AN OBJECT-ORIENTED APPROACH OF THE FINITE ELEMENT METHOD FOR THE SIMULATION OF 3D MULTIBODY SYSTEMS

Dufossé François*, Kromer Valérie*, Mikolajczak Alain*, and Gueury Michel*

* Equipe de Recherches en Interfaces Numériques (ERIN)
ESSTIN, Université Henri Poincaré, Nancy 1
2, rue Jean Lamour 54500 Vandoeuvre-lès-Nancy, France
e-mail : dufosse@esstin.uhp-nancy.fr

Key words : Object-Oriented Programming, Finite Element Method, Multibody Systems.

Abstract. In this paper, we present the object-oriented architecture of a software dedicated to the finite element simulation of 3D Multibody systems, with large rotations and large strains. The main features of the Object-Oriented Programming (OOP) are first quickly summarized. Then we present the architecture of the "solver" part of the software. Stress will be laid on the adequacy between the OOP and the finite element method used for the treatment of multibody flexible mechanisms, within a given formalism and given hypotheses. We will show that the OOP greatly simplifies the choice and the implementation of other formalisms concerning polyarticulated systems, thus conferring high flexibility and adaptability to the developed software.
1 INTRODUCTION

The aim of our work is to apply the object-oriented approach to a specific software, dedicated to the simulation of polyarticulated systems. The main features of the software are the possibilities to treat flexible as well as rigid mechanisms, and to consider various formalisms and hypotheses (particularly for the treatment of finite rotations, of the joints, of strain and stress calculations, etc).

Typically, finite element softwares are written in FORTRAN language, with a so-called “procedural” programming. These softwares contain numerous data structures, which can be called anywhere in the code. Because of this global access, the flexibility of the software is decreased. Thus, it becomes difficult to maintain the code and even more difficult to implement new models or new solving procedures. In of fact, it is necessary to have an overall understanding of the software architecture. Many interdependences can be hidden and difficult to detect. A few changes in data structures may give rise to unpredictable effects in the code.

These difficulties can be solved with the use of the OOP, as will be shown in the second part of this paper, which presents the main features of the OOP.

The third part deals with the general architecture and the main characteristics of the developed software.

2 OBJECT-ORIENTED PROGRAMMING

The use of object-oriented languages for finite element softwares first appeared in the late 1980’s, with simple linear formulations (Fenves1, Forde2). Object-oriented architectures were then progressively introduced for the treatment of various and much more complex problems (Miller3, Zimmermann4,5,…).

The most important component of the OOP is the object entity. This entity associates data (the attributes) and operators acting on these data (the methods). It is able to send and to receive messages. As a consequence, an object is an entity whose state depends on the received messages. The attributes and methods can be used by other objects, through the object’s interface (public methods). The interface prevents the user from having direct access to the object’s details. This data abstraction is called encapsulation. It appears to be one of the most prominent concepts of the OOP and allows an explicit definition of the relations between various objects.

The objects are completely defined through classes. The objects are individual instances of the classes. The methods represent the actions, which means the code executed as a specific message is sent to a particular object. Arguments can be included with the messages. From an existing class (superclass), it is possible to create a new class (subclass), which incorporates the features of the superclass as well as new characteristics. The subclass is derived from the first one. It inherits the messages and behaviours of the superclass. Class inheritance is also one of the most prominent concepts of OOP. If a method of the superclass has to be defined once again in a derived class, it must be declared as virtual. This is called polymorphism. If a virtual method has no implementation in the superclass, but only in the derived classes, it is
called a pure virtual method. A superclass possessing at least a pure virtual method is an abstract class. It is used only for the definition of the interface belonging to all the derived classes.

Contrary to procedural programming, what is required to modify the object-oriented software is restricted by the data abstraction. Changes brought to a specified object class do not affect other classes.

3 SOFTWARE ARCHITECTURE

3.1 Introduction

The evolutivity is probably the most important characteristic for any calculation software, since the software’s perenniality directly depends on it. The opportunities brought by the inheritance and the encapsulation are the keys to evolutivity for a calculation software, as they allow the introduction of new functionality from the derivation of classes constituting the heart of the software’s architecture.

A finite element calculation software may advance in different directions. In particular, for softwares dealing with polyarticulated 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 rotations, the choice of the solving algorithm, the formalism and the treatment of the joints, the flexibility or the rigidity of the bodies, etc.

Thus, the definition of few basic classes, such as ELEMENT_FORMULATION, MATERIAL_BEHAVIOUR, etc, which contain the common treatments (calculation of internal forces, calculation of mass and stiffness matrices, etc) becomes natural. Specific methods corresponding to a particular formalism will be defined in classes deriving from these basic classes.

3.2 Global description of the software architecture

The developed software is written in C++, because the C++ implementation provides portability, flexibility and easy-to-use features which are not easily obtained for large codes implemented in other languages.

We have decided to separate the operations performed by the preprocessor, the solver and the postprocessor. A graphic preprocessor developed in Borland C++ Builder is used to define the files needed for a finite element analysis (elements file, nodes file, joints file, initial conditions file, boundary conditions file, etc). The solver is written in C++ANSI, independently of the preprocessor and the compiler. Specific methods are defined to read and interpret the data files. In the same manner, specific methods are used to create results files, which are then read and interpreted by the interface of the graphic postprocessor.

The class DOMAIN has been created to represent an elementary problem (corresponding to a specific finite element analysis). It contains and manages the classical steps of a finite
element analysis (meshing, boundary conditions, initial conditions, interactions, solving, updating, writing results).

The function Main() is used only to create an object whose type is the class DOMAIN. The analysis is performed through the message solve() (Figure 1).

Main()
{
    Domain *domaine ;
    domaine=new Domain ;
    domaine->solve() ;
    delete domaine ;
}

Figure 1. : Main function

3.3 Flexible beam dynamics formulation

In order to describe the dynamics of a flexible beam, an inertial reference frame is used for the description of the translational motion, whereas a body-fixed frame is used for the rotary motion. The motion due to rigid rotation 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 final discrete equations of motion of a flexible beam element are given as

\[
\begin{bmatrix}
    m & 0 \\
    0 & J
\end{bmatrix}
\begin{bmatrix}
    \ddot{u} \\
    \ddot{\omega}
\end{bmatrix}
+ \begin{bmatrix}
    0 \\
    D(\omega)
\end{bmatrix}
\begin{bmatrix}
    S^c \\
    S^b
\end{bmatrix}
= \begin{bmatrix}
    f^c \\
    f^b
\end{bmatrix}
\]

(1)

m and J represent the assembled mass and inertia matrices; \( \ddot{u} \) and \( \ddot{\omega} \) represent the nodal acceleration vectors; \( D(\omega) \) represents the assembled non-linear acceleration; \( S^{c,b} \) and \( f^{c,b} \) represent the assembled internal and external force vectors partitioned into translational and rotational parts, respectively.

These equations can be specialized to the case of static equilibrium as :

\[ S = f \]  

(2)

The equations of motion (1) can also represent a rigid body by setting the internal force \( S \) to zero. Thus, 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.
3.4 Description of the basic classes

In what follows, we present the most important basic classes of the solver (Table 1, Figure 2). These classes are all abstract classes. They are used in the method \texttt{solve()} of the class \texttt{DOMAIN}.

<table>
<thead>
<tr>
<th>Description</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite element formulation</td>
<td>\texttt{ELEMENT_FORMULATION}</td>
</tr>
<tr>
<td>Material behaviour</td>
<td>\texttt{MATERIAL_BEHAVIOUR}</td>
</tr>
<tr>
<td>Joint formulation</td>
<td>\texttt{JOINT_FORMULATION}</td>
</tr>
<tr>
<td>Choice of the generalized coordinates integration procedure</td>
<td>\texttt{TIME_INTEGRATION_SCHEME}</td>
</tr>
<tr>
<td>Choice of the constraint force solution procedure</td>
<td>\texttt{SOLVE_LAGRANGE}</td>
</tr>
</tbody>
</table>

Table 1.: Definition of the basic classes

![Diagram](image.png)

Figure 2.: Class DOMAINDOMAINDOMAIN

a) Common tools

Some utility classes have been developed to manage mathematical tools (vectors, matrices), variable size of objects, stacks, buffers, pools for memory management, read and write protection on objects and execution time measurement, among others.

The implementation of these classes is based on the templates mechanism which allows the definition of concepts (methods, algorithms) independently of the type of object being used.

Contrary to the virtual inheritance mechanism, the connection between data and methods is made by the compiler, which receives enough information to produce an optimized code. This technique not only increases legibility without losing efficiency, but also allows the methods to be reused without considering how data are implemented.
b) Parametrization of the rotations

The ROTATION class is of particularly great importance (Figure 3). This class manages the information for the different parametrizations of the rotations (rotational vector, Euler parameters, Rodrigues parameters). As a matter of fact, for the modelization of 3D multibody systems, special attention must be given to the adequate representation of the rotations. The rotations are used in several steps of the calculation (calculation of the internal forces, calculation of the constraint Jacobian matrices, etc).

The different results (especially the rotation matrices) are stored in objects of type ELEMENT and JOINT. The ELEMENT class is an class used for the representation of finite elements, whereas the JOINT class is used for the representation of constraints between bodies. Classes working with rotation matrices contain objects of type ROTATION.

In the chosen formalism, the rotation vectors, the Euler parameters and the Rodrigues parameters are used for the computation of strain increments and curvatures. The rotational orientation update procedure is performed by the use of the Euler parameters and an implicit trapezoidal algorithm. This procedure is developed in a method of the class EULER_PARAMETERS (Figure 4). Other useful methods can be found in this class, for example the method allowing the calculation of the average rotation matrix, or the method defining the multiplication of two rotation matrices with the quaternions law.

```
Class EULER_PARAMETERS
{
    calculate_rotation_matrix() ;
    calculate_rotation_increment () ;
    calculate_rotation_average() ;
    multiplication_matrix_rotation_quaternion() ;
    update_parameters() ;
    .............
}
```

Figure 4.: Class EULER_PARAMETERS
c) Finite element formulation

The class ELEMENT_FORMULATION is the abstract master class of the finite elements library (Figure 5). It provides virtual methods of computation of the different finite element arrays and tables \((\text{calculate\_internal\_forces()}, \text{calculate\_mass\_matrix()}, \text{calculate\_strain\_tensor()})\), etc. It also provides methods of access to their attributes, for example a pointer to an object of the class ELEMENT, which in turn contains pointers to objects of the classes MATERIAL and SECTION.

The abstract class ELEMENT_TYPE possesses an attribute which is a pointer to an ELEMENT object. From ELEMENT_TYPE, subclasses corresponding to specific finite elements are derived (for example, BEAM2 for a two-nodes beam element). They contain methods for the calculation of the element’s shape functions and its derivatives, or the calculation of the element’s length.

The class BEAM_FORMULATION is derived from ELEMENT_FORMULATION (Figure 6). 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.
d) Behaviour law

In most object-oriented calculation softwares, the material behaviour is included in the finite element formulation. In our code architecture, a separated class MATERIAL_BEHAVIOUR has been created (Figure 7), which contains the virtual methods for the calculation of the stresses according to a given material behaviour law.

The methods for a linear elastic material (calculate_stress_increments(), calculate_stress_resultant_forces(), calculate_stress_resultant_moments()) are defined in the derived class LINEAR_ELASTIC_BEHAVIOUR.

A pointer to an object MATERIAL_BEHAVIOUR is set as an attribute of the class ELEMENT_FORMULATION (and reciprocally), in order to allow the use of the methods of one class in the other one.

```cpp
Class MATERIAL_BEHAVIOUR
{
    calculate_stress_resultant_forces (virtuelle);
    calculate_stress_resultant_moments (virtuelle);
    calculate_stress_increments (virtuelle);
    .......... 
    ELEMENT_FORMULATION *.. ;
    MATRIX<> *.. ;
    VECTOR<> * .. ;
    ......
};
```

Figure 7.:Class MATERIAL_BEHAVIOUR

e) Joints formulation procedure

The abstract class JOINT_FORMULATION allows the treatment of the joints with different formalisms (Lagrange multipliers, Master/Slave method). Currently, the only available class is the derived class LAGRANGE_MULTIPLIERS, for the Lagrange multipliers procedures (Figure 8).

By incorporating the Lagrange multipliers $\lambda$, the equations of motion for constrained flexible multibody systems are written as follows:

$$
\begin{bmatrix}
    m & 0 \\
    0 & J
\end{bmatrix}
\begin{bmatrix}
    \ddot{u} \\
    \dot{\omega}
\end{bmatrix}
+ B^T \lambda =
\begin{bmatrix}
    Q_u \\
    Q_\omega
\end{bmatrix}
$$

In the above equation, the notation contains both the holonomic and non-holonomic constraints in the constraint force vector $B^T \lambda$. The B matrix, called the constraint Jacobian
matrix, is deduced from the kinematic relationship between the bodies of the system. The right-hand side vector contains the remaining force-type terms as

$$\begin{bmatrix} Q_u \\ Q_\omega \end{bmatrix} = \begin{bmatrix} f^e - S^e \\ f^b - S^b - D(\omega) \end{bmatrix}$$

(4)

The constraint Jacobian matrices calculated by methods of the class LAGRANGE_MULTIPLIERS are stored as attributes of the class JOINT.

f) Time integration scheme

In the software architecture, an attractive modularity has been introduced so that the generalized coordinates (translational coordinates and angular velocities) are solved separately from the constraint forces.

The different calculation algorithms for the generalized coordinates are defined in classes deriving from the abstract class TIME_INTEGRATION_SCHEME. This class is constituted by attributes and methods which are common for all algorithms. The attributes are vectors, for the storage of different information (displacements, velocities, angular velocities and angular accelerations). The methods manage these vectors for different time steps.

The class MODIFIED_CENTRAL_DIFFERENCE has been derived from TIME_INTEGRATION_SCHEME (Figure 9). It is dedicated to the implementation of a special algorithm called «Two-stage Staggered Explicit-Implicit Algorithm», first developed by Park et al.8,9. A two-stage modification of the explicit central difference algorithm is used to integrate the translational coordinates and the angular velocity vector.
Once the angular velocities are calculated, the angular orientations are updated with the implicit trapezoidal algorithm defined in the class EULER_PARAMETERS. Nevertheless, it is possible to develop purely implicit algorithms in classes derived from TIME_INTEGRATION_SCHEME.

**g) Lagrange multipliers solution procedure**

The class SOLVE_LAGRANGE is created to solve the Lagrange multipliers and the corresponding constraint forces. Its attributes contain the constraint Jacobian matrices calculated in the class LAGRANGE_MULTIPLIERS. Specific classes (BAUMGARTE, PENALTY), corresponding to specific constraint force solution procedures, are derived from the class SOLVE_LAGRANGE (Figure 10).

![Figure 10: Organization of the class SOLVE_LAGRANGE](image)

**4 COMPUTATIONAL STRATEGY**

The computational procedure used for a dynamic analysis of flexible multibody systems is driven by the method `solve()` of the class DOMAIN. The most important steps of the procedure are the following:

- Update according to the chosen time integration scheme
- Update Euler parameters
- Update internal forces vector
- Update non-linear acceleration
- Update remaining force-type vector
- Calculate Lagrange multipliers
- Calculate nodal acceleration vector

**5 CONCLUSIONS**

In this paper, the architecture of a new finite element software for the simulation of flexible mechanisms has been presented. The program has been written following object-oriented programming techniques and using the C++ programming language.

The flexibility of the software is made possible thanks to the clear modularization that can be reached in OOP, with a marked separation of functionalities. Extensibility and reusability of OOP are clearly shown: contrary to procedural programming, users are able to naturally
incorporate new features and formalisms, enlarging the program’s possibilities. The most obvious advantage of OOP appears to be its capability to accurately represent real physical entities, thanks to the « object » notion. Thus, the boundary between human thinking and computer science becomes narrower.

REFERENCES