

The Design of SOFA

The SOFA team

December 19, 2011

Abstract

This document contains discussions and references explaining the design of SOFA. Comments and suggestions are welcome on the SOFA level mailing list.

1 Introduction

This chapter present the design of SOFA, as well as its recent evolutions. It also includes discussions justifying some of the decision that are behind this design.

The conventions used in UML class diagrams are presented in Fig 1.

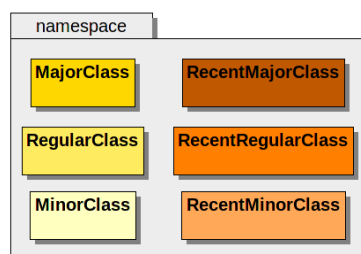


Figure 1: Color conventions in UML diagrams.

2 Core Class Diagrams

2.1 Object Model (sofa::core::objectmodel)

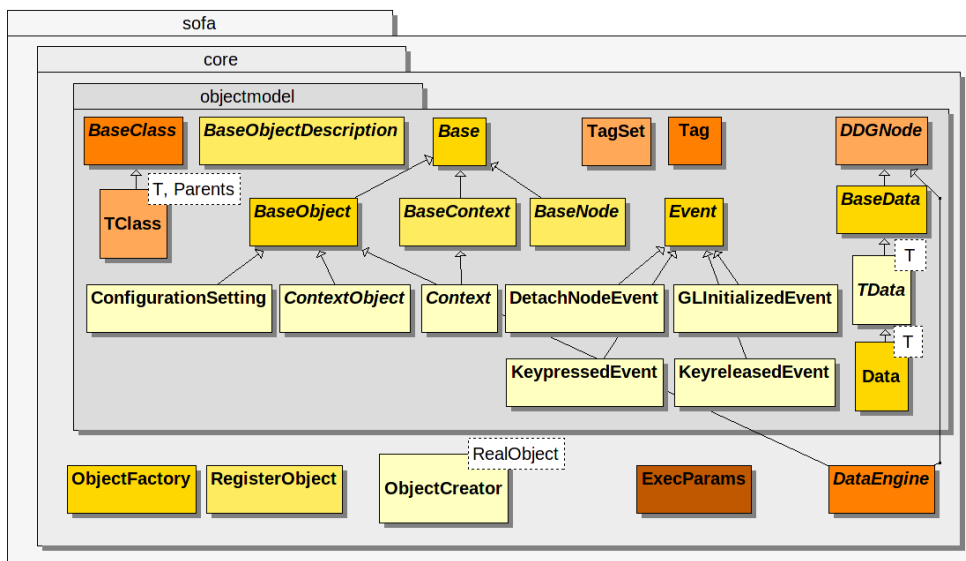


Figure 2: Classes of the sofa::core::objectmodel namespace.

2.1.1 Changes compared to the 1.0 beta 4 version

New class description system. It is based on the `BaseClass` and `TClass` classes, and requires to add the `SOFA_CLASS` macro in the declaration of all classes deriving from `Base`. The benefits are that it is now possible to follow the full hierarchy of classes from the final components, instead of having just a fixed set of categories. This macro is also necessary in classes with a templated parent class to be able to use methods and member variables defined in `Base` such as `initData` or `sout`. This removed all the previously redundant direct heritage to `BaseObject` that was previously required.

Objects and node tagging (`Tag` and `TagSet`). The goal of the introduction of tags is to provide one of the pieces necessary to support non-mechanical states (electrical potentials, contrast agent concentrations) as well as cleaner non-geometrical mechanical states (fluid dynamics, reduced-coordinate articulations). For example, in a simulation involving blood in deformable vessels, we would use two tags to distinguish the different states : mechanical, fluid. These tags will be used to easily work with only a subset of the components, so that the mechanical solver works on positions and forcefields but don't interferes with blood flow and pressure, and inversely for the fluid solver (see ¹). We decided on using there tags instead of extending the class hierarchy as was done before with the `State` and `MechanicalState` classes. A hierarchy is fine when we have only one feature that we want to differentiate on (such as base vs mechanical vs electrical), but when we add other criteria (lagrangian geometry vs eulerian vs reduced generalized coordinates, velocity vs vorticity, independent vs mapped DOFs) it is no longer manageable as specialized classes. A secondary use of these tags is to replace existing subsets mechanisms within `CollisionModels` (r2441) and `Constraints` (r3121). The design is based on the following elements. Tags are added to `BaseObject`, as a list of string (internally converted to a list of unique ids for faster processing). All visitors now filter the objects they process based on their list of tags. All solvers by default copy their own list of tags to the visitors they execute, so that they only affect the objects with the same tags as they have (TODO: this is currently broken).

Dependencies between Data (`DDGNode` and `DataEngine`). The goal is to be able to specify simple links between datas or through computation engines. To function correctly, the methods `getValue()` or `beginEdit()/endEdit()` are required to be called in all codes accessing values contained in `Data` instances. To enforce this, the class `DataPtr` was removed. Note also that it is very inefficient to call `getValue()` repetively within computation loops. Instead, the recommended method is to use the helper classes `ReadAccessor` and `WriteAccessor` that can hold a reference to a `Data` value, provides the same API as regular vectors, and automatically calls `endEdit()` at the end of the call block in case of `WriteAccessor`.

TODO: `TData<T>` was useful as a common parent for `Data<T>` and `DataPtr<T>`, but now that `DataPtr` no longer exist, it would simplify the hierarchy to merge `TData<T>` and `Data<T>`.

Copy-on-Write (CoW) mechanism (`DataContainer`). The goal is to reduce copies of datas when using engines and/or multi-threading. It is completely transparent to other codes, since everything should now respect the data access API (see above). There is however one important side-effect that may break some existing code: the pointer to the value of a `Data` can now change during a call to `beginEdit()`. This means that it is now illegal to retrieve the pointer to the value in a `Data` at init time and then reuse it after edits might have been made.

¹<http://wiki.sofa-framework.org/tdev/wiki/Notes/ProposalGenericStates>

Support for asynchronous multi-threading (ExecParams). This feature is an important goal of the new design, however it is not yet completely functional and thus may change shortly.

2.2 Physical Behavior (sofa::core::behavior)

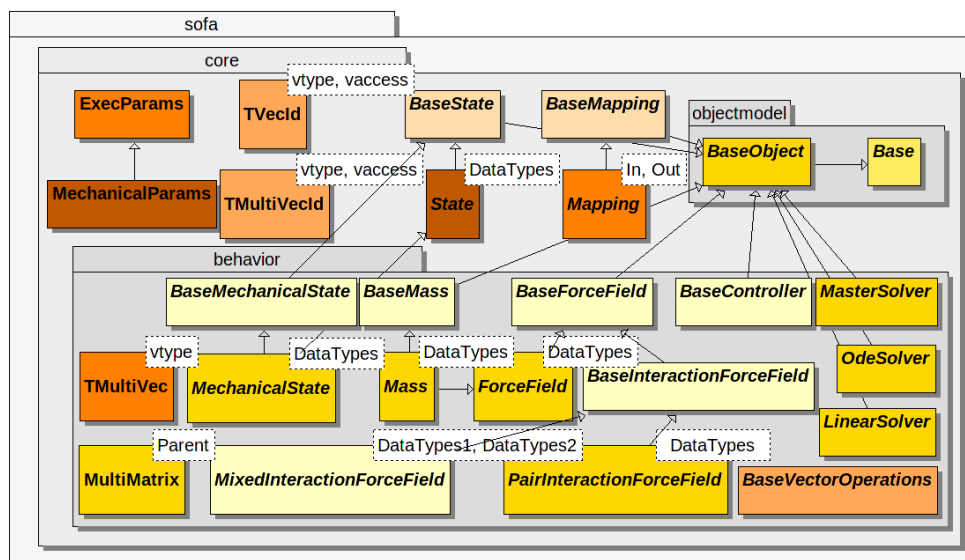


Figure 3: Classes of the sofa::core::behavior namespace.

2.2.1 Changes compared to the 1.0 beta 4 version

VecId is replaced by templated classes to specify vector types and read/write access. Previously, the VecId class was used by solvers and visitors to specify on which state vectors should an operation be applied. However, this API did not express the requirements for these vectors (which type, i.e. Coord, Deriv, or MatDeriv), nor if they were supposed to be read or written. Now, methods and visitors can use the following classes instead (all typedefs from a common TVecId templated class), so that developers will now explicitly what to expect:

```

1  /// Identify one vector stored in State
2  /// A ConstVecId only provides a read-only access to the underlying vector.
3  typedef TVecId<V_ALL, V_READ> ConstVecId;
4
5  /// Identify one vector stored in State
6  /// A VecId provides a read-write access to the underlying vector.
7  typedef TVecId<V_ALL, V_WRITE> VecId;
8
9  /// Typedefs for each type of state vectors
10 typedef TVecId<V_COORD, V_READ> ConstVecCoordId;
11 typedef TVecId<V_COORD, V_WRITE> VecCoordId;
12 typedef TVecId<V_DERIV, V_READ> ConstVecDerivId;
13 typedef TVecId<V_DERIV, V_WRITE> VecDerivId;
14 typedef TVecId<V_MATDERIV, V_READ> ConstMatrixDerivId;
15 typedef TVecId<V_MATDERIV, V_WRITE> MatrixDerivId;

```

Also, a new class TMultiVecId is introduced to be able to specify different IDs for specific states or groups of states. Similar typedefs are defined as above, replacing Vec with MultiVec.

New State API and class hierarchy. The previous State API was based on `getX()/getV()/...` methods that returned pointers to the vectors last specified (using `setX()/setV()/...` methods) as the current position, velocity, ... This is now replaced by `read(ConstVec TYPEid)` and `write(Vec TYPEid)` methods, where `TYPE` is either `Coord`, `Deriv`, or `MatDeriv`. These methods return a pointer to a `Data` instance containing the vector, instead of the vector itself. This was necessary to respect the `Data` access API (see section 2.1.1). For compatibility with existing codes (especially `draw()` methods), the read-only `getX()/getV()/...` methods still exist but they are deprecated (and there are strictly equivalent to calling `read()` with the default ID for position, velocity, ...). The non-const versions however are removed, so all codes modifying state vectors will have to use the new `Data`-based API. Note that with this change, the state components no longer store the information about which vectors should be considered as the current position, velocity, ... So another mechanism is required to specify these associations (`MechanicalParams`, see below).

A new state class hierarchy is also defined, introducing a new parent class `BaseState`, common to all states (mechanical, visual, ...). Methods allowing to read and write vectors given their IDs are specified in `State<DataTypes>`. The `MappedModel` class is removed, non-mechanical state components (such as visual models) now directly derive from `State<DataTypes>`.

Simplified Mapping class hierarchy. Previously, a different base class (`Mapping< State<TIn>, MappedModel<TOut> >` or `MechanicalMapping< MechanicalState<TIn>, MechanicalState<TOut> >`) were used for mechanical versus non-mechanical (i.e. visual or one-way) mappings. This introduced complexities and redundancies in the code of the final components (they were templated by the type of their parent class and compiled twice for each pair of input/output datatypes). Now, a single base class is used (`Mapping<TIn,TOut>`, templated only by the input and output datatypes). Components are also templated by the same types, similarly to other classes such as forcefields. A new set of flags is used to check if a given mapping is mechanical or not. Different flags are used to activate the mapping of forces, masses, and constraints separately (although by default they have the same value). Components that do not implement `applyJT` (i.e. visual mappings) can force them to false to indicate they do not support mechanical computations.

New mechanical component API based on Data and MechanicalParams. This is the most important change from the previous design. To provide all the required vector IDs that were previously stored within each `MechanicalState` by calls to `setX()/...` methods, we define a `MechanicalParam` class. This class is given to all mechanical-related methods, specified by `OdeSolvers` and transmitted by `MechanicalVisitors`. It hides the `VecId` system from most component codes, providing the same abstraction of accessing the current position and velocity vectors as was previously handled within `MechanicalState`. For example, where a component such as a `ForceField` implementation used :

```
1 const VecCoord* x = this->mstate->getX();
```

It will now use `MechanicalParams` as follows :

```
1 helper::ReadAccessor<VecCoord> x = *mparams->readX(this->mstate);
```

This API is preferable to directly manipulating `VecIds` (such as in `this->mstate->read(mparams->getXId(this->mstate))`) because :

- The code is very similar to the previous version.

- If the API of the `VecId` or `MechanicalState` class is further changed, the `MechanicalParam::readX()` method can handle it.
- Different `VecId`s for different states can be chosen within `MechanicalParam` (such as what is currently used for free vs mapped states, and animated vs external obstacle state).

However it is still possible to get the ids (using `mparams->x().getId(mstate)`).

All vector accesses given though `MechanicalParams` are read-only. Vectors actually modified by each method are given as separate `MultiVecId` parameters. This is useful to make the API clearer (we can see explicitly what each method is supposed to write) and safer (no accidental modifications to `X` or `V` within `addForce()` or `draw()` for example). Finally, other interesting pieces of information can be queried though `MechanicalParams`. For example, a `ForceField` can know as soon as the `addForce()` call if the current solver is implicit or explicit, if the kinematic and potential energy should be computed, ...

New solver hierarchy and vector manipulation API (`TMultiVec` and `BaseVectorOperations`). Previously, the solver classes had a complex hierarchy, with several parent classes (`SolverImpl`, `OdeSolverImpl`, `MasterSolverImpl`) that where used to launch each type of visitors (mechanics, linear algebra, ...). In the new design, these classes are replaced by separate helper classes (`VectorOperations`, `MechanicalOperations`) that are instanciated at runtime and allow to launch each type of visitors while holding and updating the persistant parameters (in the form of a `MechanicalParams` for mechanical visitors for example, or `ExecParams` for simple vector operations).

2.3 Topology (`sofa::core::topology`)

The design of this namespace is still a work in progress, so major changes should still be expected...

2.3.1 Changes compared to the 1.0 beta 4 version

To be documented...

2.4 Collision (`sofa::core::collision`)

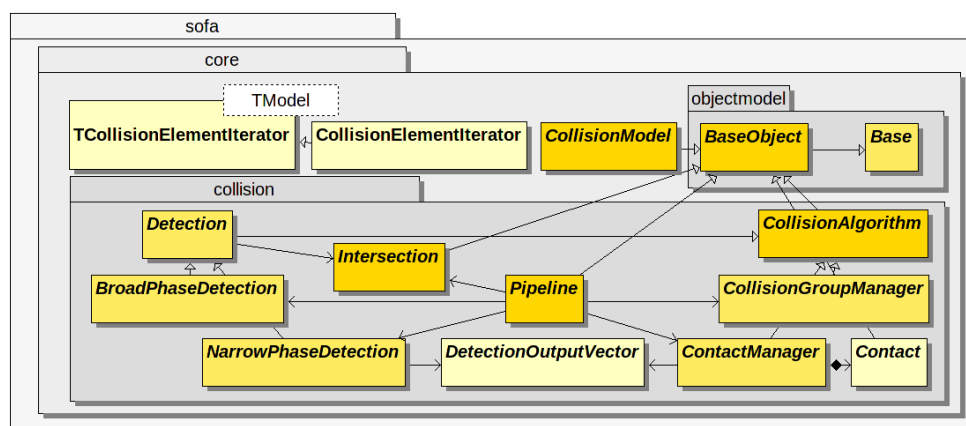


Figure 4: Classes of the `sofa::core::collision` namespace.

2.4.1 Changes compared to the 1.0 beta 4 version

No major changes.

2.5 Mesh and Image Loaders (sofa::core::loader)

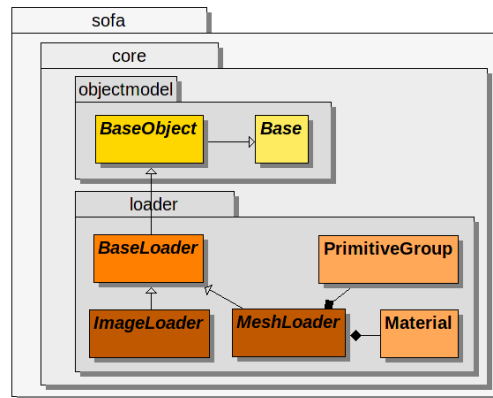


Figure 5: Classes of the `sofa::core::loader` namespace.

2.5.1 Changes compared to the 1.0 beta 4 version

This namespace is completely new.

To be documented...

3 Main classes

3.1 sofa::core::objectmodel

3.1.1 Base

```
51
52 namespace objectmodel
53 {
54
55 /**
56  * \brief Base class for everything
57  *
58  * This class contains all fonctionnality shared by every objects in SOFA.
59  * Most importantly it defines how to retrieve information about an object (name, type, data fields).
60  * All classes deriving from Base should use the SOFA_CLASS macro within their declaration (see BaseClass.h).
61  *
62  */
63 class SOFA_CORE_API Base
64 {
65 public:
66     typedef Base* Ptr;
67     typedef boost::intrusive_ptr<Base> SPtr;
68
69     typedef TClass< Base, void > MyClass;
70     static const MyClass* GetClass() { return MyClass::get(); }
71     virtual const BaseClass* getClass() const { return GetClass(); }
72
73     template<class T>
74     static void dynamicCast(T*& ptr, Base* b)
75     {
76         ptr = dynamic_cast<T*>(b);
77     }
78
79 protected:
80     /// Constructor cannot be called directly
81     /// Use the New() method instead
82     Base();
83
84     /// Direct calls to destructor are forbidden.
85     /// Smart pointers must be used to manage creation/destruction of objects
86     virtual ~Base();
87
88 private:
89     /// Copy constructor is not allowed
90     Base(const Base& b);
91
92     sofa::helper::system::atomic<int> ref_counter;
93     void addRef();
94     void release();
95
96     friend inline void intrusive_ptr_add_ref(Base* p)
97     {
98         p->addRef();
99     }
100
101     friend inline void intrusive_ptr_release(Base* p)
102     {
103         p->release();
104     }
105
106 public:
107
108
109
110     /// Accessor to the object name
111     const std::string & getName() const
```

```

112 {
113     return name.getValue();
114 }
115
116 /// Set the name of this object
117 void setName(const std::string& n);
118
119 /// Set the name of this object, adding an integer counter
120 void setName(const std::string& n, int counter);
121
122 /// Get the type name of this object (i.e. class and template types)
123 virtual std::string getTypeName() const;
124
125 /// Get the class name of this object
126 virtual std::string getClassName() const;
127
128 /// Get the template type names (if any) used to instantiate this object
129 virtual std::string getTemplateName() const;
130
131 /// @name fields
132 /// Data fields management
133 /// @{
134
135 /// Assign one field value (Data or Link)
136 virtual bool parseField( const std::string & attribute, const std::string & value);
137
138 /// Check if a given Data field or Link exists
139 virtual bool hasField( const std::string & attribute) const;
140
141 /// Parse the given description to assign values to this object's fields and potentially other parameters
142 virtual void parse ( BaseObjectDescription* arg );
143
144 /// Assign the field values stored in the given list of name + value pairs of strings
145 void parseFields ( const std::list<std::string>& str );
146
147 /// Assign the field values stored in the given map of name -> value pairs
148 virtual void parseFields ( const std::map<std::string, std::string*>& str );
149
150 /// Write the current field values to the given map of name -> value pairs
151 void writeDatas (std::map<std::string, std::string*>& str);
152
153 /// Write the current field values to the given XML output stream
154 void xmlWriteDatas (std::ostream& out, int level = 0);
155
156 /// Find a data field given its name. Return NULL if not found.
157 /// If more than one field is found (due to aliases), only the first is returned.
158 BaseData* findData( const std::string &name ) const;
159
160 /// @deprecated
161 BaseData* findField( const std::string &name ) const { return findData(name); }
162
163 /// Find data fields given a name: several can be found as we look into the alias map
164 std::vector< BaseData* > findGlobalField( const std::string &name ) const;
165
166 /// Find a link given its name. Return NULL if not found.
167 /// If more than one link is found (due to aliases), only the first is returned.
168 BaseLink* findLink( const std::string &name ) const;
169
170 /// Find link fields given a name: several can be found as we look into the alias map
171 std::vector< BaseLink* > findLinks( const std::string &name ) const;
172
173 /// Update pointers in case the pointed-to objects have appeared
174 virtual void updateLinks();
175
176 /// Helper method used to initialize a data field containing a value of type T
177 template<class T>
178 BaseData::BaseInitData( Data<T>* field, const char* name, const char* help, bool isDisplayed=true,
179     bool isReadOnly=false )

```

```

179     {
180         BaseData::BaselnitData res;
181         this→initData0(field, res, name, help, isDisplayed, isReadOnly);

```

3.1.2 BaseObject

```

61
62 namespace objectmodel
63 {
64
65 class Event;
66 class BaseNode;
67
68 /**
69  * \brief Base class for simulation objects.
70  *
71  * An object defines a part of the fonctionnality in the simulation
72  * (stores state data, specify topology, compute forces, etc).
73  * Each simulation object is related to a context, which gives access to all available external data.
74  * It is able to process events, if listening enabled (default is false).
75  *
76 */
77 class SOFA_CORE_API BaseObject : public virtual Base
78 #ifndef SOFA_SMP
79  , public BaseObjectTasks
80 #endif
81 {
82 public:
83     SOFA_CLASS(BaseObject, Base);
84 protected:
85     BaseObject();
86
87     virtual ~BaseObject();
88 public:
89     /// @name Context accessors
90     /// @{
91
92     //void setContext(BaseContext* n);
93
94     const BaseContext* getContext() const;
95
96     BaseContext* getContext();
97
98     const BaseObject* getMaster() const;
99
100    BaseObject* getMaster();
101
102    typedef helper::vector<BaseObject::SPtr> VecSlaves;
103
104    const VecSlaves& getSlaves() const;
105
106    BaseObject* getSlave(const std::string & name) const;
107
108    virtual void addSlave(BaseObject::SPtr s);
109
110    virtual void removeSlave(BaseObject::SPtr s);
111
112    virtual void copyAspect(int destAspect, int srcAspect);
113
114    virtual void releaseAspect(int aspect);
115
116    /// @}
117
118    /// @name control
119    /// Basic state control
120    /// @{

```

```

121
122     /// Pre-construction check method called by ObjectFactory.
123     template<class T>
124     static bool canCreate(T* /*obj*/, BaseContext* /*context*/, BaseObjectDescription* /*arg*/)
125     {
126         return true;
127     }
128
129     /// Construction method called by ObjectFactory.
130     template<class T>
131     static typename T::SPtr create(T*, BaseContext* context, BaseObjectDescription* arg)
132     {
133         typename T::SPtr obj = sofa::core::objectmodel::New<T>();
134         if (context) context->addObject(obj);
135         if (arg) obj->parse(arg);
136         return obj;
137     }
138
139     /// Parse the given description to assign values to this object's fields and potentially other parameters
140     virtual void parse ( BaseObjectDescription* arg );
141
142     /// Initialization method called at graph creation and modification, during top-down traversal.
143     virtual void init();
144
145     /// Initialization method called at graph creation and modification, during bottom-up traversal.
146     virtual void bwdInit();
147
148     /// Update method called when variables used in precomputation are modified.
149     virtual void reinit ();
150
151     /// Save the initial state for later uses in reset ()
152     virtual void storeResetState();
153
154     /// Reset to initial state
155     virtual void reset();
156
157     /// Called just before deleting this object
158     /// Any object in the tree below this object that are to be removed will be removed only after this call ,
159     /// so any references this object holds should still be valid .
160     virtual void cleanup();

```

3.2 sofa::core

3.2.1 BaseState

```

39 *
40 * This class define the interface of components used as source and
41 * destination of regular (non mechanical) mapping. It is then specialized as
42 * MechanicalState (storing other mechanical data) or MappedModel (if no
43 * mechanical data is used, such as for VisualModel).
44 */
45 class SOFA_CORE_API BaseState : public virtual objectmodel::BaseObject
46 {
47 public:
48     SOFA_ABSTRACT_CLASS(BaseState, objectmodel::BaseObject);
49 protected:
50     virtual ~BaseState() { }
51 public:
52     /// Current size of all stored vectors
53     virtual int getSize() const = 0;
54
55     /// Resize all stored vector
56     virtual void resize(int vsize) = 0;
57
58     /// @name BaseData vectors access API based on VecId

```

3.2.2 State

```
38 namespace core
39 {
40
41 /**
42 * \brief Component storing position and velocity vectors.
43 *
44 * This class define the interface of components used as source and
45 * destination of regular (non mechanical) mapping. It is then specialized as
46 * MechanicalState (storing other mechanical data) or MappedModel (if no
47 * mechanical data is used, such as for VisualModel).
48 *
49 * The given DataTypes class should define the following internal types:
50 * \li \code Real \endcode : scalar values (float or double).
51 * \li \code Coord \endcode : position values.
52 * \li \code Deriv \endcode : derivative values (velocity).
53 * \li \code VecReal \endcode : container of scalar values with the same API as sofa::helper::vector.
54 * \li \code VecCoord \endcode : container of Coord values with the same API as sofa::helper::vector.
55 * \li \code VecDeriv \endcode : container of Deriv values with the same API as sofa::helper::vector.
56 * \li \code MatrixDeriv \endcode : vector of Jacobians (sparse constraint matrices).
57 *
58 */
59 template<class TDataTypes>
60 class State : public virtual BaseState
61 {
62 public:
63     SOFA_CLASS(SOFA_TEMPLATE(State,TDataTypes), BaseState);
64
65     typedef TDataTypes DataTypes;
66     /// Scalar values (float or double).
67     typedef typename DataTypes::Real Real;
68     /// Position values.
69     typedef typename DataTypes::Coord Coord;
70     /// Derivative values (velocity, forces, displacements).
71     typedef typename DataTypes::Deriv Deriv;
72     /// Container of scalar values with the same API as sofa::helper::vector.
73     typedef typename DataTypes::VecReal VecReal;
74     /// Container of Coord values with the same API as sofa::helper::vector.
75     typedef typename DataTypes::VecCoord VecCoord;
76     /// Container of Deriv values with the same API as sofa::helper::vector.
77     typedef typename DataTypes::VecDeriv VecDeriv;
78     /// Vector of Jacobians (sparse constraint matrices).
79     typedef typename DataTypes::MatrixDeriv MatrixDeriv;
80 protected:
81     virtual ~State() { }
82 public:
83     /// @name New vectors access API based on VecId
84     /// @{
85
86     virtual Data< VecCoord >* write(VecCoordId v) = 0;
87     virtual const Data< VecCoord >* read(ConstVecCoordId v) const = 0;
88
89     virtual Data< VecDeriv >* write(VecDerivId v) = 0;
90     virtual const Data< VecDeriv >* read(ConstVecDerivId v) const = 0;
91
92     virtual Data< MatrixDeriv >* write(MatrixDerivId v) = 0;
93     virtual const Data< MatrixDeriv >* read(ConstMatrixDerivId v) const = 0;
94
95     /// @}
96
97     /// @name BaseData vectors access API based on VecId
98     /// @{
99
100     virtual objectmodel::BaseData* baseWrite(VecId v);
101
102     virtual const objectmodel::BaseData* baseRead(ConstVecId v) const;
```

3.2.3 BaseMapping

```
48 *
49 * This Interface is used for the Mappings. A Mapping can convert one model to an other.
50 * For example, we can have a mapping from a BehaviorModel to a VisualModel.
51 *
52 */
53 class SOFA_CORE_API BaseMapping : public virtual objectmodel::BaseObject
54 {
55 public:
56     SOFA_ABSTRACT_CLASS(BaseMapping, objectmodel::BaseObject);
57 protected:
58     /// Constructor
59     BaseMapping();
60
61     /// Destructor
62     virtual ~BaseMapping();
63 public:
64     Data<bool> f_mapForces;
65     Data<bool> f_mapConstraints;
66     Data<bool> f_mapMasses;
67     Data<bool> f_mapMatrices;
68
69     /// Apply the transformation from the input model to the output model (like apply displacement from BehaviorModel
70     to VisualModel)
71     virtual void apply(const MechanicalParams* mparams /* PARAMS FIRST = MechanicalParams::defaultInstance()*/,
72         MultiVecCoordId outPos = VecCoordId::position(), ConstMultiVecCoordId inPos = ConstVecCoordId::position() ) =
73         0;
74     virtual void applyJ(const MechanicalParams* mparams /* PARAMS FIRST = MechanicalParams::defaultInstance()*/,
75         MultiVecDerivId outVel = VecDerivId::velocity(), ConstMultiVecDerivId inVel = ConstVecDerivId::velocity() ) = 0;
76
77     /// Accessor to the input model of this mapping
78     virtual helper::vector<BaseState*> getFrom() = 0;
79
80     /// Accessor to the output model of this mapping
81     virtual helper::vector<BaseState*> getTo() = 0;
82
83     // BaseMechanicalMapping
84     virtual void applyJT(const MechanicalParams* mparams /* PARAMS FIRST */, MultiVecDerivId inForce,
85         ConstMultiVecDerivId outForce) = 0;
86     virtual void applyDJT(const MechanicalParams* mparams /* PARAMS FIRST */, MultiVecDerivId inForce,
87         ConstMultiVecDerivId outForce) = 0;
88     virtual void applyJT(const ConstraintParams* mparams /* PARAMS FIRST */, MultiMatrixDerivId inConst,
89         ConstMultiMatrixDerivId outConst) = 0;
90     virtual void computeAccFromMapping(const MechanicalParams* mparams /* PARAMS FIRST */, MultiVecDerivId
91         outAcc, ConstMultiVecDerivId inVel, ConstMultiVecDerivId inAcc) = 0;
92
93     virtual bool areForcesMapped() const;
94     virtual bool areConstraintsMapped() const;
95     virtual bool areMassesMapped() const;
96     virtual bool areMatricesMapped() const;
97
98     virtual void setForcesMapped(bool b);
99     virtual void setConstraintsMapped(bool b);
100     virtual void setMassesMapped(bool b);
101     virtual void setMatricesMapped(bool b);
102
103     virtual void setNonMechanical();
104
105     /// Return true if this mapping should be used as a mechanical mapping.
106     virtual bool isMechanical() const;
107
108     /// Return true if the destination model has the same topology as the source model.
109     ///
110     /// This is the case for mapping keeping a one-to-one correspondance between
111     /// input and output DOFs (mostly identity or data-conversion mappings).
112     virtual bool sameTopology() const { return false; }
```

```

106 /// Get the (sparse) jacobian matrix of this mapping, as used in applyJ/applyJT.
107 /// This matrix should have as many columns as DOFs in the input mechanical states
108 /// (one after the other in case of multi-mappings), and as many lines as DOFs in
109 /// the output mechanical states.
110 ///
111 /// applyJ(out, in) should be equivalent to $out = J in$.
112 /// applyJT(out, in) should be equivalent to $out = J^T in$.
113 ///
114 /// @TODO Note that if the mapping provides this matrix, then a default implementation
115 /// of all other related methods could be provided, or optionally used to verify the
116 /// provided implementations for debugging.

```

3.2.4 Mapping

```

43 * This Interface is used for the Mappings. A Mapping can convert one model to an other.
44 * For example, we can have a mapping from a BehaviorModel to a VisualModel.
45 *
46 */
47
48 template <class TIn, class TOut>
49 class Mapping : public BaseMapping
50 {
51 public:
52     SOFA_CLASS(SOFA_TEMPLATE2(Mapping,TIn,TOut), BaseMapping);
53
54     /// Input Data Type
55     typedef TIn In;
56     /// Output Data Type
57     typedef TOut Out;
58
59     typedef typename In::VecCoord InVecCoord;
60     typedef typename In::VecDeriv InVecDeriv;
61     typedef typename In::MatrixDeriv InMatrixDeriv;
62     typedef Data<InVecCoord> InDataVecCoord;
63     typedef Data<InVecDeriv> InDataVecDeriv;
64     typedef Data<InMatrixDeriv> InDataMatrixDeriv;
65
66     typedef typename Out::VecCoord OutVecCoord;
67     typedef typename Out::VecDeriv OutVecDeriv;
68     typedef typename Out::MatrixDeriv OutMatrixDeriv;
69     typedef Data<OutVecCoord> OutDataVecCoord;
70     typedef Data<OutVecDeriv> OutDataVecDeriv;
71     typedef Data<OutMatrixDeriv> OutDataMatrixDeriv;
72
73 protected:
74     /// Input Model, also called parent
75     SingleLink<Mapping<In,Out>, State< In >, BaseLink::FLAG_STOREPATH|BaseLink::FLAG_STRONGLINK>
76         fromModel;
77     ///State< In >* fromModel;
78     /// Output Model, also called child
79     SingleLink<Mapping<In,Out>, State< Out >, BaseLink::FLAG_STOREPATH|BaseLink::FLAG_STRONGLINK>
80         toModel;
81     ///State< Out >* toModel;
82 public:
83     /// Name of the Input Model
84     ///Data< std::string > object1;
85     ///objectmodel::DataObjectRef m_inputObject;
86     /// Name of the Output Model
87     ///Data< std::string > object2;
88     ///objectmodel::DataObjectRef m_outputObject;
89
90     Data<bool> f_applyRestPosition;
91     Data<bool> f_checkJacobian;
92 protected:
93     /// Constructor, taking input and output models as parameters.
94     ///

```

```

93  /// Note that if you do not specify these models here, you must called
94  /// setModels with non-NULL value before the initialization (i.e. before
95  /// init() is called).
96  Mapping(State< In >* from=NULL, State< Out >* to=NULL);
97  /// Destructor
98  virtual ~Mapping();
99 public:
100  /// Specify the input and output models.
101  virtual void setModels(State< In > * from, State< Out > * to);
102
103  /// Set the path to the objects mapped in the scene graph
104  void setPathInputObject(const std::string &o){fromModel.setPath(o);}
105  void setPathOutputObject(const std::string &o){toModel.setPath(o);}
106
107  /// Return the pointer to the input model.
108  State< In >* getFromModel();
109  /// Return the pointer to the output model.
110  State< Out >* getToModel();
111
112  /// Return the pointer to the input model.
113  helper::vector<BaseState*> getFrom();
114  /// Return the pointer to the output model.
115  helper::vector<BaseState*> getTo();

```