Elements of an Object-Oriented FEM Program for Multibody Systems Analysis

KROMER Valérie, DUFOSSÉ François, GUEURY Michel
Equipe de Recherche en Interfaces Numériques – Modélisation des Matériaux du Génie Civil
Université Henri Poincaré, Nancy 1
ESSTIN, 2 rue Jean Lamour, 54500 Vandoeuvre-lès-Nancy
FRANCE

Abstract: -The principal objective of this paper is to give elements of the object-oriented architecture of a finite element code dedicated to multibody systems analysis. Emphasis is placed on the adequacy between object-oriented programming and the finite element method used for the treatment of three-dimensional multibody flexible mechanisms with large rotations and large strains. Numerical examples are given in order to demonstrate the software capabilities.

Key-Words: - Object-oriented programming, Finite element code, Multibody systems analysis

1 Introduction

Several researchers have worked in the application of object-oriented programming techniques to the finite element method in recent years, in various domains of interest in finite element developments (constitutive law modeling, rapid dynamics, coupled problems, nonlinear analysis, symbolic computation, among others). The recent article of Mackerle [1] gives a bibliography with more than 150 references covering the period 1996-1999.

However, like many other engineering applications, multibody systems analysis codes (ADAMS [2], DADS [3], MBOSS [4]) 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 first section. It 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 formalism used for the treatment of multibody systems is presented in the second section. The third section deals with the object-oriented architecture of the computational engine of the software. Finally, numerical examples are given in the last part of the paper.

2 Object-oriented concepts and multibody systems analysis

The object-oriented philosophy 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 [5,6].

The *abstraction* of the data type is realized by the means of a *class*, which 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. 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.

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. 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.

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.

The last concept is *polymorphism*, which designates the capability of object-oriented applications to interpret the same request differently depending on the object being processed. It is realised through the definition of abstract objects using virtual member functions. Abstract objects allow the writing of generic algorithms and the easy extension of the existing code. For instance, through polymorphism, it is possible to describe any constraint between two bodies without regard to joint type (revolute joint, translational joint, etc) and body type (rigid, flexible, etc). The implementation details of the specific bodies and joints involved in the connection are hidden by abstraction in the joint and body objects.

3 Multibody system analysis

The choices concerning mechanical formalisms have been made in order to favor concepts as modularity, polyvalence and evolutivity, which fit particularly well to the object-oriented philosophy.

3.1 Flexible beam dynamic 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 [7,8]. 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.

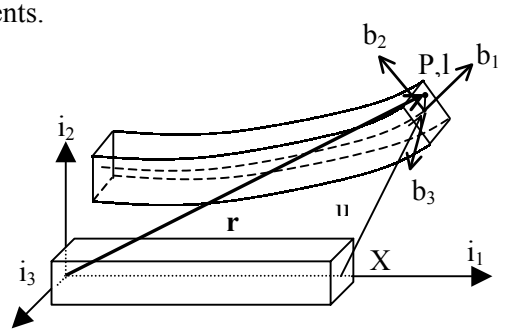


Fig.1. Spatial beam kinematics

The location from the inertial origin of an arbitrary point P on the beam (Fig.1) is represented by the following position vector :

$$\vec{r} = \vec{X}^{\,i} + \vec{u}^{\,i} + \vec{l}^{\,b} \qquad (1)$$

where $\vec{X}$ is the position vector of a point of the original neutral axis, $\vec{u}$ is the total translational displacement vector of the neutral axis, $\vec{l}$ 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 inertial frame or with respect to the body-fixed frame, respectively.

The orientation of the body-fixed reference frame is expressed with respect to the inertial reference frame through an orthogonal transformation matrix $[R]$.

The body frame components of the angular velocity tensor are obtained by :

$$[\tilde{\omega}] = -[\dot{R}][R]^T = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \qquad (2)$$

where ω_i (i = 1 to 3) are the components of the angular velocity vector $\vec{\omega}$, the notation $[\dot{x}]$ indicates the time derivative, the notation $[\tilde{x}]$ indicates a skew symetric tensor and the notation $[x]^T$ indicates a transpose matrix.

The final discrete equations of motion of a flexible beam element are given as :

$$\begin{bmatrix} m & 0 \\ 0 & J \end{bmatrix} \begin{Bmatrix} \ddot{\vec{u}} \\ \dot{\vec{\omega}} \end{Bmatrix} + \begin{Bmatrix} \vec{0} \\ \vec{D} \end{Bmatrix} + \begin{Bmatrix} \vec{S}^{\,i} \\ \vec{S}^{\,b} \end{Bmatrix} = \begin{Bmatrix} \vec{F}^{\,i} \\ \vec{F}^{\,b} \end{Bmatrix} \qquad (3)$$

where [m] and [J] represent the mass and inertia matrices ; $\ddot{\vec{u}}$ and $\dot{\vec{\omega}}$ represent the nodal accelerations vectors ; $\vec{D}$ represents the non-linear acceleration, $\vec{S}^{\,i,b}$ and $\vec{F}^{\,i,b}$ represent the internal and external force vectors partitioned into translational and rotational parts, respectively.

Thus, the unconstrained equations of an arbitrary configuration of flexible beams and rigid bodies can be 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.

3.2 Equations of motion for multibody systems

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

$$\begin{bmatrix} m & 0 \\ 0 & J \end{bmatrix} \begin{Bmatrix} \ddot{\vec{u}} \\ \dot{\vec{\omega}} \end{Bmatrix} + [B]^T \vec{\lambda} = \begin{Bmatrix} \vec{Q}^{\,u} \\ \vec{Q}^{\,\omega} \end{Bmatrix} \qquad (4)$$

In order to alleviate the equations, the same notation has been used (in (4) or in (3)) to represent elementary or assembled matrices and vectors. The right-hand side vector of (4) contains the remaining force-type terms as :

$$\begin{Bmatrix} \vec{Q}^{\,u} \\ \vec{Q}^{\,\omega} \end{Bmatrix} = \begin{Bmatrix} \vec{F}^{\,i} \\ \vec{F}^{\,b} \end{Bmatrix} - \begin{Bmatrix} \vec{S}^{\,i} \\ \vec{S}^{\,b} \end{Bmatrix} - \begin{Bmatrix} \vec{0} \\ \vec{D} \end{Bmatrix} \qquad (5)$$

In equation (4), the notation contains both the holonomic and non-holonomic constraints in the constraint force vector $[B]^T \vec{\lambda}$. The [B] matrix, called the constraint Jacobian matrix, is deduced from the kinematic relationship between the bodies of the system. Classical Jacobian matrices modeling standard joints (universal, revolute, spherical, translational joints) are described in [9].

Thus, given the Jacobian matrices for the joints and beam connections, the equations (4) can be employed in a systematic manner in order to represent an arbitrary assemblage of articulated flexible and rigid components.

3.3 Resolution of multibody systems equations

The resolution of multibody systems equations requires the use of an adapted algorithm for the time integration of the generalized coordinates (translational coordinates $\vec{u}$ and angular velocities $\vec{\omega}$). It must also include a proper treatment of three-dimensional finite rotations [10], as well as a method to satisfy the kinematic constraints conditions. In order to introduce an attractive modularity in the solution procedures, which favors the object-oriented philosophy, the integration of the generalized coordinates is performed separately from the calculation of the constraint forces (Lagrange multipliers $\vec{\lambda}$) [11].

The generalized coordinates are calculated with a particular explicit integration procedure, called « two-stage staggered algorithm », first developed by Park [11], which 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.

The rotational orientation parameters are not directly integrable from the angular velocity vector. As a consequence, a procedure must be developed to update the configuration orientation given the angular velocity. Various methods have been proposed for the parametrization of large rotations [12]. The Euler parameters representation has been chosen in this work, because of their algebraic nature and especially because they do not possess any singularity limitation. Once the angular velocities are calculated, the angular orientations are updated with the implicit trapezoidal formula.

The Lagrange multipliers are solved by the use of a stabilized constraint force procedure associated to the implicit forward Euler formula. Details about the procedures used for the updating of the rotational orientation and of the Lagrange multipliers can be found in [13].

As one can see, the updating of the generalized coordinates, of the rotational orientation and of the Lagrange multipliers are performed separately. These separated treatments lead to an attractive modular software implementation, which favors the object-oriented philosophy and eases the introduction of new algorithms and new formalisms in the future.

4 Description of the computational engine

4.1 Global description

The computational engine is written in C++ ANSI, independently of the preprocessor and the compiler. We selected C++, which seems to be, currently, the language providing numerical efficiency, portability, flexibility and is easy to use.

Moreover, C++ gives the possibility to use specific features, as the *overloading* of functions and operators, or the *templates mechanism*, which can greatly ease programming.

The current architecture is based on the hypotheses and formalism presented in the previous section. However, 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 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, writing results).

The management of the different stages of the analysis is performed by the major class **Domain**, which represents the heart of the software architecture (Fig.2). It has been created to represent an elementary problem (corresponding to a specific multibody system analysis).

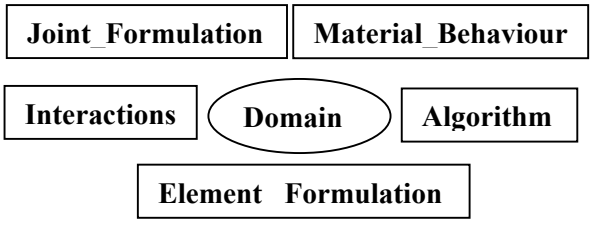


Fig.2. Global architecture of the software

4.2 Description of the major basic classes

4.2.1 Parametrization of the rotations

The abstract **Rotation** class is of particularly great importance. This class manages the information for the different parametrizations of the finite rotations (rotational vector, Euler parameters, Rodrigues parameters, among others), which are used in several steps of the calculation (calculation of the internal forces, calculation of the constraint Jacobian matrices).

The classes corresponding to a specific parametrization of the rotations are derived from the **Rotation** base class (Fig.3). The virtual methods of **Rotation** are implemented in the derived classes, as well as specific methods.

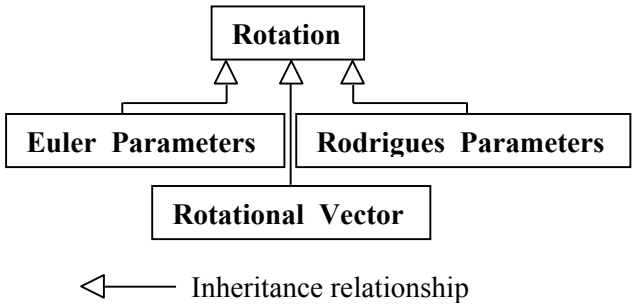


Fig.3. Derived classes from the **Rotation** virtual class

4.2.2 Finite element formulation

The class **Element_Formulation** (Fig.4) is the abstract master class of the finite elements library. It provides virtual methods of computation of the different finite element arrays and tables (mass matrix, strain operator, …). The effective computation depends on the chosen formalism and may vary appreciably. In these conditions, the object-oriented philosophy eases the implementation of new formalisms, without requiring a complete redefinition of the software's architecture.

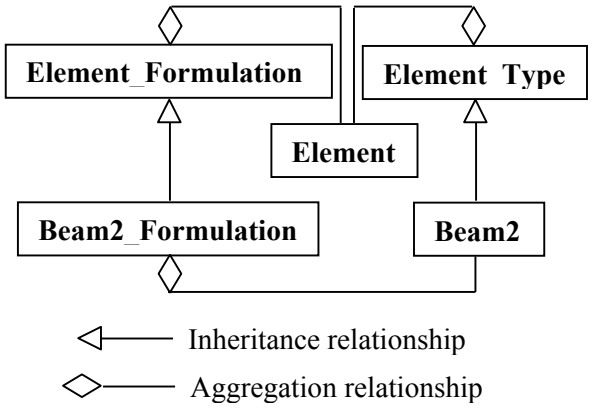


Fig.4. Classes associated with the finite element modelization

The abstract class **Element_Type** contains virtual methods for the calculation of the element's shape function and its derivatives. 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.

4.2.3 Joints formulation procedure

The **Joint** class has been created for the modelization of the joints between bodies. The abstract class **Joint_Formulation** allows the treatment of the joints according to different formalisms (Fig.5). Currently, the only available class is the derived class **Jacobian**, for the calculation of the constraint Jacobian matrices required in the Lagrange multipliers technique described in section 3.3.

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**.

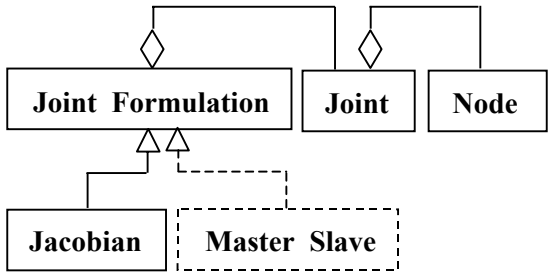


Fig.5. Classes associated with constraints modelization

4.2.4 Algorithms

The different solving algorithms presented in section 3.3 are defined in classes derived from the abstract class **Algorithm** (Fig.6).

Algorithm contains methods and attributes which are common to the different algorithms. Thus, the implementation of new algorithms can be achieved easily.

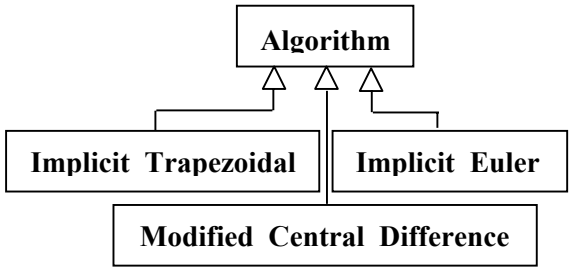


Fig.6. Abstract class **Algorithm**

4.3 Description of the class Domain

The summarized description of the major class **Domain** is given in figure 7. The *Main()* function (Fig.8) is used only to create an object whose type is the class **Domain.**

```
Class Domain
{
private :
  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 :
  int nb_node_,nb_element_,nb_joint_,nb_eq_ ;
  ….
public :
  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() ;
  ….
} ;
```

Fig.7. **Domain** class definition

```
Main()
{
  Domain *domain ;
  domain = new Domain ;
  domain->Solve() ;
  delete domain ;
}
```

Fig.8. Main function

The analysis is performed through the message S*olve()* of the class **Domain** (Fig.9). 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.

```
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() ;
}
```

Fig.9. *Solve()* method of the class **Domain**

At each time step, the calculation is performed by the method *Solve_at()*. The figure 10 shows the operations which follow the activation of *Solve_at()*. Summarized explanations of these operations are given as inline comment remarks.

```
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());
  ….
  // iteration on all elements
  for (this->TopElement() ;
          elem_=this->GetElement()) ;)
  {
  // 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() ;
    int *loc ;
    // localization table of the element
    loc=elem_Loce() ;
```

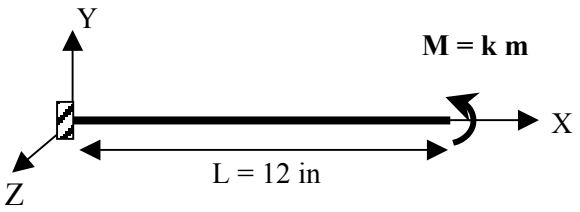
```
    // calculation of the global mass matrix
    this->Assemble
          (global_mass_matrix_,mass_elem,loc) ;
    delete mass_elem ;
    ….
  }
  // calculation of the external force vector from
  loading_list_
  this->CreateFext(time);
  // definition of the right-hand side vector Q
  /(equation 5)
  this->Create_System_one(time) ;
  // calculation of the Lagrange multipliers λ
  //according to the chosen algorithm
  //(for example, Euler formula)
  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. 4)
  this->Solve_Generalized_Coordinates(time) ;
  // saves the current solution and frees memory
  this->Terminate(time) ;
}
```

Fig.10. *Solve_at()* method of the class **Domain**

5 Numerical examples

The two numerical examples presented in this paper are purely academic. Nevertheless, they illustrate the potential of the software, whose computational engine has been designed in order to integrate flexible and rigid bodies into the architecture in a unified manner. Moreover, it is able to deal with both open-loop and closed-loop systems in a systematic way.

5.1 Example 1 : Winding of an embedded beam



$E = 30.10^6$ lb/in^2
$G = 15.10^6$ lb/in^2
$S = 1$ in^2
$I = 1/12$ in^4
$m = \pi EI/L$

Fig.11. Pure bending of an embedded beam

The first example concerns the pure bending of a beam whose first extremity is embedded and whose second extremity is submitted to an external torque (Fig. 11). The beam is discretized in twelve linear beam elements.

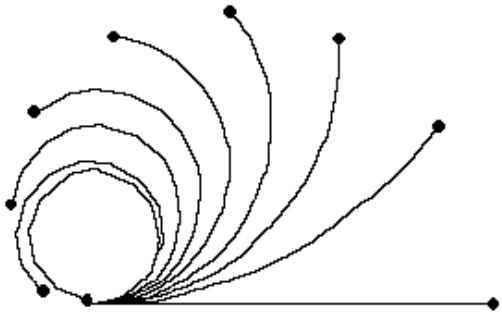
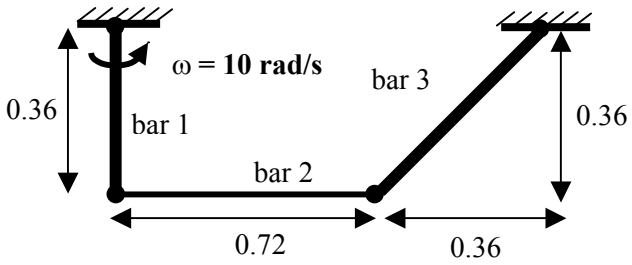


Fig.12. Successive deflections of the embedded beam

The torque is applied incrementally, the k coefficient ($0 \le k \le 2$) being assimilated to the time. The time step is equal to $\Delta k = 0{,}0002$. The successive deflections are shown on figure 12. According to the Euler classical formula [14], the deflection of the beam is a circle portion. For $\theta = \frac{ML}{EI} = 2\pi$, the deformed configuration of the beam is a closed circle.

5.2 Example 2 : Three-bar mechanism

The second example is a three-bar mechanism (Fig.13). The bars 1 and 3 are rigid. They are connected through revolution joints to the flexible bar 2. Densities and inertia are the same for the three bars. The first bar is submitted to a constant angular velocity ω. The rigid bars are discretized in two linear beam elements, while the flexible bar is discretized in four linear beam elements. The time step is equal to 0.0002. The successive deflections are shown on figure 14. The numerical results are in perfect agreement with the results obtained by Ibrahimbegovic [15].



ES = GS =1 E+6
EI = GJ = 1 E+5
Sρ = 1
Jρ = Diag(20,10,10)

Fig.13. Three-bar mechanism

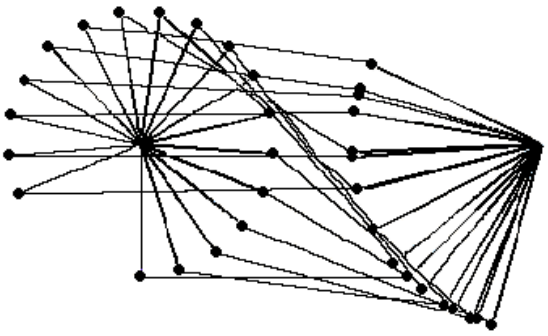


Fig.14. Successive deformed configurations of the three-bar mechanism

6 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 designed according to object-oriented principles. This approach allows us to simplify the architecture of the program and to take advantage of the inherent synergy between object-oriented design and multibody systems analysis. The flexibility of the software is made possible thanks to the clear modularization that can be reached with object-oriented programming, with a marked separation of functionalities. Extensibility and reusability of object-oriented programming are clearly shown : 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.

References:

[1] J. Mackerle, Object-oriented techniques in FEM and BEM. A bibliography (1996-1999), *Finite Element in Analysis and Design*, Vol.36, 2000, pp.89-196.

[2] R.R Ryan, ADAMS : Multibody system analysis software. In : Scheihlen W, editor, *Multibody Systems Handbook*, Berlin: Springer, 1990.

[3] P.E. Nikravesh, I.S. Chung, Application of Euler parameters for the dynamic analysis of three-dimensional constrained mechanical systems, *Journal of Mechanical Design,* Vol.104, 1982, pp.785-791.

[4] P.E. Nikravesh, G. Gim, Systematic construction of the equations of motion for multibody systems containing closed kinematic loops, *Proceedings of the ASME Design Automation Conference*, 1989.

[5] B. Meyer, *Object-Oriented Software Design*, ISBN : 0-13-629049-3 or 0-13-629031-0 PBK, Prentice-Hall, 1988.

[6] G.L. Fenves, Object-oriented programming for engineering software development, *Engineering with Computers*, Vol.6, 1990, pp.1-15.

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

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

[9] J.C. Chiou, Constraint treatment techniques and parallel algorithms for multibody dynamic analysis. *PHD Thesis, University of Colorado*, 1990.

[10] A. Ibrahimbegovic, On the choice of finite rotation parameters, *Computer Methods in Applied Mechanics and Engineering*, Vol.149, 1997, pp.49-71.

[11] K.C. Park, J.C. Chiou, J.D. Downer, Explicit-Implicit staggered procedure for multibody dynamics analysis, *Journal of Guidance, Control and Dynamics*, Vol.13, 1990, pp.562-570.

[12] A. Cardona, M. Géradin, A beam finite element non-linear theory with finite rotations, *International Journal for Numerical Methods in Engineering*, Vol.26, 1988, pp.2403-2438.

[13] F. Dufossé, Approche orientée objet appliquée à la conception d'un logiciel dédié à l'analyse des systèmes multicorps*, PHD Thesis, Université Henri Poincaré, Nancy I*, 2001.

[14] K.S. Surana, R.M. Sorem, Geometrically non-linear formulation for three dimensional curved beam elements with large rotations, *International Journal for Numerical Methods in Engineering*, Vol.28, 1989, pp.43-73.

[15] A. Ibrahimbegovic, S. Mamouri, 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, pp.805-831.