Combining Equation-based and Multibody Models

Andrea Neumayr[1] Martin Otter[1]

[1]German Aerospace Center (DLR), Institute of System Dynamics and Control (SR), Germany,
{andrea.neumayr,martin.otter}@dlr.de

Abstract

This article highlights the combination of equation-based modeling with multibody models. In other words, it combines the equation-based modeling language Modia and the multibody module Modia3D. The multibody system is defined in an object-oriented way and parts of it are defined by equations. Algebraic loops are treated that appear due to the connection of multibody and equation-based components. A new approach to variable structure systems are so-called predefined acausal components which consist of pre-compiled causal parts and acausal equations. To generalize the concepts for variable structure systems, the multibody module is defined as a predefined acausal component. As a result, the number of degrees of freedom of the multibody system can vary during simulation. This is demonstrated with a non-trivial example of a walking space robot from the MOSAR space project.

Keywords: Modia, Modia3D, Modelica, Julia, multibody, variable structure systems, segmented simulation

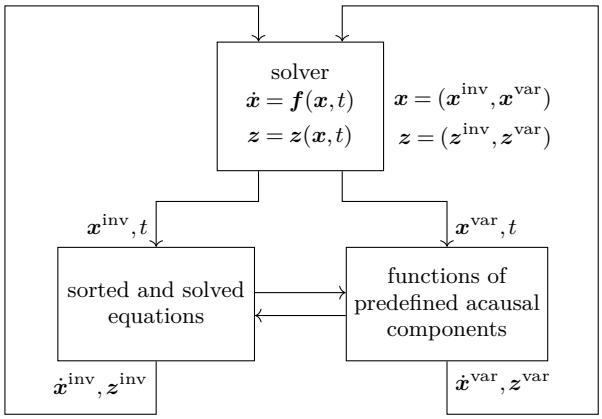


Figure 1. Predefined acausal component. Communication between the solver, the sorted and solved equations, and the functions of the predefined acausal components. The state vector $\boldsymbol{x}$ and the event indicators $\boldsymbol{z}$ are split into an invariant and a variant part: $\boldsymbol{x} = (\boldsymbol{x}^{\text{inv}}, \boldsymbol{x}^{\text{var}})$, $\boldsymbol{z} = (\boldsymbol{z}^{\text{inv}}, \boldsymbol{z}^{\text{var}})$. The variant parts consist of the states defined and used in the causal partitions of all predefined acausal components. The dimensions of the invariant parts are fixed before simulation begins. The dimensions of the variant parts can change at events during simulation.

1 Introduction

This publication is about generating Ordinary Differential Equations (ODEs) from equation-based models that are combined with variable structure multibody models. It summarizes, combines, and provides a deep insight into the findings of previous publications on symbolic transformations in Modia (Otter and Elmqvist 2017; Elmqvist et al. 2021), and iterative solving of multibody systems with Modia3D (Neumayr and Otter 2019), and variable structure systems (Neumayr and Otter 2023a; Neumayr and Otter 2023b). For this purpose, parts of previous publications are briefly repeated.

Modia is an equation-based modeling and simulation environment. It is inspired by the modeling language Modelica and has similar semantics. Modia is a domain-specific extension of the Julia programming language[1] (Bezanson et al. 2017). Modia3D is an open source multibody module with a modular and customizable component-based design pattern and is closely integrated with Modia. Furthermore, Modia supports a new approach to variable structure systems with predefined acausal components. Modia3D is one such component.

All current proposals for variable structure systems, e.g., (Mehlhase 2014; Mattsson, Otter, and Elmqvist 2015; Tinnerholm, Pop, and Sjölund 2022) require prior knowledge of all models and all modes in order to switch between these models during simulation. If this information is not available, and whenever the equation structure changes, the entire model is reprocessed and its code is regenerated and recompiled (or interpreted), e.g., (Zimmer 2010; Tinnerholm, Pop, and Sjölund 2022).

Neumayr and Otter (2023a) and Neumayr and Otter (2023b) introduce a new general concept for dealing with variable structure systems in which variables can appear and disappear during simulation. The two previous publications are briefly summarized below. There is no need to regenerate and recompile code when the number of equations and states changes at events. The method can be applied to declarative, equation-based modeling languages, such as Modia and Modelica. The transition between the modes,

[1]The pseudocode snippets in this publication are Julia-like.

called segments, is triggered by specific commands. Both the number of variables and the number of equations can vary from segment to segment.

The idea is to introduce predefined acausal components. Their equations are split into causal and acausal partitions. The causal partition is always evaluated in the same order, regardless of how the component is connected to components. This partition is sorted, explicitly solved for the unknowns, and implemented with one or more functions. The acausal partition is a set of equations that is sorted and solved. A large part of the variables in the causal partitions are hidden as local variables in functions and passed directly to the solver. This leads to the concept in Figure 1.

Based on this generic concept, this article shows how it can be applied to a class of multibody models implemented as a predefined acausal component.

2 Mathematical Descriptions

2.1 DAEs and ODEs

In equation-based modeling languages physical systems are described mathematically by Differential Algebraic Equations (DAEs) (1)

$$\boldsymbol{F}(\dot{\boldsymbol{x}}_{\mathrm{DAE}}, \boldsymbol{x}_{\mathrm{DAE}}, \boldsymbol{w}_{\mathrm{DAE}}, \boldsymbol{u}, t) = 0, \tag{1}$$

where $\boldsymbol{x}_{\mathrm{DAE}}(t)$ are variables appearing differentiated in the model, $\boldsymbol{w}_{\mathrm{DAE}}(t)$ are algebraic variables that are not differentiated, and $\boldsymbol{u}(t)$ are model inputs. These vectors depend on time $t \in \mathcal{R}$. $\boldsymbol{F}$ represents the equations of the system.

On the one hand, DAEs (1) can be solved numerically with DAE solvers such as DASSL (Brenan, Campbell, and Petzold 1996) or IDA from the Sundials suite (Hindmarsh, Serban, and Collier 2015). This approach has some limitations. For this reason, there are solvers for DAEs with a particular structure (Arnold 2017) that have much better numerical properties.

On the other hand, a system of DAEs $\boldsymbol{F}$ (1) can be transformed into Ordinary Differential Equations (ODE) in state-space form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}, t), \tag{2}$$

and solved with ODE solvers. The non-trivial transformation from an implicit DAE system to an explicit ODE system can be performed symbolically and automated by any compiler of equation-based modeling languages. If the structure of the physical system changes during simulation – known as a variable structure system – so does its underlying mathematical description represented by DAEs and its corresponding ODEs. Therefore, it would be required to execute the computationally expensive transformation and compilation from DAEs to ODEs again.

2.2 Multibody Equations

The equations of motion of a multibody system with kinematic loops are described as follows, see e.g., (Arnold 2017):

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{G}^T(\boldsymbol{q},t)\boldsymbol{\lambda} + \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) &= \boldsymbol{\tau} \\ \boldsymbol{0} &= \boldsymbol{g}(\boldsymbol{q},t), \end{aligned} \tag{3}$$

where $\boldsymbol{q}$ are the generalized coordinates of the joints of the spanning tree (such as the angle of a revolute joint), $\boldsymbol{v}$ are the derivatives of $\boldsymbol{q}$, $\boldsymbol{\tau}$ are the generalized forces in the joints of the spanning tree (such as the driving torque of a revolute joint), $\boldsymbol{\lambda}$ are the generalized forces/torques in the cut-joints, $\boldsymbol{M} = \boldsymbol{M}^T$ is the positive definite mass matrix, $\boldsymbol{g}$ are the kinematic constraint equations of the cut-joints on position level, $\boldsymbol{G} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{q}}$ are the partial derivatives of the constraint equations with respect to $\boldsymbol{q}$ and has full row rank, and $\boldsymbol{h}$ are applied generalized forces. This DAE with index 3 gives rise to numerical problems when integrating it directly. Instead, with the method of Gear, Leimkuhler, and Gupta (1985) and Gear (1988) it can be transformed to a DAE with index 1 (4), see (Otter and Elmqvist 2017; Neumayr and Otter 2019) with much more beneficial numerical properties:

$$\begin{aligned} \boldsymbol{0} &= \dot{\boldsymbol{q}} - \boldsymbol{v} + \boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\mu}}_{int} \\ \boldsymbol{0} &= \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{G}^T(\boldsymbol{q},t)\dot{\boldsymbol{\lambda}}_{int} + \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) - \boldsymbol{\tau} \\ \boldsymbol{0} &= \boldsymbol{g}(\boldsymbol{q},t) \\ \boldsymbol{0} &= \boldsymbol{G}(\boldsymbol{q},t)\boldsymbol{v} + \boldsymbol{g}^{(1)}(\boldsymbol{q},t), \end{aligned} \tag{4}$$

where:

1. The derivative of the constraint equations $\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{q},t)$ are added as new equations.
2. New unknowns $\dot{\boldsymbol{\mu}}_{int}$ are introduced to stabilize the DAE.
3. The generalized constraint forces $\boldsymbol{\lambda}$ are replaced by $\dot{\boldsymbol{\lambda}}_{int}$ the derivatives of its integral.

In the following, the focus is on the special case of tree-structured multibody systems where (3) and (4) simplify to the index 1 DAE

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) &= \boldsymbol{\tau}. \end{aligned} \tag{5}$$

This equation can be transformed into the ODE

$$\begin{aligned} \dot{\boldsymbol{q}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \boldsymbol{M}^{-1}(\boldsymbol{q},t)\left(\boldsymbol{\tau} - \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t)\right) \\ &= \boldsymbol{f}_{\mathrm{mbs}}(\boldsymbol{q},\boldsymbol{v},\boldsymbol{\tau},t). \end{aligned} \tag{6}$$

Conceptually, it is easy to define this multibody model as a predefined acausal component: The ODE

DOI
10.3384/ecp21717

(6) is part of the sorted and solved equations. The function $\boldsymbol{f}_{\text{mbs}}$ is part of the functions of the predefined acausal component, in Figure 1. However, this approach has serious drawbacks. Therefore, it is handled differently in Modia/Modia3D. The following issues are discussed in detail in the following sections:

1. *Object-oriented definition of multibody system.*
   A multibody model consists of various components, such as bodies, joints, and force elements. Users expect to drag and combine these elements individually with equation-based components. An example is shown in Figure 2 as a Modelica object-diagram. The multibody components `body, rev, world` are combined with components from equation-based libraries. The corresponding Modia/Modia3D model is in Listing 1. In section 3 is explained, how to treat the multibody components as specially marked parameters. These are used to inject equations before symbolic processing begins.

2. *Algebraic loops between multibody and equation-based models.*
   Algebraic loops can occur between multibody systems (5) and equation-based components. For example, if $\boldsymbol{\tau} = \boldsymbol{\tau}(\boldsymbol{q}, \boldsymbol{v}, \dot{\boldsymbol{v}}, t)$ due to the connection structure. Figure 2 is an example that contains an algebraic loop due to the connection of the rotational components `motorInertia, gear` to the flange of the revolute joint `rev`. Modia/Modia3D treat such algebraic loops efficiently, see section 4. For example, code-size grows linearly with the number of iteration variables.

3. *Variable structure multibody systems.*
   In Modia3D the structure of the multibody system and its degrees of freedom can vary during simulation. A non-trivial example explains how to generalize the newly introduced concepts, in section 5.

3 Object-Oriented Definitions of Multibody Systems

A Modia/Modia3D model[2] of a one-arm robot with a drive train is sketched in Listing 1 to briefly recap the object-oriented definitions of multibody systems. Parts are already published in Elmqvist et al. (2021). A corresponding Modelica object-diagram is shown in Figure 2.

Listing 1. Modia/Modia3D model of a one-arm robot with motor, ideal gear and cascaded P-PI controller that drives the flange of a revolute joint.

```
Servo = Model3D(
  world = Object3D(feature=Scene()),
  body  = Object3D(feature=Solid(...)),
  rev   = RevoluteWithFlange(
    obj1=:world, obj2=:body, axis=3,
    phi=Var(init=0.0), w=Var(init=0.0)),

  ramp          = Ramp,
  ppi           = Controller,
  wSensor       = UnitlessSpeedSensor,
  motorInertia  = Inertia,
  gear          = IdealGear,
  connect = :[
    (ramp.y, ppi.refGain)
    (gear.flangeB, rev.flange)
    ...])

servo = @instantiateModel(Servo)
simulate!(servo, stopTime=...)
```

A Modia model is defined with the predefined dictionary `Model`. All parts of the model are declared with name/value pairs. Parameters are defined with the predefined dictionary `Par`[3]. A Modia3D model is defined with the predefined dictionary `Model3D`. It may contain Modia components, see Listing 1. The instances `world, body, rev` of multibody components are individually defined and combined with instances `ramp, ppi, wSensor, motorInertia, gear` of equation-based Modia components.

Multibody components, such as `Object3D, RevoluteWithFlange`, are defined as very simple Modia components, see Listing 2 and Listing 3. They contain enough information to transform an instance of such a component into acausal and causal partitions before symbolic processing begins. This is a generic Modia approach for predefined acausal components and not specific to multibody systems.

Listing 2. Definitions of multibody components as special parameters.

```
Object3D(; kwargs...) = Par(; kwargs...,
  _constructor = :(Modia3D.Object3D))

Solid(; kwargs...) = Par(; kwargs...,
  _constructor = :(Modia3D.Solid))
```

The components in Listing 2 are defined as Julia functions with keyword arguments. All provided keyword arguments are collected by variable `kwargs...`. The function body consists of one constructor `Par`. It creates a dictionary that defines a parameter consisting of the specified keyword arguments `kwargs...`, and the additional keyword argument `_constructor = <name>`. Before a model is symbolically processed, all parameter definitions that contain a `_constructor` keyword are replaced by a reference to a Julia object. It is generated with `_constructor` and all keyword arguments of the parameter. For example, in a

[2] Modia3D.jl, v0.12.2, test/Robot/ServoWithRampAndRevolute.jl

[3] For more details, see e.g., Elmqvist et al. (2021, section 2) and the Modia tutorial https://modiasim.github.io/Modia.jl/stable/tutorial/Tutorial.html.

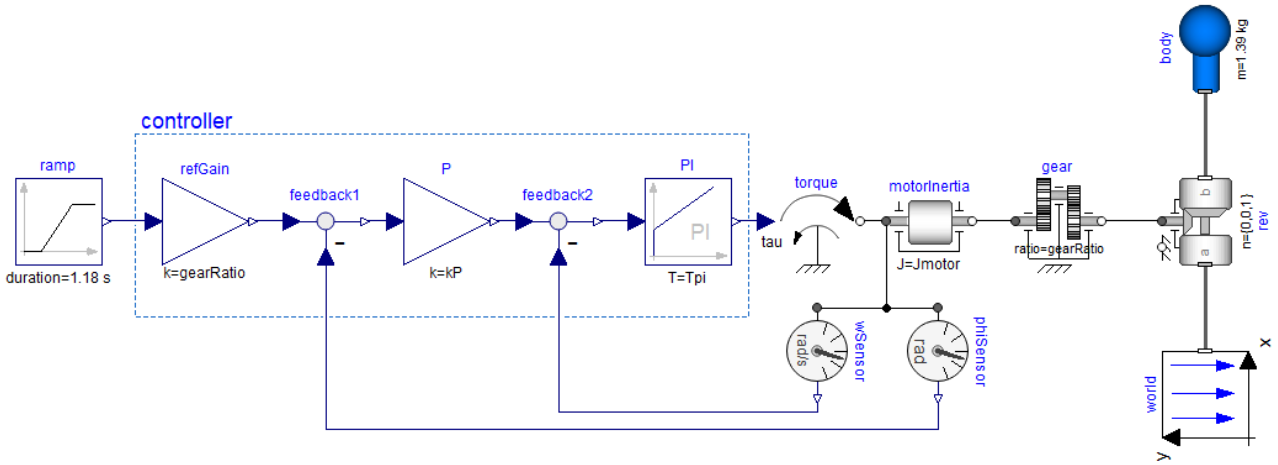


Figure 2. A single revolute joint of a manipulator rotates around the z-axis and is driven by a servo motor via an ideal gear. The revolute angle is controlled by a cascaded P-PI controller, that tracks the reference ramp.

first step `Object3D(feature = Solid())` is replaced by `Modia3D.Object3D(feature = Modia3D.Solid())`. In a second step, this constructor is executed and returns a reference to a Julia object that is associated with key `body`. This can be regarded as a generalization of the concept of External Objects in Modelica.

Listing 3. Definition of multibody components as Modia Models.

```
Flange = Model(phi=Var(potential=true),
               tau=Var(flow=true))
RevoluteWithFlange(; obj1, obj2, axis=3,
    phi=Var(init=0.0), w=Var(init=0.0)) =
  Model(;
    _constructor = Par(value =
      :(Modia3D.Joints.Revolute),
      _jointType = :RevoluteWithFlange),
    obj1    = Par(value = obj1),
    obj2    = Par(value = obj2),
    axis    = Par(value = axis),
    flange  = Flange,
    phi     = phi,
    w       = w,
    equations = :[
      phi = flange.phi
      w   = der(phi)])
```

The multibody component `RevoluteWithFlange` in Listing 3 is defined as a Modia `Model`. It consists of parameters `obj1, obj2, axis`, local variables `phi,w` (that are initialized with zero), an instance `flange` of a rotational flange, and two equations `phi = flange.phi` and `w = der(phi)`. These equations are the acausal part of a revolute joint. The causal part is defined with parameter `_constructor` together with all parameters (defined with keyword `Par`).

During instantiation of a Modia model (before its equations are symbolically processed), all parameter definitions are evaluated. For example, if a parameter `p` is defined with an equation `p = 2*Lx + 3`, assuming that `Lx = 4` is defined as a parameter, then this expression is replaced by `p = 11`.

Listing 4. Constructor generated for `RevoluteWithFlange`.

```
ref = Modia3D.Joints.Revolute(obj1, obj2,
      axis=3)
```

During the parameter evaluation, a special action is taken for parameters with name `_constructor`: A constructor call is assembled from the constructor name and any defined parameters. For example, the `RevoluteWithFlange` definition of Listing 3 results in the constructor call of Listing 4. This constructor is called on the fly resulting in an instance of Julia struct `Revolute`. The call returns a reference `ref` to the created instance. A statement like `rev = RevoluteWithFlange()` in Listing 1 is a key/value pair with the key `rev` and the value is an instance of a `Model` dictionary. This value is replaced by an instance of a parameter dictionary, resulting in `rev = Par(value = ref)`. So, the generated instance of the revolute joint is stored as a parameter. The evaluated parameters are displayed with e.g., `simulate!(logEvaluatedParameters = true)`.

The keys of other instances are referenced in the argument list, e.g., `RevoluteWithFlange(obj1 = :world)`. During parameter evaluation, symbols like `:world` are searched for on the left side of the equal signs. They are then replaced by the corresponding value of this keyword. For example, `:world` is replaced by the Julia reference created by the constructor call `Modia3D.Object3D(feature = Modia3D.Scene())`. Once all parameters are evaluated, all keyword arguments of multibody components contain a reference to the instantiated Julia objects.

A multibody model inside a Modia model is defined with dictionary `Model3D`, see Listing 1. This dictionary is a `Model` dictionary with two additional parameters `_buildFunction` and `_initSegmentFunction`, see

DOI
10.3384/ecp21717

Listing 5. Before symbolic processing begins, a model is recursively inspected. For each subdictionary containing the parameter `_buildFunction`, the function defined with `functionName` is called with the subdictionary as an argument. The items returned by this function call, are added to the subdictionary.

Listing 5. Definition of Model3D model.

```
# predefined acausal component
Model3D = Model(
  # called once before symb. processing
  _buildFunction = Par(functionName
    = :(buildModel3D!)),
  # called before each new sim. segment
  _initSegmentFunction = Par(functionName
    = :(initSegmentModel3D!))
)
```

Furthermore, the entire model hierarchy is flattened. Alias variables are eliminated. The set of all equations is generated, as sketched in Listing 6 for the model of Listing 1.

Listing 6. Flattened model equations with equations injected by `buildModel3D!`.

```
# Equation from RevoluteWithFlange
rev.w = der(rev.phi)

# Equations injected with buildModel3D!
mbs1 = Modia3D.openModel3D!(model, _x,
    time)
mbs2 = Modia3D.setStatesRevolute!(mbs1,
    rev.phi, rev.w)
mbs3 = Modia3D.setAccelerationsRevolute!(
    mbs2, der(rev.w))
genForces = ...

success =
    Modia3D.setHiddenStatesDerivatives!(
    model, mbs3, ...)
```

Function `openModel3D!` creates an instance of the multibody system. It contains, e.g., the generated instance of the revolute joint. The instantiated top level `model` is passed as an argument. So, the function `openModel3D!` has access to the complete model definition. Function `setStatesRevolute!` stores the current values of the angles and angular velocities of all revolute joints of the multibody model. These variables are states in the Modia equations, due to their definition in Listing 3. Function `setAccelerationsRevolute!` stores the angular accelerations of all revolute joints in the multibody model. Further function calls basically construct (5) in residue form. So, $\boldsymbol{f}_{\text{gen}} = \boldsymbol{M}(\boldsymbol{q},t)\dot{\boldsymbol{v}} + \boldsymbol{h}(\boldsymbol{q},\boldsymbol{v},t) - \boldsymbol{\tau}$. The set of flattened equations is processed symbolically, i.e., equations are differentiated, sorted and simplified. The result is stored as a Julia function. It is compiled into binary code that is called by the `simulate!` function.

4 Symbolic Transformations

In Modia, models are symbolically transformed to ODEs (2) in state-space form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{p},t), \quad \boldsymbol{x}_0 = \boldsymbol{x}(t_0), \tag{7}$$

where $\boldsymbol{x}(t)$ is the state vector, $\boldsymbol{p}$ is a hierarchical dictionary of parameters, and t is the time. The algorithms and the symbolic transformations are described in Otter and Elmqvist (2017) and Elmqvist et al. (2021). After the symbolic processing a Julia function called `getDerivatives` is generated and compiled to calculate the derivatives $\dot{\boldsymbol{x}}$.

Physical models often lead to linear equation systems. Modia generates very compact code to solve them numerically during execution of the model.

Assume, a nonlinear equation system

$$\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{w},\boldsymbol{u}), \tag{8}$$

with unknown local variables $\boldsymbol{w}$ and known variables $\boldsymbol{u}$ are identified by structural analysis. With the tearing algorithm of Otter and Elmqvist (2017) this equation system can be transformed to

$$\boldsymbol{w}_1 = \boldsymbol{g}_1(\boldsymbol{w}_{\text{eq}},\boldsymbol{u}) \tag{9}$$

$$\boldsymbol{0} = \boldsymbol{g}_{\text{eq}}(\boldsymbol{w}_1,\boldsymbol{w}_{\text{eq}},\boldsymbol{u}). \tag{10}$$

where the unknowns and equations are split into an explicitly evaluable part $\boldsymbol{w}_1$ and $\boldsymbol{w}_{\text{eq}}$ is solved implicitly.

If $\boldsymbol{g}$ is linear in the unknowns $\boldsymbol{w}$, it is possible to rearrange (conceptually) equation (10) into a linear equation system

$$0 = \boldsymbol{A}(\boldsymbol{w}_1,\boldsymbol{u})\boldsymbol{w}_{\text{eq}} - \boldsymbol{b}(\boldsymbol{w}_1,\boldsymbol{u}). \tag{11}$$

In (11), $\boldsymbol{A},\boldsymbol{b}$ are functions of the explicitly solved variables $\boldsymbol{w}_1$ and the known variables $\boldsymbol{u}$. The equation has to be solved for variables $\boldsymbol{w}_{\text{eq}}$. In the worst case, $\boldsymbol{A}$ would have n^2 elements ($n = \dim(\boldsymbol{w}_{\text{eq}})$). Therefore, the size of the rearranged code would be $O(n^2)$. So, the code size would increase quadratically with the number of iteration variables n.

Listing 7. Conceptual implementation of linear equation iteration.

```
# Initialize memory m (m.w_eq = 0, ...)
while true
  w_eq = m.w_eq
  w_1 = g_1(w_eq, u)
  m.r = g_eq(w_1, w_eq, u)
  if lEqIteration(m); break; end
end
```

Instead, the concept is to generate the code in Listing 7 and `lEqIteration` in Listing 8. Together they construct and solve the linear equation system (11). Residues $\boldsymbol{r}$ are computed and stored in the memory m. The linear equation system is solved to compute $\boldsymbol{w}_{\text{eq}}$ and $\boldsymbol{w}_1$ from this solution. The code size of this approach is $O(n)$.

Listing 8. Linear equation iteration `lEqIteration`.

```
function lEqIteration(m)
  n = length(m.w_eq)
  if m.mode == QUIT
    return true
  elseif m.mode == COMPUTE_B
    # compute b with w_eq = 0
    # r = A*0 - b => b = -r
    m.b = -copy(m.r)
    m.j = 1
    m.w_eq = e_1
    m.mode = COMPUTE_A
  else # m.mode == COMPUTE_A
    # compute column j of A with w_eq =
        e_j
    # r = A*e_j - b => A[:,j]
    m.A[:,j] = m.r + m.b
    if m.j != n
      m.j += 1      # j+1
      m.w_eq = e_j # j+1-th unit vector
    else
      # solve linear equation system
      # A*w_eq = b
      m.w_eq = m.A \ m.b
      m.mode = QUIT
    end
  end
  m.r = zeros(n)
  return false
end
```

The function `lEqIteration` in Listing 8 is called in a while loop from Listing 7. It iteratively computes vector $\boldsymbol{b}$, matrix $\boldsymbol{A}$, and finally $\boldsymbol{w}_{\text{eq}}$, depending on the actual mode (`COMPUTE_B`, `COMPUTE_A`, `QUIT`). All vectors $\boldsymbol{b}, \boldsymbol{r}, \boldsymbol{w}_{\text{eq}}$, matrix $\boldsymbol{A}$, column counter j, and the actual mode are stored in a memory m, and are updated when needed. To compute vector $\boldsymbol{b}$, the first mode is `COMPUTE_B`. The residues $\boldsymbol{r}$ are computed with $\boldsymbol{w}_{\text{eq}} = \boldsymbol{0}$. This allows to set $\boldsymbol{b} = -\boldsymbol{r}$. To compute matrix $\boldsymbol{A}$, the next mode is `COMPUTE_A`. To iteratively calculate the columns of $\boldsymbol{A}$, the residues are computed with $\boldsymbol{w}_{\text{eq}} = \boldsymbol{e}_j$ that is the j-th unit vector from $j = 1, \ldots, n$. When the n-th column of $\boldsymbol{A}$ is computed, so $\boldsymbol{A}$ is known, the linear equation system is solved for $\boldsymbol{w}_{\text{eq}}$. One final iteration of the while loop is needed to evaluate $\boldsymbol{w}_1$.

Moreover, symbolic processing analyses if $\boldsymbol{A}$ is a function of the parameters $\boldsymbol{p}$, so it does not change after initialization. In this case, the LU-decomposition of $\boldsymbol{A}$ is computed once at initialization and stored in the memory m. During simulation, only a (cheap) backwards solution is applied to compute the solution. If the size of the residual equation is one, a simple division is done, instead of using a linear equation solver. These special cases are not shown in Listing 8 to keep the description simple.

Modia uses the linear equation solver of the Julia package RecursiveFactorization.jl[4] with the left-looking LU-algorithm of (Toledo 1997) for dimensions up to $n = 500$ by default. Benchmarks show a large speed-up compared to the linear standard solver based on OpenBLAS[5] which is otherwise used.

The ODE and DAE solvers of Julia package DifferentialEquations.jl[6] (Rackauckas and Nie 2017) are used for the generated `getDerivatives` function. The `getDerivatives` function is called (automatically) as required by the interface of the selected solver.

One powerful technique for DAE solvers increases the simulation speed enormously. It is applicable when the size n of a linear system of equations exceeds a certain limit ($n \geq 50$), and the unknowns $\boldsymbol{w}_{\text{eq}}$ are a subset of the derivatives of the DAE states. The relevant DAE state derivatives are used as solutions $\boldsymbol{w}_{\text{eq}}$ of the linear system of equations. The residuals $\boldsymbol{r}$ are used for the DAE solver. For each model evaluation, the residuals of the linear equation system are calculated only once instead of solving a linear equation system. At events (including initialization), the linear equation system is constructed and solved, and providing consistent initial conditions for the DAE solver.

To demonstrate the outlined approach, the model in Listing 1 resp. Figure 2 is symbolically processed resulting in the `getDerivatives` function of Listing 9. In the first statements of this function, all used parameters are inquired. The states $\_x$ provided by the solver are assigned to the corresponding model variables. Afterwards, all explicitly solved equations are present. To solve the algebraic loop present in the sorted equations, a new memory m is allocated and its stored data is initialized with zero values before entering the while loop. The while loop computes the residues iteratively, to solve the multibody equations (6), the equations of components `motorInertia`, and `gear` with `lEqIteration` in Listing 8 for the iteration variable $\boldsymbol{w}_{\text{eq}}$.

Listing 9. Generated function for model in Listing 1.

```
# _x states vector from solver
function getDerivatives(_x, model, time)
  < get parameters: startTime, duration,
      kRefGain, gearRatio, ...>

  # states
  rev.phi = _x[1]
  rev.w = _x[2]
  ppi.PI.x = _x[3]

  # explicitly solved equations
  # f1 from eq (6)
  der(rev.phi) = rev.w
  ppi.refGain.u =
    ramp(time, startTime, duration)
  ppi.refGain.y =
    kRefGain * ppi.refGain.u
  motorInertia.phi = gearRatio * rev.phi
```

[4] https://github.com/YingboMa/RecursiveFactorization.jl

[5] https://www.openblas.net/

[6] https://github.com/SciML/DifferentialEquations.jl

DOI
10.3384/ecp21717

```
wSensor.flange.phi = motorInertia.phi
ppi.P.u =
  ppi.refGain.y - wSensor.flange.phi
ppi.P.y = kP * ppi.P.u
der(motorInertia.phi) =
  gearRatio * der(rev.phi)
der(wSensor.flange.phi) =
  der(motorInertia.phi)
wSensor.w = der(wSensor.flange.phi)
ppi.PI.u = ppi.P.y - wSensor.w
der(ppi.PI.x) = ppi.PI.u / Tpi
motorInertia.flangeA.tau =
  kpi * (ppi.PI.x + ppi.PI.u)
motorInertia.w =
  der(motorInertia.phi)

# open 3D model
mbs1 = openModel3D!(model, _x, time)

# set states in revolute joints
mbs2 = setStatesRevolute!(mbs1,
    rev.phi, rev.w)

begin
# new memory m: m.A=zeros(1,1),
# m.b=zeros(1), m.w_eq=zeros(1),
# m.r=zeros(1), m.j=0
# m.mode = COMPUTE_B
m = initlEqIteration(model)

while true
  # explicitly solved equations
  der(rev.w) = m.w_eq[1]
  der(der(rev.phi)) = der(rev.w)
  der(der(motorInertia.phi)) =
    gearRatio * der(der(rev.phi))
  der(motorInertia.w) =
    der(der(motorInertia.phi))
  motorInertia.a =
    der(motorInertia.w)
  gear.flangeA.tau =
    -Jmotor * motorInertia.a +
      motorInertia.flangeA.tau
  gear.flangeB.tau =
    -gearRatio * gear.flangeA.tau

  # set acceleration in joints
  mbs3 = setAccelerationsRevolute!(
    mbs2, der(rev.w))

  # f2 from eq (6): compute generalized
  # forces in joints from position,
  # velocity, acceleration, collisions
  genForces = computeGeneralizedF(mbs3)

  # compute residue vector
  if m.mode != QUIT
    m.r[1] =
      genForces[1] + gear.flangeB.tau
  end
  if lEqIteration(m); break; end
end

# report derivatives to solver
model.der_x[1] = der(rev.phi)
model.der_x[2] = der(rev.w)
model.der_x[3] = der(ppi.PI.x)
return nothing
end
```

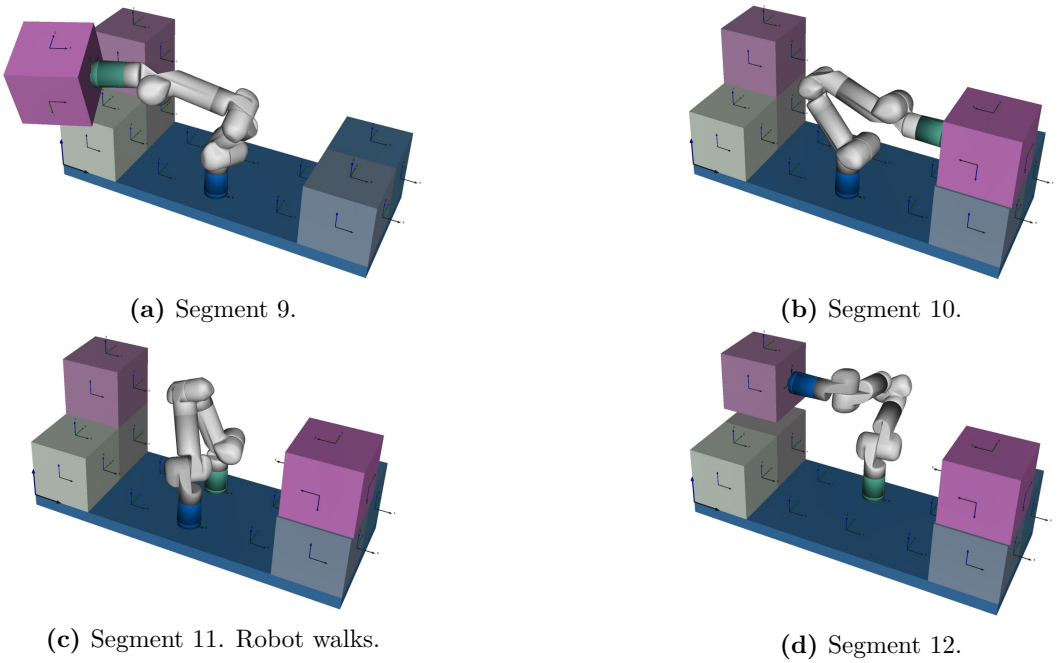
5 Variable Structure Systems: Relocatable Space Robot

Modia3D is designed as a predefined acausal component of Modia. It offers invariant and variant joints. The latter ones can be changed during simulation of variable structure systems. Currently, the category of variant joints consists of a joint type that rigidly fixes two Object3Ds and a joint type that allows a free motion between two Object3Ds. The second joint type can be replaced exclusively by another joint from this category with action commands e.g., `actionAttach`, `actionReleaseAndAttach`, `actionRelease`, `actionDelete`.

In this article, a sophisticated application with a new action command `ActionFlipChain` is discussed. This new action command allows flipping a kinematic chain with segmented simulation. It is demonstrates with a relocatable space robot (Deremetz et al. 2020). The symmetric, 7 DoF robotic manipulator belongs to the MOSAR project (Modular and Re-Configurable Spacecraft, see (Letier et al. 2019)). The robotic manipulator consists of one arm with 7 joints, and two end effectors. One end effector is colorized blue while the other is colorized green, see Figure 3. It enables the detection, manipulation and positioning of spacecraft modules. The robot relocates itself on the interfaces of the spacecraft or on the modules. The visualization data and trajectory for each joint of the robot are taken from Reiner (2022). The drive of each joint has gear dynamics that is modeled by a spring/damper pair with Modia. The 3D mechanics is modeled with Modia3D.

The described behavior above is simulated with the upcoming model. A robot places two modules and walks on a platform, such as a spacecraft. The robot uses the end effectors of its arm in Figure 3. The model shows the robot's ability of gripping the modules with either one of its end effectors and to alternate between attaching of its end effectors to the platform. This allows the robot to walk. In doing so, the kinematic chain of the robot's joints across its span of arm must be reversed. This means that the parent-child relationship between the Object3Ds is flipped. Special treatments of the joints are required to appropriately implement this.

The platform program for the robot and six modules is sketched in Listing 10. Hereby, segments 9–12 correspond to Figure 3a – Figure 3d. At initialization, the robot and the six modules are not rigidly attachted to the platform. In segment 2, the blue end effector is rigidly attached to the platform. In segments 3–8, the six modules are rigidly attached to the



(a) Segment 9. **(b)** Segment 10. **(c)** Segment 11. Robot walks. **(d)** Segment 12.

Figure 3. Walking robot on a platform. One end effector of the arm is fastened to the platform while the other one is able to place one of the two modules or it can walk on the platform.

platform. In segment 9 and 10, the green end effector is moving and replacing a module. In segment 11, the robot is walking. This means that the attachment to the platform alternates between the blue and the green end effector. The blue one is released and the green one is attached. The kinematic chain spanned between the end effectors is reversed. In segment 12, the blue end effector is gripping a module.

The relocatable space robot places two modules and walks on the platform. This scenario lasts 86 s and the simulation is performed in 2.2 s. This is much faster than real-time, since collision handling with point contacts is neglected. Moreover, it is impossible to represent collisions between two parallel surfaces with a collision algorithm that computes point contact like the Minkowski Portal Refinement (MPR) algorithm (Snethen 2008; Neumayr and Otter 2017).

Listing 10. Platform program for relocatable robot.

```
function platformProgram(actions)
  # segment 1 (from initialization)
  # segment 2
  # attach blue end effector to platform
  ActionAttach(actions,
    "blueEnd", "platform.X2Y2")
  EventAfterPeriod(actions, 1e-10)
  # segment 3 - 8
  # attach 6 modules to platform
  ActionAttach(actions,
    "boxX1Y1Z1.Zneg", "platform.X1Y1")
  ...
  EventAfterPeriod(actions, 7.0)

  # segment 9
  # attach box to green end effector
  ActionReleaseAndAttach(actions,
    "boxX1Y1Z2.Xpos", "greenEnd")
  EventAfterPeriod(actions, 17.0)

  # segment 10
  # release box off green end effector,
  # attach box to other box
  ActionReleaseAndAttach(actions,
    "boxX1Y1Z2.Zpos", "boxX5Y1Z1.Zpos")
  EventAfterPeriod(actions, 6.0)

  # segment 11
  # attach green end effector to platform
  # flip kinematic chain between blue and
  # green end effector
  ActionFlipChain(actions, "greenEnd",
    "platform.X2Y2", "blueEnd")
  EventAfterPeriod(actions, 14.0)

  # segment 12
  # attach box to blue end effector
  ActionReleaseAndAttach(actions,
    "boxX1Y2Z2.Xpos", "blueEnd")
  EventAfterPeriod(actions, 23.0)

  # segment 13
  # release box off blue end effector,
  # attach box to other box
  ActionReleaseAndAttach(actions,
    "boxX1Y2Z2.Zpos", "boxX5Y2Z1.Zpos")
  ...
end
```

This application demonstrates that by introducing new features and combining them with existing ones, the new approach for variable structure systems is relatively easy to extend.

DOI
10.3384/ecp21717

6 Conclusion

In this article, equation-based modeling and multibody modeling are combined using the example of a one-armed robot. It shows how to integrate multibody equations, equation-based Modia components and a combination of both. Therefore, Modia3D's multibody components are defined as Modia parameters. In addition, Modia3D components are defined as Modia models with an equation section. All of this is processed to generate code that is solved iteratively. The iterative solution method is discussed in detail. The multibody tree is also set up during initialization, and it is processed to calculate the generalized forces needed to solve the generated code. When dealing with variable structure systems, parts of the multibody tree are rebuilt when a new segmented is initialized.

References

Arnold, Martin (2017). "DAE Aspects of Multibody System Dynamics". In: *Surveys in Differential-Algebraic Equations IV*. Cham: Springer International Publishing, pp. 41–106. DOI: 10.1007/978-3-319-46618-7_2.

Bezanson, Jeff et al. (2017). "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1, pp. 65–98. DOI: 10.1137/141000671.

Brenan, Kathryn Eleda, Stephen L Campbell, and Linda Ruth Petzold (1996). *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*. Vol. 14. SIAM. ISBN: 0-89871-353-6.

Deremetz, Mathieu et al. (2020). "MOSAR-WM: A relocatable robotic arm demonstrator for future on-orbit applications". In: *71st International Astronautical Congress, IAC 2020*. IAF. URL: https://elib.dlr.de/139962/.

Elmqvist, Hilding et al. (2021). "Modia - Equation Based Modeling and Domain Specific Algorithms". In: *Proceedings of the 14th International Modelica Conference*. LiU Electronic Press, pp. 73–86. DOI: 10.3384/ecp2118173.

Gear, Charles William (1988). "Differential-Algebraic Equation Index Transformations". In: *SIAM Journal on Scientific and Statistical Computing* 9.1, pp. 39–47. DOI: 10.1137/0909004.

Gear, Charles William, Ben Leimkuhler, and Gopal K Gupta (1985). "Automatic integration of Euler-Lagrange equations with constraints". In: *Journal of Computational and Applied Mathematics* 12, pp. 77–90. DOI: 10.1016/0377-0427(85)90008-1.

Hindmarsh, A.C., R. Serban, and A. Collier (2015). *User Documentation for IDA v2.8.2*. Tech. rep. UCRL-SM-208112. Lawrence Livermore National Laboratory.

Letier, Pierre et al. (2019). "MOSAR: Modular spacecraft assembly and reconfiguration demonstrator". In: *15th Symposium on Advanced Space Technologies in Robotics and Automation*.

Mattsson, Sven Erik, Martin Otter, and Hilding Elmqvist (2015). "Multi-mode DAE systems with varying index". In: *11th International Modelica Conference*, pp. 89–98. DOI: 10.3384/ecp1511889.

Mehlhase, Alexandra (2014). "A Python framework to create and simulate models with variable structure in common simulation environments". In: *Mathematical and Computer Modelling of Dynamical Systems* 20.6, pp. 566–583. DOI: 10.1080/13873954.2013.861854.

Neumayr, Andrea and Martin Otter (2017). "Collision Handling with Variable-step Integrators". In: *8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. EOOLT'17. ACM, pp. 9–18. DOI: 10.1145/3158191.3158193.

Neumayr, Andrea and Martin Otter (2019). "Algorithms for Component-Based 3D Modeling". In: *13th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp19157383.

Neumayr, Andrea and Martin Otter (2023a). "Modelling and Simulation of Physical Systems with Dynamically Changing Degrees of Freedom". In: *Electronics* 12.3. DOI: 10.3390/electronics12030500.

Neumayr, Andrea and Martin Otter (2023b). "Variable Structure System Simulation via Predefined Acausal Components". In: *Proceedings of the 15th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp204.

Otter, Martin and Hilding Elmqvist (2017). "Transformation of Differential Algebraic Array Equations to Index One Form". In: *Proceedings of the 12th International Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ecp17132565.

Rackauckas, Christopher and Qing Nie (2017). "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia". In: *Journal of Open Research Software* 5.1. DOI: 10.5334/jors.151.

Reiner, Matthias J. (2022). "Simulation of the on-orbit construction of structural variable modular spacecraft by robots". In: *Proceedings of the American Modelica Conference*. LiU Electronic Press. DOI: 10.3384/ECP2118638.

Snethen, Gary (2008). "Xenocollide: Complex collision made simple". In: *Game Programming Gems 7*. Course Technology. Charles River Media, pp. 165–178. ISBN: 978-1-58450-527-3.

Tinnerholm, John, Adrian Pop, and Martin Sjölund (2022). "A Modular, Extensible, and Modelica-Standard-Compliant OpenModelica Compiler Framework in Julia Supporting Structural Variability". In: *Electronics* 11.11, p. 1772. ISSN: 2079-9292. DOI: 10.3390/electronics11111772.

Toledo, Sivan (1997). "Locality of Reference in LU Decomposition with Partial Pivoting". In: *SIAM Journal on Matrix Analysis and Applications* 18.4, pp. 1065–1081. DOI: 10.1137/S0895479896297744.

Zimmer, Dirk (2010). "Equation-based modeling of variable-structure systems". PhD thesis. ETH Zurich. DOI: 10.3929/ethz-a-006053740.