Abstract

The objective is to model and simulate larger and more complex 3-dimensional systems as it is possible with a pure equation-based modeling system such as Modelica. The approach shall combine component-based 3D modeling, as used in modern game engines, with equation-based modeling. The proposed methodology has been evaluated and tested in the experimental modeling environment Modia3D that is implemented with the Julia programming language.

Keywords: Modelica, Modia, Modia3D, Julia, DAE, equation-based modeling, component-based modeling, multibody, collision handling

1 Introduction

The objective is to model and simulate larger and more complex 3-dimensional systems as it is practically possible with a pure equation-based modeling system such as the current Modelica language version 3.4. Issues are:

- The data structures of an equation-based modeling system are limited as compared to a programming language such as C++ or Julia. For example, it is virtually impossible to define 3D meshes and collision handling algorithms in Modelica.

- Specialized operations in the 3D world are hard to use, such as to remove redundant constraints of a planar loop automatically, solve kinematic loops analytically, or use an O(n) multibody algorithm. In Modelica, a user has to explicitly model such situations with specialized elements or use a pre-processor that generates Modelica code, see e.g. (Elmqvist et al., 2009).

- Since Modelica compilers expand the models for the symbolic engine, the same equation is analyzed many times. Thus, the number of expanded equations grows at least linearly with the number of model instances and therefore the compilation time grows at least linearly with the model size.

The goal of this article is to propose an approach how to combine 3D modeling techniques with equation-based modeling à la Modelica. This procedure has been evaluated and tested with the open source prototype Modia3D\(^1\) (version 0.2.0-beta.1). It is implemented with the Julia programming language\(^2\) (Bezanson et al., 2017) taking advantage of Julias powerful language features such as multiple dispatch and set-based types\(^3\). Modia\(^4\) (Elmqvist et al., 2016, 2017) shall be used for the equation-based modeling. The intention is to utilize the results of this prototyping in the design of the next Modelica language generation.

Modia3D has no graphical user interface. It would be useful to have 3D schematics as proposed by (Elmqvist et al., 2015a). The textual representation of Modia3D is designed for 3D schematics and not for Modelica 2D schematics. Modia3D provides a generic interface to visualize simulation results with different 3D renderers. Currently, the free community edition as well as the professional edition of the DLR Visualization library\(^5\) (Bellmann, 2009; Hellerer et al., 2014) are supported.

2 Component-Based 3D Modeling

Modern game engines, such as Unity or Unreal Engine, have a component-based design, so the architecture is based on composition and aggregation. Basically, in this context a coordinate system is located in the 3D world that has a container of optional components (such an object is called GameObject\(^6\) in Unity, Actor\(^7\) in Unreal Engine, Object3D\(^8\) in Three.js). Each of these components has properties such as geometry, visualization, dynamics, collision properties, light, camera, sound, etc., see e.g. (Nystrom, 2014)\(^9\). This design has the advantage that many optional components and variants can be defined and treated in a very flexible and unified way. In this paper, this very special variant of the generic component-based design pattern is called component-based 3D modeling.

Modelica 3.4 supports component-based design via replaceable components. Unfortunately, this language construct has limitations and is not sufficient for component-based design as needed below. On the other hand, Julia is particularly designed to support this programming pattern\(^10\) and is thus very well suited for the implementation of Modia3D.

\(^1\)https://github.com/ModiaSim/Modia3D.jl
\(^2\)https://julialang.org
\(^3\)https://docs.julialang.org/en/stable/manual/types/
\(^4\)https://github.com/ModiaSim/Modia.jl
\(^5\)https://visualization.ltx.de/, http://www.systemcontrolinnovationlab.de/the-dlr-visualization-library/
\(^6\)https://docs.unity3d.com/Manual/GameObjects.html
\(^7\)https://docs.unrealengine.com/en-us/Engine/Components
\(^8\)https://threejs.org/docs/index.html#api/core/Object3D
\(^9\)http://gameprogrammingpatterns.com/component.html
\(^10\)http://www.stochasticlifestyle.com/type-dispatch-design-post-object-oriented-programming-julia
2.1 Object3D

In Modia3D, component-based 3D modeling is performed with Object3D objects. An Object3D object consists of a 3D coordinate system that has associated, optional properties collected in the data container (see Figure 1). The code-snippet\(^\text{11}\) of the following constructor call \(^\text{12}\) creates a new Object3D object obj:

\begin{verbatim}
1  obj = Object3D(parent, data, r=[0,0,0],
2  R=[1,0,0;0,1,0;0,0,1], fixed=true)
\end{verbatim}

Hereby, obj is defined relative to a parent object3D, with the position vector \(r\) and the rotation matrix \(R\). It is rigidly connected to its parent if fixed=true, and can move freely if fixed=false. In the latter case, initial position and rotation matrix is defined with \(r, R\).

Argument data is of the abstract type AbstractObject3Ddata. Therefore, all objects can be used which are a subtype of this type. There are further constructor functions for Object3D, therefore the arguments parent and data are also optional (e.g. line 3). An Object3D is said to be a reference Object3D, if no parent Object3D is given. The world-object3D can be defined as, for example

\begin{verbatim}
3  world = Object3D().
\end{verbatim}

In Figure 2, the current abstract and concrete subtypes of AbstractObject3Ddata are shown. Instances of the concrete subtypes can be used for the positional argument data. In Figure 2, the concrete types are printed in light blue, abstract types in black, and types that are currently under implementation are printed in grey color. The most important concrete types are discussed below. Note, a data object consists of a set of optional components, providing in a flexible way variants and different functionality. All these components are positioned and moved with the same concept - the coordinate system to which the components are attached. The conceptual difference to current Modelica is that the Modelica.Mechanics.MultiBody library defines coordinate systems and properties (such as visualization data) with respect to various Part objects. As a result, the equations to define coordinate systems relative to each other are present many times in many different components, whereas in Modia3D these equations are present only once in the Object3D definition (and Object3D objects support much more flexible part definitions).

2.2 Visual Objects

Visualization objects are subtypes of AbstractVisualElement, which is also a subtype of AbstractObject3Ddata. These elements are used for animation purposes only. Basically, their Julia implementation is an interface to the DLR Visualization library (Bellmann, 2009; Hellerer et al., 2014). The concrete types which have a geometry and associated visualization properties are subtypes of AbstractGeometry. Their material is defined with a Material object. For example, the following constructor call generates a new Material object:

\begin{verbatim}
4  vmat = Material(color=[0,0,255],
5  wireframe=false, transparency=0.5,
6  shininess=0.7, reflectslights=true)
\end{verbatim}

Concrete subtypes with a geometry and a material are shown in Figure 3.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Object3D defined relatively to its parent.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{Overview of AbstractObject3Ddata types.}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure3.png}
\caption{Visual elements with Material.}
\end{figure}

The following example defines an Object3D, which is positioned at \([0,0,0.8]\) in the world-object3D (line 3), is displayed with the visualization material vmat (lines 4 - 6), and has a sphere geometry with diameter = 0.9 m.
Additionally to the subtypes of AbstractGeometry, Modia3D supports currently the concrete types shown in Figure 4. These types do not have a Material object. Here, a grid with 0.7 m length and 0.6 m width, is defined.

grid = Object3D(world, Grid(0.7, 0.6))

It is positioned at the origin of the world-object3D.

Figure 4. Visual elements without Material.

Remark 1. All objects which are subtypes of AbstractVisualElement are mutable objects. Therefore, they still can be changed after instantiation, especially during simulation.

2.3 Solid Objects

The type Solid is directly derived from AbstractObject3DData and defines solid physical objects. A solid object can have geometry, mass properties, can be visualized and can be used in collision handling, and all of these properties are optional. Solid objects are immutable to guarantee constant mass properties during simulation. The following constructor call creates a new Solid object.

solid = Solid(geo, massProperties, material, contactMaterial=nothing)

The arguments have the following meaning:

geo defines the geometry of the solid. It is either nothing (= no geometry defined) or it is a subtype of AbstractSolidGeometry.

massProperties defines the mass properties of the solid. It is either nothing (= is massless) or there are various options to define these properties (see lines 19 - 21 below).

material defines the visualization properties of geo. It is either nothing (= geo is not visualized) or it is a Material object (e.g. lines 4 - 6).

contactMaterial defines the contact response characteristics of geo. It is either nothing (= geo is not included in the collision handling) or it is a subtype of AbstractContactMaterial.

Since all these properties are optional, there is a great flexibility to define the desired solid. Below, more details about the arguments of the solid constructor are given.

Argument: geo

Solid geometry objects geo are subtypes of the abstract type AbstractSolidGeometry and are shown in Figure 5. For example, a SolidFileMesh object can be defined with the following constructor call.

mesh = SolidFileMesh("pascal.obj", 0.2)

Assuming that a file in obj-format is available as "pascal.obj" and the mesh shall be scaled by a factor of 0.2.

Currently, Modia3D supports collision handling only for convex objects. If a SolidFileMesh object is concave, collision handling is performed with respect to the convex hull of the mesh. Alternatively, the open source V-HCAD library\(^{14}\) can be used to approximate a concave mesh by a set of convex parts. Then, Modia3D utilizes these convex parts in collision handling and the original concave mesh for non-collision operations.

The following functions\(^{15}\) compute key properties for rigid-body computations or collision handling and they are provided for all solid geometry objects displayed in Figure 5.

- volume(geo) returns the volume of a solid geometry object geo.
- centroid(geo) returns the position of the centroid of geo. If the solid is homogeneous, the centroid’s position is identical to the center of mass.
- inertiaMatrix(geo, mass) returns the inertia matrix of a solid geometry object geo with mass mass.
- boundingBox!(geo, ..) updates the bounding box of geo. This operation is used for collision handling (see below).
- supportPoint(geo, ..) returns the support point of geo. This operation is a key property for collision handling (see below).

In the following example some geometric properties of a SolidSphere object with diameter D are computed with the above mentioned functions.

\(^{14}\text{V-HCAD: https://github.com/kmammou/v-hacd}\)

\(^{15}\text{As usual in Julia, function names with a ! at the end indicate that one or more of the input arguments are changed by the function call.}\)
Currently, only mesh-data from wavefront (*.obj) files are supported. It is planned to generalize the support of meshes as proposed in (Elmqvist et al., 2015b), to directly define them in Modia3D and provide CSG (Constructive Solid Geometry) operations on them.

**Argument: massProperties**

There are several variants to define the optional mass properties: mass, center of mass, and inertia matrix. Examples of the different variants are:

```plaintext
19 mesh1 = SolidFileMesh("Aluminium")
20 massProp = MassProperties(m=2.1, Ixx=0.1, ...)
21 mesh3 = SolidFileMesh(massProp)
```

In the first case a string is given (line 19), such as "Aluminium". This string is a key in a dictionary in which some key data of materials is stored, such as density, Young's modulus, heat capacity, and thermal conductivity. The density of the material is used together with the geometry geo to compute the needed mass properties (see lines 14 - 18). Alternatively, a number (line 20) can be given that is interpreted as the mass of the solid. Again, together with the geometry geo the needed mass properties are calculated. Finally, also an instance of type MassProperties (line 21) can be provided, in which all mass properties are explicitly given.

**Argument: contactMaterial**

The contact material cmat (line 23) defines how a solid behaves in contact cases. At the moment elastic contacts can be handled, with a spring - damper module. Therefore, a spring constant c and a damper constant d can be provided, as shown in the next example.

```plaintext
23 cmat = ElasticResponse(c=1e5, d=100.0),
24 sphere = SolidSphere(0.2), "Aluminium",
        contactMaterial=cmat)
```

In the example above, a sphere basicSphere is defined that has a diameter of 0.2 m and is made of Aluminium. It is visualized with material vmat, and takes place in collision handling using contact material cmat for the response calculation. This definition is used to declare three spheres: sphere1, sphere2, sphere3. These spheres can move freely in space and are initially placed at different positions in the world-object3D (line 3). Note, although three spheres are declared, all the position-independent properties of the spheres, like visualization material, contact material etc. are defined only once by the reference object basicSphere. In Modelica, one could construct something similar by using replaceable record constructors in modifiers. The conceptual difference is that the data and equations of a basicSphere Modelica model would be present three times in the generated code and not once as in the Modia3D code.

If only one sphere shall be defined, the above definition (line 26 - 32) can also be given without auxiliary variables:

```plaintext
18 D = 0.3; density = 7700
19 geo = SolidSphere(D)
20 V = volume(geo)
21 m = density*V
22 IM = inertiaMatrix(geo, m)
```

In the previous section it was shown how to define Object3D objects and how to associate properties to an Object3D in a very flexible manner. In this section the aggregation of Object3D is discussed.

Hierarchical structures are defined with the Modia3D metaprogramming construct of Julia 16. It generates an

```plaintext
```

Depending on the underlying types of the elements of an Object3D object, type-specific methods are called (based on Julias multiple dispatch feature).
abstract syntax tree of Julia code that is automatically compiled and executed at the line where the macro is called.

```julia
45 @assembly AssemblyName(.,.) begin
46   name = constructor(.,.)
47   < other statements >
48 end
```

The `@assembly` macro generates a new Julia type `AssemblyName` (it is a mutable struct) that (a) contains all left-hand side "name" definitions as elements, (b) uses the code of the `@assembly` for the constructor function for its struct, and (c) initializes support for hierarchical names of the elements of this new type. For example,

```julia
@assembly Bar(;Lx=0.1,Ly=Lx/5,Lz=Ly) begin
  obj0 = Object3D(Solid(SolidBeam(Lx,Ly,Lz),
                   "Aluminium",
                   Material(color="Blue")),
  obj1 = Object3D(obj0,r=[-Lx/2,0,0.0]),
  obj2 = Object3D(obj0,r=[ Lx/2,0,0.0]),
end
```

The reference `Object3D` `obj0` (line 50) is defined as a solid with a `SolidBeam` geometry. The two other `Object3Ds` `obj1`, `obj2` (lines 53 - 54) have `obj0` as parent `Object3D` and their positions are defined according to Figure 6. To check this definition, an instance of the new `Bar` type is constructed (line 56) and it is an input argument of the function call `visualizeAssembly!(bar)` that visualizes the assembly with the default renderer.

Since all left-hand side variables of `@assembly Bar` are elements of the new type, these elements can be accessed via the `bar` instance (line 56) as, e.g. `bar.obj1`. Since the code of an `@assembly` definition is used as a constructor function, order matters and thus the statements are executed in the given order.

Assembly objects can also be elements in other assembly objects and therefore hierarchical structures can be built. For demonstration, the following planar four-bar mechanism is defined. It consists of three bars and the ground

```julia
@assembly Fourbar(;Lx=0.1) begin
  world = Object3D(CoordinateSystem(0.6))
  pos1 = Object3D(world,r=[Lx/2,0.0,Lx/2])
  pos2 = Object3D(pos1,r=[Lx,0.0,0.0])
  ground = Object3D(world,Box(..),..)
  bar1 = Bar(Lx=Lx)
  bar2 = Bar(Lx=Lx)
  bar3 = Bar(Lx=Lx)
  rev1 = Revolute(pos1,bar1.obj1,
                   phi_start=pi/2)
  rev2 = Revolute(bar1.obj2,bar2.obj1,
                   phi_start=pi/2)
  rev3 = Revolute(bar3.obj2,bar2.obj2,
                   phi_start=pi/2)
  rev4 = Revolute(pos2,bar3.obj1,
                   phi_start=pi/2)
end

fourbar = Fourbar(Lx=1.0)
visualizeAssembly!(fourbar)
```

A revolute joint is defined, with the constructor

```julia
Revolute(object1,object2).
```

It constrains `object2` (line 78), so that the z-axis of `object2` coincides with the z-axis of `object1` (line 78) and can rotate around it. Via additional keyword arguments, the joint can be configured further. For example, `phi_start = angle` initially rotates `object1` along its z-axis for the given angle to arrive at `object2`. The revolute joints are visualized in Figure 7 with red cylinders.

Note, in Modelica and Modia a user has to treat one of the revolute joints differently. For example defining one of them to be a revolute cut-joint in a planar loop (Modelica model: `RevolutePlanarLoopConstraint`), since otherwise a redundant set of equations would be generated that cannot be handled with current symbolic engines.

Contrary, in Modia3D no special action is needed by the user. Instead, there is the requirement that the configuration defined by the assembly constructor must be consistent. For example, if `phi_start = pi` would be defined for `rev4`, then this start angle would not be consistent to the already defined configuration, and an error would occur. However, it would be possible to define the angle as `phi_start =`
4 Actuator Objects

The main purpose of Modia3D is to model the 3D-part of a system. All other parts of a system model shall be defined with the equation-based modeling language Modia. Modia3D and Modia shall be combined in the following ways:

1. using Modia models in Modia3D (e.g. a Modia actuator model that drives a Modia3D revolute joint),
2. using Modia3D models in Modia,
3. transforming Modia3D models to Modia equations (to be used, e.g. in embedded systems), and
4. defining force elements directly with simple Julia macros, mainly to develop the interface to Modia (but without actually using Modia).

Currently, only item 4 has been implemented by providing the Julia macros @signal and @forceElement. The usage of these macros is sketched below with two examples:

To move the generalized coordinate of a joint kinematically, a @signal macro with one output signal is defined:

```
@signal Sine(;y_off=0.5,w=1.0,A=1.0) begin
  y = RealScalar(causality=Output)
end
```

Here, one output variable y (line 98) is declared as RealScalar and it is computed in Julia function `computeSignal(sine::Sine,sim)` (line 100). All parameters that are defined in the header declaration (line 97) as well as all variables of the SimulationState sim, e.g. sim.time, sim.startTime, can be used for computing the signal (lines 100 - 103). The new type Sine can be used in an assembly component e.g. to drive one joint of the four-bar mechanism (Figure 7).

First, bar1, bar2 (lines 81 - 82) are defined with the Bar assembly (lines 49 - 55) and as well as their connection with revolute joints rev1, rev2 (lines 83 - 86). As a result, the initial positions of bar1, bar2, as well as pos2 (line 61) on the ground are known. In a second step, the distance L3 between the origin of pos2 and the origin of bar2.obj2 is computed (line 87). If bar3 (line 87) is placed between these two objects, it must have Lx=L3. Furthermore, the angle phi30 (line 88) between the x-axis of pos2 and the position vector from the origin of pos2 to the origin of bar2.obj2 is computed and used as start angle for rev4.

Note, the result is similar to a system that is defined by a parameterized CAD system: Whenever Fourbar2 is instantiated with different arguments (e.g. Lx=0.5 or Lx=10.1), consistent initial configurations of the mechanism are constructed always.
The next example shows how a P-PI cascade controller can be defined that drives a rotational flange of a Modia3D assembly:

```plaintext
@forceElement Controller(;k1=10.0,k2=10.0, T2=0.01,freqHz=0.5,A=1.0) begin
  PI_x = RealScalar(...)
  PI_derx = RealScalar(...)
  sine_y = c.sin(2*pi*c.freqHz*sim.time)
  gain_y = c.k1*(c.sine_y.value - c.phi.value)
  PI_u = gain_y - c.w.value
  w = RealScalar(...)
  phi = RealScalar(causality=Input)
  tau = RealScalar(causality=Output)

end

function computeTorque(c::Controller,sim)
  c.sine_y.value = c.A* sin(2*pi*c.freqHz*sim.time)
  gain_y = c.k1*(c.sine_y.value - c.phi.value)
  PI_u = gain_y - c.w.value
  c.PI_derx.value = PI_u/c.T2
  c.tau.value = c.k2*(c.PI_x.value + PI_u)
end
```

The Controller model uses the angle \( \phi \) and angular velocity \( w \) as inputs (lines 119 - 120) to compute the driving torque \( \tau \) (line 121) as output. This is performed with a P-PI cascade controller where a sine is used as a reference angle. Here, all parameters have to be defined in a `@forceElement` model, and can be used for computing the driving torque with function `computeTorque(c,sim)` (line 123). This function (lines 123 - 131) takes `c` as an instance of `Controller` and `sim` as an instance of `SimulationState` as input values.

```plaintext
@assembly MoveFourbar2(;Lx=0.1) begin
  fourbar = Fourbar(Lx=Lx)
  c = Controller()
  flange = AdaptorForceElementToFlange(
    phi=c.phi, w=c.w, tau=c.tau)
  Modia3D.connect(flange, fourbar.rev1)
end

fourbar2 = MoveFourbar2(Lx=1.0)
model = SimulationModel(fourbar2)
result = simulate!(model,stopTime=3.0)
```

In `MoveFourbar2` an instance of `Controller` is created (line 134). For connecting this instance with a revolute joint (line 137), an adaptor between a force element and a flange is needed. This is done with function `AdaptorForceElementToFlange(...)` (lines 135 - 136) that uses keywords \( \phi \), \( w \), \( a \), and \( \tau \) (see line 136) to associate the controller signals with the corresponding flange variables. Constructor function `SimulationModel(...)` generates a `simulation model`. Since no keyword argument is provided, the default `analysis=DynamicAnalysis` is used. Function `simulate!(..)` (line 141) performs a dynamic simulation of the simulation model `model`. At every time instant, all `computeTorque(...)` functions of each assembly component are executed.

## 5 Prototype Implementation

In this section some details about the implementation of the Modia3D prototype are given.

### 5.1 Handler Objects

Independent `handler objects` are responsible for the various computations that have to be carried out. In a first step, the components for the handler objects are identified.

When instantiating an assembly object, the parent-child-relationships between the Object3Ds are updated and stored in them. For example, when instantiating a `Bar` assembly (lines 49 - 55), `obj0` is the parent of `obj1`. However, when connecting `bar2.obj1` to `bar1.obj2` with a revolute joint `rev2` (lines 68 - 69), then the parent-child-relationship is updated, so that `Object3D bar1.obj2` becomes the parent of `bar2.obj1`, and `bar2.obj1` becomes the parent of `bar2.obj0`.

During the update process, kinematic loops are also identified. For example the revolute joint `rev4` (lines 72 - 73) introduces a constraint between two Object3Ds that are connected in a tree-structure having the same root-object3D. Joints which close a loop are just referenced in the corresponding Object3Ds, without changing the parent-child-relationship of the Object3Ds. The first Object3D in the top-most assembly that is not defined with respect to another Object3D, is treated as the `world-object3D`. Due to this approach, a tree of connected Object3Ds is constructed having the `world-object3D` as its root. As an inspiration the open-source Javascript library `Three.js`\(^19\), was used, to design a similar tree. The Modia3D `Object3D data structure` is hereby similar to the `Three.js base class Object3D`.

In a second step, the kinematic loops are analyzed. Currently, the joints that close a kinematic loop are treated as cut-joints. Hereby, the corresponding kinematic loop is analyzed and if the loop is planar, a reduced set of equations are used for the cut-joint. It is planned to significantly improve this phase by analytically solving a large class of loops with the technique described in (Oetter et al., 2003) that is used in the Modelica Standard Library (`MultiBody.Joints.Assemblies`) and is based on the more general `characteristic pair of joints method`.

\(^19\)https://threejs.org/ "Lightweight cross-browser JavaScript library/API used to create and display animated 3D computer graphics on a Web browser"
Collision detection in Modia3D is based on the MPR (Minkowski Portal Refinement) algorithm (Snethen, 2008), which computes the shortest penetration depth of two convex shapes. The MPR-algorithm is much simpler to implement and has less numerical problems than the often used GJK/EPA-standard algorithms (Gilbert et al., 1988; Bergen, 2003), because it only works with triangles and not with tetrahedrons.

DAE (1) generated by Modia3D is solved with a variable-step integrator. Variable-step integrators are sensitive to drastic changes of the DAE, as in the case of collisions. To speed up the simulation and to improve the robustness of the integration, Modia3D uses the distances between convex shapes as zero-crossing functions $z_i(t)$ (1b). In the original version of the MPR-algorithm (Snethen, 2008) only penetration depths are determined. In Modia3D improvements of the MPR-algorithm are utilized that have been proposed in (Kenwright, 2015; Neumayr and Otter, 2017), in particular to compute the distances of shapes that are not in contact, treating special collision situations properly and introducing a new termination condition to speed up the algorithm in some situations.

In Modia3D collision handling of $n$ potentially colliding shapes is performed in the following (mostly standard) way:

1. **Broad Phase**
   The shapes are approximated by bounding volumes where potential collisions can be very cheaply determined resulting in $O(n^2)$ cheap tests. When using special data structures (such as octrees or kd-trees), it is possible to reduce the number of cheap tests to $O(n \log(n))$.

2. **Narrow Phase**
   For the potentially colliding shape pairs as identified in the broad phase, the signed distances are computed with the improved MPR-algorithm (Neumayr and Otter, 2017).

3. **Response Calculation**
   If two shapes are penetrated, a force and/or torque is applied at the contact point, such as a spring - damper
force element, depending on the penetration depth (section 2.3).

The MPR-algorithm computes the contact points \( C_A, C_B \), the Euclidean distance \( \delta \) if shapes are not in contact, and otherwise the penetration depth \( \delta \) (Figure 9). The distance \( \delta \) calculated by the MPR-algorithm is used as zero-crossing function \( z_i \) for the integrator. This means it detects the transition between penetration and non-penetration of a shape pair. A brute force method for the integrator would be to use the distances between any two shapes as zero-crossing functions, resulting in an \( O(n^2) \) number of zero-crossing functions. Since the number of crossing functions would grow quadratically with the number of collision objects, the maximum number of zero-crossing functions is bounded by \( n_{z,\text{max}} \), which defines the maximum number of objects that can be in contact at the same time instant. This number can be adapted by the user. If more shapes get in contact, the simulation is currently halted with an error (alternatively, the simulation could be halted and could be restarted with an enlarged z vector). The zero-crossing functions are computed with the following scheme (for more details, see (Neumayr and Otter, 2017)):

- The function \( \text{selectContactPairs!}(...) \) is called before every integrator step.
  - Execution of broad and narrow phase.
  - Selection and ordering of \( n_z \leq n_{z,\text{max}} \) shape pairs according to their distances.

- The function \( \text{getDistances!}(...) \) is called whenever the integrator requests a new zero-crossing function evaluation.
  - Execution of broad and narrow phase.
  - Storing the distances of the contact pairs in \( z \) that have been selected in the last call of \( \text{selectContactPairs!}(...) \) and checking that none of the remaining distances is negative.

The broad phase in Modia3D uses \( \text{AABBs} (= \text{Axis Aligned Bounding Boxes}) \) (see e.g. (Bergen, 2003)). Each AABB approximates one shape and only if the AABB’s are intersecting, the distance between these two possibly colliding shape pairs is calculated in the narrow phase. In the narrow phase, support points (Bergen, 2003; Snethen, 2008) are computed. A support point is a point on a shape which is farthest away in the search direction \( e \) and is computed as

\[
\text{supportPoint}(\text{geo}, r_{\text{abs}}, R_{\text{abs}}, e) = r_{\text{abs}} + R_{\text{abs}}' \cdot (\text{centroid}(\text{geo}) + \text{supportPoint}_\text{ref}(\text{geo}, R_{\text{abs}}e))
\]

where \( \text{supportPoint}_\text{ref}(..) \) is the shape-specific function to compute a support point in the reference coordinate system of the shape.

The AABB of a shape is calculated by calling the \( \text{supportPoint}_\text{ref} \) function specialized for one axis \( i = 1, 2, 3 \) in a particular axis direction \( \text{dir} = -1, 1 \).

Therefore, no shape specific AABB function is needed. The best fitting AABB’s are not useful when zero-crossing functions shall be computed, because if some surfaces or edges of a shape are also parallel to an axis, and these shapes would incidentally collide, they are already penetrating each other (see Figure 10). Therefore, it will not be possible for the variable-step integrator to detect the transition between penetration and non-penetration. Hence to avoid such scenarios, each edge length of the best fitting AABB gets enlarged by a specific factor of the longest edge length. In Figure 11 there are four shapes \( A_1, A_2, B_1, B_2 \) and each have its AABB’s shown as a grey box. Collision handling for shape pairs is switched off.

![Figure 9. Shapes A, B are not in contact.](image)

![Figure 10. Best fitting AABB’s.](image)

![Figure 11. Two rigidly attached shapes with AABB’s.](image)
are checked in the broad phase. In Figure 11, there are two rigidly attached shapes $A$, that consists of $A_1, A_2$, and $B$, that consists of $B_1, B_2$. The joints, which connects them to the ground are visualized with red cylinders. Without any assumptions, there would be 6 possible pairs to check in the broad phase. But by pre-processing the structure of the computational tree, it is reduced to 4 pairs, that have to be looked at in the broad phase whether the AABB’s are intersecting. Here only for one pair $A_2 − B_2$ the narrow phase (MPR-algorithm) has to be executed.

5.3 Compilation Time

All equations to compute the movement of Object3Ds and joints are implemented in Julia functions that can be compiled once and then just called for the actual model. Therefore, basically the same compiled code is used for any model, independent of its size. Only the @assembly code that describes which Object3Ds, joints etc. are used and how they are connected and parameterized is compiled for an actual model. But this code part is very small as compared to all the other equations. Hence, compiling a Modia3D model should be fast and nearly independent of model size. In an equation-based modeling system the equations of every instance need to be symbolically processed and translated. Therefore, the translation time grows with model size. To clarify this behavior, the following experiment was carried out:

The mechanical part of the 6 degree of freedom r3-robot present in the Modelica Standard library\(^{21}\) was used in a comparison test. In Table 1 the translation/compilation time (= time from requiring to simulate the model, until the simulation starts) of OpenModelica (1.13.0 nightly build) and of a commercial Modelica tool were compared with the compilation time of the corresponding Modia3D r3-robot model.

As expected, the simulation of the Modia3D model starts nearly immediately even for large models, whereas a waiting time is present for a Modelica model before simulation starts and this can be significant for large models.

### Table 1. Translation/compilation time for 1...100 robots (= 6...600 degrees of freedom) on a standard notebook.

<table>
<thead>
<tr>
<th>Number of robots</th>
<th>1</th>
<th>10</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenModelica</td>
<td>17 s</td>
<td>194 s</td>
<td>3600 s</td>
<td>—</td>
</tr>
<tr>
<td>commercial Modelica tool</td>
<td>5 s</td>
<td>20 s</td>
<td>80 s</td>
<td>170 s</td>
</tr>
<tr>
<td>Modia3D</td>
<td>0.3 s</td>
<td>0.4 s</td>
<td>0.5 s</td>
<td>0.6 s</td>
</tr>
</tbody>
</table>

6 Relation to other Work

Multibody systems software\(^{22}\) is designed to simulate mechanical systems, often in offline simulations. A large number of multibody codes exist such as ADAMS, RecurDyn, SIMPACK and many others\(^{23}\). Typically, specialized integration methods based on variable-step integrators are used. Furthermore, it is standard to support mechanisms with kinematic loops in a numerically sound way.

Modia3D has these features of a multibody program. However, the architecture of a typical multibody program is centered around rigid or flexible bodies where points on the body are specially marked and then objects (joints, forces, visual elements, etc.) are connected to these markers. Modia3D instead is centered around component-based design where optional components are associated to coordinate systems. The advantage is that models with many variants can be much more flexibly configured without code-duplication. For example, in the Modelica MultiBody library there are many parts, such as BodyShape, BodyBox etc. and every part defines a fixed variant (e.g. BodyBox defines a rigid body and a visual shape from a geometric box). Obviously, the number of manageable variants is limited by this design and similar code fragments are used at many places (e.g. to locate a shape object relatively to the part reference frame). Furthermore, it is planned to extend Modia3D also in non-mechanical domains (such as optionally adding heat transfer to a solid) which is straightforward with the component-based design. On the other hand, Modia3D is an experimental prototype and features are missing that are available in widely used multibody codes and are important in industrial applications.

Game engines\(^{24}\), such as Unity or Unreal engine, are used to develop games. Typically, fixed-step integrators are used in game engines, collision handling is a key element and simulation of mechanisms with kinematic loops is either not or only approximately supported. Modia3D supports collision handling in a similar way as in a game engine (currently, only elastic response calculation is supported, but it is planned to add optional impulse-based response computations). Due to the component-based design it is easy to configure the geometries that shall be treated in the collision handling (=by providing a contact material). There had been several attempts to support collision handling in Modelica, such as (Otter et al., 2005; Hofmann

---


\(^{22}\) [https://en.wikipedia.org/wiki/Multibody_system](https://en.wikipedia.org/wiki/Multibody_system)

\(^{23}\) see e.g.: [https://www.iftomm-multibody.org/software](https://www.iftomm-multibody.org/software)

\(^{24}\) [https://en.wikipedia.org/wiki/Game_engine](https://en.wikipedia.org/wiki/Game_engine)
et al., 2014; Elmqvist et al., 2015b; Bardaro et al., 2017). These approaches use external C or C++ programs for the collision handling and interface these programs to Modelica. The drawback is that a close integration into a model is hard. For example, new parts are provided that support collision handling (existing parts, such as BodyBox do not get this feature), and the same geometry is present three times: For collision handling, for animation, and for computing the rigid body properties. In Modia3D, a geometry, such as a box, is only present once. In the constructor call it is defined whether mass properties are computed from the geometry, whether the geometry is shown in the animation or whether it is utilized in collision handling, or any variant of these options.

7 Conclusion

In this article a new technique is proposed to improve modeling of 3D systems for a modeling language. Ingredients from different communities are used: The basic architecture is taken from game engines, in particular to use component-based 3D modeling to achieve a very flexible way to build-up 3D systems, to model collisions and to use various handlers for the different computational tasks. Kinematic and dynamic simulation is performed with multibody algorithms, in particular to simulate systems with kinematic loops, and by utilizing variable-step integrators with zero-crossing functions. Constructing consistent initial configurations is performed by using ideas from parameterized CAD systems. The hierarchical modeling and naming of sub-components follows the Modelica-Modia approach. The equation-based modeling language Modia shall be used to provide dynamic models from other domains, e.g. as actuators to drive a joint. On the other hand, it is planned that Modia3D models can be utilized as components in a Modia model. As a résumé it can be noted that the proposed approach seems to considerably improve the 3D modeling features of an equation-based language and could therefore be used as one building block of the next Modelica generation.

Modia3D is still an early prototype and several important parts are under development, especially the integration with Modia is missing. Furthermore, the code was currently mainly developed for its functionality and not yet tuned for efficiency. For these reasons, benchmarks about the simulation efficiency have not yet been performed, especially also not for large models (e.g. sparse matrix handling in the simulation engine was tested, but is not yet available in the publicly available prototype).

References


