Proceedings
of the 3rd International Modelica Conference,
Linköping, November 3-4, 2003,
Peter Fritzson (editor)

Martin Otter, Hilding Elmqvist and Sven Erik Mattsson
DLR; Dynasim:
The New Modelica MultiBody Library
pp. 311-330

Paper presented at the 3rd International Modelica Conference, November 3-4, 2003, Linköpings Universitet, Linköping, Sweden, organized by The Modelica Association and Institutionen för datavetenskap, Linköpings universitet

All papers of this conference can be downloaded from
http://www.Modelica.org/Conference2003/papers.shtml

The New Modelica MultiBody Library

Martin Otter[1], **Hilding Elmqvist**[2], **and Sven Erik Mattsson**[3]
[1]DLR, Oberpfaffenhofen, Germany, Martin.Otter@dlr.de
[2]Dynasim AB, Lund, Sweden, Elmqvist@dynasim.se
[3]Dynasim AB, Lund, Sweden, SvenErik@dynasim.se
http://www.robotic.dlr.de/Martin.Otter and http://www.dynasim.se

Abstract

A new Modelica library for the modeling and simulation of 3-dimensional mechanical systems has been developed. It will be freely available in the Modelica standard library. Furthermore, the Dymola simulation environment has been considerably enhanced to support the needed features. The MultiBody library is first presented from a user's point of view. Furthermore, all essential details of the implementation are described. The library includes features that are usually not available in other multi-body software, such as analytic handling of a large class of kinematical loops, or the arbitrary connection feature of objects. For example, series connection of 3D line force components is possible.

1 Introduction

The *MultiBody* library is a free Modelica package providing 3-dimensional mechanical components to conveniently model mechanical systems, such as robots, mechanisms, or vehicles. It will be accessible as Modelica.Mechanics.MultiBody and is a replacement of the Modelica library ModelicaAdditions.MultiBody which has been used for a long time. The main design goal of the library and of the supporting features in Dymola [7] was that standard applications can be carried out in a convenient way without knowledge of the Modelica language. The MultiBody library has the following important features:

- Components can be connected together in a nearly arbitrary fashion. If kinematical loop structures occur, they are automatically handled in an efficient way by a new technique explained in section 5. Also force components can be connected directly together, a feature that is usually not available in other multi-body software.
- The non-linear equations occurring in kinematical loops are solved *analytically*, i.e., in a robust and efficient way, for a large class of mechanisms, such as a 4 bar and slider-crank mechanism, or a MacPherson suspension by constructing such loops with elements from the MultiBody.Joints.Assemblies sub package.
- Most joints and all bodies have potential states. A Modelica translator, such as Dymola, will use the generalized coordinates of joints as states if possible. If this is not possible, e.g., because bodies are moving freely in space, states are selected from body coordinates. An advanced user may select states manually from the "Advanced" menu of the corresponding components.
- Whenever a multi-body system model is constructed, all defined components are automatically visualized in an animation using appropriate default sizes and colors. This allows an easy visual check of the constructed model, without extra work of the modeler. Both, the complete animation as well as individual component animation can be switched off. In this case the equations defining animation are removed from the generated code.
- Annotations and assert statements have been introduced that provide in many cases warning or error messages that are related to the library components and not to specific equations as it is usual in Modelica libraries.

2 A First Example

In a first example it shall be demonstrated how to build up, simulate and animate a simple pendulum, consisting of a body and a revolute joint with linear damping in the joint. In Figure 1 the composition diagram of this model is shown. It uses components from the MultiBody library, see figure on next page. Every model utilizing the MultiBody library must

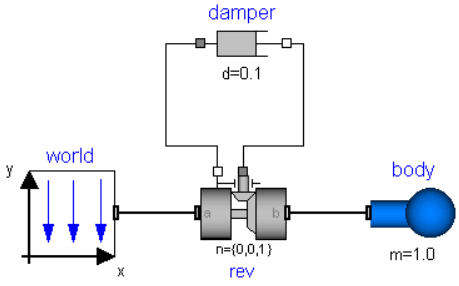


Figure 1. Composition diagram of pendulum

have an instance of the MultiBody.*World* model on top level. The reason is that in the world object the gravity field is defined (no gravity, uniform gravity or point gravity), as well as the default sizes of animation shapes and this information is reported to all used components. Joint "rev" is dragged from Joints.*ActuatedRevolute*, "body" from Parts.*Body* and the "damper" as 1-dimensional force element from "Modelica.Mechanics.Rotational.*Damper*". All components are connected together according to the physical connection structure. After translation, automatically the animation from Figure 2 is shown:

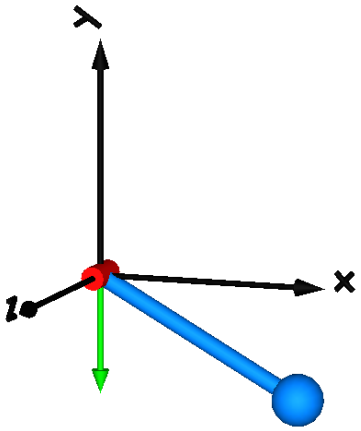


Figure 2. Automatic animation of pendulum

The coordinate system represents the world frame, the green arrow pointing in negative y-axis characterizes the direction of the gravity acceleration, the red cylinder in the world origin is directed along the axis of rotation of the revolute joint, and the light blue cylinder and sphere characterize the body (the center of the sphere is located in the center of mass of the body).

Before translation, the parameters of the dragged components need to be defined. Some parameters are vectors that have to be defined with respect to a local coordinate system of the corresponding component. A convenient way is often a definition of the multi-body model in a configuration where all local frames are parallel to the world frame. This is usually the case when all joint variables, such as the angle of a revolute joint, are zero. Since in such a reference configuration only one coordinate system is essential, the definition is easier as if n frames of n components would have to be taken into account.
The reference configuration for the simple pendulum shall be defined in the following way: The y-axis of the world frame is directed upwards, i.e., the opposite direction of the gravity acceleration. The revolute joint is placed in the origin of the world frame. The rotation axis of the revolute joint is directed along the z-axis of the world frame. The body is placed on the x-axis of the world frame (i.e., the rotation angle of the revolute joint is zero, when the body is on the x-axis). In the following figure, the Dymola menu to define the revolute joint according to this definition is shown:

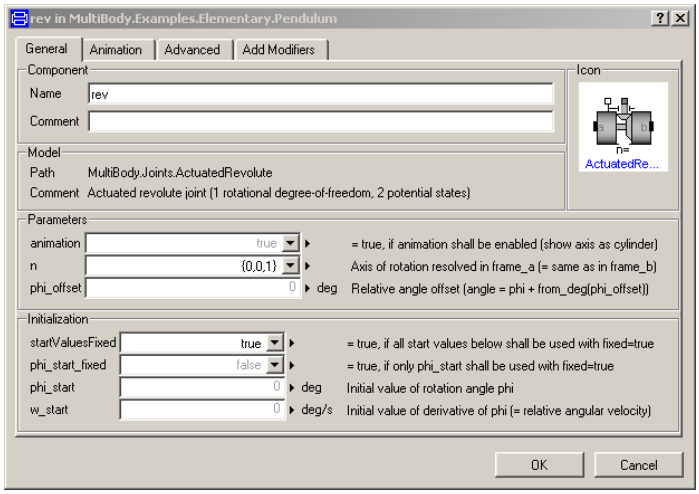


Figure 3. Dymola menu to define a revolute joint

The axis of rotation is defined as `"n = {0,0,1}"` meaning that it is directed into the direction of the z-axis of the World coordinate system in the reference configuration. Accordingly, the body component is defined in Figure 4.

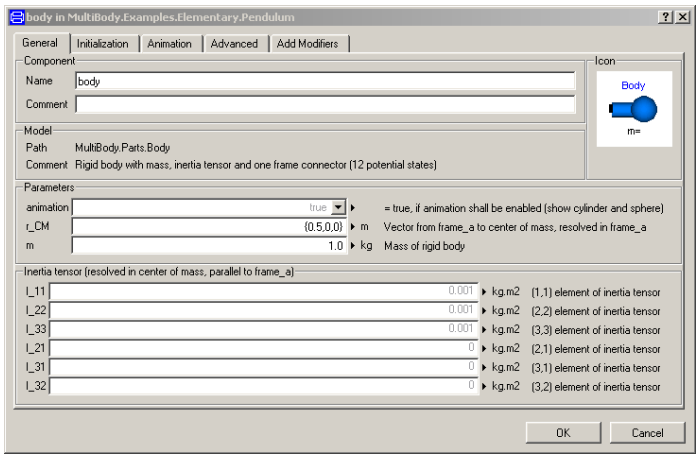


Figure 4. Dymola menu to define a body

The vector "r_CM" from the origin of the "left" coordinate system of the body called "frame_a" to the center of mass of the body is defined as `"r_CM = {0.5, 0, 0}"`, meaning that it is directed 0.5 m along the x-axis of the world frame in the reference configuration. Note, for subsystems in a hierarchical model, e.g., a MacPherson suspension, it is also often convenient to use a local reference configuration for the vector definitions.

3 Describing Orientation

In mechanical systems many variables have to be described with respect to coordinate systems. The notation used in the MultiBody library for this purpose is discussed at hand of Figure 5.

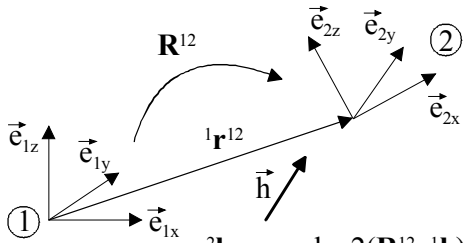


Figure 5. Notation for coordinate systems

For notational convenience the word "frame" is used in the sequel as a synonym for "coordinate system". Frame 1 in Figure 5 is described by 3 unit vectors $\vec{e}_{1x}, \vec{e}_{1y}, \vec{e}_{1z}$ that are orthogonal to each other and Frame 2 is described in a similar fashion by unit vectors $\vec{e}_{2x}, \vec{e}_{2y}, \vec{e}_{2z}$. Frame 2 is defined relatively to frame 1 by the position vector ${}^1\mathbf{r}^{12}$ that is directed from the origin of frame 1 to the origin of frame 2 and is resolved in frame 1, i.e.,

$$\vec{r}^{12} = {}^1\mathbf{r}^{12} \cdot \vec{\mathbf{e}}_1 = \{{}^1r_x^{12}, {}^1r_y^{12}, {}^1r_z^{12}\} \cdot \{\vec{e}_{1x}, \vec{e}_{1y}, \vec{e}_{1z}\}$$

Note, that ${}^1\mathbf{r}^{12}$ is a one-dimensional (Modelica) array that holds the 3 coordinates of vector $\vec{r}^{12}$ with respect to frame 1. In the sequel, (Modelica) arrays with one or two dimensions are always characterized by bold face characters if the complete array is referenced.

The relative orientation of frame 2 with respect to frame 1 is defined by the "orientation object" $\mathbf{R}^{12}$ (also called "rotation object"). There are different ways to mathematically describe orientation. To ease usage, the MultiBody library is designed such that knowledge about the actual description form of orientation is not necessary. This is achieved by providing a pre-defined type

`MultiBody.Frames.Orientation`

and utility functions in `MultiBody.Frames` operating on instances of this type. The two most important functions are shown in Figure 5: An arbitrary vector $\vec{h}$ might be represented by its coordinates with respect to frame 1 (${}^1\mathbf{h}$) or with its coordinates with respect to frame 2 (${}^2\mathbf{h}$), respectively. If either of the two representations is given, the other one can be computed in the following way:

```
import MultiBody.Frames;
Frames.Orientation R12;
Real h1[3] "h resolved in frame 1"
Real h2[3] "h resolved in frame 2"
equation
  h2 = Frames.resolve2(R12, h1);//or
  h1 = Frames.resolve1(R12, h2);
```

There are about 30 of these utility functions in sub library MultiBody.Frames. We will explain some more of them when needed. Note, that with every orientation object a direction is associated. E.g., the inverse orientation R21 of R12 is computed by "`R21 = Frames.inverseRotation(R12)`".

During the development of the MultiBody library, 3 different representation forms of the orientation object have been implemented:

1. Transformation matrix $\mathbf{T}$ (${}^2\mathbf{h} = \mathbf{T}^{12} \cdot {}^1\mathbf{h}$).
2. Two rows of the transformation matrix.
3. Quaternions (see, e.g., [16]).

Benchmark tests revealed that the transformation matrix leads usually to the most efficient code and therefore this representation form was selected. Since in some situations quaternions are useful, the implemented functions operating on quaternions are provided in the MultiBody library under MultiBody.Frames.Quaternions. Also some quite involved functions are present, e.g., to compute quaternions from a transformation matrix in a numerically robust way (Quaternions.from_T).

Dymola has the built-in rule that functions with one statement are always "inlined" before they are used. Most of the utility functions in MultiBody.Frames are therefore defined just with one statement to enforce inlining, in order (a) to **not** have any function call overhead, (b) to allow symbolic rearrangement of terms and (c) that symbolic differentiation is possible. Other tools using the MultiBody library should also have support for inlining in order to get efficient code.

4 MultiBody Frame Connector

We are now in the position to present the design of the "Frame" connector that is used to connect multi-body components together. All variables used in this connector are displayed in Figure 6: A coordinate system "frame a" is rigidly fixed at an attachment point of a mechanical part. This Frame is described with respect to the world frame by the

- position vector ${}^0\mathbf{r}^{0a}$ that is directed from the origin of the world frame to the origin of frame a and is resolved in the world frame and by the
- orientation object $\mathbf{R}^{0a}$ describing the relative orientation between the world frame and frame a.

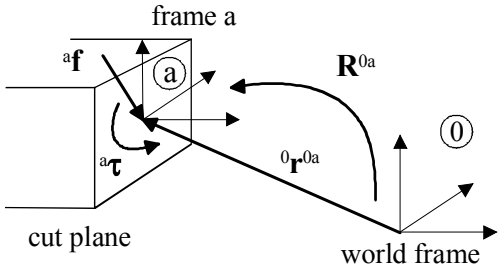


Figure 6. MultiBody "Frame" connector

It is assumed that a free body diagram is constructed, i.e. that a cut is performed between mechanical parts that shall be connected together at frame a. In the cut plane a resultant cut force $^{a}\mathbf{f}$ and a resultant cut torque $^{a}\boldsymbol{\tau}$ act on frame a. Both vectors are resolved in this frame.

```
connector Frame
  import SI = Modelica.SIunits;
  SI.Position        r_0[3]"= 0r0a";
  Frames.Orientation R     "= R0a";
  flow SI.Force      f[3]  "= af";
  flow SI.Torque     t[3]  "= aτ";
end Frame;

connector Frame_a = Frame;
connector Frame_b = Frame;
```

The four previously defined variables are used in the connector. The additional connectors Frame_a and Frame_b have the identical definition as connector Frame. The only difference is that Frame_a and Frame_b have different icons in order to be able to distinguish Frame connectors more easily in a composition diagram.

The cut force and cut torque are **flow** variables in order that the force and torque balance at a point where several components are connected together is fulfilled. Note, that two connected frames (a and b) coincide, since a.r_0 = b.r_0 and a.R = b.R due to the connection rules of Modelica.

The orientation between two frames can be described by 3 independent variables, see, e.g., [16][18]. Unfortunately, every such description form has a singularity and therefore cannot be used in a connector. For this reason, an orientation object has to be described by a set of **redundant** variables that are related to each other with constraint equations. In the MultiBody library the orientation object is described by a transformation matrix that has 9 entries, i.e., a highly redundant description form. This property leads to significant difficulties and is one of the reasons why it needed so long time to come up with a "truly" object-oriented multi-body library (E.g. the first Dymola multi-body library was developed in 1994 [17]).

In several components, such as a body or a sensor, velocities or accelerations of connector variables are needed. These derivatives can be easily obtained in the following way:

```
  import SI = MultiBody.SIunits;
  import MultiBody.Interfaces;
  import MultiBody.Frames;
  Interfaces.Frame_a     frame_a;
  SI.Velocity            v_0[3];
  SI.Acceleration        a_0[3];
  SI.AngularVelocity     w_a[3];
  SI.AngularAcceleration z_a[3];
equation
  v_0 = der(frame_a.r_0);
  a_0 = der(v);
  w_a = Frames.angularVelocity2(
          frame_a.R,der(frame_a.R));
  z_a = der(w_a);
```

As can be seen, the velocity v_0 and the acceleration a_0 of the origin of frame_a (resolved in the world frame) are simply computed by applying the derivative operator **der**(..). The angular velocity of frame_a is computed with a function that requires as input the orientation object **R** and its derivative d**R**/dt and returns the angular velocity $^{a}\boldsymbol{\omega}^{a}$ resolved in frame_a according to Poisson's equation. With $\mathbf{R}^{T} = [\mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z]$, $^{a}\boldsymbol{\omega}^{a}$ is computed as:

$$^{a}\boldsymbol{\omega}^{a} = \{\mathbf{e}_z^T \cdot \dot{\mathbf{e}}_y, \quad -\mathbf{e}_z^T \cdot \dot{\mathbf{e}}_x, \quad \mathbf{e}_y^T \cdot \dot{\mathbf{e}}_x\}$$

Applying the derivative operator **der**(...) on w_a results in the angular acceleration, resolved in frame_a, since according to Euler's differentiation rule ($^{i}d\vec{h}/dt = {}^{k}d\vec{h}/dt + \vec{\omega}^{ik} \times \vec{h}$):

$$\begin{aligned} {}^{0}d\vec{\omega}^{a}/dt &= {}^{a}d\vec{\omega}^{a}/dt + \vec{\omega}^{a} \times \vec{\omega}^{a} \\ &= {}^{a}d\vec{\omega}^{a}/dt \end{aligned}$$

where $^{i}d\vec{h}/dt$ is the derivative of vector $\vec{h}$ with respect to coordinate system i and $\vec{\omega}^{a}$ is the absolute angular velocity of frame_a.

In books about multi-body systems it is usually recommended to compute the angular velocity by recursive calculations and it is claimed that this is much more efficient as using the direct application of Poisson's equation as it is performed with function "angularVelocity2" above. For a "truly" object-oriented library it is difficult or not possible to apply a recursive calculation directly since in an object only relations between connector variables can be formulated. It turns out that the generated code of the MultiBody library is nearly as efficient as from the ModelicaAdditions.MultiBody library where the angular velocity is computed recursively. This is due to the particular implementation of Poisson's equation and Dymola's symbolic capabilities.

5 Overdetermined DAEs

By collecting together all explicit equations in a Modelica model and its submodels and all equations due to "connect" statements, a Modelica model is mapped to a DAE (= Differential Algebraic Equation system) of the following form:

$$\mathbf{0} = \mathbf{f}(d\mathbf{x}/dt, \mathbf{x}, \mathbf{y}, t)$$

where **x** contains all variables appearing differentiated and **y** contains all pure algebraic variables. To get efficient code, this DAE has to be symbolically processed and transformed to state space form (at least numerically) with a subset of **x** as states. This is performed by BLT partitioning [8] to get a sequential model evaluation and to identify algebraic loops, the Pantelides algorithm [19] to determine equations to be differentiated and the dummy derivative method [13] to select independent states (this method can be interpreted as a variant of the currently popular "projection methods" of higher index DAEs). All these algorithms require that dim(**f**) = dim(**x**) + dim(**y**), i.e., the number of equations has to be identical to the number of unknown variables.

Whenever the variables in a connector are **not** independent from each other, connection structures that have loops may result in a DAE where there are more equations as unknowns, i.e., dim(**f**) > dim(**x**) + dim(**y**). Usually, this overdetermined set of equations is still consistent, so that a unique mathematical solution exists. Since the Frame connector has an overdetermined set of variables due to the orientation object, also models of the MultiBody library may result in an overdetermined DAE.

It seems unlikely that the symbolic algorithms from above can be generalized to directly handle such DAEs, because it is not possible to distinguish consistently overdetermined DAEs from erroneous DAEs (that are a result of modeling errors), by pure structural information. For this reason, the only practical way seems to be to mark the overdetermined equation subset in the model and transform this set of equations before the standard algorithms from above are applied. One such way of marking and transforming an overdetermined set of equations has been designed for the next version 2.1 of the Modelica language and has been implemented in Dymola version 5.1. This approach is sketched in the rest of the section.

It is assumed that overdetermined DAEs are due to overdetermined sets of (non **flow**) variables **v** in connectors. Such connectors will be called "overdetermined connectors" in the sequel. When connecting two or more overdetermined connectors together, equality equations for corresponding overdetermined variable sets are generated, such as "**v**1 = **v**2". Whenever, say, **v**1 is computed in one component and then passed to the next component via a "connect" statement, everything is fine, because **v**2 is uniquely computed from **v**1 by "**v**2 := **v**1". Difficulties arise, if both **v**1 = **v**1(**x**) and **v**2 = **v**2(**x**) are computed from potential state variables **x**, since a connection equation **v**1 = **v**2 imposes an overdetermined (but consistent) set of constraints on the variables **x**.

The basic requirement is that the developer of an overdetermined connector provides a function called "equalityConstraint(**v**1,**v**2)" that returns a **non-redundant** set of residues that should be zero if the equality constraint **v**1 = **v**2 is fulfilled. In a pre-processing step of the model equations, a translator has then to decide for every connection set whether an equation of the form "**v**1 = **v**2" or an equation of the form "0 = equalityConstraint(**v**1,**v**2)" has to be added to the DAE. Let us demonstrate this by considering the Frame connector.

Modelica is enhanced such that a type or record declaration may optionally contain a definition of function "equalityConstraint(...)":

```
type Orientation
  extends Real[3,3];

  function equalityConstraint
    input  Orientation R1;
    input  Orientation R2;
    output Real residue[3];
  protected
    Orientation R_rel;
  algorithm
    R_rel = R2*transpose(R1);
    residue := {R_rel[2,3],
                R_rel[3,1],
                R_rel[1,2]};
  end equalityConstraint;
end Orientation;
```

An orientation object is defined by a transformation matrix of dimension [3,3]. Two orientation objects, i.e., transformation matrices, $\mathbf{R}_1$ and $\mathbf{R}_2$ are identical ($\mathbf{R}_1 = \mathbf{R}_2$) if the relative transformation matrix between $\mathbf{R}_1$ and $\mathbf{R}_2$, i.e., $\mathbf{R}_{rel} = \mathbf{R}_2 \cdot \mathbf{R}_1^T$ is the unit matrix. A transformation matrix describing a small rotation can be approximated by (see, e.g., [18])

$$\mathbf{R}_{rel} \approx \begin{bmatrix} 1 & \varphi_3 & -\varphi_2 \\ -\varphi_3 & 1 & \varphi_1 \\ \varphi_2 & -\varphi_1 & 1 \end{bmatrix}$$

where $\varphi_1,\varphi_2,\varphi_3$ are a set of 3 **independent** variables describing the deviation from the unit matrix. As a result, if the outer diagonal elements [2,3], [3,1] and

[1,2] of $\mathbf{R}_{rel}$ vanish, then $\mathbf{R}_1 = \mathbf{R}_2$. Therefore, these 3 outer diagonal elements are returned as residues by function equalityConstraint(...). To summarize, a connection between two Frame connectors will either result in **9** equations $\mathbf{R}_1 = \mathbf{R}_2$ to define the equality between two orientation objects or in **3** equations by calling function equalityConstraint(...). If appropriately selected, the result is a regular DAE where the number of equations is identical to the number of unknowns. A call to function equalityConstraint(...) will usually result in a non-linear system of equations that has only the desired solution $\mathbf{R}_1 = \mathbf{R}_2$, if the initial guess values of the iteration variables are close enough to this solution.

The remaining open question is how a tool can decide which connection equations to use? An informal description is given below. Details of the algorithm are sketched in the appendix.

A new package called "Connections" is introduced in Modelica, containing a set of built-in operators to mark overdetermined equations. Let us sketch these operators using the orientation object R as an example:

- **root**(A.R) defines that the orientation object R in connector A is computed in a consistent way. The world object has such a definition because R is defined as identity matrix.
- **branch**(A.R, B.R) defines that there is an algebraic relationship between the orientation object A.R in connector A and the orientation object B.R in connector B. Joint objects have such a definition, if there is an algebraic constraint between frame_a.R and frame_b.R.

These two operators are already sufficient, since a tool can determine whether the graph constructed with root(...), connect(...) and branch(...) statements contains loops. These loops have to be cut and for every cut the equalityConstraint(...) function has to be used to state the equality of orientation objects.

If there is a free flying body, coordinates of the body should be used as states from which the orientation object in the body connector can be computed. This in turn means that a free flying body is also a root in the graph. Formally, this situation is defined by operators:

- **potentialRoot**(A.R) defines that the orientation object R in connector A might be computed in a consistent way, if this is necessary. Body objects have such a definition.
- **isRoot**(A.R) returns true if the orientation object A.R has been selected as a root. This means that different equations have to be provided.

The sketched method to handle overdetermined DAEs with symbolic transformation techniques is not specific to multi-body systems. For example, efficient implementations of electric power systems use the Park transformation to define currents and voltages in the connector **relatively** to the harmonic, high-frequency signal of a power source that is described by the angle of the rotor of the source. This allows much faster simulations, since the basic high frequency signal of the power source is not part of the differential equations. On the other hand, the source angle has to be included into the connector leading to an overdetermined description that can be handled with the method presented in this section.

6 Elementary Components

Using the "Frame" connector and the utility functions in MultiBody.Frames, it is straightforward to implement the elementary components that are usually available in multi-body programs.

The MultiBody library has about 40 components. The most important ones are shown in Table 1. Contrary to approaches described in text books about this topic, equations are **only** defined on "position" level. A tool has enough information to figure out via the Pantelides algorithm [19] which equations have to be differentiated in order to transform the DAE to state space form with the dynamic dummy derivative method [13][14]. This feature simplifies the implementation and the understanding of the MultiBody library considerably.

In the left column of Table 1, the icon of the respective model is shown whereas in the right column the essential equations are given that are mapped directly to Modelica equations in the library. Abbreviations which are used for variable and function names in the right column (to save space) are stated at the top row of Table 1. The new built-in operators "root", "isRoot", "branch", "potentialRoot" from Table 1 are actually within a package "Connections" (the correct name would therefore be, e.g., Connections.root). All other used functions are from subpackage MultiBody.Frames. Let us discuss the components in a bit more detail, see Table 1.

6.1 MultiBody.World

In the World model essentially the position vector of its frame connectors is set to zero and the orientation object of the frame is set to a null rotation (e.g., the transformation matrix is the identity matrix). When dragging MultiBody.World into a model, the following declaration is generated (this behavior is defined via an annotation):

```
inner MultiBody.World world;
```

This is necessary since nearly all components have a corresponding "**outer**" declaration to access the definitions in the world object, such as defaults for animation and the gravity function. In components that have a mass, the function world.gravityAcceleration(**r**) is called to inquire the gravity acceleration at position **r**. Depending on user input, different gravity fields can be used. Currently, no gravity field, parallel and point gravity field is supported. This allows, e.g., to easily simulate a satellite in the gravity field of the earth. An example is given in Figure 7.

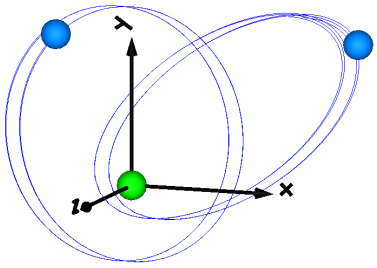


Figure 7. Two point masses in a point gravity field

If the World object is missing in a model, a warning message is printed and an instance of the World object with default settings is automatically utilized. This feature is again defined via an annotation (this is useful for any type of inner declaration).

6.2 MultiBody.Parts.FixedTranslation

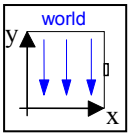
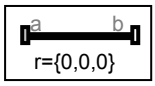
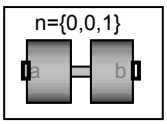
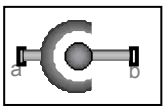
This component defines a fixed translation of a frame. It is, e.g., used to define frames for several attachment points on a body. The equations state that the position vector of frame_b is defined from the position vector of frame_a and the relative position vector ${}^a\mathbf{r}^{ab}$ from frame_a to frame_b (${}^a\mathbf{r}^{ab}$ is defined as parameter "r"). Since frames are translated, the orientation objects in the two frames are set equal. This in turn requires a "Connections.branch(...)", see section 5. Finally, a force and torque balance of this massless part is present in the Modelica model.

6.3 MultiBody.Joints.Revolute

This component defines a rotation along an axis vector $\mathbf{n} = {}^a\mathbf{n} = {}^b\mathbf{n}$ via angle φ. When φ = 0, frame_a and frame_b coincide. As with most other joints, the generalized coordinates (here: φ and $\omega = \dot{\varphi}$) have the attribute stateSelect = StateSelect.prefer in order that they are selected as states if possible. Since the origins of both frames are located at the same point on the axis of rotation, the position vectors in the two frames are identical. The relative orientation object $\mathbf{R}^{rel}$ is computed with **n** and φ. It is used to define the relationship between the orientation objects from frame_a and frame_b. It is also stated

Abbreviations:

$\mathbf{r}^a, \mathbf{R}^a, \mathbf{f}^a, \boldsymbol{\tau}^a$:= frame_a.r_0, .R, .f, .t
$\mathbf{r}^b, \mathbf{R}^b, \mathbf{f}^b, \boldsymbol{\tau}^b$:= frame_b.r_0, .R, .f, .t
absRotation := Frames.absoluteRotation
relRotation := Frames.relativeRotation
angVel2 := Frames.angularVelocity2
Q.angVel2 := Frames.Quaternions.angularVelocity2
Q.constraint := Frames.Quaternions.orientationConstraint
grav := world.gravityAcceleration

World 	root(frame_b.R) $\mathbf{r}^b = \mathbf{0}$ $\mathbf{R}^b = \text{nullRotation}()$
Parts.FixedTranslation 	branch(frame_a.R, frame_b.R) $\mathbf{r}^b = \mathbf{r}^a + \text{resolve1}(\mathbf{R}^a, {}^a\mathbf{r}^{ab})$ $\mathbf{R}^b = \mathbf{R}^a$ $\mathbf{0} = \mathbf{f}^a + \mathbf{f}^b$ $\mathbf{0} = \boldsymbol{\tau}^a + \boldsymbol{\tau}^b + {}^a\mathbf{r}^{ab} \times \mathbf{f}^b$
Joints.Revolute 	branch(frame_a.R, frame_b.R) $\mathbf{r}^b = \mathbf{r}^a$ $\mathbf{R}^{rel} = \text{planarRotation}(\mathbf{n}, \varphi)$ $\mathbf{R}^b = \text{absRotation}(\mathbf{R}^a, \mathbf{R}^{rel})$ $\omega = \dot{\varphi}$ $0 = \mathbf{n}^T \cdot \boldsymbol{\tau}^b$ $\mathbf{0} = \mathbf{f}^a + \text{resolve1}(\mathbf{R}^{rel}, \mathbf{f}^b)$ $\mathbf{0} = \boldsymbol{\tau}^a + \text{resolve1}(\mathbf{R}^{rel}, \boldsymbol{\tau}^b)$
Joints.Spherical 	//**no** branch(...) $\mathbf{r}^b = \mathbf{r}^a$ $\mathbf{R}^{rel} = \text{relRotion}(\mathbf{R}^a, \mathbf{R}^b)$ $\mathbf{0} = \mathbf{f}^a + \text{resolve1}(\mathbf{R}^{rel}, \mathbf{f}^b)$ $\boldsymbol{\tau}^a = \mathbf{0}$ $\boldsymbol{\tau}^b = \mathbf{0}$

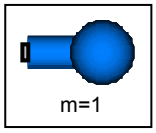
Parts.Body
 
	potentialRoot(frame_a.R)
	if isRoot(frame_a.R) **then**
	$0 = \text{Q.constraint}(\mathbf{p})$
	$\boldsymbol{\omega}^a = \text{Q.angVel2}(\mathbf{p}, \dot{\mathbf{p}})$
	$\mathbf{R}^a = \text{Frames.from\_}(\mathbf{p})$
	else
	$\boldsymbol{\omega}^a = \text{angVel2}(\mathbf{R}^a, \dot{\mathbf{R}}^a)$
	$\mathbf{p} = \text{Q.nullRotation}()$
	end if
	$\mathbf{v} = \dot{\mathbf{r}}^a$
	$\mathbf{g} = \text{grav}(\mathbf{r}^a + \text{resolve1}(\mathbf{R}^a, \mathbf{r}^{CM}))$
	$\mathbf{a} = \text{resolve2}(\mathbf{R}^a, \dot{\mathbf{v}} - \mathbf{g})$
	$\mathbf{f}^a = m \cdot (\mathbf{a} + \dot{\boldsymbol{\omega}}^a \times \mathbf{r}^{CM} + \boldsymbol{\omega}^a \times (\boldsymbol{\omega}^a \times \mathbf{r}^{CM}))$
	$\boldsymbol{\tau}^a = \mathbf{I}\dot{\boldsymbol{\omega}}^a + \boldsymbol{\omega}^a \times \mathbf{I}\boldsymbol{\omega}^a + \mathbf{r}^{CM} \times \mathbf{f}^a$

Table 1. Elementary components of MultiBody library

that the projection of the cut-torque on **n** must vanish. Finally, the force and torque balance of this massless part is present. Besides model "Revolute" there is also a joint "ActuatedRevolute" that has an additional 1-dim. flange connector. Via this flange, a drive train can be attached driving the revolute

joint, e.g., with components from the Modelica.-Mechanics. Rotational library (see Figure 1).

There is an additional utility function "rooted(…)" to inquire whether there is a path in the spanning trees of the virtual connection graphs from a selected root to the frame under consideration. This is used here and at some other places to give two equation variants depending on the actual connection structure in order to avoid small linear algebraic equations. For example, if rooted(frame_a.R) = **true** then the force and torque at frame_a are computed from the frame_b quantities. Otherwise, the force and torque at frame_b are computed from the frame_a quantities.

6.4 MultiBody.Joints.Spherical

This component defines a spherical joint, i.e., the origins of frame_a and frame_b coincide and the two frames can freely rotate relative to each other. No torques are transmitted via this joint. Since frame_a.R and frame_b.R are not related together in an algebraic equation, **no** "branch(...)" statement is present. No states are defined for this joint.

6.5 MultiBody.Parts.Body

This component defines the mass and inertia properties of a body. It has one frame_a that is usually used as reference coordinate system of a part which is associated with a specific geometric position on the part. Other points on the part are often defined via FrameTranslation components connected to frame_a of the body component. The mass m, the position vector $\mathbf{r}^{CM} = {}^{a}\mathbf{r}^{aCM}$ from the origin of frame_a to the center of mass (resolved in frame_a) and the inertia tensor $\mathbf{I} = {}^{a}\mathbf{I}^{CM}$ with respect to the center of mass are given as parameters and define the body properties, see also Table 1.

The body component is defined as "potentialRoot", i.e., it may be selected as root of a spanning tree of the virtual connection graph. Whether it is selected or not can be inquired via function "isRoot(...)". If the body frame is **not** selected as root, the orientation object in the frame is defined somewhere else. In this case the second branch of the if clause in Table 1 is used and the angular velocity of the body frame is determined by frame_a.R and its derivative which for example means that it is computed (indirectly) by the generalized position and velocity variables of joints.

If "isRoot(...) = **true**", it is required that frame_a.R is calculated within the body object. This is only possible if variables of the body are used as states from which frame_a.R can be determined. By default, quaternions **p** are used as potential states. Consequently frame_a.R is computed from **p** and the angular velocity is computed from **p** and its derivative $\dot{\mathbf{p}}$. The 4 coordinates of the quaternion vector **p** have to fulfill the constraint equation "$\mathbf{p}^T \cdot \mathbf{p} = 1$". This non-linear equation is added in the first if-clause. Since there is a non-linear equation relating potential states, a tool has to use the dynamic dummy derivative method to dynamically select 3 states out of 4 potential states during simulation. Whenever the selection comes close to its singularity, Dymola changes the states at a completed step of the integrator. The 4th potential state has to be computed by solving the non-linear quaternion constraint equation. Dymola performs this in an efficient and robust way, because it can detect that the special non-linear equation of quaternions is present and solves this equation analytically. E.g., if p[1:3] are selected as states, then

p[4] = sqrt(1 – p[1:3]*p[1:3])*signAtLastStep(p[4]).

Via a parameter in the "Advanced" menu of the body object, it is possible to alternatively also use the 3 Cardan angles as states. They are defined with respect to a coordinate system "Fix" fixed in frame_a. Whenever the Cardan angles come close to their singularity, frame "Fix" is changed such that the new Cardan angles are far away from their singularity. The advantage of this approach is that no dynamic dummy derivative method is needed. The disadvantage is that every change of states results in a state event which is less efficient as the state change performed with the dynamic dummy derivative method. Furthermore, several variables are discontinuous (especially the Cardan angles) which can lead to problems if equations are further differentiated, e.g., for inverse models.

The non-standard feature to have potential states both in **joints** and in **bodies** is especially useful for inexperienced users, since they do not have to introduce a "virtual" joint with 6 degrees of freedom. For example, it is easy to just build up a system as in Figure 8, where a body is connected via a spring to the environment.

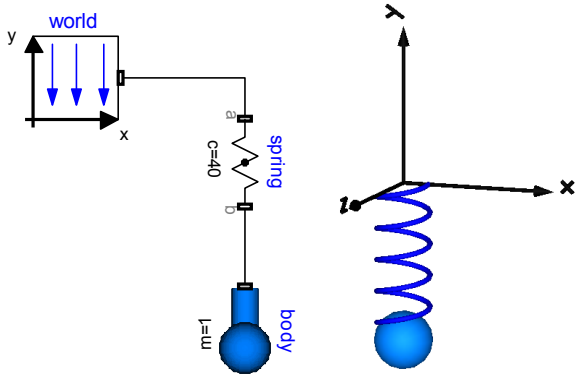


Figure 8. Free body with spring

In the left part of the figure the Modelica schematic and in the right part the default animation is shown. No "non-physical" joint has to be introduced to build up such a model, as it is usually the case in other multi-body programs.

Let us now return to the body equations in Table 1. Once the orientation object and the angular velocity of the body frame are determined, all other kinematical quantities are derived by differentiation and used in the Newton/Euler equations that are formulated with respect to frame_a of the body (and not with respect to the center of mass).

7 Loop Structures

Due to the new handling of overdetermined DAEs, the modeler does not have to take special actions if loop structures occur (contrary to the ModelicaAdditions.MultiBody library). An example is presented in Figure 9. It is available as MultiBody.Examples.Loops.Fourbar1. In the upper

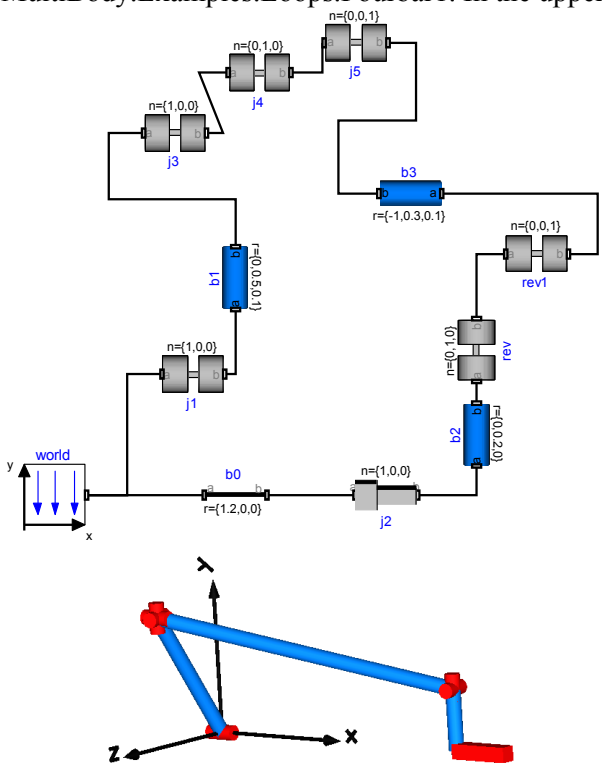


Figure 9. Four bar mechanism with 7 joints and 1 dof

part of the figure the Modelica schematic of a four bar mechanism is shown constructed with the MultiBody library. It consists of 6 revolute, 1 prismatic joint and forms a kinematical loop. This mechanism has one degree of freedom.

In the lower part of the figure the default animation is shown. Note, that the axes of the revolute joints are represented by the red cylinders and that the axis of the prismatic joint is represented by the red box on the lower right side.

Whenever loop structures occur, non-linear algebraic equations are present on "position level". It is then usually not possible by structural analysis to select states during translation (which is possible for non-loop structures). In the example above, Dymola detects a non-linear algebraic loop of 57 equations and reduces this to a system of 7 coupled algebraic equations. Note, that this is performed without using any "cut-joints" as it is usually done in multi-body programs, but by just appropriate symbolic equation manipulation. Via the dynamic dummy derivative method the generalized coordinates on position and velocity level from one of the 7 joints are dynamically selected as states during simulation. Whenever, these two states are no longer appropriate, states from one of the other joints are selected.

The efficiency of loop structures can usually be enhanced, if states are statically fixed at translation time. For this mechanism, the generalized coordinates of joint j1 can always be used as states. This can be stated by setting parameter "enforceStates = **true**" in the "Advanced" menu of the desired joint. This flag sets the attribute stateSelect of the generalized coordinates of the coresponding joint to "StateSelect.always". When setting this flag to **true** for joint j1 in the four bar mechanism, Dymola detects a non-linear algebraic loop of 40 equations and reduces this to a system of 5 coupled non-linear algebraic equations.

7.1 Planar Loops

In Figure 10 the model of a V6 engine is shown that has a simple combustion model. It is available as MultiBody.Examples.Loops.EngineV6. The Modelica schematic of one cylinder is given in the middle part of the figure. Connecting 6 instances of this cylinder appropriately together results in the engine schematic displayed at the upper part of the figure. In the lower part the animation of the engine is shown. Every cylinder consists essentially of 1 prismatic and 2 revolute joints that form a planar loop, since the axes of the two revolute joints are parallel to each other and the axis of the prismatic joint is orthogonal to the revolute joint axes. All 6 cylinders together form a coupled set of 6 loops that have together 1 degree of freedom.

All planar loops, and especially the engine, result in a DAE that does not have a unique solution. The reason is that, e.g., the cut forces in direction of the axes of the revolute joints cannot be uniquely computed. Any value fulfills the DAE equations.

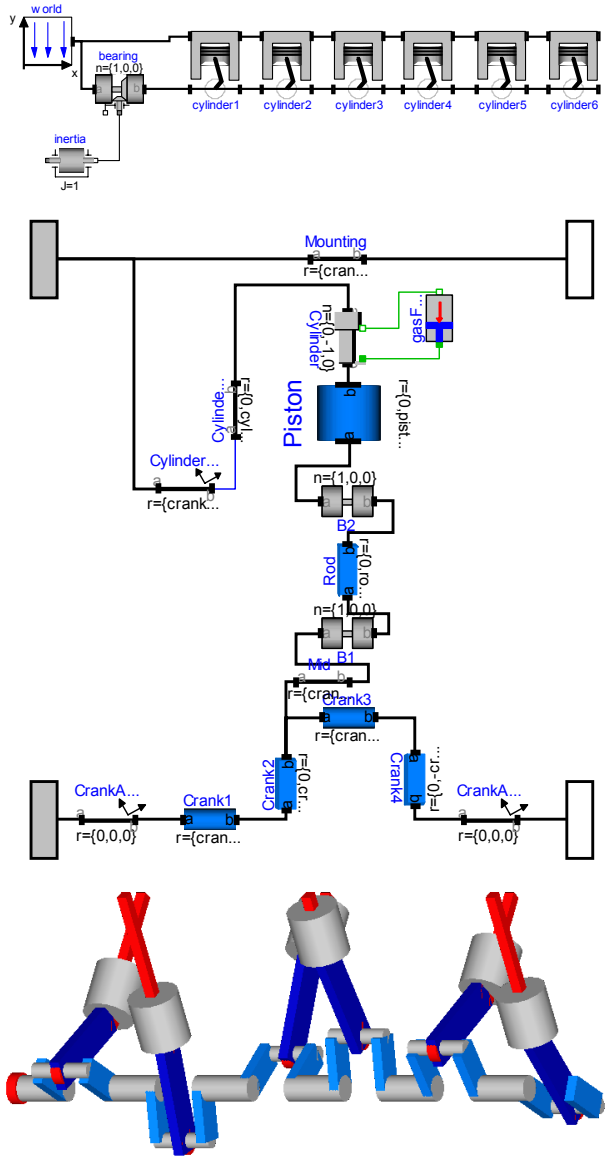


Figure 10. V6 engine with 6 planar loops and 1 dof

This is a structural property that is determined by the symbolic algorithms. Since they detect that the DAE is structurally singular, a further processing is not possible. Without additional information it is also impossible that the symbolic algorithms could be enhanced because if the axes of rotations of the revolute joints are only slightly changed such that they are no longer parallel to each other, the planar loop can no longer move and has 0 degrees of freedom. Algorithms based on pure structural information cannot distinguish these two cases.

The usual remedy is to remove superfluous constraints, e.g., along the axis of rotation of **one** revolute joint. Since this is not easy for an inexperienced modeler, the flag "planarCutJoint" is provided in the "Advanced" menu of a revolute joint that removes these constraints. This flag must be set to **true** for one revolute joint in every planar loop. In the engine example, this flag is set in the revolute joint B2 in the cylinder model.

If a modeler is not aware of the problems with planar loops and models them without special consideration, Dymola displays an error message and points out that a planar loop may be the reason and suggests to use the "planarCutJoint" flag. This error message is due to an annotation in the Frame connector:

```
flow SI.Force f[3] annotation(
        unassignedMessage="..");
```

If no assignment can be found for some forces in a connector, the "unassignedMessage" is displayed. In most cases the reason for this is a planar loop or two joints that constrain the same motion. Both cases are discussed in the message.

Note that the non-linear algebraic equations occurring in planar loops can be solved analytically in most cases and therefore it is highly recommended to use the techniques discussed in the next two sections for such systems.

7.2 Analytic Loop Handling: User's View

It is well known that the non-linear algebraic equations of most mechanical loops in technical devices can be solved analytically. It is, however, difficult to perform this fully automatically and therefore none of the commercial, general purpose multi-body programs, such as MSC ADAMS[1], LMS DADS[5], SIMPACK[21], have this feature. These programs solve loop structures with pure numerical methods. Multi-body programs that are designed for real-time simulation of the dynamics of specific vehicles, such as ve-DYNA[23], usually contain manual implementations of a particular multi-body system (the vehicle) where the occurring loops are either analytically solved, if this is possible, or are treated by table look-up where the tables are constructed in a pre-processing phase. Without these features the required real-time capability would be difficult to achieve.

In a series of papers and dissertations, especially [10][24][11][15], Prof. Hiller and his group in Duisburg have developed systematic methods to handle mechanical loops analytically. The "characteristic pair of joints" method [10][24] basically cuts a loop at two joints and uses geometric invariants to reduce the number of algebraic equations, often down to one equation that can be solved analytically. Also several multi-body codes have been developed that are based on this method, e.g., MOBILE [12]. Besides the very desired feature to solve non-linear algebraic equations analytically, i.e., efficiently and in a robust way, there are several drawbacks: It is

difficult to apply this method automatically. Even if this would be possible in a good way, there is always the problem that it cannot be guaranteed that the statically selected states lead to no singularity during simulation. Therefore, the "characteristic pair of joints" method is usually manually applied which requires know-how and experience.

In the MultiBody library the "characteristic pair of joints" method is supported in a restricted form such that it can be applied also by non-specialists. The idea is to provide joint aggregations in package MultiBody.Joints.**Assemblies** as one object that either have 6 degrees of freedom or 3 degrees of freedom (for usage in planar loops).

As an example, a variant of the four bar mechanism from Figure 9 is given in Figure 11. In the upper part of the figure, the mechanism is modeled with standard joints. In the lower part, the two spherical joints and the prismatic joint are collected together in an assembly object called "jointSSP" that is defined in

MultiBody.Joints.Assemblies.JointSSP.

This joint aggregation has a frame at the left side of the left spherical joint (frame_a) and a frame at the right side of the prismatic joint (frame_b). JointSSP, as all other objects from the Joints.Assemblies package, has the property, that **the generalized coordinates, and all other frames defined in the assembly, can be calculated given the movement of frame_a and of frame_b**. This is performed by **analytically** solving non-linear systems of equations (details are given in the next subsection). From a structural point of view, the equations in an assembly object are written in the form

$$\mathbf{q} = \mathbf{f}_1(\mathbf{r}^a, \mathbf{R}^a, \mathbf{r}^b, \mathbf{R}^b)$$

where $\mathbf{r}^a$, $\mathbf{R}^a$, $\mathbf{r}^b$, $\mathbf{R}^b$ are the variables defining the position and orientation of the frame_a and frame_b connector (see also Table 1) and $\mathbf{q}$ are the generalized positional coordinates inside the assembly, e.g., the angle of a revolute joint. Given angle φ of revolute joint j1 from the four bar mechanism, frame_a and frame_b of the assembly object can be computed by a forward recursion

$$(\mathbf{r}^a, \mathbf{R}^a, \mathbf{r}^b, \mathbf{R}^b) = \mathbf{f}(\varphi)$$

Since this is a structural property, the symbolic algorithms can automatically select φ and its derivative as states and then all positional variables can be computed in a forwards sequence. It is now understandable that Dymola transforms the equations of the four bar mechanism to a recursive sequence of statements that has neither linear nor non-linear algebraic loops (remember, the previous "straightforward" solution had a nonlinear system of equations of order 5).

The aggregated joint objects consist of a combination of either a revolute or prismatic joint and of a rod that has either two spherical joints at its two ends or a spherical and a universal joint, respectively. For all combinations, analytic solutions can be determined. For planar loops, combinations of 1, 2 or 3 revolute joints with parallel axes and of 2 or 1 prismatic joint with axes that are orthogonal to the revolute joints can be treated analytically. The currently supported combinations are listed in Table 2. The missing combinations (such as JointSUP or Joint RPP) will be added in one of the next releases.

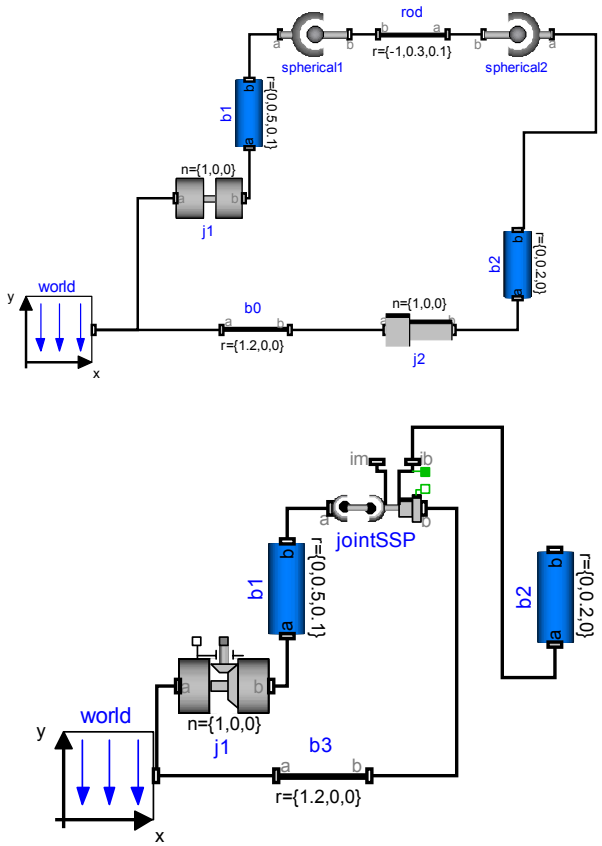


Figure 11. Analytic handling of four bar mechanism

3-dimensional Loops:	
JointSSR	Spherical – Spherical – Revolute
JointSSP	Spherical – Spherical – Prismatic
JointUSR	Universal – Spherical – Revolute
JointUSP	Universal – Spherical – Prismatic
JointUPS	Universal – Prismatic – Spherical
Planar Loops:	
JointRRR	Revolute – Revolute – Revolute
JointRRP	Revolute – Revolute – Prismatic

Table 2. MultiBody.Joints.Assemblies aggregations

On first view this seems to be quite restrictive. However, mechanical devices are usually built up with rods connected by spherical joints on each end, and additionally with revolute and prismatic joints. Therefore, the combinations of Table 2 occur frequently. The universal joint is usually not present in actual devices but is used (a) if two JointXXX components can be connected such that a revolute and a universal joint together form a spherical joint, see Figure 12 and (b) if the orientation of the connecting rod between two spherical joints is needed, e.g., since a body shall be attached. In this case one of the spherical joints might be replaced by a universal joint. This approximation is fine as long as the mass and inertia of the rod is not significant.

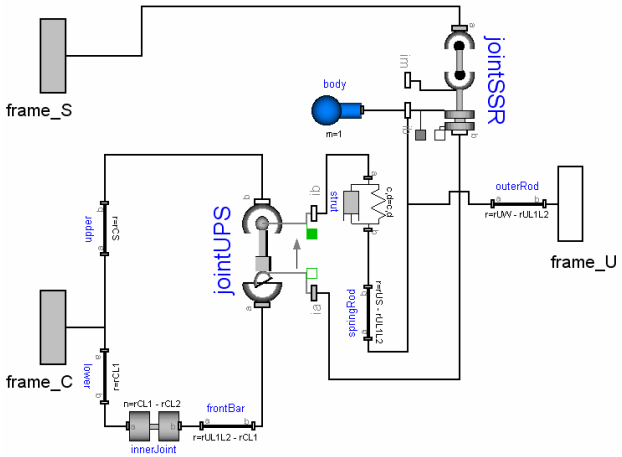


Figure 12. MacPherson with analytic loop handling

Let us discuss item (a) in more detail: The MacPherson suspension in Figure 12 is from the Modelica VehicleDynamics library [2]. It has three frame connectors. The lower left one (frame_C) is fixed in the vehicle chassis. The upper left one (frame_S) is driven by the steering mechanism, i.e., the movement of both frames are given. The frame connector on the right (frame_U) drives the wheel. The three frames are connected by a mechanism consisting essentially of two rods with spherical joints on both ends. These are built up by a jointUPS and a jointSSR assembly, see Figure 12. As can be seen, the universal joint from the jointUPS assembly is connected to the revolute joint of the jointSSR assembly. Therefore, we have 3 revolute joints connected together at one point and if the axes of rotations are chosen appropriately, this describes a spherical joint. In other words, the two connected assemblies define the desired two rods with spherical joints on each ends.

The movement of the chassis, frame_C, is computed somewhere else. When the generalized coordinates of revolute joint "innerJoint" (lower left part in figure) are used as states, then frame_a and frame_b of the jointUPS joint can be calculated. After the non-linear loop with jointUPS is solved, all frames on this assembly are known, especially, the one connected to frame_b of the jointSSR assembly. Since frame_b of jointSSR is connected to frame_S which is computed from the steering mechanism, again the two required frame movements of the jointSSR assembly are calculated, meaning in turn that also all other frames on the jointSSR assembly can be computed, especially, the one connected to frame_U that drives the wheel. From this analysis it is clear that a tool is able to solve these coupled loops analytically.

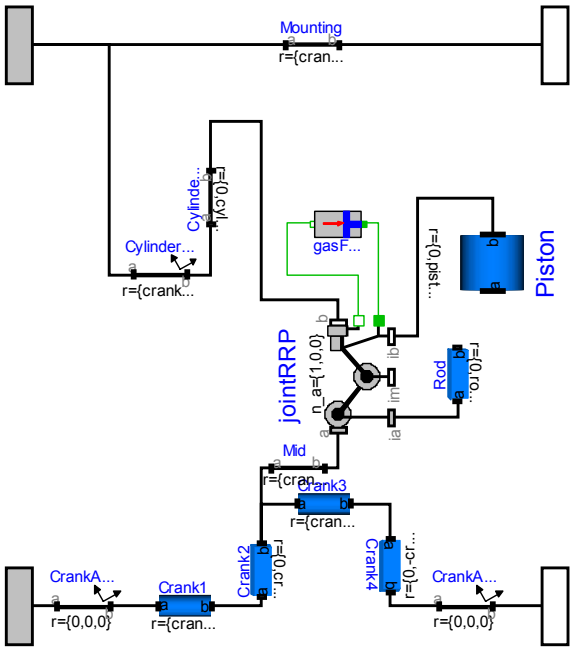


Figure 13. Cylinder of engine with analytic loop handling

Another example is the engine model from Figure 10. It is sufficient to rewrite the basic cylinder model by replacing the joints with a JointRRP object that has two revolute and one prismatic joint, see Figure 13. Since 6 cylinders are connected together, 6 coupled loops with 6 JointRRP objects are present. This model is available as MultiBody.Examples.Loops.EngineV6_analytic.

From Figure 10 it can be seen that the revolute joint of the crank shaft (left part of upper subfigure in Figure 10) might be selected as degree of freedom. Then the 4 connector frames of all cylinders can be computed. As a result the computations of the cylinders are decoupled from each other. Within one cylinder, see Figure 13, the position of frame_a and frame_b of the jointRRP assembly can be computed and therefore the generalized coordinates of the two revolute and the prismatic joint in the jointRRP object can be determined. From this analysis it is not surprising that Dymola is able to transform the DAE equations

into a sequential evaluation without any linear or non-linear loop. Compare this nice result with the model from Figure 10 that leads to a DAE with 6 algebraic loops and 5 non-linear equations per loop. Additionally, a linear system of equations of order 43 is present. The simulation time is about 5 times faster with the analytic loop handling.

7.3 Analytic Loop Handling: How it works

The basic technique for the analytic loop handling is explained at hand of the JointSSR (Spherical – Spherical – Revolute) assembly shown in Figure 14. It consists of two spherical joints connected by a rigid massless rod and a revolute joint connected by an additional massless rod to the spherical joint in the middle (optionally, a point mass can be present on the rod connecting the two spherical joints). At the upper part of Figure 14 the Modelica icon of the JointSSR object and in the lower part an animation view with some important position vectors is shown. The following derivation is a special case of the "characteristic pair of joints" method and is based on [24].

It is assumed that the positions and orientations of frame_a and of frame_b of the JointSSR object are calculated as a function of states. This means that the position vectors ${}^0\mathbf{r}_{s1}$, ${}^0\mathbf{r}_{rev}$ from the origin of the world frame to the origins of frame_a and of frame_b of the JointSSR object are known. Using the orientation objects of frame_a and of frame_b it is easy to compute position vector ${}^a\mathbf{r}_1$ that is directed from the origin of the revolute joint (= frame_b) to the origin of the first spherical joint (= frame_a) and is resolved in **frame_a** of the **revolute** joint (this frame is identical to frame_b of the JointSSR object). Position vector ${}^b\mathbf{r}_2$ is a parameter of the JointSSR object and is directed from the origin of the revolute joint to the origin of the second spherical joint and is resolved in **frame_b** of the

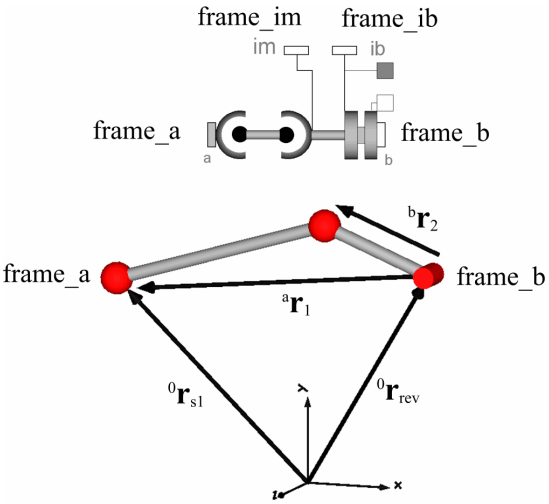


Figure 14. Analytic loop handling for JointSSR

revolute joint. The two spherical joints are connected together by a rod with a fixed length L which is a parameter of the JointSSR object. The length L can be also calculated by computing the vector from spherical joint 1 to spherical joint 2 with vectors ${}^a\mathbf{r}_1$, ${}^b\mathbf{r}_2$ and taking its length. The square of this length results in:

$$L^2 = \left({}^b\mathbf{r}_2 - \mathbf{T}(\varphi)\cdot {}^a\mathbf{r}_1\right)^T \cdot \left({}^b\mathbf{r}_2 - \mathbf{T}(\varphi)\cdot {}^a\mathbf{r}_1\right)$$

Since ${}^a\mathbf{r}_1$ and ${}^b\mathbf{r}_2$ are resolved in different frames, ${}^a\mathbf{r}_1$ has first to be transformed from frame_a to frame_b of the revolute joint using the relative transformation matrix **T** between these two frames. This matrix is solely a function of the unknown rotation angle φ. In the equation above all variables are known (or are calculated somewhere else) with exception of φ. Therefore, we have one non-linear algebraic equation for one unknown, φ, and the goal is to solve this equation analytically. Multiplying out all terms and taking into account that $\mathbf{T}(\varphi)^T \cdot \mathbf{T}(\varphi)$ is the unit matrix, since transformation matrices are orthogonal, we arrive at

$$0 = {}^b\mathbf{r}_2{}^T \cdot {}^b\mathbf{r}_2 + {}^a\mathbf{r}_1{}^T \cdot {}^a\mathbf{r}_1 - L^2 - 2\cdot {}^b\mathbf{r}_2 \cdot \mathbf{T}(\varphi)\cdot {}^a\mathbf{r}_1$$

The relative transformation matrix T can be mathematically described as, see, e.g., [18]:

$$\mathbf{T} = \mathbf{n}\cdot\mathbf{n}^T + \left(\mathbf{E} - \mathbf{n}\cdot\mathbf{n}^T\right)\cdot\cos(\varphi) - \begin{bmatrix} 0 & -n_3 & n_2 \\ n_3 & 0 & -n_1 \\ -n_2 & n_1 & 0 \end{bmatrix}\cdot\sin(\varphi)$$

where **E** is the identity matrix and **n** is a unit vector in direction of the axis of rotation. **n** has the same coordinates with respect to frame_a and to frame_b. Inserting this formula in the constraint equation and rearranging terms results in

$$0 = A\cdot\cos(\varphi) + B\cdot\sin(\varphi) + C$$

with

$$A = -2\cdot\left({}^b\mathbf{r}_2{}^T \cdot {}^a\mathbf{r}_1 - (\mathbf{n}^T \cdot {}^b\mathbf{r}_2)\cdot(\mathbf{n}^T \cdot {}^a\mathbf{r}_1)\right)$$
$$B = 2\cdot {}^b\mathbf{r}_2 \cdot \mathbf{n} \times {}^a\mathbf{r}_1$$
$$C = {}^a\mathbf{r}_1{}^T \cdot {}^a\mathbf{r}_1 + {}^b\mathbf{r}_2{}^T \cdot {}^b\mathbf{r}_2 - L^2 - 2\cdot(\mathbf{n}^T \cdot {}^b\mathbf{r}_2)\cdot(\mathbf{n}^T \cdot {}^a\mathbf{r}_1)$$

Note, that the coefficients A, B, C are computed from known quantities. This non-linear equation has two solutions in the range: $-180^\circ \le \varphi \le 180^\circ$:

$$\varphi_{1/2} = \text{atan2}(-B\cdot C - k\cdot A\cdot\sqrt{A^2+B^2-C^2},\ -A\cdot C + k\cdot B\cdot\sqrt{A^2+B^2-C^2})$$
$$k = \pm 1$$

In the JointSSR object a guess value φ_{guess} is defined as a parameter. From the two solutions the one is selected during initialization that is closest to φ_{guess}. This determines the value of the constant k at initial

time. During simulation, the value of k is kept constant. The term under the square root may become negative so that no (real) solution exists anymore. This is the case when the length of ${}^a\mathbf{r}_1$ becomes larger as the sum of the lengths of the two rods of the JointSSR object, see Figure 14. This case is checked with an assert statement and if it is no longer valid, the simulation is stopped and an appropriate error message is given in which this situation is explained.

Note, for the JointSSP (Spherical – Spherical – Prismatic) assembly, a similar derivation leads to a simple quadratic equation that has two solutions.

Once angle φ is determined with the above formulas, all other desired positional quantities of the JointSSR object can be computed in a straightforward way. By differentiating the equations twice also the first and second derivative of the angle can be determined. The differentiation is automatically performed by the tool. Finally, the (unchanged) equations of the revolute joint and of the other components in the JointSSR object are used to build up the DAE system. It turns out that this approach results in a linear system of equations where at least the second derivative of φ and the as yet unknown force in the rod connecting the two spherical joints is contained. The dimension of this loop is reduced or the loop is even completely eliminated in some cases by the following approach:

In the revolute joint there is an equation that states that the projection of the cut-torque $\boldsymbol{\tau}$ of frame_b on the axis of rotation $\mathbf{n}$ of the revolute joint is zero, see Table 1: $\mathbf{n}^T \cdot \boldsymbol{\tau} = 0$. By a torque balance around the origin of frame_b of the JointSSR object, the cut-torque $\boldsymbol{\tau}$ at frame_b can be expressed as a function of the cut-forces and cut-torques at the other frame connectors of the JointSSR object and the unknown force in the rod connecting the two spherical joints (assuming this rod is cut for the torque balance). Inserting these relationships in the equation $\mathbf{n}^T \cdot \boldsymbol{\tau} = 0$, results in one linear equation in the unknown rod force from which the rod force can be computed analytically as function of the cut-forces and -torques of frame_im and frame_ib (see Figure 14).

8 Force Elements

Force elements exert forces and torques between two frames. The icon of the most general one available in the MultiBody library (model MultiBody.Forces.ForceAndTorque) is displayed in Figure 15

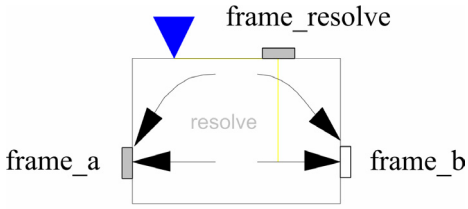


Figure 15. General force element

The 6 elements in the input signal vector are interpreted as the 3 coordinates of a force and the 3 coordinates of a torque acting at the component to which frame_b of the ForceAndTorque component is connected. The force and torque defined with the 6 elements of the input are assumed to be resolved in the frame to which connector frame_resolve is connected. If frame_resolve is not connected, it is assumed that the force and torque are resolved in frame_b. Additionally the force and torque act with "opposite sign" on frame_a (or more precisely, the force and torque on frame_a is computed by a force/torque balance between the two frames). Via sensor elements, any type of kinematical or force/torque information can be inquired. This can be used to compute the force and torque of a force element. Note, since the MultiBody library is purely equation based, also accelerations (e.g., from an acceleration sensor), and cut-forces and cut-torques (e.g., the normal force of a Coulomb friction element) can be utilized to compute the force and torque of a ForceAndTorque element.

8.1 Line Force Elements With Mass

More often, line force elements are needed, that exert a force on the line between the origins of two frames. The two basic line force elements of the MultiBody library are displayed in Figure 16.

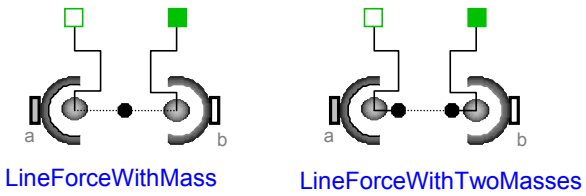


Figure 16. Line force elements that may have mass

The force acting between the origins of frame_a and of frame_b (on the line between these two points) is defined via the two 1-dimensional flange connectors at the top part of the icons (the two green filled and non filled squares). Here, models of the Modelica.Mechanics.Translational library can be connected. An example is given in Figure 17 where a 1-dimensional translational spring is connected between the 1D flange connectors.

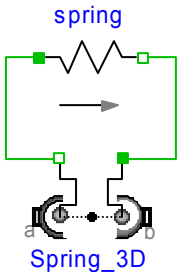


Figure 17. Line force with 1D spring

This approach has several advantages: (1) the distance between frame_a and frame_b is reported in the 1D flange connectors and can therefore be directly utilized in the force law without having to use a sensor object to inquire kinematical information. (2) For more complicated force laws, e.g., a hydraulic cylinder that is driven by a hydraulic circuit, it is advisable to first test the whole force law separately with 1-dim. elements and additional libraries such as a hydraulic or an electrical library. When this works, the force object is just connected to the 3-dimensional line force element of Figure 16.

In multi-body programs the assumption is usually made that force elements are massless. In reality this is not always justified since, e.g., a spring or a hydraulic cylinder has mass that might be significant in some applications. For example, the counter balance systems of large robots have usually a mass that is 5 – 10 % of the mass of the moving parts. By just examining the reaction force to the ground, it is clear that it is not possible to neglect this mass.

For these practical requirements, the line force elements provided in the MultiBody library have optionally one or two point masses on the line from the origin of frame_a to the origin of frame_b. The usage of a point mass is usually sufficient and has the advantage that not much data is required from the user (additionaly data: mass of the point mass and its location) and that it can be handled very efficiently with only a small overhead in the computation compared to a force element without a point mass.

In element "LineForceWithMass" the point mass is located at a fixed relative distance between the two frame origins. Default is "in the middle". This is useful, e.g., for a spring. In element "LineForceWithTwoMasses" two point masses are present that are located at an absolute distance with respect to frame_a and to frame_b, respectively. For example, point mass 1 might be located 0.5 m away from the origin of frame_a on the line to frame_b. This is useful, e.g., for a hydraulic cylinder.

8.2 Direct Coupling of Force Elements

Nearly all multi-body programs have the restriction that two force elements cannot be directly connected together. When this is desired, the user has to introduce a body with a small mass between the force elements leading usually leading to an unnecessary stiff model. Since the Modelica MultiBody library is purely equation based, there are **no** such **restrictions** and it is possible to connect 3-dimensional force elements directly together, such as a series connection of the "ForceAndTorque" element from Figure 15. This usually leads to non-linear systems of equations.

It is also possible to connect line force elements directly together as demonstrated in Figure 18. This example is available from MultiBody. Examples.Elementary.ThreeSprings. In the upper part of this figure the Modelica schematic is shown consisting of three springs that are connected together at one point. The other ends of the springs are connected to the environment and to a body moving freely in space. In the lower part of the figure the animation of this system is shown.

Without special action difficulties would occur, since in every "line force element" there is an equation stating that the cut-torques at both ends of the line force element (= frame_a.t and frame_b.t) are zero. If three line force elements are connected together as in Figure 18, there is additionally the zero sum equation of flow variables stating that the sum of the cut-torques of the connected springs is zero. This is one equation too much, since all torques in this equation are already set to zero in the spring elements. On the other hand, the orientation

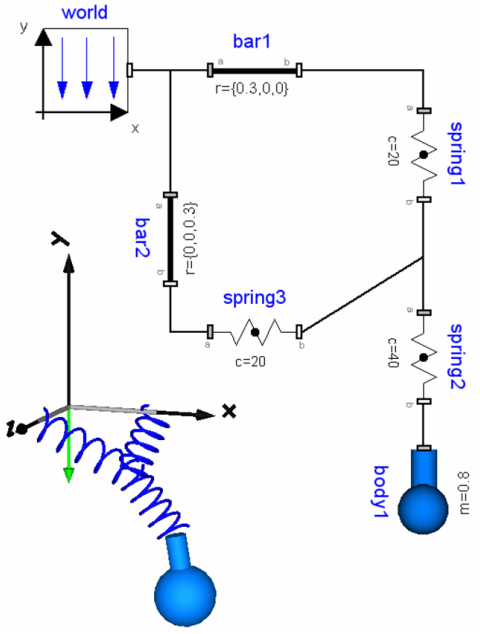


Figure 18. Springs connected directly together

object in the frame connector is not defined because a line force element does not compute it, which means that the orientation object in the connection point of the three springs is not defined. Therefore, the resulting DAE of Figure 18 would be structurally singular and has both overdetermined and underdetermined sets of equations.

It is possible to automatically fix this problem. One line force element that is directly connected at one point to other line force elements has to define that the orientation object in the frame connector defines a null rotation and on the other hand has to remove the equation that states that the cut-torque is zero. This is defined in the following way with Modelica:

```
model LineForceWithMass
  ...
equation
  potentialRoot(frame_a.R, 100);
  potentialRoot(frame_b.R, 100);
    ...
  if isRoot(frame_a.R) then
    frame_a.R=Frames.nullRotation();
  else
    frame_a.t=zeros(3);
  end if;

  if isRoot(frame_b.R) then
    frame_b.R=Frames.nullRotation();
  else
    frame_b.t=zeros(3);
  end if;
end LineForceWithMass;
```

A frame connector of a line force element is a potential root of a virtual connection graph (see section 5). The priority of this potential root is set to 100, as opposed to potential roots of bodies that have a priority of 0. This means that, whenever possible, a body is selected as a root. If this is not possible, a frame connector of a line force element is selected as root (meaning that only line force elements are connected together). Since exactly one frame of a connection point is selected as root, the corresponding line force element can provide the necessary equations as shown in the Modelica code fragment above.

9 Animation

The MultiBody library provides sub library "Visualizers" that contains models to visualize geometric parts, see Figure 19. All visualizer objects have a frame connector to connect the object to any other frame connector in a model. The properties of the visualizer object are described with respect to

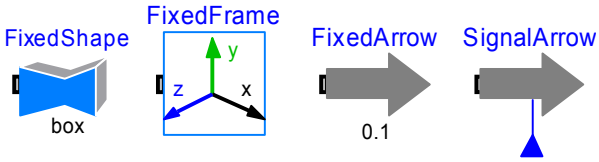


Figure 19. Visualizer objects

the frame to which the object is connected. All visualizer objects have a Boolean parameter "animation" with default "animation = **true**". If "animation = **false**" is set, the animation of this object is switched off and all equations of this object are removed from the generated code. Additionally, in the World object there is a global flag "enableAnimation". If this flag is set to **false**, the animation of all objects is removed (this is especially important for real-time simulation).

Visualizer components "FixedArrow" and "SignalArrow" display an arrow at a frame. "FixedFrame" displays a coordinate system with axes labels, see Figure 2. "FixedShape" displays either one of the geometric shapes from Figure 20 or it displays a 3D shape from a DXF or STL file. All models in the MultiBody library, such as a joint, a body, a force element or a sensor, have built-in

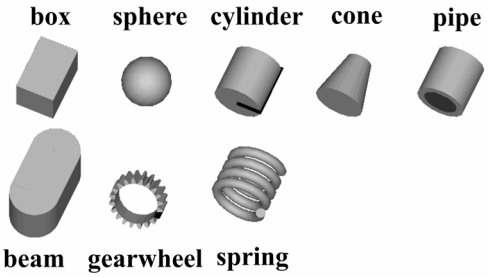


Figure 20. Geometric shapes visualized by "FixedShape"

animation properties that are based on the visualizer objects. Appropriate default values are available such that, without any additional action from the user, always an animation of the defined elements is displayed that can be further refined to get a nicer drawing. The main advantage of this approach is that a defined multi-body model can be quickly checked visually. This feature is implemented in the following way (which might be useful also for other applications):

```
  ...
protected
  outer MultiBody.World world;
  parameter Integer ndim =
     if world.enableAnimation and
        animation then 1 else 0;
  Visualizers.Advanced.Shape
    shape[ndim](
      each shapeType=shapeType,
      each color=color,
        ...
    )
```

Via an **outer** declaration the world object is accessed. The Visualizers.Advanced.Shape model is a shape without a frame connector that may have a fixed or dynamic shape using all the elements from Figure 20. An instance of this model is declared as an array with dimension "ndim". This dimension is either zero or one, depending whether animation is enabled or not. A variable of array shape, such as "color" has the same value for all array indices and therefore it is defined as "**each** color = ...". Modelica supports zero-sized component arrays and therefore the above definition just states that no object "shape" is present, when the dimension of the array is zero, i.e., when animation is disabled.

10 Summary and Outlook

It is expected that the new and free Modelica MultiBody library will be very helpful for the modeling of simple and complex 3-dimensional mechanical systems, especially for non-experts in the multi-body field, since the library is easy to use (in contrast to the previous ModelicaAdditions.-MultiBody library) and it is very powerful. Especially, several features are present to get real-time simulation performance. The MultiBody library is designed to work closely together with other Modelica libraries, in particular with the libraries:

- **Modelica.Mechanics.Translational** for 1-dim. translational line force elements.
- **Modelica.Mechanics.Rotational** for 1-dim. rotational elements to define drive trains driving, e.g., revolute joints. This library contains sophisticated elements such as bearing friction, torque dependent friction in gears, clutches, brakes.
- **PowerTrain** [20] which is an extension of the Rotational library dedicated to vehicle power trains and complicated planetary gears with losses. The Rotational, MultiBody and PowerTrain library are extended in the next version such that all 3D effects of 1-dim. drive trains attached to MultiBody models are taken into account in an efficient and user convenient way [22]. In particular support torques of drive train elements are calculated.
- **HyLib** [3][4] for the modeling of hydraulic systems. Hydraulic cylinders of HyLib can be directly attached to the 1D flanges of MultiBody line force elements.
- **VehicleDynamics** [2] for the modeling of the dynamics of vehicles providing a large set of components and also complete vehicles in different levels of model details. The free VehicleDynamics library is currently based on the ModelicalAdditions.MultiBody library. It will soon be converted to the new MultiBody library.
- Import filters from **AutoDesk Mechanical desktop** [5] and from **SolidWorks** [9] to Modelica are available for the Modelica-Additions.MultiBody library. It is planned to convert them soon to the new MultiBody library, see http://www.mathcore.com.

We plan to further continue the development of the MultiBody library in different directions. Since the field of possible improvements is large, e.g., modeling of elastic bodies, modeling of contact, interfaces to finite element and CAD programs, aero-elastic couplings of wings, etc., we are interested in cooperations. Please, feel free to contact the authors if you plan to use the MultiBody library as a basis for enhancements, especially if you provide your work also in the public domain.

Acknowledgements

Developments in the EU Project RealSim "Real-time simulation for design of multi-physics systems", in the years 2000-2002 under contract IST-1999-11979, have influenced the design of this library, e.g., the close integration of animation in all objects.

The idea to provide a general line force element with 1D-translational connectors has been taken from the VehicleDynamics library [2]. In the ModelicaAdditions.MultiBody library a user had to inherit from a "LineForce" superclass and always implement the force law with Modelica equations. The usage of the 1D flange connectors is more user friendly.

Appendix: Algorithm to Transform Overdetermined DAEs

In this appendix the algorithm is sketched to transform an overdetermined DAE to a standard DAE where the number of equations and unknowns are identical.

In Table 3, the set of Modelica built-in operators introduced in section 5 are formally defined. These operators are utilized to describe the relationships of the overdetermined types or records in the connector instances of a model: **Every** instance of an *overdetermined type* or *record* in an *overdetermined connector* is a **node** in a *virtual connection graph* that is used to determine when the standard equation "$\mathbf{R}_1 = \mathbf{R}_2$" or when the equation "$\mathbf{0}$ = equalityConstraint($\mathbf{R}_1$, $\mathbf{R}_2$) "has to be used for the generation of connect(...) equations. The **branches** of the virtual connection graph are implicitly defined by "connect(...)" and explicitly by "Connections.branch(...)" statements, see Table 1.

For example, a revolute joint has two connectors frame_a and frame_b. In this model, there is an algebraic relationship between the orientation objects of these two frames: frame_b.R = f(frame_a.R, φ), where φ is the relative rotation angle. A definition of the form

```
Connections.branch
              (frame_a.R, frame_b.R);
```

has to be present in this joint model in order to state that the overdetermined variables frame_a.R and frame_b. R are algebraically coupled.

Additionally, corresponding nodes of the virtual connection graph have to be defined as **roots** or as **potential roots** with functions "root(...)" and "potentialRoot(...)", respectively, see Table 3. For example, connector frame_a in the World model has to be defined as "Connections.root(frame_a.R)" because all elements of frame_a.R are explicitly given in the World model (frame_a.R = nullRotation()). A "potential root" is, for example, a body object, since if the body is freely flying in space, body coordinates may be used as states from which the orientation object can be computed. It is a "potential root", because body states should for efficiency reasons only be selected as states, if no other possibility exists.

Note, that branch(...), root(...), potentialRoot(...) do not generate equations. They only define nodes and branches in the virtual connection graph for analysis purposes to be discussed now.

Before connect(…) equations are generated, the virtual connection graph is transformed into a **set** of **spanning trees** by removing breakable branches (connections) from the graph. This is performed in

connect(A,B);	Defines **breakable branches** from the overdetermined type or record instances in connector instance A to the corresponding overdetermined type or record instances in connector instance B for a virtual connection graph.
branch(A.R,B.R);	Defines a **non-breakable branch** from the overdetermined type or record instance R in connector instance A to the corresponding overdetermined type or record instance R in connector instance B for a virtual connection graph. This function can be used at all places where a connect(..) statement is allowed. *[This definition shall be used, if in a model with connectors A and B the overdetermined records A.R and B.R are algebraically coupled in the mode]*.
root(A.R);	The overdetermined type or record instance R in connector instance A is a (definite) **root node** in a virtual connection graph. *[This definition shall be used if in a model with connector A the overdetermined record A.R is (consistently) assigned, e.g., from a parameter expressions]*
potentialRoot(A.R); **potentialRoot** (A.R, priority = prior);	The overdetermined type or record instance R in connector instance A is a **potential root node** in a virtual connection graph with priority "prior" (prior ≥ 0). If no second argument is provided, the priority is zero. "prior" shall be a parameter expression of type Integer. In a virtual connection subgraph without a Connections.root definition, one of the potential roots with the lowest priority number is selected as root *[This definition is, e.g., used in a body, see Parts.Bodys in Table 2]*.
b = **isRoot**(A.R);	Returns true, if the overdetermined type or record instance R in connector instance A is selected as a root in the virtual connection graph.

Table 3. Operators "Connections.XXX" (e.g. Connections.branch) to define the set of overdetermined equations

the following way:

1. Every root node defined via the "Connections.root(…)" statement is a definite root of one spanning tree.
2. The virtual connection graph may consist of sets of subgraphs that are not connected together. Every subgraph in this set shall have at least one root node or one potential root node. If a graph of this set does not contain any root node, then **one potential root** node in this subgraph with the lowest priority number is selected to be the root of the subgraph. The selection can be inquired in a class with function Connections.isRoot(…), see Table 1.
3. If there are n selected roots in a subgraph, then breakable branches have to be removed such that the result shall be a set of n spanning trees with the selected root nodes as roots.

After this analysis, the connect(…) equations for overdetermined variables are generated in the following way:

1. For every breakable branch in one of the spanning trees, i.e., connect(A,B) statements, the usual "equality" connect equations are generated, "A.R = B.R".
2. For every breakable branch **not** in any of the spanning trees, the equations "**0** = R.equalityConstraint(A.R,B.R)" are generated instead of "A.R = B.R".

An example for a virtual connection graph is given in Figure 21. This example contains two independent subgraphs that are analyzed separately. The left subgraph has two (definite) roots. Four breakable branches, i.e., connect(...) statements have to be removed to arrive at two spanning trees. For every removed connect(...) statement the equalityConstraint(...) function is used to generate the connection equation. In the right subgraph of Figure 21 no definite root is present. Therefore, the potential root with the lowest priority has to be selected as root. If there are several roots with the same lowest priority, one of them is selected arbitrarily. Starting from the selected root, only one branch has to be removed to also arrive at a spanning tree in this subgraph.

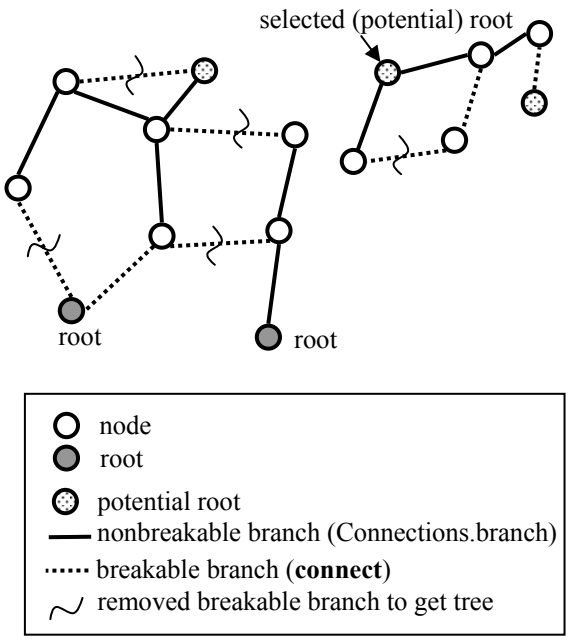


Figure 21. Example for virtual connection graph

Bibliography

[1] ADAMS: **MSC ADAMS** at http://www.mscsoftware.com/products/quick_prod.cfm

[2] Andreasson: **VehicleDynamics library**. Proceedings of the 3rd Int. Modelica Conference, Modelica'2003. http://www.Modelica.org

[3] Beater P. (2003): **HyLib version 2.1.** http://www.HyLib.com

[4] Beater P., and Otter M. (2003): **Multi-Domain Simulation: Mechanics and Hydraulics of an Excavator**. Proceedings of the 3rd Int. Modelica Conference, Modelica'2003. http://www.Modelica.org

[5] Bunus B., Engelson V., and Fritzson P. (2000): **Mechanical Models Translation, Simulation and Visualization in Modelica**. Proc. of Modelica 2000 workshop, Lund, 2000. http://www.modelica.org/-workshop2000/proceedings/Bunus.pdf

[6] DADS: **LMS DADS** at http://www.lmsintl.com/

[7] Dynasim (2003)*:* **Dymola Users Guide, Version 5.1**, http://www.dynasim.se.

[8] Elmqvist, H. (1978): **A Structured Model Language for Large Continous Systems**. PhD-Thesis, Lund Institute of Technology, Lund, Sweden.

[9] Engelson V. (2000): **Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing**. Linköping Studies in Science and Technology. Dissertation No 627. Department of Computer and Information Science, Linköping University (chapter 5).

[10] Hiller M., and Woernle C. (1987): **A Systematic Approach for Solving the Inverse Kinematic Problem of Robot Manipulators.** Proceedings 7th World Congress Th. Mach. Mech., Sevilla.

[11] Kecskemethy A. (1993): **Objektorientierte Modellierung der Dynamik von Mehrkörpersystemen mit Hilfe von Übertragungselementen**. Dissertation, VDI Fortschritt-Berichte, Reihe 20, Nr. 88.

[12] Kecskemethy A. (1993): **Mobile - An Object-Oriented Tool-Set for the Efficient Modeling of Mechatronic Systems**. Proc. of the Second Conference on Mechatronics and Robotics, pp. 447-462, Duisburg/Moers, Sept. 27.-29. MOBILE homepage: http://www.mechanik.tu-graz.ac.at/mobile

[13] Mattsson S.E., and Söderlind G. (1993): **Index reduction in differential-algebraic equations using dummy derivatives**. SIAM Journal of Scientific and Statistical Computing, Vol. 14, pp. 677-692.

[14] Mattsson S.E., Olsson H., and Elmqvist H. (2000): **Dynamic Selection of States in Dymola**. Modelica Workshop 2000 Proceedings, pp. 61-67, http://www.modelica.org/workshop2000/-proceedings/Mattsson.pdf

[15] Möller M. (1992): **Ein Verfahren zur automatischen Analyse der Kinematik mehrschleifiger räumlicher Mechanismen**. Dissertation, Institut A für Mechanik der Universität Stuttgart.

[16] Nikravesh, P.E (1988): **Computer-Aided Analysis of Mechanical Systems**. Prentice Hall.

[17] Otter M., Elmqvist H., and Cellier F. (1996): **Modeling of MultiBody Systems with the Object-Oriented Modeling Language Dymola**. Nonlinear Dynamics, Vol. 9, pp. 91-112.

[18] Roberson R.E., and Schwertassek R (1988): **Dynamics of Multibody Systems**. Springer Verlag.

[19] Pantelides C. (1988): **The Consistent Initialization of Differential-Algebraic Systems**. SIAM Journal of Scientific and Statistical Computing, pp. 213-231.

[20] PowerTrain (2002): **PowerTrain Library 1.0 – Tutorial**. DLR, www.dynasim.se/www/Power-TrainTutorial.pdf

[21] SIMPACK: http://www.simpack.de/

[22] Schweiger C., and Otter M. (2003): **Modelling 3D Mechanical Effects of 1D Powertrains**. Proceedings of the 3rd Int. Modelica Conference, Modelica'2003. http://www.Modelica.org

[23] TESIS ve-DYNA: http://www.tesis.de/en

[24] Woernle C. (1988): **Ein systematisches Verfahren zur Aufstellung der geometrischen Schließbedingungen in kinematischen Schleifen mit Anwendung bei der Rückwärtstransformation für Industrieroboter**. Fortschritt-Berichte VDI, Reihe 18, Nr. 59, Düsseldorf: VDI-Verlag , ISBN 3-18-145918-6.