D.L., "An object-oriented approach to multibody systems analysis", *Computers and Structures*, vol. 69, 1998, p. 209-217.

Lai M., *Conception orientée objet. Pratique de la méthode HOOD*, Dunod, 1991.

Mackerle J., "Object-oriented techniques in FEM and BEM. A bibliography (1996-1999)", *Finite Element in Analysis and Design*, vol. 36, 2000, p. 189-196.

Mackie R.I., "Object-oriented programming and numerical methods", *Microcomputers in Civil Engineering*, vol. 6, 1991, p. 123-128.

Miller GR., "A LISP-based object-oriented approach to structural analysis", *Engineering with Computers*, vol. 4, 1988, p. 197-203.

Nikravesh P.E., Chung I.S., "Application of Euler parameters for the dynamic analysis of three-dimensional constrained mechanical systems", *Journal of Mechanical Design*, vol. 104, 1982, p. 785-791.

Otter M., Elmqvist H., Cellier F.E., "Modeling of multibody systems with the object-oriented modeling language Dymola", *Proceedings of the MATO/ASI, Computer-aided analysis of rigid and flexible mechanical systems*, Troia, Portugal, June 27-July 9, 1993.

Park K.C., Chiou J.C., "Stabilization of computational procedures for constrained dynamical systems", *Journal of Guidance, Control and Dynamics*, vol. 11, 1988, p. 365-370.

Park K.C., Chiou J.C., Downer J.D., "Explicit-Implicit staggered procedure for multibody dynamics analysis", *Journal of Guidance, Control and Dynamics*, vol. 13, 1990, p. 562-570.

Park K.C., Downer J.D., Chiou J.C., Farhat C., "A modular multibody analysis capability for high precision, active control and real-time applications", *International Journal for Numerical Methods in Engineering*, vol. 32, 1991, p. 1767-1798.

Remy P., Devloo B., Alves Filho J.S.R., "An object-oriented approach to finite element programming (phase I): a system independent windowing environment for developing interactive scientific programs", *Advances in Engineering Software*, vol. 14, 1992, p. 41-46.

Rumbaugh J., *Object oriented modeling and design*, Prentice Hall, 1991.

Rumbaugh J., Jacobson I., Booch G., *The unified modeling language reference manual*, Reading, MA, Addison-Wesley, 1999.

Ryan R. R., "ADAMS: Multibody system analysis software", in *Multibody Systems Handbook*, Scheihlen W. (ed), Berlin, Springer, 1990.

Scholz S.P., "Elements of an object-oriented FEM++ program in C++", *Computers and Structures*, vol. 43, n° 3, 1992, p. 517-529.

Tisell C., Orsborn K., "A system for multibody analysis based on object-relational database technology", *Advances in Engineering Software*, vol. 31, 2000, p. 971-984.