Sofa Documentation

The Sofa Team

November 22, 2007

Abstract

This is the Sofa documentation

Contents

Chapter 1

Introduction to SOFA

1.1 Brief overview

SOFA is an open-source C++ library for physical simulation, primarily targeted to medical simulation. It can be used as an external library in another program, or using one of the associated GUI applications.

The main feature of SOFA compared with other libraries is its high flexibility. It allows the use of multiple interacting geometrical models of the same object, typically, a mechanical model with mass and constitutive laws, a collision model with simple geometry, and a visual model with detailed geometry and rendering parameters. Each model can be designed independently of the others. During run-time, consistency is maintained using mappings.

Additionally, SOFA scenes are modeled using a data structure similar to hierarchical scene graphs commonly used in graphics libraries. This allows the splitting of the physical objects into collections of independent components, each of them describing one feature of the model, such as mass, force functions and constraints. For example, you can replace spring forces with finite element forces by simply replacing one component with another, all the rest (mass, collision geometry, time integration, etc.) remaining unchanged.

Moreover, simulation algorithms, such as time integration or collision detection and modeling, are also modeled as components in the scene graph. This provides us with the same flexibility for algorithms as for models.

Flexibility allows one to focus on its own domain of competence, while re-using the other's contributions on other topics. However, efficiency is a major issue, and we have tried to design a framework which allows both efficiency and flexibility.

1.2 Commented example

Figure 1.1 shows a simple scene composed of two different objets, one rigid body and one particle system, and linked by a spring. This scene is modeled and simulated in C++ as shown in section 1.6.1. The corresponding scene graph is shown in figure 1.2. Note that the graph in the left of figure 1.1 only displays a hierarchical view, while the whole graph includes additional pointers displayed as dashed arrows in figure 1.2.

The scene is modeled as a tree structure with four nodes:

- `root`
- `deformableBody` corresponds to the elastic string
- `rigidBody` corrsponds to the rigid object
- `rigidParticles` corresponds to a set of particles (only one in this case) attached to the rigid body

Each node can have children nodes and *components*. Each component implements a reduced set of functionalities.

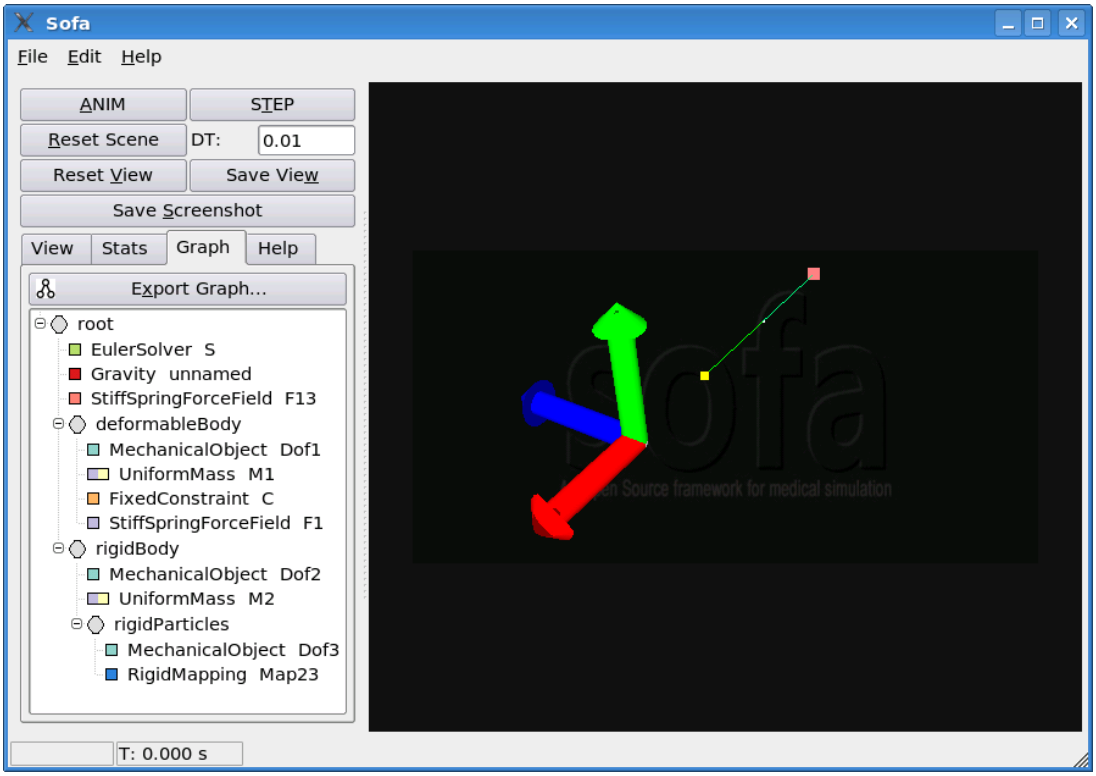

Figure 1.1: A pendulum composed of a rigid body (reference frame and yellow point) attached to an elastic string (green) fixed at one end (pink point). The corresponding scene graph is displayed on the left.

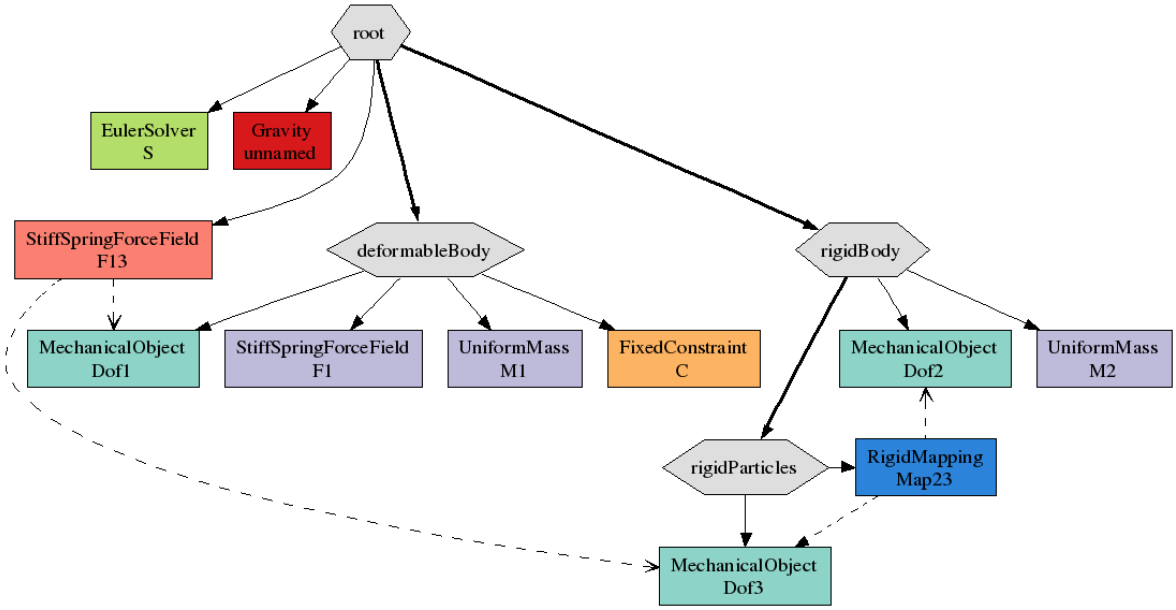

Figure 1.2: The scene graph of the mixed pendulum. The nodes are displayed as grey hexagons, while the components are displayed as rectangles with colors associated with their types or roles. The bold plain arrows denote node hierarchy, while the thin plain arrows point to the components attached to the nodes, and the dotted arrows denote pointers between components.

One of the most important type of component is the `MechanicalObject`, which contains a list of *degrees of freedom* (DOF), i.e. coordinates, velocities, and associated auxiliary vectors such as forces and accelerations. All the coordinates in a `MechanicalObject` have the same type, e.g. 3D vectors particles, or (translation, rotation) pairs for rigid bodies. `MechanicalObject`, like many other SOFA classes, is a generic (C++ template) class instantiated on the types of DOF it stores. The particle DOFs are drawn as white points, whereas the rigid body DOFs are drawn as red, green, blue reference frame axes. There can be at most one `MechanicalObject` attached to a given node. This guarantees that all the components attached to the same node process the same types of DOF. Consequently, the particles and the rigid body necessarily belong to different nodes.

In this example, the masses are stored in `UniformMass` components. The types of their values are related to the types of their associated DOF. `UniformMass` is derived from the abstract `Mass` class, and stores only one value, for the case where all the associated objects have the same mass. If necessary, it can replaced by a `DiagonalMass` instanciated on the same DOF types, for the case where the associated objects have different masses. This is an important feature of SOFA : each component can be replaced by another one deriving from the same abstract class and instanciated on the same DOF types. This results in a high flexibility.

The `FixedConstraint` component attaches a particle to a fixed point in world space, drawn in pink. The constraints act as filters which cancel the forces and displacements applied to their associated particle(s). They do not model more complex constraints such as maintaining three points aligned.

The `StiffSpringForceField` stores a list of springs, each of them modeled by a pair of indices, as well as the standard physical parameters, stiffness, damping and rest length.

The rigid body is connected to the deformable string by a spring. Since this spring is shared by the two bodies, it is modeled in the `StiffSpringForceField` attached to a common ancestor, the graph root in this example. Our springs can only connect particles. We thus need to attach a particle to the rigid body. Since the particle DOFs types are different from the rigid body DOF types, they have to be stored in another `MechanicalObject`, called `rigidParticleDOF` in this example, and attached to a different node. However, `rigidParticleDOF` is not a set of independent DOF, since they are fixed in the reference frame of the rigid body. We thus attach it to a child node of the rigid body, and connect it to `rigidDOF` using a `RigidMapping`. This component stores the coordinates of the particle in the reference frame of the rigid body. Its task is to propagate the position, velocity and displacement of the rigid body down to the yellow particle, and conversely, to propagate the forces applied to the particle up to the rigid body.

Mappings are one of the major features of SOFA . They allow us to use different geometric models for a given body, e.g. a coarse tetrahedral mesh for viscoelastic internal forces, a set of spheres for collision detection and modeling, and a fine triangular mesh for rendering.

The gravity applied to the scene is modeled in the `Gravity` component near the root. It applies to all the scene, unless locally overloaded by another gravity component inside a branch of the tree.

The abstract component classes are defined in namespace `core::componentmodel`.

So far, we have discussed the physical model of the scene. To animate it, we need to solve an *Ordinary Differential Equation* (ODE) in time. There are plenty of ODE solvers, and SOFA allows the design and the re-use of a wide variety of them. Here we use a simple explicit Euler method, modeled using an `EulerSolver` component. It triggers computations such as force acumulation, acceleration computation and linear operations on state vectors. More sophisticated solvers are available in SOFA , and can be used by simply replacing the `EulerSolver` component by another one, e.g `RungeKutta4` or `CGImplicit`.

Other capabilities of SOFA , such as collision detection and response, will be discussed in subsequent sections.

1.3 Multi-model objects

An important feature of Sofa is the possibility of using different models of a single physical object. Figure 1.3 shows a scene graph representing a liver, and three different images of it. The liver exhibits three different geometries for mechanics, rendering and collision. The corresponding xml code is given in section 1.6.2.

On top of the scene, collision-related components allow a user to interact with the collision models using rays casted from the mouse pointer and hitting collision models.

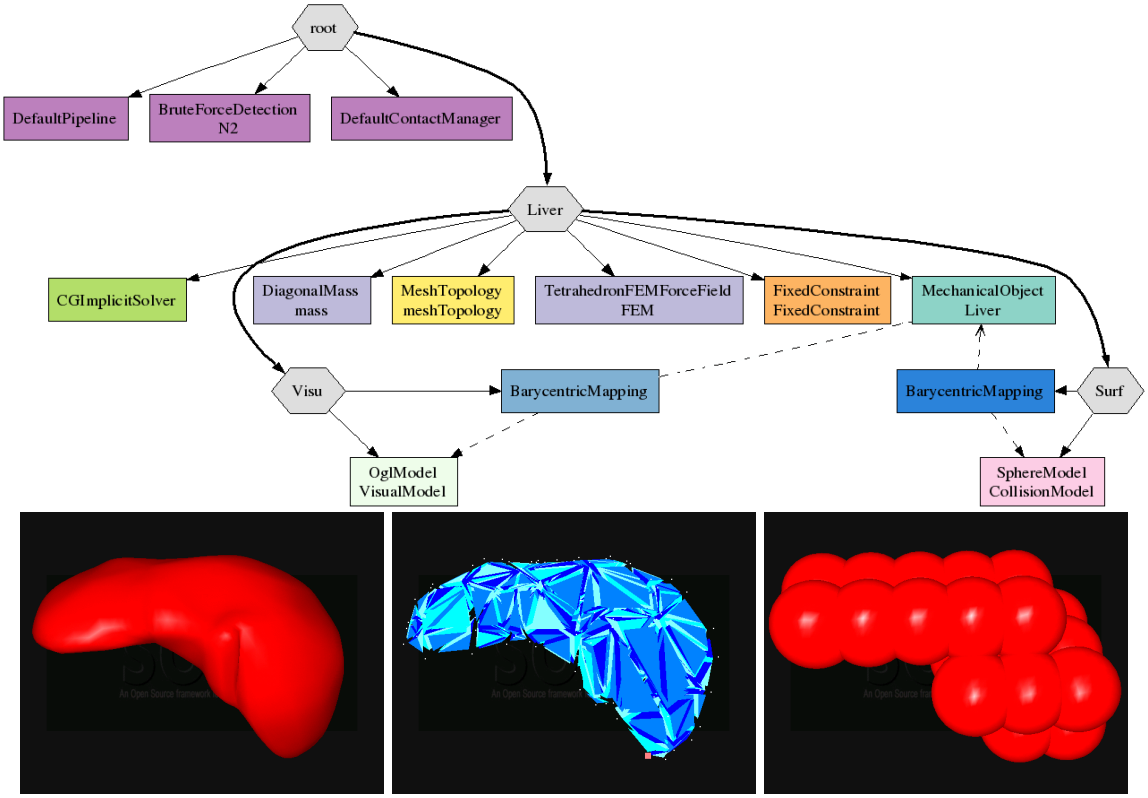

Figure 1.3: A liver. Top: scene graph. Bottom: visual model, mechanical model, collision model, respectively.

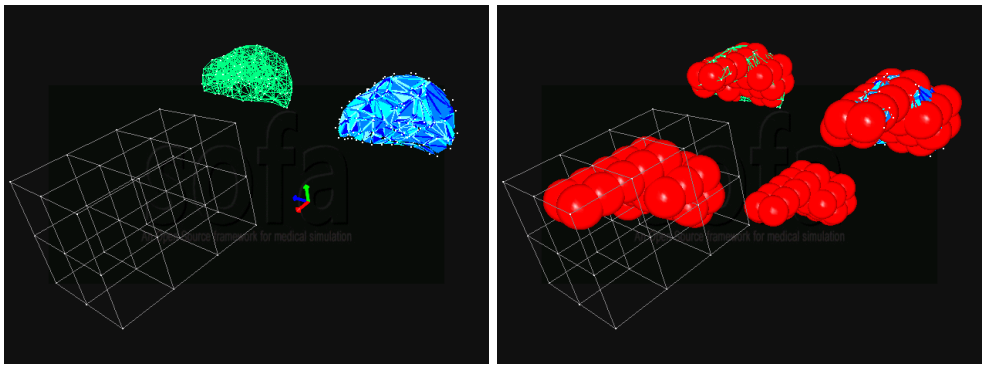

Figure 1.4: Left: four behavior models (from left to right: deformable grid, springs, rigid, tetrahedral FEM) combined with the same collision model (right).

The liver is modeled using three nodes, in two levels. The parent level contains the mechanical dofs (particle positions and velocities) in a `MechanicalObject` component. These dofs are the mechanically independent degrees of freedom of the object, in Lagrange's formalism. The node also contains components related to the dynamics of the particles, such as mass and internal forces. We call it the *behavior model*.

The two other nodes are in the lower level because during the simulation, their coordinates are totally defined by the coordinates of their parent node. Thus, they do not belong to the set of mechanically independent dofs. *Mappings* are used to compute their positions and velocities based on their parent's, using the pointers represented as dashed arrows. Mappings are not symmetric. The motion of the parent dofs is mapped to the children dofs, whereas the motion of the children dofs is not mapped to their parent. This ensures consistency.

The `VisualModel` has vertices which are used for rendering, along with other rendering data such as a list of polygons, normals, etc. The mapping is one-way and the mapped dofs have no mechanical influence.

The `SphereModel` class derives from `MechanicalObject`, with an additional radius value. It also derives from `CollisionModel`, which allows it to be processed by the collision detection and modeling pipeline. When contact or mouse interaction forces are applied to the spheres, the forces are propagated bottom-up to their parent dofs by the mapping. This allows the contact forces to be taken into account in the dynamics equations. The mapping is thus two-ways and derives from `MechanicalMapping` instead of `Mapping`. This is why it has a different color in the image of the scene graph. Again, the mechanical mappings are not symmetric: the forces are propagated from the children to the parents, not the other way round.

Mappings only propagate positions top-down, whereas MechanicalMappings additionally propagate velocities top-down and forces bottom-up.

Mapped models can be designed independently of their parent models, provided that the adequate (mechanical) mapping is available. This results in a high flexibility. For example, collision spheres can be replaced by collision triangles without changing anything in the behavior model or in the visual model. Similarly, other visual models can be used without modifying the behavior and collision models, and different behavior models can be used with the same collision and visual models, as illustrated in figure 1.4.

1.4 Recursive data processing

A typical simulation program, controlled by an application such as the Graphics User Interface (GUI), looks like the one given in figure 1.5. In SOFA , each of the simulation methods is implemented as a recursive graph traversal, `InitVisitor`, `AnimateVisitor` and `VisualDrawVisitor`, respectively. Visitors are explained in the next section.

```
init();
repeat {
    animate();
    draw();
}
```

Figure 1.5: Pseudocode for a standard simulation program.

```
f = 0
accumulateForces(f,x,v);
a = f/M;
a = filter(a);
x += v * dt;
v += a * dt;
```

Figure 1.6: Pseudocode for explicit Euler integration.

1.4.1 Visitors

The data structure is processed using visitors called *visitors*. They recursively traverse the tree structure and call appropriate virtual methods to a subset of components during the *Top-Down Traversal* (TDT), using virtual method `Visitor::processNodeTopDown`, then during the *Bottom-Up Traversal* (BUT), using virtual method `Visitor::processNodeBottomUp`.

For example, the `VisualDrawVisitor` draws the `VisualModel` components during the TDT, and does nothing during the BUT. The `MechanicalComputeForceVisitor` accumulates the forces in the appropriate dof vectors during the TDT, then propagates the forces to the parent dofs using the mechanical mappings during the BUT.

When processed by a visitor$a$, a component can fire another visitor b through its associated sub-tree. Visitor a can continue once visitor b is finished. During the TDT, each traversed component decides wether the calling visitor continues, or prunes the sub-tree associated with the component, or terminates.

The components directly access their sibling components only, except for the mappings. A component traversed by a visitor can indirectly access the data in its associated sub-tree in read-write mode using visitors, whereas data in its parent graph is read-only and only partially accessible using method `getContext`. Sibling nodes of the same type can be traversed by visitors in arbitrary order.

The visitors belong to namespace `simulation::tree`.

1.4.2 ODE Solvers

When an `AnimateVisitor` traverses a node with an `OdeSolver` component, the solver takes the control of its associated subtree and prunes the `AnimateVisitor`. The solver triggers visitors in its associated subtree to perform the standard mechanical computations and integrate time.

The simplest solver is the explicit Euler method, implemented in `EulerSolver`. The algorithm is shown as pseudocode in figure 1.6. Net force is computed in the first line. In the second line, the acceleration is deduced by dividing the force by the mass. Then the accelerations of the fixed points are canceled. Finally, position and velocity are updated.

This algorithm can not be directly implemented in SOFA because there are no state vectors x,v,f,a which gather the state values of all the objects in the scene. The solver processes an arbitrary number of objects, of possibly different types, such as particles and rigid bodies. Each physical object carries its state values and auxiliary vectors in its own `MechanicalObject` component, which is not directly accessible to the solver.

The solvers represent state vectors as `MultiVector` objects using symbolic identificators implemented in class `VecId`. There are four staticly predefined identificators: `VecId::position()`, `VecId::velocity()`, `VecId::force()` and `VecId::dx()`. A `Multivector` declared by a solver with a given VecId implicitly refers to all the state vectors in the different `MechanicalObject` components with the same `VecId` in the solver's subtree.

Vector operations can be remotely triggered by a solver using a visitor of a given type, which defines the operator, and given `VecId`s, which define the operands. During the subtree traversal, the operator is

applied to the given vectors of the traversed MechanicalObject components.

For example, let us comment the visitors performed by the EulerSolver shown in figure 1.1. Its implementation is in method component::odesolver::EulerSolver::solve(double). First, multivectors are declared.

Then method core::componentmodel::behavior::OdeSolver::computeForce(VecId) is called. It first fires a MechanicalResteForceVisitor to reset the force vectors of all the MechanicalObject components. It then fires a MechanicalComputeForceVisitor. During the TDT, each component derived from core::componentmodel::behavior::BaseForceField computes and accumulates its force in its sibling MechanicalObject. In the example shown in figure 1.1, F13 adds its contribution to Dof1 and Dof2, then F1 and M1 add their contributions to Dof1, then M2 to Dof2. Then during the BUT, the mechanical mappings sum up the forces of their chid dof to their parent dof, *i.e* the force in Dof3 to Dof2 through M23 in the same example. Note that branches deformableBody and rigidBody can be processed in parallel. At the end, the force vector in Dof1 contains the net force applied to the particles, and the force vector in Dof2 contains the net (six-dimensional) force applied to the rigid body.

Then method OdeSolver::accFromF(VecId,VecId) fires a MechanicalAccFromFVisitor. Each component derived from core::componentmodel::behavior::BaseMass computes the accelerations corresponding to the forces in its sibling MechanicalObject.

Then method OdeSolver::projectResponse fires a MechanicalApplyConstraintsVisitor. All the core::componentmodel::behavior::BaseConstraint components (component C in the example) filter the acceleration vector to maintain some points fixed.

Once the acceleration is computed, multivector methods are used to update the positions and velocities. Here again, visitors are used to perform the desired operation in each traversed MechanicalObject.

MultiVector operations are pruned at the first level for efficiency, because the solvers deal with the mechanically independent state variables rather than the mapped variables. Moreover, the mapped coordinates can not be assumed to vary linearly along with their parent variables. Applying a MechanicalPropagatePositionAn is thus necessary to update the mapped dofs based on the mechanically independent dofs. This visitor is automatically performed after time integration, as one can see in the code of method MechanicalIntegrationVisitor::fwdOdeSolver. It is also used by some solvers when auxiliary states are needed, as discussed in section 1.5, in order to update the mapped dofs.

1.5 State vectors

The state vectors contain the coordinates, velocites, and other dof-related values such as force and acceleration. They are stored in MechanicalObject components. This template class can be instanciated on a variety of types to model particles, rigid bodies or other types of bodies. The template parameter is a DataTypes class which describes data and data containers, such as the the type of coordinates and coordinate derivatives used. These two types are the same in the case of particles, but they are different in the case of rigid bodies.

Each MechanicalObject can represent a set of physical objects of the same type, such as particles. The coordinate state vectors are defined by the VecCoord type, while the derivatives (velocity, acceleration, force, small displacement) are defined by the VecDeriv type. Each MechanicalObject stores two arrays of state vectors, one for coordinates and the other for derivatives, as illustrated in figure 1.7.

Auxiliary vectors are necessary for complex solvers, such as RungeKutta2Solver. This solver first performs a half-length Euler step, then evaluates the derivative of this new state (called the *midpoint*), and finally uses this derivative to update the initial state over a whole time step.

To compute the forces at the midpoint while keeping the initial state for further use, we use the auxiliary vectors newX and newV. However, components such as forces and constraints use state vectors, and we have to make sure that they use the right ones. To ensure consistency and make the use of auxiliary states transparent, the other components get access to the state vectors using methods MechanicalObject::getX(), getV(), getF() and getDx(). These methods return pointers to the appropriate vectors, as illustrated in figure 1.7.

Internal MechanicalObject switches are performed by methods MechanicalObject::setX(), setV(), setF() and setDx(). These methods are applied by the visitors which take multivectors as parameters,

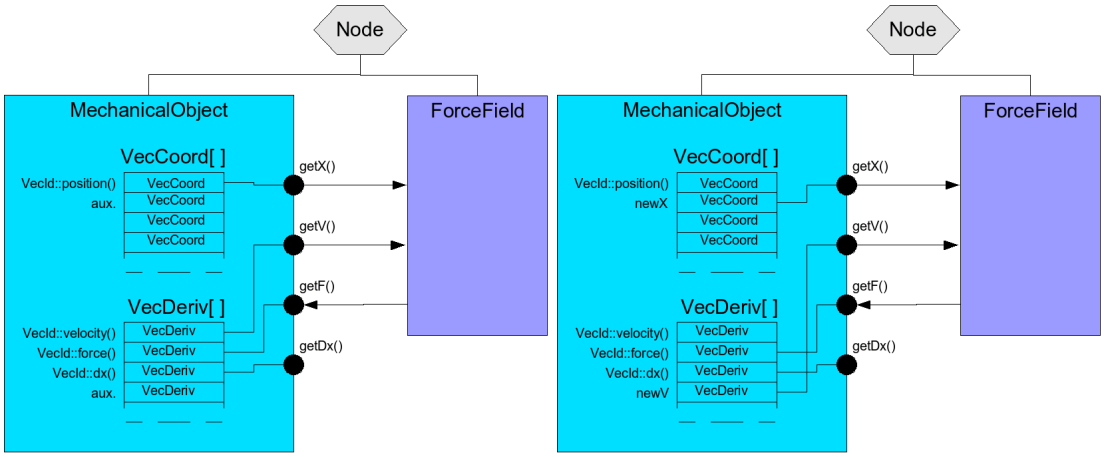

Figure 1.7: A `MechanicalObject` and a component addressing it. Left: using the default state vectors. Right: using auxiliary state vectors.

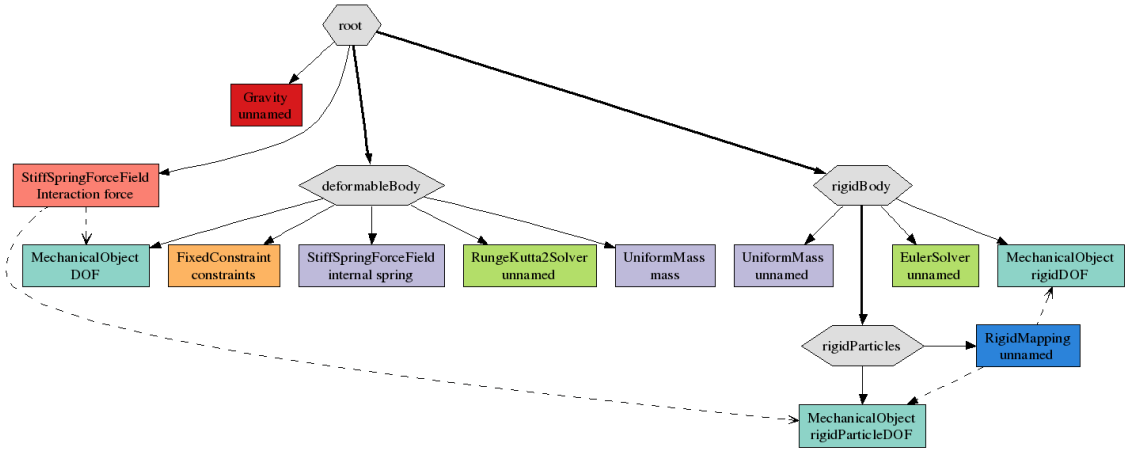

Figure 1.8: A scene graph with objects animated using different ODE solvers.

before they use other components. See, for example, method `MechanicalPropagatePositionAndVelocityVisitor::fwdMechanicalState`.

Note that some constraint-based animation methods require large state vectors and matrices encompassing all the mechanical objects of the scene. Such methods are currently under development in SOFA , and they are not yet documented. They use visitors to count the total number of scalar dofs and to gather them in large state vectors, as well as to build mechanical matrices such as mass, stiffness, damping and compliance etc.

1.5.1 Mechanical groups

During the simulation, each solver prunes the `AnimateVisitor` which traverses it and manages its associated subtree by itself using other visitors. We call *mechanical group* the objects animated by a given solver. Each mechanical group corresponds to a subtree in the scene graph. In the example discussed in section 1.2, there is one mechanical group because a single solver located near the root manages the whole scene. However, using separate solvers for different objects can sometimes increase efficiency. In the example shown in figure 1.8, the same deformable body is animated using a `RungeKutta2Solver` while the rigid body is animated using an `EulerSolver`.

A mechanical group can include interaction forces between elements of the group, and such interaction forces are handled by the solver as expected. Interaction forces can also occur between objects which

do not belong to the same group. In this case, the interaction force is located at a higher hierarchical level than the objects it applies to, as shown in figure 1.8. It can not be traversed by visitors fired by the solvers. Its evaluation is performed by the `AnimateVisitor`, and accumulated as external forces in the associated `MechanicalObject` components. Consequently, it acts as a constant constant force during each whole animation step. In a `RungeKutta2Solver`, during the force computation at midpoint, its value is the same as at the starting point. In a `CGImplicitSolver`, its stiffness is not taken into account, which may introduce instabilities if its actual stiffness is high.

The default collision manager of Sofa circumvents this problem by dynamically gathering the objects in contact in a common mechanical group.

1.6 Code of the examples

1.6.1 The hybrid pendulum

This is the code of the example commented in section 1.2 :

```
/** A sample program. Laure Heigeas, Francois Faure, 2007. */
// scene data structure
#include <sofa/simulation/tree/Simulation.h>
#include <sofa/component/contextobject/Gravity.h>
#include <sofa/component/odesolver/CGImplicitSolver.h>
#include <sofa/component/odesolver/EulerSolver.h>
#include <sofa/component/odesolver/StaticSolver.h>
#include <sofa/component/MechanicalObject.h>
#include <sofa/component/mass/UniformMass.h>
#include <sofa/component/constraint/FixedConstraint.h>
#include <sofa/component/forcefield/StiffSpringForceField.h>
#include <sofa/component/mapping/BarycentricMapping.h>
#include <sofa/component/visualmodel/OglModel.h>
#include <sofa/defaulttype/RigidTypes.h>
#include <sofa/component/mapping/RigidMapping.h>
// gui
#include <sofa/gui/SofaGUI.h>

using sofa::simulation::tree::GNode;
typedef sofa::component::odesolver::EulerSolver OdeSolver;
using sofa::component::contextobject::Gravity;

// deformable body
typedef sofa::defaulttype::Vec3Types ParticleTypes;
typedef ParticleTypes::Deriv Vec3;
typedef sofa::core::componentmodel::behavior::MechanicalState<ParticleTypes> ParticleStates;
typedef sofa::component::MechanicalObject<ParticleTypes> ParticleDOFs;
typedef sofa::component::mass::UniformMass<ParticleTypes,double> ParticleMasses;
typedef sofa::component::constraint::FixedConstraint<ParticleTypes> ParticleFixedConstraint;
typedef sofa::component::forcefield::StiffSpringForceField<ParticleTypes> ParticleStiffSpringForceField;
typedef sofa::component::visualmodel::GLExtVec3fTypes OglTypes;
typedef sofa::core::componentmodel::behavior::MappedModel<OglTypes> OglMappedModel;

// rigid body
typedef sofa::defaulttype::StdRigidTypes<3,double> RigidTypes;
typedef RigidTypes::Coord RigidCoord;
typedef RigidTypes::Quat Quaternion;
typedef sofa::defaulttype::RigidMass<3,double> RigidMass;
typedef sofa::component::mass::UniformMass<RigidTypes,RigidMass> RigidUniformMasses;
typedef sofa::core::componentmodel::behavior::MechanicalState<RigidTypes> RigidStates;
typedef sofa::component::MechanicalObject<RigidTypes> RigidDOFs;
typedef sofa::core::componentmodel::behavior::MechanicalMapping<RigidStates, ParticleStates >
    RigidToParticleMechanicalMapping;
typedef sofa::component::mapping::RigidMapping< RigidToParticleMechanicalMapping >
    RigidToParticleRigidMechanicalMapping;

int main(int, char** argv)
{
    sofa::gui::SofaGUI::Init(argv[0]);
    //========================== Build the scene
    double endPos = 1.;
    double attach = -1.;
    double splength = 1.;

    //-------------------- The graph root node
    GNode* groot = new sofa::simulation::tree::GNode;
    groot->setName( "root" );

    // One solver for all the graph
    OdeSolver* solver = new OdeSolver;
    groot->addObject(solver);
    solver->setName("S");

    //-------------------- Deformable body
    GNode* deformableBody = new GNode;
    groot->addChild(deformableBody);
    deformableBody->setName( "deformableBody" );

    // degrees of freedom
    ParticleDOFs* DOF = new ParticleDOFs;
    deformableBody->addObject(DOF);
    DOF->resize(2);
    DOF->setName("Dof1");
    ParticleTypes::VecCoord& x = *DOF->getX();
    x[0] = Vec3(0,0,0);
    x[1] = Vec3(endPos,0,0);
```

```
    // mass
    ParticleMasses* mass = new ParticleMasses;
    deformableBody->addObject(mass);
    mass->setMass(1);
    mass->setName("M1");

    // Fixed point
    ParticleFixedConstraint* constraints = new ParticleFixedConstraint;
    deformableBody->addObject(constraints);
    constraints->setName("C");
    constraints->addConstraint(0);

    // force field
    ParticleStiffSpringForceField* spring = new ParticleStiffSpringForceField;
    deformableBody->addObject(spring);
    spring->setName("F1");
    spring->addSpring( 1,0, 10., 1, splength );


    //-------------------- Rigid body
    GNode* rigidBody = new GNode;
    groot->addChild(rigidBody);
    rigidBody->setName( "rigidBody" );

    // degrees of freedom
    RigidDOFs* rigidDOF = new RigidDOFs;
    rigidBody->addObject(rigidDOF);
    rigidDOF->resize(1);
    rigidDOF->setName("Dof2");
    RigidTypes::VecCoord& rigid_x = *rigidDOF->getX();
    rigid_x[0] = RigidCoord( Vec3(endPos-attach+splength,0,0), Quaternion::identity() );

    // mass
    RigidUniformMasses* rigidMass = new RigidUniformMasses;
    rigidBody->addObject(rigidMass);
    rigidMass->setName("M2");


    //-------------------- the particles attached to the rigid body
    GNode* rigidParticles = new GNode;
    rigidParticles->setName( "rigidParticles" );
    rigidBody->addChild(rigidParticles);

    // degrees of freedom of the skin
    ParticleDOFs* rigidParticleDOF = new ParticleDOFs;
    rigidParticles->addObject(rigidParticleDOF);
    rigidParticleDOF->resize(1);
    rigidParticleDOF->setName("Dof3");
    ParticleTypes::VecCoord& rp_x = *rigidParticleDOF->getX();
    rp_x[0] = Vec3(attach,0,0);

    // mapping from the rigid body DOF to the skin DOF, to rigidly attach the skin to the body
    RigidToParticleRigidMechanicalMapping* rigidMapping = new
        RigidToParticleRigidMechanicalMapping(rigidDOF,rigidParticleDOF);
    rigidParticles->addObject( rigidMapping );
    rigidMapping->setName("Map23");


    // ----------------- Interaction force between the deformable and the rigid body
    ParticleStiffSpringForceField* iff = new ParticleStiffSpringForceField( DOF, rigidParticleDOF );
    groot->addObject(iff);
    iff->setName("F13");
    iff->addSpring( 1,0, 10., 1., splength );

    // Set gravity for the whole graph
    Gravity* gravity =  new Gravity;
    groot->addObject(gravity);
    gravity->f_gravity.setValue( Vec3(0,-10,0) );



    //============================ Init the scene
    sofa::simulation::tree::getSimulation()->init(groot);
    groot->setAnimate(false);
    groot->setShowNormals(false);
    groot->setShowInteractionForceFields(true);
    groot->setShowMechanicalMappings(true);
    groot->setShowCollisionModels(false);
    groot->setShowBoundingCollisionModels(false);
    groot->setShowMappings(false);
    groot->setShowForceFields(true);
    groot->setShowWireFrame(false);
    groot->setShowVisualModels(true);



    //============================ Run the main loop

    sofa::gui::SofaGUI::MainLoop(groot);
}
```

1.6.2 A liver

This is the XML code of the liver discussed in section 1.3 page 4 :

```
<Node name="root" dt="0.02" showBehaviorModels="0" showCollisionModels="0" showMappings="0"
      showForceFields="0">
    <Object type="Simulation" name="Simulation" />
    <Object type="CollisionPipeline" verbose="0" />
    <Object type="BruteForceDetection" name="N2" />
    <Object type="CollisionResponse" response="default" />
    <!--<Object type="CollisionGroup" />-->
```

```
    <Node name="Liver">
        <Object type="CGImplicit" iterations="25" />
        <Object type="MechanicalObject" template="Vec3f" name="Liver" filename="BehaviorModels/liver.xs3" />
        <Object type="DiagonalMass" name="mass" filename="BehaviorModels/liver.xs3" />
        <Object type="Mesh" name="meshTopology" filename="Topology/liver.mesh" />
        <Object type="TetrahedronFEMForceField" name="FEM" youngModulus="500" poissonRatio="0.3"
              computeGlobalMatrix="false" method="large" />
                <Object type="FixedConstraint" name="FixedConstraint" indices="3 39 64" />
        <Node name="Visu">
            <Object type="OglModel" name="VisualModel" filename="VisualModels/liver-smooth.obj" color="red" />
            <Object type="BarycentricMapping" object1="../.." object2="VisualModel" />
        </Node>
        <Node name="Surf">
            <Object type="Sphere" name="CollisionModel" filename="CollisionModels/liver.sph" />
            <Object type="BarycentricMapping" />
        </Node>
    </Node>
    <!--<Object type="Sphere" name="Floor" filename="CollisionModels/floor.sph" static="1" dy="-2.5" />
    <Object type="OglModel" name="FloorV" filename="VisualModels/floor.obj"
          texturename="VisualModels/floor.bmp" dy="-2.5" />-->
</Node>
```

Chapter 2

Design

2.1 Framework

2.1.1 UML Diagrams

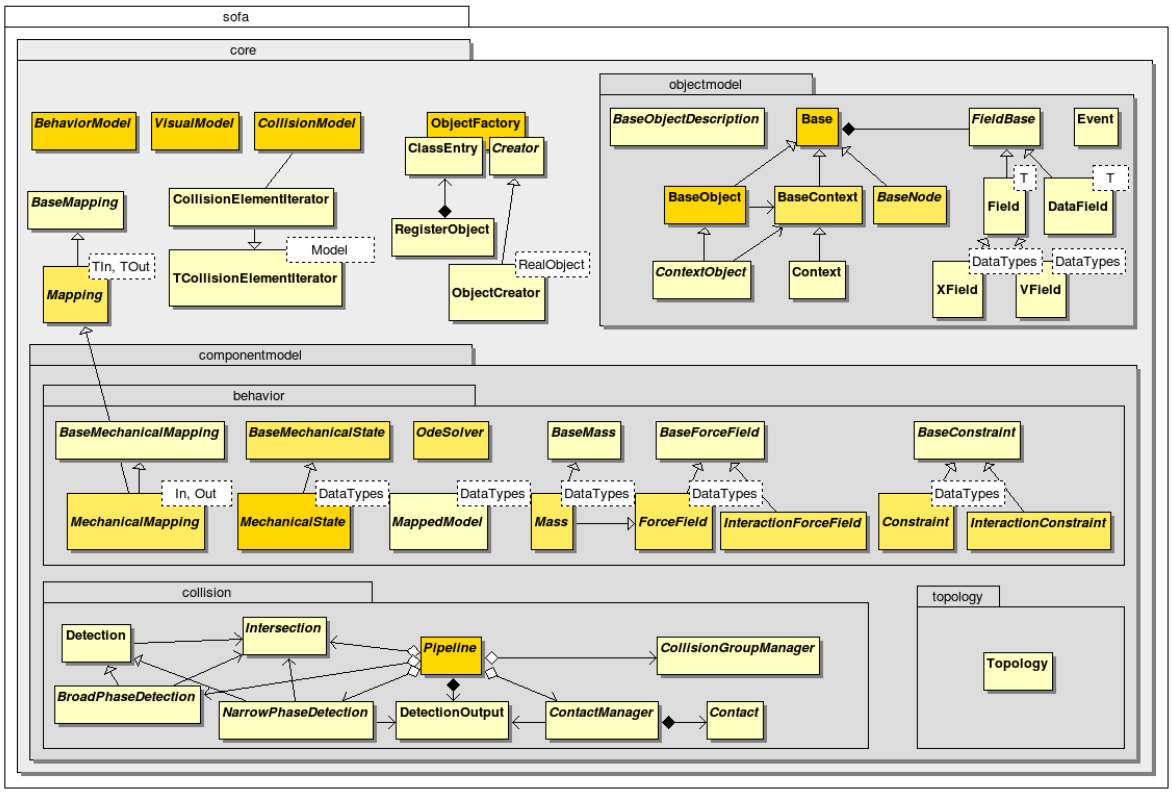

Figure 2.1: Classes of the sofa::core namespace.

Chapter 3

Modules

In this document we explain the usage and the fucntionalities of the modules developped using the Core Sofa Framework.

3.1 Collision Models

3.1.1 Ray Traced Collision Detection

This module implements the algorithm described in the paper entitled "Ray-traced collision detection for deformable bodies" by E.Hermann F.Faure and B.Raffin. When two objects are in collision, a ray is shot from each surface vertex in the direction of the inward normal. A collision is detected when the first intersection belongs to an inward surface triangle of another body. A contact force between the vertex and the matching point is then created. Experiments show that this approach is fast and more robust than traditional proximity-based collisions.

To speedup the searching of elements that cross the ray, we stored all the triangles of each colliding objects in an octree. Therefore we can easily navigate inside this octree and efficiently find the points crossing the ray. The octree structure allow us to have a satisfying performance independently from the size of the triangles used, which is not the case for a regular grid.

Using this module

An exemple showing the usage of the Ray Traced collision detection can be found in the RayTraceCollision.scn file in the *scene* directory. The collision detection mechanism must be set as **RayTraceDetection**, and instead of using a TriangleModel one must use a **TriangleOctreeModel**. The TriangleOctreeModel will create an Octree that contains all the Triangles from the collision model.

3.2 Soft Articulations

3.2.1 Concepts

The objective of this method is to use stiff forces to simulate joint articulations, instead of classical constraints.

To do this, a joint is modeled by a 6 degrees of freedom spring. By the way, the user specify a stiffness on each translation and rotation.

- A null stiffness defines a free movement.
- A huge stiffness defines a forbidden movement.
- All nuances are possible to define semi constrained movements.

2 main advantages can be extracted from this method :

- A better stability. As we don't try to statisfy constraints but only apply forces, there is always a solution to resolve the system.
- more possibilities to model articulations are allowed. As the stiffnesses define the degrees of freedom of the articulations, a better accuracy is posssible to simulate free movements as forbidden movements, i.e. an articulation axis is not inevitably totally free or totally fixed.

3.2.2 Realization

To define physically an articulated body, we first have a set of rigids (the bones). *cf fig. 1*

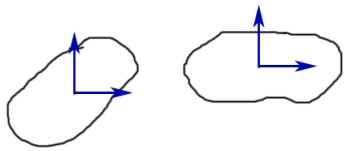

Figure 3.1: two bones

Each of these bones contains several articulations points, also defined by rigids to have orientation information. *cf fig. 2*

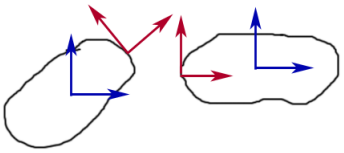

Figure 3.2: two bones (blue) with their articulation frames (red)

As seen previously, a joint between 2 bones is modeled by a 6-DOF spring. These springs are attached on the articulations points. *cf fig. 3*

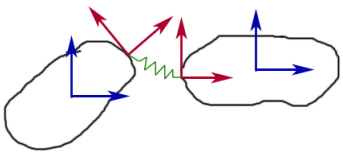

Figure 3.3: two bones linked by a joint-spring

3.2.3 Sofa implementation

To simulate these components in Sofa, we first need 2 mechanical objects : one for the bones (independent DOFs), and an other for the articulation points (mapped DOFs). Each of them contains a list of rigid DOFs (respectively all the bones and all the articulations of the articulated body). A mapping performs the link between the two lists, to know which articulations belong to which bones.

Corresponding scene graph

Example

The example softArticulations.scn shows a basic pendulum :

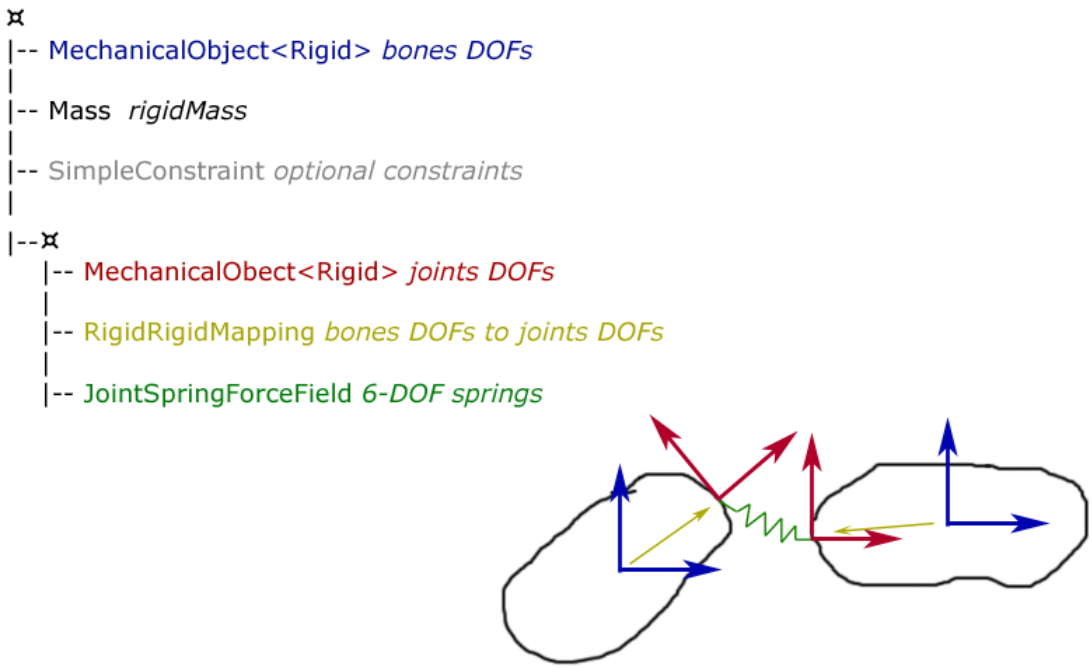

Figure 3.4: a simple articulated body scene

```
<Node>
  <Object type="BruteForceDetection"/>
  <Object type="DefaultContactManager"/>
  <Object type="DefaultPipeline"/>
  <Object type="ProximityIntersection"/>

  <Node>
    <Object type="CGImplicitSolver" />
    <Object type="MechanicalObject" template="Rigid" name="bones DOFs"
            position="0 0 0  0 0 0 1
                      1 0 0  0 0 0 1
                      3 0 0  0 0 0 1
                      5 0 0  0 0 0 1
                      7 0 0  0 0 0 1" />
    <Object type="UniformMass" template="Rigid" name="bones mass"
            mass="1 1 [1 0 0,0 1 0,0 0 1]" />
    <Object type="FixedConstraint" template="Rigid" name="fixOrigin"
            indices="0" />

    <Node>
      <Object type="MechanicalObject" template="Rigid" name="articulation points"
              position="0 0 0  0.707914 0 0 0.707914
                       -1 0 0  0.707914 0 0 0.707914
                        1 0 0  0.707914 0 0 0.707914
                       -1 0 0  0.707914 0 0 0.707914
                        1 0 0  0.707914 0 0 0.707914
                       -1 0 0  0.707914 0 0 0.707914
                        1 0 0  0.707914 0 0 0.707914
                       -1 0 0  0.707914 0 0 0.707914
```

```
                    1 0 0  0.707914 0 0 0.707914" />
      <Object type="RigidRigidMapping"
              repartition="1 2 2 2 2" />
      <Object type="JointSpringForceField" template="Rigid" name="joint springs"
              spring="0 1   0 0 0 0 1 0   0 30000  0 200000   0  0 0 0  0 0 0 1
                      2 3   0 0 0 0 1 0   0 30000  0 200000   0  0 0 0  0 0 0 1
                      4 5   0 0 0 0 1 0   0 30000  0 200000   0  0 0 0  0 0 0 1
                      6 7   0 0 0 0 1 0   0 30000  0 200000   0  0 0 0  0 0 0 1" />
    </Node>
    <Node>
      <Object type="MechanicalObject" template="Vec3d"
              position="-1 -0.5 -0.5  -1 0.5 -0.5 ..." />
      <Object type="MeshTopology"
              lines="0 1  1 2  ..."
              triangles="3 1 0  3 2 1  ..." />
      <Object type="TriangleModel"/>
      <Object type="LineModel"/>
      <Object type="RigidMapping"
              repartition="0 8 8 8 8" />
    </Node>
  </Node>
</Node>
```

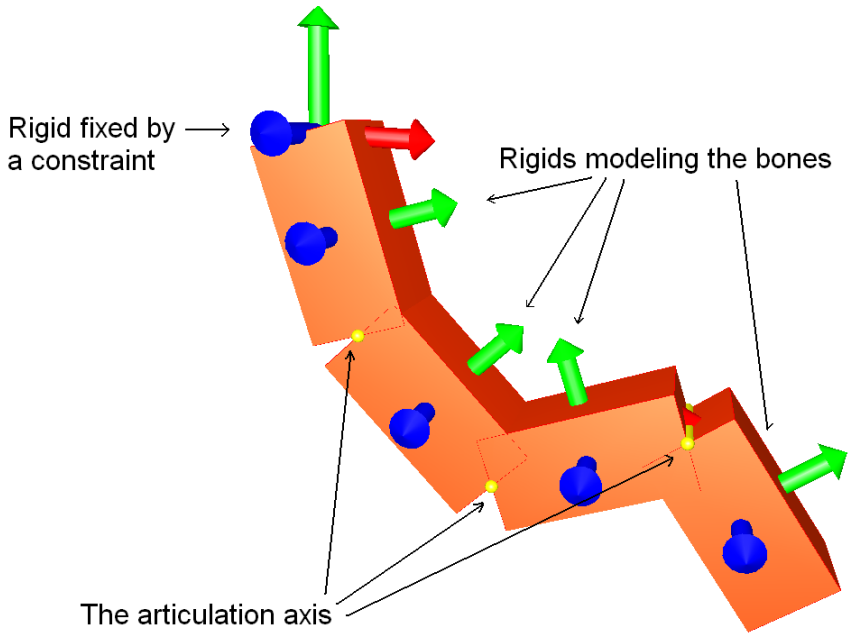

Figure 3.5: The pendulum is composed by 4 rigids linked one by one by articulations

In this example, we have under the first node the components to manage collisions, as usual. Under the second node, we have :

- the solver,
- the mechanical object modeling the independent rigid DOFs (5 rigids here),
- the rigid mass,

- a constraint, to fix the first rigid.

The third node (a child of the previous one) contains the components relative to the articulations :

- the mechanical object modeling articulation points. Positions and orientations are relative to their parents.
- the mapping to link the two mechanical objects, as explained before. To know which articulations belong to which bones, a repartition vector is used. Several cases for this vector are possible :
  - no value specified : every articulations belong to the first bone (classic rigid mapping).
  - one value specified (ex: repartition="2") : each bone has the same number of articulations.
  - number of bones values (like here, repartition="1 2 2 2 2") : the number of articulations is specified for each bone. For instance, here the first bone has 1 articulation, the next has 2 articulations, the next 2, Etc.
- the JointSpringForceField containing the springs (4 springs here). Each spring is defined by a list of parameters. For instance for the first spring we have "0 1 0 0 0 0 1 0 0 30000 0 200000 0 0 0 0 0 0 0 1".
  - "0 1" are the indices of the two articulations the spring is attached to
  - "0 0 0 0 1 0" design the free axis for the movements. "0 0 0" mean that the 3 translation axis are constrained, and "0 1 0" mean that only the Y rotation axis is free.
  - "0 30000 0 2000000" are the stiffnesses for each kind of movement: "0 30000" are respectively for free translation and for constrained translation", and "0 2000000" are respectively for free rotation and for constrained rotation.
  - "0" is the damping factor
  - "0 0 0" is to specify the initial translation
  - "0 0 0 1" is to specify the initial rotation (quaternion)

The last node contains the collision model. Nothing special here.

3.2.4 Skinning

The articulated body described previously models the skeleton of an object. To have the external model (for the visual model or the collision model), which follows correctly the skeleton movements, it has to be mapped with the skeleton. A skinning mapping allows us to do this link. The external model is from this moment to deform itself smoothly, i.e. without breaking points around the articulations.

The influence of the bones on each point of the external model is given by skinning weights. 2 ways are possible to set the skinning weights to the mapping :

- Either the user gives directly the weights list to the mapping. It is useful if good weights have been pre computed previsouly, like in Maya for instance.
- Else, the user defines a number of references n that will be used for mapped points. Then, each external model point will search its n nearest bones (mechanical DOFs), and then compute the skinning weights from the relation :

$$W = \frac{1}{d^2}$$

with d : the distance between the external point and the rigid DOF.

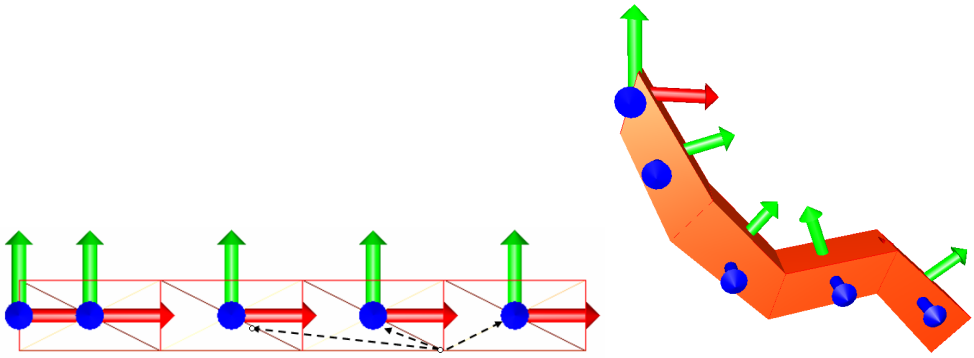

Figure 3.6: In the example "softArticulationsSkinned.scn" the external points compute their skinning weights from the 3 nearest DOFs

3.3 How to use mesh topologies in SOFA

H. Delingette, B. André

3.3.1 Introduction

BTW, What is a mesh topology ?

A mesh is usually described as a set of points that are connected by edges, triangles or any other type of mesh element. Thus it is useful to make a clear a distinction between two different aspects of a mesh :

- **Mesh Geometry** : the mesh geometry consists in the position of the mesh vertices. This information depends on the space where the mesh in embedded. For instance in a 2D triangulation, each vertex position is a 2D vector while in a 3D triangulation, each vertex position is a 3D vector. Therefore the mesh geometry can be described as an array of vector whose size is the number of mesh vertices. In SOFA, we use the word Degree Of Freedom (DOF) to describe such an array because it can be used to store other geometric information (rigid transformation, first or second derivatives, etc.).
- **Mesh Topology** : the mesh topology describes how the vertices are connected with each other. For instance, it describes the set of triangles by specifying the 3 vertex indices that make each triangle. A mesh topology manipulates vertex indices (as unsigned int) and therefore is independent of the embedding space. For instance, a 2D and a 3D triangulation may have the same mesh topology but with different mesh geometry.

Why do I need to bother with mesh topologies ?

As discuss above, mesh topology is an essential part of a mesh and therefore any computation task that requires a mesh needs to know how to use a mesh topology.
This includes:

- **Mesh Visualization**,
- **Collision detection** : some collision detection are mesh based (e.g. triangles or edges),
- **Mechanical Modeling** : deforming a mesh also requires to the knowledge of a mesh topology. For instance a spring mass model requires knowing about the edges that connects pair of vertices,
- **Haptic rendering**,
- **Description of scalar** (temperature, electric potential, etc.) or vectorial fields (speed, fiber orientation, etc.)

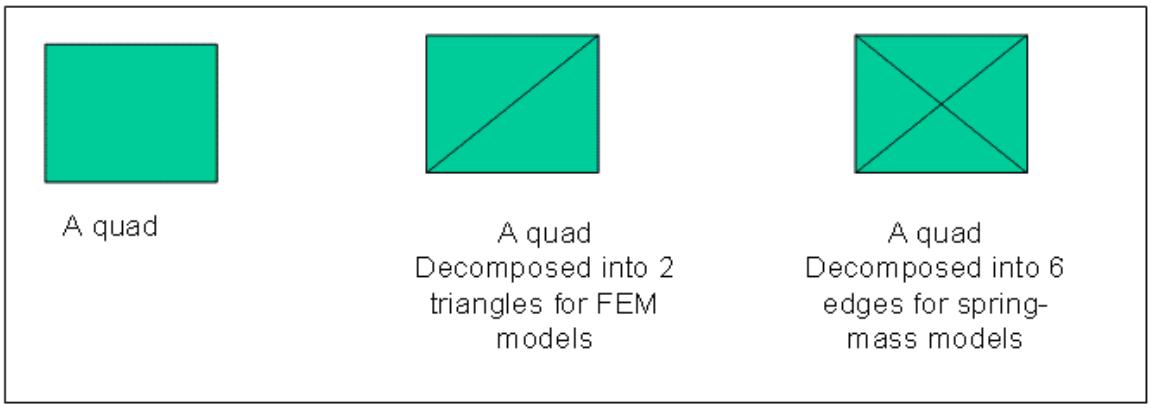

Figure 3.7: Multiple topology descriptions of the Quad.

Using a mesh topology is relatively simple since it consists in having access to arrays of indices corresponding to vertex indices or edge indices or other topological items.

A more tricky part consists a) in changing locally or globally this topology (adding a triangle, removing an edge) and b) in propagating those changes to all objects using the mesh topology to perform a task (visualization, deformation, etc.)

How are Mesh Topologies designed in SOFA ?

The mesh geometry in SOFA is stored in a MechanicalObject which is a template class because it depends on the embedding space (2D or 3D Euclidian space), the vector class and the required floating point accuracy (float vs double).

A mesh topology is stored in a different object than the mesh geometry.

One important aspect of the design of mesh topologies in SOFA is the fact that they are organized in a class hierarchy. For instance, a triangulation object derives from an edge set object since a triangulation can be also viewed as a set of edges, each triangle having 3 edges. This is very important to design generic software components. Indeed, following the same example, with this design, a spring mass mechanical model that only requires the knowledge of edges (pairs of vertices) can also be used on a triangulation or any other mesh (quad, hexahedral, tetrahedral mesh) that derives from an edge set object.

Another interesting feature in SOFA is the ability to provide multiple topology descriptions for the same mesh. For instance a quad element (see figure below) has four DOFs which can be connected with 2 triangles or 6 edges. Thus, the same mesh geometry can be described by 3 different mesh topologies. SOFA uses the mechanism of topological mapping to provide multiple topologies associated with the same mesh geometry. Those mappings also apply to map a subset of the mesh topology into a new mesh topology. For instance the border of a tetrahedral mesh can be mapped into triangulation mesh or edges of a triangulated mesh can be mapped into a polygonal mesh.

Another important aspect of the design is the fact that topological changes (mesh cutting or refinement) are handled in SOFA. For the programmer, it implies that specific containers must be used to store data for each software component. For instance, a spring mass model must store the spring stiffness of each edge. Therefore the container of spring stiffness must have the same size than the number of edges in the mesh. In SOFA, to cope with topological changes that can add or remove the number of edges, it is mandatory to use a specific container (in such case EdgeData container) that will automatically resize itself when topological changes occur.

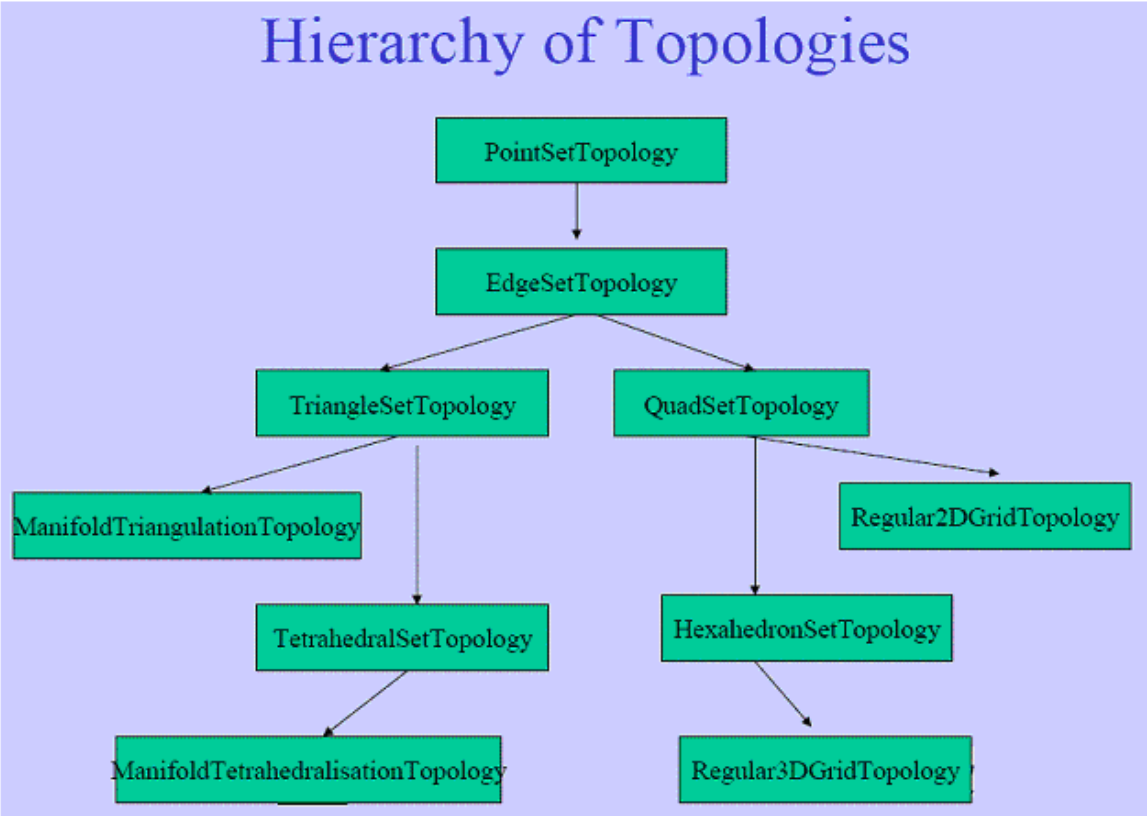

Figure 3.8: A generic and hierarchical topology is described from a class called BaseTopology.

What are the different mesh topologies supported in SOFA ?

3.3.2 Using Mesh Topologies

What is a mesh topology object ?

What is the difference with the old MeshTopology object ?

What are the different mesh topology objects in SOFA ?

BaseTopology class provides an implementation which handles topological changes, full topological relationships and geometric computation.

How do I have access to the adjacency information between items ?

What to do if I need to know only basic topology information ?

What are the geometry algorithms stored in each topology classes ?

3.3.3 Handling Topological Changes

How does it work ?

What container should I use to handle the topology changes ?

How to write the different callback functions associated with the containers ?

Topological changes are handled in a way which is as much transparent for the user as possible.

The 4 components of a BaseTopology object

```
Class BaseTopology<DataTypes> {

// A container for info to be stored and methods to access adjacency :
//   - Adjacency Information is only computed when needed
//   - Non template class
//   - Store TopologicalChange list

TopologyContainer *container ;

// A modifier for low-level methods to change topology :
//   - Cannot be accessed from user
//   - Modifier also changes the DOFs in the Mechanical Object
//   - Low level methods to add or to remove an item
TopologyModifiers<DataTypes> *modifier ;

// TopologyAlgorithms for high-level methods to change topology (user access) :
//   - Accessed from the user
//   - High level algorithms to refine, cut mesh
TopologyAlgorithms<DataTypes> *topologyAlgorithms ;

// Geometry Algorithms methods to get geometry information :
//   - Compute geometric information (normal, curvature, area, length)
GeometryAlgorithms<DataTypes> *geometryAlgorithms ;

};
```

Implementation for objects inherited from each other (Point, Edge, Triangle, Tetrahedron

- PointSetTopology (inherited from BaseTopology) :
  - Container : For each point gives its global index. This is useful for subset topologies (subset triangulation, ...) where the number of vertices involved in the topology may not be the same as the total number of vertices.
  - Modifier : addPointsProcess, removePointsProcess, renumberPointsProcess, addPointsWarning, removePointsWarning, propagateTopologicalChanges
  - Geometry : computeCenter, computeRadius, getAABB()
- EdgeSetTopology (inherited from PointSetTopology)
  - Container : array of edges, array of vertex-edge shell
  - Modifier : addEdgesProcess, removeEdgesProcess, fuseEdgesProcess, splitEdgesProcess, addEdgesWarning, removeEdgesWarning
  - Geometry : getEdgeLength, getRestEdgeLength
- TriangleSetTopology (inherited from EdgeSetTopology)
  - Container : array of triangles, of vertex- and edge-triangle shell
  - Modifier : addTrianglesProcess, removeTrianglesProcess, addTrianglesWarning, removeTrianglesWarning
  - Topology Algorithms : InciseAlongPointsList, RemoveAlongTrianglesList

  - Geometry : computeTriangleNormal

- TetrahedronSetTopology (inherited from TriangleSetTopology)
  - Container : array of tetrahedra, array of vertex-, edge-, triangle-tetrahedra shell
  - Modifier : addTetrahedraProcess, removeTetrahedraProcess, addTetrahedraWarning, removeTetrahedraWarning
  - Geometry : computeTetrahedronVolume

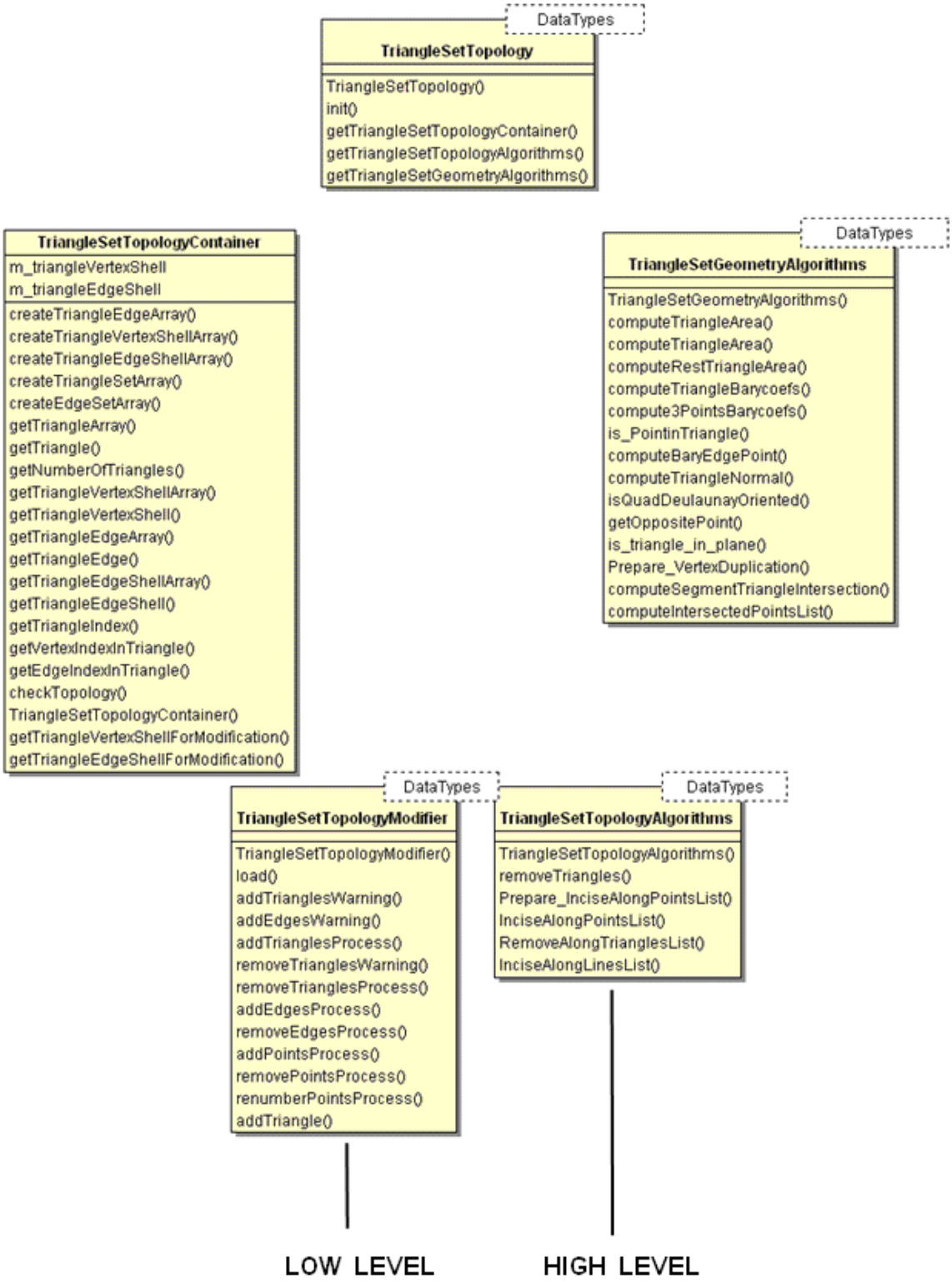

Figure 3.9: UML diagram describing the 4 components of TriangleSetTopology class.

Definition of data structures to be "aware" of topological changes

Force Fields, Constraints, Mapping and other modules may require to store information for each topological item (point, edge, triangle, etc.).

Two container data structures are defined to handle topological changes by matching the types of TopologyChanges :

- PointData<MyType>, EdgeData<MyType> are arrays (same as std::vector) of item of type MyType
- PointSubset, EdgeSubset are arrays of points or edges

Used-defined functions are called when an item is created or destroyed.

In higher level classes (for example TriangularQuadraticSpringForceField, DiagonalMass or FixedConstraint classes), the user only provides callback functions to handle :

- the creation of a topological item
- the destruction of a topological item

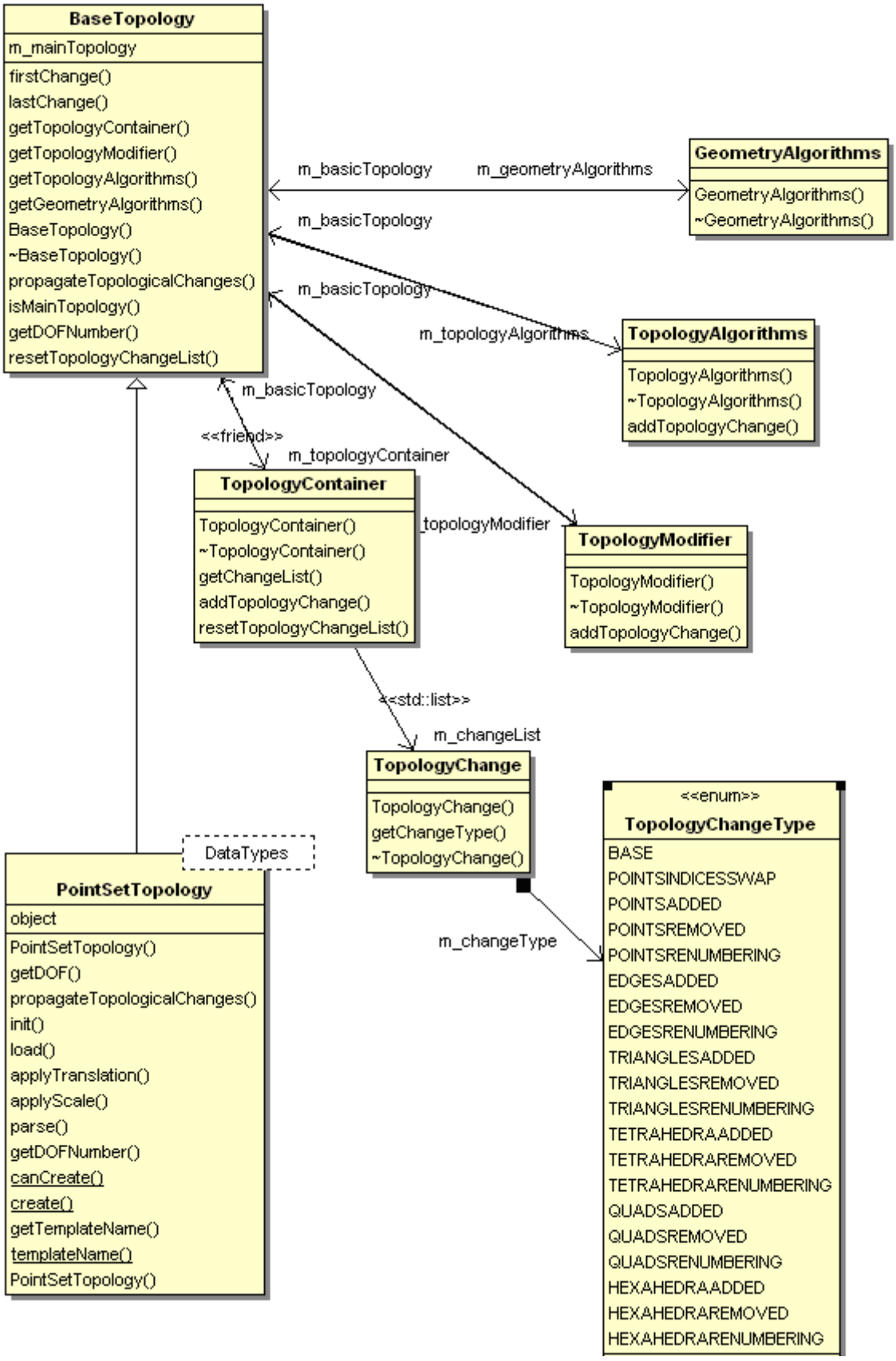

Figure 3.10: UML diagram showing the use of a TopologyChanges List from BaseTopology class.

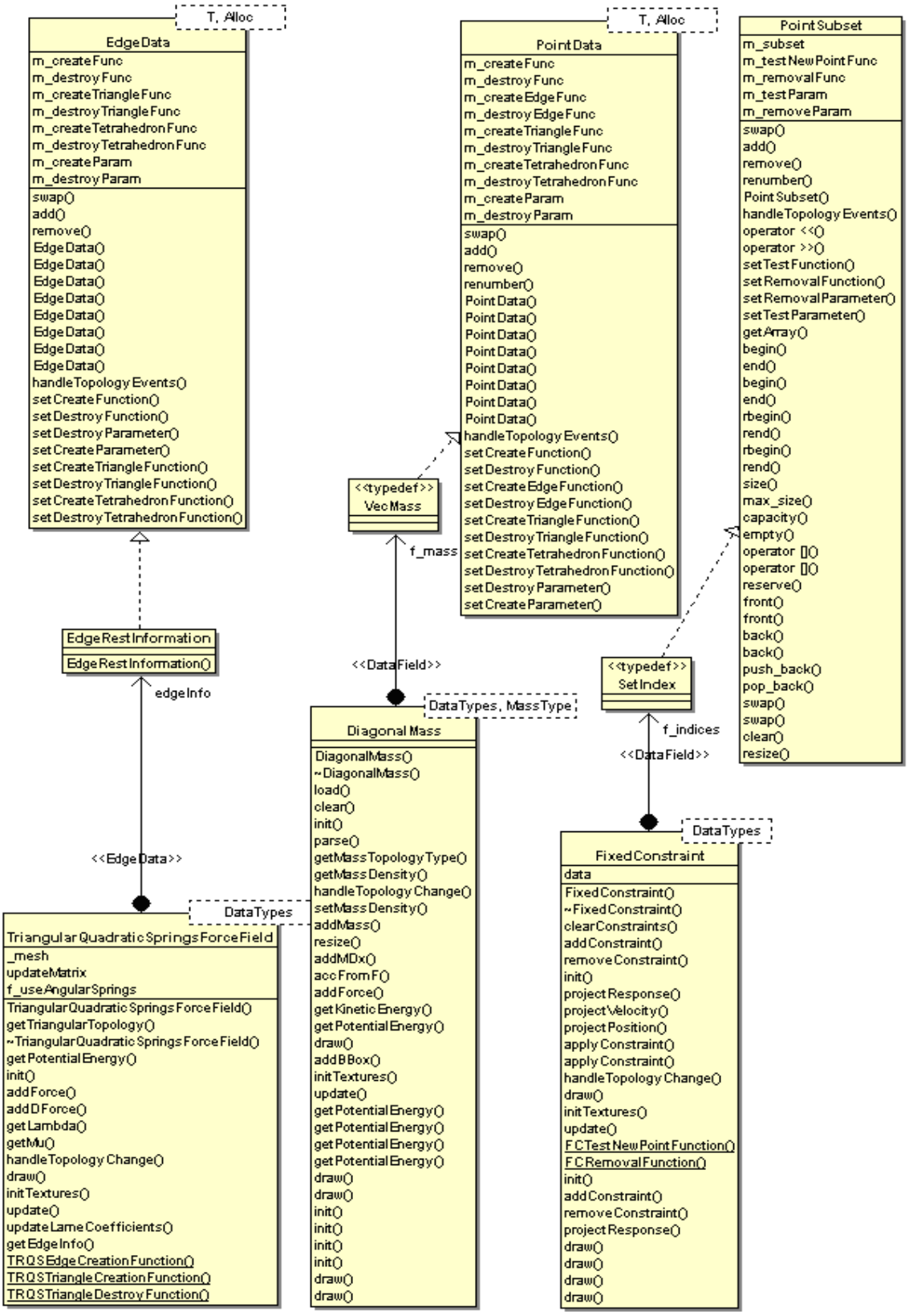

Figure 3.11: These UML diagrams show the use of EdgeData, PointData and PointSubset to handle topological changes implying modifications respectively in ForceField, Mass and Constraint modules.

Adding a list of items	Removing a list of items
1. ADD	1. WARN
2. WARN	2. PROPAGATE
3. PROPAGATE	3. REMOVE

Figure 3.12: Order to respect when adding or removing an item (see the explanations in the following example).

- "WARN" means : add the current topological change (add or delete a list of items) in the list of TopologyChanges
- "PROPAGATE" means : traverse the simulation tree with a TopologyChangeVistor to send the current topological change event to all force fields, constraints, mappings, etc.

Example : What happens when I split an Edge ?

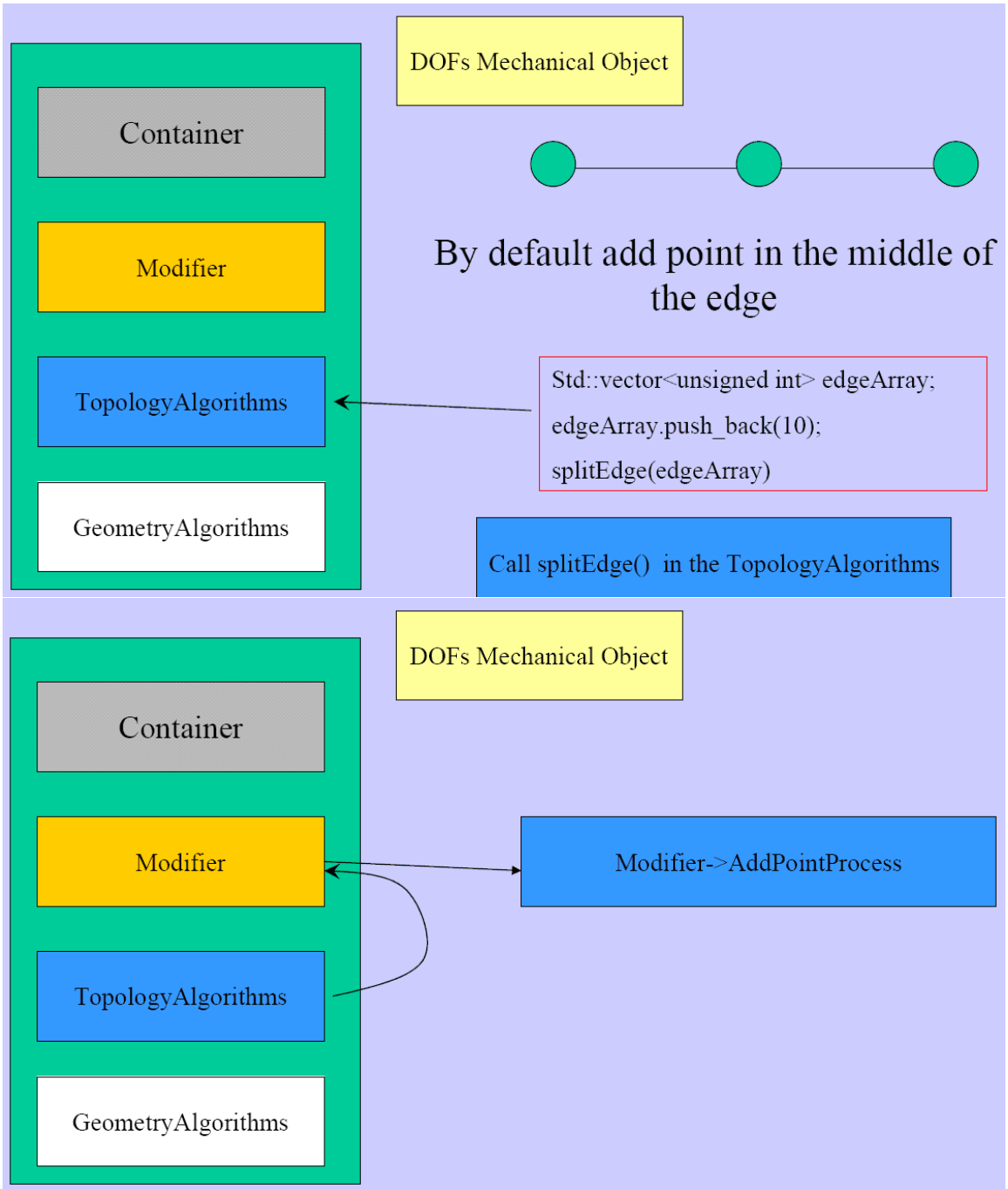

Figure 3.13: What happens when I split an Edge ? - Step 1. 2.

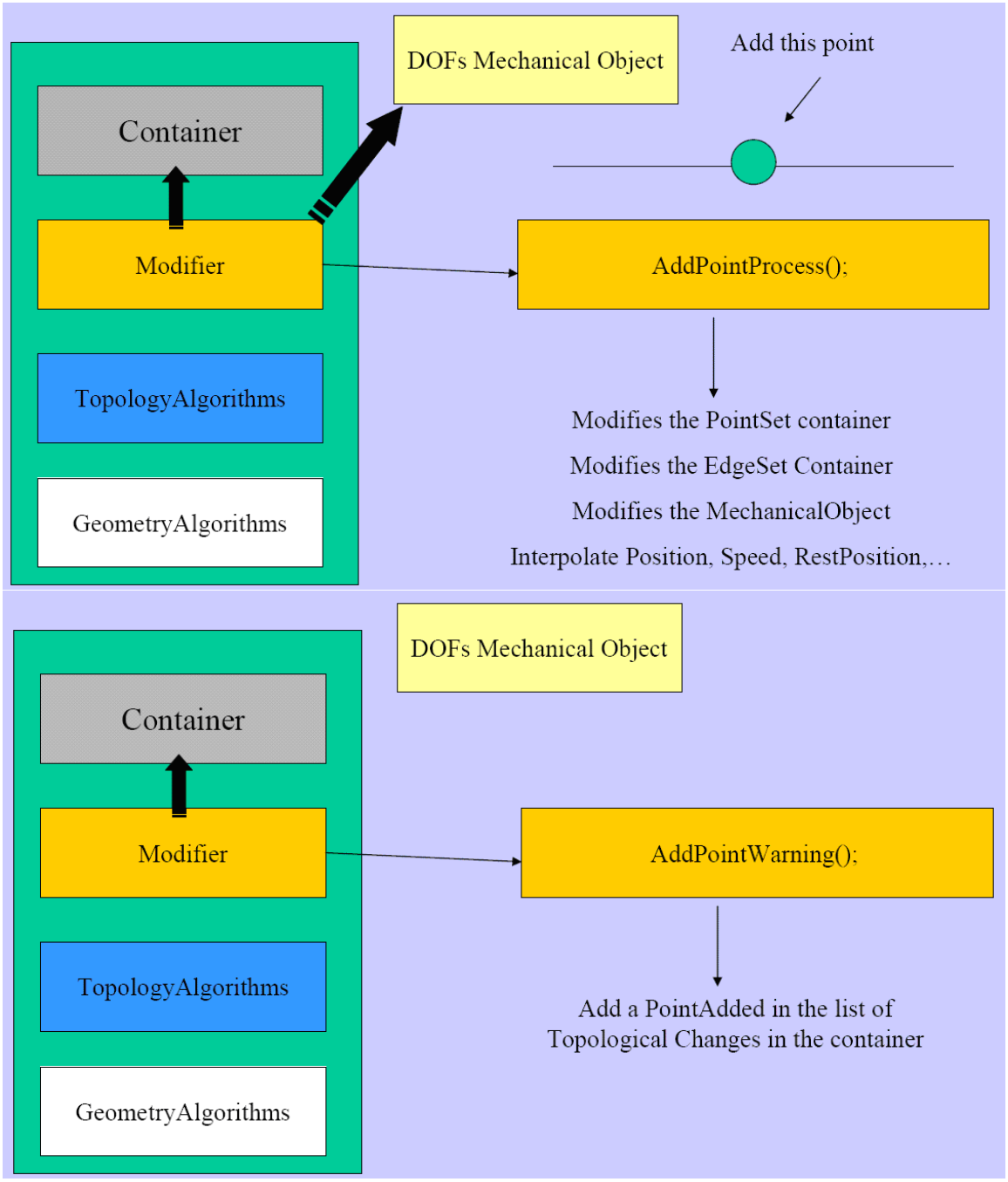

Figure 3.14: What happens when I split an Edge ? - Step 3. 4.

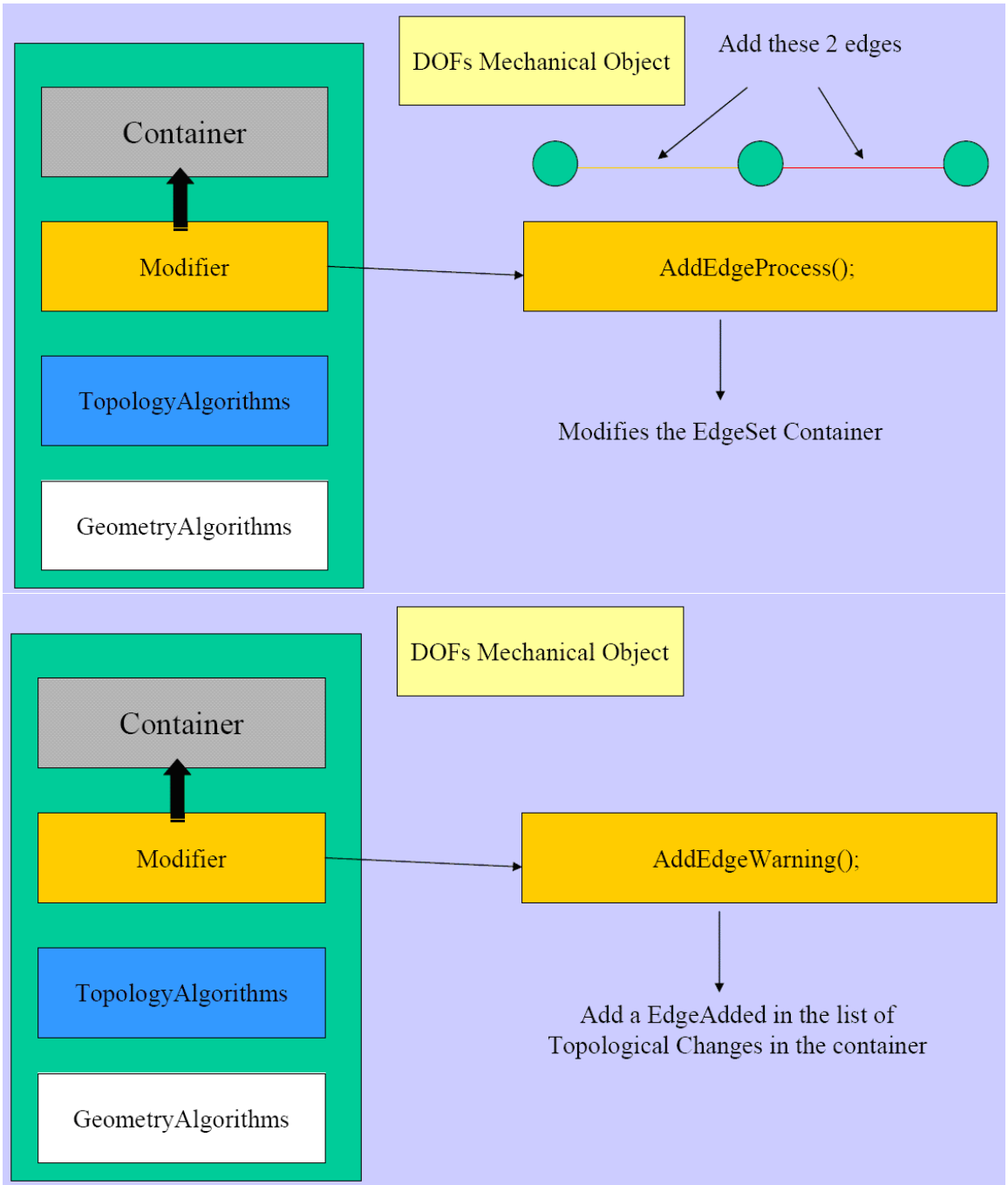

Figure 3.15: What happens when I split an Edge ? - Step 5. 6.

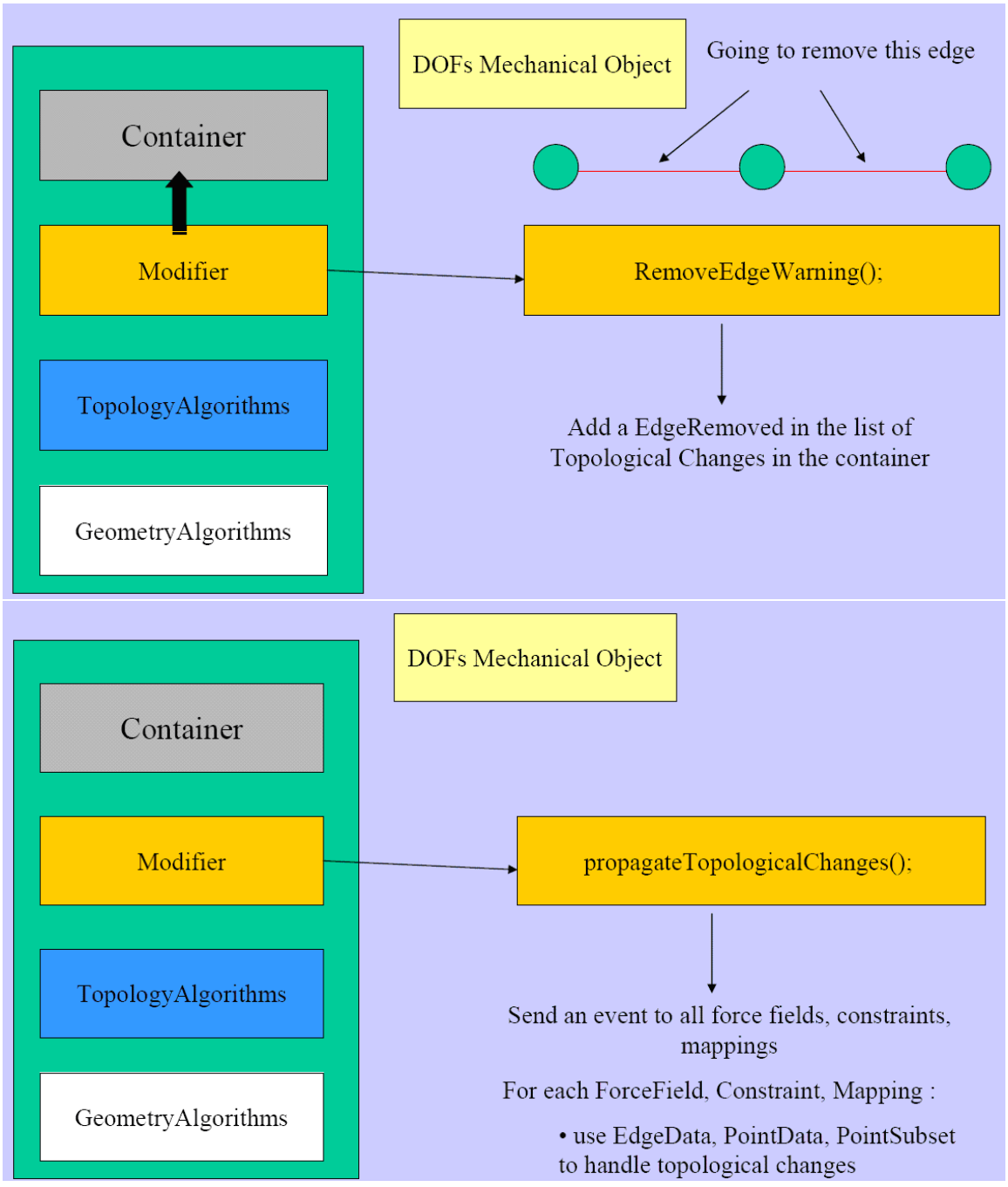

Figure 3.16: What happens when I split an Edge ? - Step 7. 8.

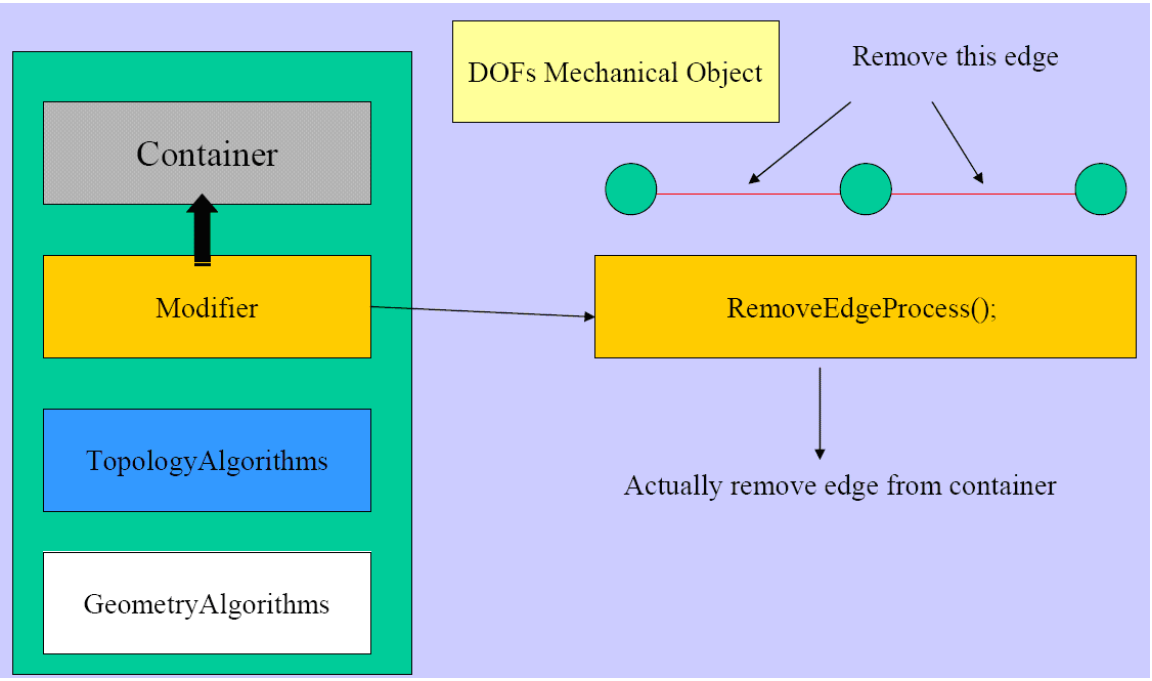

Figure 3.17: What happens when I split an Edge ? - Step 9.

Chapter 4

How to contribute to this documentation

4.1 Document structure

This document gathers the content of other documents located in different subdirectories. These documents can also be compiled as stand-alone documents. The structure can be illustrated as follows:

- `sofadocumentation.tex` : the root document.
- `macros_docu.tex` : custom commands and macros. This file is included by the root document.
- `introduction` : a subdirectory containing a chapter of this document.
  - `introduction.tex` : a stand-alone article containing the same. This file may include `../macros_docu.tex` to use the common custom commands.
  - `introduction_body.tex` : the text of the chapter/article. This file is included as a chapter by `sofadocumentation.tex`, and included as full article text by `introduction.tex`.
- there could (and hopefully will !) be other subdirectories with a similar structure.

4.2 Compiling the document

4.2.1 File formats

The graphics are handled by: `\usepackage[pdftex]{graphicx}`. This allows the inclusion of `.png` images rather than `.eps`, which makes the image files much smaller, and compilation of `html` faster. The result of the compilation is a `.pdf` rather than a `.dvi` file.

4.2.2 Include paths

The root document includes files in the subdirectories, which in turn include files too. The problem is that the path from the subdirectory (used when compiling a stand-alone article) is not the same as from the parent directory (used when compiling the whole report). It seems that LaTeXhas no include path command to circumvent this problem.

Fortunately, LaTeXuses an environment variable named TEXINPUTS which represents a list of directories to search. We use this variable to allow file inclusion, by adding the parent directories to the list. The command lines depend on the shell, e.g.:

- using bash: `export TEXINPUTS=${TEXINPUTS}:..:../..  ; latex sofadocumentation.tex`
- using tcsh: `setenv TEXINPUTS ${TEXINPUTS}:..:../..  ; latex sofadocumentation.tex`

Using the `Kile` graphical front-end, you can customize the command line in menu `Settings/Kile/Tools/Build`, select tool `LATEX` and write the appropriate command line in the `Command` field.

4.2.3 HTML

HTML can be generated using the following command:
`latex2html sofadocumentation.tex -mkdir -dir ./html -show_section_numbers -split 1`
Currently, the listings do not appear in html.