Modia – Equation Based Modeling and Domain Specific Algorithms

Hilding Elmqvist\textsuperscript{1} Martin Otter\textsuperscript{2} Andrea Neumayr\textsuperscript{2} Gerhard Hippmann\textsuperscript{2}

\textsuperscript{1}Mogram AB, Sweden
\textsuperscript{2}DLR Institute of System Dynamics and Control, Germany

Abstract

A new design of the Modia experimental modeling language based on Julia is presented. It has simple yet powerful syntax and semantics. A unified means of describing the fundamental semantics, which is similar to Modelica, is outlined. Furthermore, it is shown how domain specific algorithms can be combined with equation based modeling. It is demonstrated for multibody systems and enables more efficient translation since much repetitive analysis and transformations are avoided and faster simulation. The drive train of a robot model was automatically translated from Modelica to Modia. Modern simulation algorithms from the Julia community allow working with automatic differentiation and uncertainties.

Keywords: Modelica, Julia, Modia, Uncertainties, Multibody

1 Introduction

Modelica\textsuperscript{1} (Modelica Association 2021), the equation and object-oriented modeling language makes it easy to model large industrial systems with millions of equations. However, the transformation of the equations to executable form does not scale well since the current transformation technology is based on flattening of the hierarchical model structure. This means that certain structural and symbolic algorithms have to make repetitive work since the inherent structure is lost in the flattening process.

Furthermore, in the design of Modelica, several compromises needed to be made, for example, for multibody systems to handle spanning trees, closed kinematic loops, planar loops, over-determinism with quaternions, and reversing and zero flows for fluid systems. But even with these compromises, non-optimal performance of model transformation and simulation speed is achieved. On the other hand, the multibody community has designed methods for efficient simulation based on the manual conversion of the basic equations of motion and constraints to efficient algorithmic code, see e.g. (Arnold 2016).

Modelica also has the restriction that the number of equations and the number of states must be constant and that array dimensions of variables, even parameters, must be constant. There is a need for varying structure modeling to enable robot gripping, satellite docking, turning off parts of a fluid system, etc.

In this paper, we propose a hybrid solution: combining equation and object-oriented modeling with specialized algorithmic treatment of certain domains such as multibody and fluid systems. This approach relies on the powerful and fast programming language Julia\textsuperscript{2} (Bezanson et al. 2017). The Julia package Modia\textsuperscript{3} provides a modeling language that is based on hierarchical collections of name/value pairs. A unifying semantics has been defined for hierarchical modifiers à la Modelica and inheritance. Certain model instances are recognized as algorithmic models, i.e. calls to Julia functions are sorted among the solved equations. This technique also opens up for closed coupling to FEM and CFD models. Using a general-purpose algorithmic language instead of Modelica algorithms and functions enables use of more advanced data structures such as trees, dictionaries, etc.

It is important to have a stable model standard enabling encoding know-how available in books and articles in a formal language in order that these models can be stored and reused over a long time. The Modelica language was designed for this purpose. However, Modelica also needs to be enhanced due to various new needs within the modeling and simulation community. Modia was introduced to provide an experimental platform for extensions to the Modelica semantics and for experimentation of new transformation and simulation algorithms.

Since Modelica is a well-established modeling language there exist ten-thousands of models and it is very important to be able to reuse this huge model knowledge base. For that reason a translator between (so far for a subset of) Modelica and Modia is developed.

The presented modeling framework based on Julia has the advantage of using a modern infrastructure around the DifferentialEquations.jl package\textsuperscript{4} (Rackauckas and Nie 2017b). For example, it enables using dual number representation for automatic differentiation and uncertainty information. Julia is also used for modeling in the ModelingToolKit (Ma et al. 2021) and experiments are made to use Julia instead of MetaModelica for the OpenModelica implementation (Tinnerholm et al. 2020).

\textsuperscript{1}https://modelica.org/modelicalanguage.html
\textsuperscript{2}https://julialang.org/
\textsuperscript{3}https://github.com/ModiaSim/Modia.jl
\textsuperscript{4}https://github.com/SciML/DifferentialEquations.jl
2 Modia Language

The Modia syntax and frontend has been redesigned since (Elmqvist, Henningsson, and Otter 2017). The new design is completely based on hierarchical collections of name/value pairs together with merging of such collections. This schema is used for models, variables, equations and hierarchical modifiers.

2.1 Variables and models

Variables are implicitly defined by their references in equations. A constructor \texttt{Var} allows defining variables with attributes:

\[
\text{name} = \text{Var}(\text{attribute}=\text{value}, \ldots)
\]

\text{Var} is a function taking name/value pairs, building and returning a corresponding dictionary.

\[
\text{Var}(\text{kwars} \ldots) = \text{OrderedDict} (\text{kwars})
\]

Presently introduced attributes are: value, min, max, init, start, and the Booleans: parameter, constant, input, output, potential and flow. Example:

\[
T = \text{Var}(\text{parameter=true}, \text{value}=0.2, \text{min}=0)
\]

Some syntactically useful shortcuts using \text{Var} from ?? have been defined:

\begin{verbatim}
Listing 1. Modia shortcuts.
Par(\text{kwars} \ldots) = \text{Var}(\text{kwars} \ldots)
input = \text{Var}(\text{kwars} \ldots)
output = \text{Var}(\text{kwars} \ldots)
potential = \text{Var}(\text{kwars} \ldots)
flow = \text{Var}(\text{kwars} \ldots)
\end{verbatim}

If the value has references to other declared variables in the model, the expressions needs to be quoted that is enclosed in \texttt{"..."}. A parameter can also be defined by name = \texttt{literal-value}. time is a reserved name for the independent variable having unit \texttt{s} for seconds. The Julia package Unitful\footnote{https://github.com/PainterQubits/Unitful.jl} provides a means for defining units and managing unit inference.

Definition of units is done with a string \texttt{u"..."}(see e.g., Listing 3). Units are given to inputs, states (\texttt{init}-attribute) and if the model equations contain systems of simultaneous equations, then approximate guess values, optionally with units, must be given as \texttt{start}-attribute to iteration variables.

A model (Listing 2) is also defined as a collection of name/value pairs with the constructor \texttt{Model} (similar to \texttt{Var}, but having a tag to enable better diagnostics).

\begin{verbatim}
Listing 2. Syntax of a Modia model.
name = \text{Model}(
    <variable-or-component-definition>,
    ...
    \text{equations} = [:
        <equation1>
        <equation2>
        ...]
)
\end{verbatim}

The equations have Julia expressions in both left and right hand side of the equal sign. Note that the entire array of equations is quoted since enclosed in :[ ]. This enables later processing such as symbolically solving the equation since an AST (abstract syntax tree) is built-up instead of evaluating the expressions.

For example, in Modia a low pass filter can be defined as:

\begin{verbatim}
Listing 3. Modia model of a low pass filter.
LowPassFilter = \text{Model}(
    T = \text{Par}(\text{value}=0.2\text{u"s"}, \text{min}=0\text{u"s"}),
    u = \text{input},
    y = \text{output},
    x = \text{Var}(\text{init}=0),
    \text{equations} = [:
        T \text{ der}(x) + x = u,
        y = x
    ]
)
\end{verbatim}

This corresponds to the following Modelica model:

\begin{verbatim}
Listing 4. Modelica model of a low pass filter.
block LowPassFilter
    \text{import} Modelica.Blocks.Interfaces;
    \text{parameter} SIunits.Time T = 0.2;
    Interfaces.RealInput u;
    Interfaces.RealOutput y;
    Real x(\text{start}=0.0, \text{fixed=true});
    \text{equation}
        T \text{ der}(x) + x = u;
        y = x;
end LowPassFilter;
\end{verbatim}

2.2 Connectors

Models which contain any flow variable, a variable having an attribute \texttt{flow=true}, are considered connectors. Connectors must have an equal number of flow and potential variables, variables that contain an attribute \texttt{potential=true}, and have matching array sizes. Connectors may not have any equations. An electrical connector with potential \texttt{v} (in Volt) and current \texttt{i} (in Ampere) is defined as:

\[
\text{Pin} = \text{Model} (\text{v=potential}, \text{i=flow})
\]

2.3 Components

Components are declared by using a model name as a value in a name/value pair.

An electrical resistor with two Pins \texttt{p} and \texttt{n} can be described as follows:

\begin{verbatim}
Listing 5. Resistor model using Pins.
Resistor = \text{Model}(
    R = 1.0\text{u"\Omega"},
    p = \text{Pin},
    n = \text{Pin},
    \text{equations} = [:
        \text{O} = p.i + n.i,
        \text{v} = p.v - n.v,
        i = p.i,
        R\text{ i} = \text{v}
    ]
)
\end{verbatim}
2.4 Merging

Models and variables are defined with hierarchical collections of name/value pairs. Setting and modifying parameters of components and attributes of variables are also naturally structured in the same way. A constructor Map is used for that. For example, modifying the parameter T of the LowPassFilter model defined in Listing 3 can be made by:

```plaintext
lowPassFilter = LowPassFilter |
    Map(T = Map(value=2u"s", min=1u"s"))
```

The achieved semantics is the same as for hierarchical modifiers in Modelica and results in:

```plaintext
lowPassFilter = Model{
    T = Var(parameter=true, value=2u"s", min=1u"s")
}
```

The used merge operator | is an overloaded binary operator of bitwise or with recursive merge semantics. In Listing 6 a sketch of the recursive merging function is given:

```plaintext
Listing 6. Merging structure of name/value pairs.

function Base.:!(x, y)
    result = deepcopy(x)
    for (key, value) in y
        if typeof(value) <: AbstractDict &&
            key in keys(result)
            value = result[key] || value
        elseif key in keys(result) &&
            key == :equations
            equa = copy(result[key])
            push!(equa.args, value.args...)
            result[key] = equa
        end
        result[key] = value
    end
    return result
end
```

Merging of equations is handled specially by concatenating the equations vectors. More details, for example, about redeclaring and deleting names are given in the Modia tutorial.

2.5 Inheritance

Various physical components sometimes share common properties. One mechanism to handle this is to use inheritance. Modia makes a semantic unification by using merging.

Electrical components such as resistors, capacitors and inductors are categorized as oneports (Listing 7) that have two pins. Common properties are: constraint on currents at pins and definitions of voltage over the component and current through the component:

```plaintext
Listing 7. Oneport model for electrical components.

OnePort = Model{
    p = Pin,
    n = Pin,
    equations = :[
        0 = p.i + n.i
        v = p.v - n.v
        i = p.i
    ]
}
```

Having such a OnePort definition (Listing 7) makes it convenient to define electrical component models by merging OnePort with specific parameter definitions with default values and equations:

```plaintext
Listing 8. Electrical components merged with OnePort.

Resistor = OnePort |
    Model( R = 1.0u"Ω",
        equations = :[ R*i = v ] )
```

```plaintext
Capacitor = OnePort |
    Model( C = 1.0u"F",
        v = Var(init=0.0u"V"),
        equations = :[ C*der(v) = i ] )
```

```plaintext
Inductor = OnePort |
    Model( L = 1.0u"H",
        i = Var(init=0.0u"A"),
        equations = :[ L*der(i) = v ] )
```

```plaintext
ConstantVoltage = OnePort |
    Model( V = 1.0u"V",
        equations = :[ v = V ] )
```

The resulting Resistor (Listing 8) defined by merging with OnePort is identical to the Resistor defined in Listing 5 due to the concatenation of the equations vector performed by the merge operator.

2.6 Connections

Connections are described as a special equation of the form:

```plaintext
connect( <connect-reference-1>,
    <connect-reference-2>, ... )
```

A 'connect-reference' has either the form 'connect instance name' or 'component instance name'. 'connect instance name' with 'connect instance name' being either a connector instance, input or output variable.

For connectors, all the corresponding potentials of the connectors in the same connect statement are set equal. The sum of all incoming corresponding flows to the model are set equal to the sum of the corresponding flows into sub-components, i.e. the same semantics as in Modelica.

Connected models

Having the electrical component models from Listing 8 enables defining a filter (Listing 9), with internal resistance R1 of the voltage source, by instantiating components, setting parameters and defining connections.

```plaintext
Listing 9. Filter model defined with electrical components.

Filter = Model{
    R  = Resistor |
        Map(R=0.5u"Ω"),
    R1 = Resistor |
        Map(R=0.1u"Ω"),
    C  = Capacitor |
        Map(C=2.0u"F"),
    V  = ConstantVoltage |
        Map(V=10.0u"V"),
    equations = :[
        connect(V.p , R1.n),
        connect(R1.p , R.p),
        connect(R.n , C.p),
        connect(C.n , V.n) ]
}
```

3 Transformation of Modelica models to Modia

A recursive-descent parser for Modelica has been developed in Julia by Hilding Elmqvist (Otter, Elmqvist, et al. 2019). It builds...
an AST which is converted to the new Modia syntax.

Figure 1. RobotR3.oneAxis drive line model.

This paper focuses on combining equation based modeling with algorithmic modeling of multibody systems. A good example of this combined need is robot modeling with the multibody part combined with rotational drive trains with inertias, gearboxes, springs, etc. and electrical motors with current amplifier electronics. All this is complemented with input/output blocks of the controllers. Such an example is available in the Modelica Standard Library\(^8\) (Modelica Association 2020): Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3. The Modelica to Modia translator has been used to automatically translate the drive line model RobotR3.OneAxis in Figure 1 to Modia. The drive line consists of a path planning component, a P-PI controller, an electrical motor with current controller, a gearbox with friction, elasticity and damping and a rotational load inertia. By this translation, the know-how stored as Modelica models (55 models, 700 lines of Modia code) can be reused in an environment which enables more analyses than regular simulation as is demonstrated below.

4 Symbolic transformations

The Modia model OneAxis is instantiated with Julia macro @instantiateModel(OneAxis) that performs structural and symbolic transformations based on the algorithms sketched in (Otter and Elmqvist 2017), generates and compiles a function called getDerivatives for calculation of derivatives and returns a reference to the instantiated model where this function is stored. Transformation is currently performed to ODE form (Ordinary Differential Equations in state space form) where derivatives are explicitly solved for ($x(t)$ is the state vector, $\mathbf{p}$ is a hierarchical dictionary of parameters and $r$ is time):

\[
x = f(x, \mathbf{p}, r), \quad x_0 = x(t_0)
\]

Physical models lead often to linear systems of equations, as the Filter model in Listing 9. Modia generates very compact code to built-up and solve linear systems of equations numerically during execution of the model, as shown for the Filter model in Listing 10.

Listing 10. Generated function for model Filter.

```julia
function getDerivatives(_der_x, _x, _m, _time) 
    _p = _m.evaluatedParameters
    _leq = nothing
    time = _time * upreferred(u"s")
    var"C.v" = -_x[1] + u"v"
    var"V.v" = (_p["V"])["V"]
    var"C.p.v" = var"C.v" + -1*var"V.v"

    begin
        local var"R.v", var"Ri.i", var"R.p.v" 
        _leq = _m.linearEquations[1]
        _leq.mode = -3
        leqIteration(_leq, <more arg.>)
        var"R.v" = _leq_mode.x[1]
        var"R.i.i" = var"R.v"/((_p["R"])["R"]*1)
        var"R.p.v" = (_p["R"])["R"]*var"R.i.i"
        append!(_leq.residuals, stripUnit{
            var"R.v"-var"R.p.v"+var"C.p.v"
        })
        _leq = nothing

        end
        _leq.mode = -3

        var"der(C.v)" = -var"R.i.i" / (_p["C"])["C"]
        _der_x[1] = stripUnit(var"der(C.v)"")
        if _m.storeResult
            addToResult!(_m, _der_x, _time, var"R.v",
            var"R.p.v", var"R.i.i", var"C.p.v", var"V.v")
        end
    return nothing
end
```

Assume a nonlinear equation system

\[
\mathbf{0} = g(\mathbf{w}, \mathbf{v})
\]

has been identified with unknowns $\mathbf{w}$ and variables $\mathbf{v}$ that are known at this stage. If $g$ is linear in $\mathbf{w}$, the tearing algorithm of (Otter and Elmqvist 2017) is applied to split this linear equation system in an explicitly solvable part $\mathbf{w}_2$ and an implicit part $\mathbf{w}_1$:

\[
w_2 := g_2(w_1, v) \quad (3)
\]

\[
\mathbf{r} := g_2(w_1, w_2) \quad (= \mathbf{0}) \quad (4)
\]

\[
\mathbf{A}(w_2, \mathbf{v})w_1 - \mathbf{b}(w_2, \mathbf{v}) \quad (5)
\]

Since it is known that (2) is linear in $\mathbf{w}$, it is possible to rearrange (conceptually) equations (3.4) into the form (5). This is, however, not actually done, because $\mathbf{A}$ has $n^2$ elements and then the size of the rearranged code would grow with $O(n^2)$. Instead, only code is generated to compute the residual $\mathbf{r}$, in order that the code size grows with $O(n)$:

\[
\text{while leqIteration(\text{leq})}
\]

\[
w_1 := \text{leq.x}
\]

\[
w_2 := g_2(w_1, v)
\]

\[
\text{leq.r := g_2(w_1, w_2)}
\]

end

Function leqIteration provides first leq.x = $\mathbf{0}$ and the while loop computes $\text{leq.r := -b}$. This vector is copied into an auxiliary vector inside the data structure \text{leq}. Afterwards, leq.x = $\mathbf{e}_i$ is set to the i-th unit vector and again the residual leq.r is computed. When varying i from 1 to n, all columns of $\mathbf{A}$ are computed. Afterwards the linear system $\mathbf{A}\mathbf{w}_1 = \mathbf{b}$ is solved and in a last

\(^8\)https://github.com/modelica/ModelicaStandardLibrary
iteration the body of the while loop is again evaluated with the solution to get \( w_1 := \text{leq.x} \) and compute \( w_2 \).

During symbolic processing it is analyzed whether \( A \) is only a function of parameters \( p \), so does not change after initialization. In this case, the LU-decomposition of \( A \) is computed once during initialization and stored in the data structure \( \text{leq} \). During simulation, only a (cheap) backwards solution is applied to compute the solution. If the residual equation has size one, a simple division is used, instead of calling a linear equation solver.

By default, the linear equation solver of Julia package RecursiveFactorization.jl\(^9\) is used that implements the left-looking LU algorithm of (Toledo 1997). This solver is used up to a dimension of \( n = 500 \), because a benchmark shows a large speed-up with respect to the default linear solver based on OpenBLAS\(^10\) that would otherwise be used.

The ODE and DAE integrators of Julia package DifferentialEquations.jl\(^11\) (Rackauckas and Nie 2017a) are used with the generated \text{getDerivatives} function. If a DAE integrator is selected, the \text{getDerivatives} function is (automatically) called as needed by the interface of the DAE integrator.

Additionally, a powerful feature is included: If a DAE integrator is used and the size \( n \) of a linear equation system exceeds a particular limit (by default \( n \geq 50 \)) and the unknowns \( w_1 \) are a subset of the derivatives of the DAE states, then the following technique is used: During integration, the relevant DAE state derivatives are used as solutions \( \text{leq.x} \) of the linear equation system and the residuals \( \text{leq.r} \) are used as residuals for the DAE solver. The effect is that during integration no linear equation system is solved, but just the residuals \( \text{leq.r} \) of the linear equation system are computed \textit{once} for every model evaluation. During \textit{events} (including \textit{initialization}), the linear equation system is constructed and solved and provides consistent initial conditions for the DAE solver. The big benefit is that simulation speed can increase tremendously, see the benchmarks in section 7.

5 Operations on Modia models

5.1 Simulation with parameter merging

The Modia model OneAxis (Listing 11) is instantiated, simulated and results plotted with the following commands:

Listing 11. Instantiate, simulate and plot results of model OneAxis.

```
using Modia
@usingModiaPlot
oneAxis = @InstantiateModel(OneAxis)
simulate!(oneAxis, Tsit5(),
    stopTime=1.6u"s",
    merge=Map\{(loadMap(J=12.0))\})
plot(oneAxis, [..])
```

Function \text{simulate!} performs one simulation of the instantiated model with a solver from the Julia package DifferentialEquations.jl\(^12\) (Rackauckas and Nie 2017b). This package contains a large set of solvers. In Listing 11 the solver \text{Tsit5} is used. With various keyword arguments the simulation run can be defined. Especially, the stop time is set in the example to 1.6 s. If no unit is given, a unit of seconds is assumed. Furthermore, parameters and initial values can be provided by a hierar-

chical \text{Map} that is merged with the current values via the \text{merge} keyword. The simulation result is stored inside the instantiated model and is plotted with function call \text{plot}.

Hierarchical parameters and initial values can also be read from file, for example from a JSON file as shown in Listing 12.

Listing 12. JSON file for OneAxis parameterization.

```
{"axis": {
    "gear": {
        "ratio": 210.0,
        "c": 8.0,
        "d": 0.01,
        "Rv0": 0.5,
        "Rv1": 7.69e-4
    },
    "motor": {
        "J": 0.0013,
        "k": 5161.0,
        "w": 5500.0,
        "D": 0.6,
        "w_max": 315.0,
        "i_max": 9.0
    }
}
```

The parameters and initial values read with function \text{readMap(..)} are stored in a hierarchical map that can be directly merged in to the model before simulation starts:

```
simulate!(oneAxis, Tsit5(),
    stopTime = 1.6u"s",
    merge=readMap("OneAxisParameters.json"))
```

Units can be defined using dictionaries (value, unit) as shown in Listing 13. These dictionaries are converted to Julia values with units before the merging is done.

Listing 13. JSON structure for parameterization with units.

```
{"axis": {
    "gear": {
        "ratio": 210.0,
        "c": {"value":8.0,"unit":"N/m"}
    }
}
```

It is also possible to encode and decode such JSON parametrizations which contains Julia expressions for parameter propagation and calculations.

This offers new possibilities not available in the Modelica language: Since parameters and initial values are stored in a data structure, this data structure can be read from file or from a database system, then manipulated and finally simply merged into the model.

Typically, in Modelica parameter values are propagated and changes are performed via modifiers. For larger model hierarchies it is always hard to figure out which of the parameters to propagate and modify, because the set of parameters is too large. Sometimes records are used for the model parameterization, but then the corresponding models must be specially designed for these records, and the modeler ends up with a large set of different record types that cannot be conveniently utilized.

5.2 Simulation with different precisions

By default, a simulation is performed with 64 bit precision (Julia type \text{Float64}). However, the generated \text{getDerivatives} function does not depend on a particular type of the floating
point variables. Julia has a very elaborate type system and it is very easy to utilize different types when calling a function, provided a concrete type is not explicitly defined in the function signature. Note, the concrete type signature is given when calling the `getDerivatives` function from the solver, and then the function is specially compiled for this signature.

There is, however, one restriction: The used integrator must be prepared to use any type of floating point variables. This is the case for solvers that are natively defined in Julia, such as integrator `Tsit5`. DifferentialEquations.jl supports also external solvers that are typically implemented in C or Fortran for 64 bit precision only. These solvers cannot be used with other floating point types.

In Modia, the floating point precision is defined with the keyword `FloatType` of macro [@instantiateModel](https://docs.julia-lang.org/en/v1.6/manual/macros.html#instantiateModel). For example, the following definition simulates with 32 bit precision:

```julia
oneAxis = @instantiateModel(OneAxis, FloatType = Float32)
simulate!(oneAxis,...)
```

Simulation can also be performed with higher precision:

- `FloatType = Double64` from package DoubleFloats.jl\(^\text{13}\) uses two Float64 numbers and double word arithmetic to perform floating point operations with roughly 30 significant digits in an efficient way.

- `FloatType = BigFloat` is a Julia built-in floating point type wrapping the GNU Multiple Precision Arithmetic Library (GMP)\(^\text{14}\) and the GNU MPFR Library\(^\text{15}\) to support computations with any type of desired floating point precision. The drawback is that the computation might be slow.

### 5.3 Simulation with uncertainties

Julia package Measurements.jl\(^\text{16}\) (Giordano 2016) provides calculations with uncertainties described by normal distributions using linear error propagation theory. This package allows to define uncertain variables with nominal value and standard deviation. For example \(v = 2.0 \pm 0.2\) defines that variable \(v\) has a nominal value of 2.0 and a standard deviation of 0.2. In other words, with a probability of about 95%, variable \(v\) is in the range \(1.6 \leq v \leq 2.4\). The package overloads the Julia operators on floating point operations to perform propagation of uncertainties. This works also for functions, such as, solving linear equation systems. An example is given in Listing 14:

**Listing 14.** Uncertainty modeling with Measurements.jl.

```julia
using Measurements

v1 = 2.0 ± 0.2
v2 = 3.0 ± 0.3
v3 = v1 + v2 # = 5.0 ± 0.36
v4 = v3 - v1 # = 3.0 ± 0.3
v5 = v1*v2 # = 6.0 ± 0.85
```

In order to utilize this feature in Modia, the setting `FloatType = Measurement{Float64}` has to be used, defining that 64 bit floating point numbers with uncertainties are treated. Note, the uncertainty propagation again goes through all code, also through the integration algorithms. Therefore, this approach only works for solvers implemented in Julia. In Listing 15 this type of uncertainty modeling is applied on the OneAxis model, where uncertainties are defined for the load inertia \(J\) and the gear stiffness \(c\):

**Listing 15.** Uncertainty modeling for OneAxis model.

```julia
class=OneAxis | Model(
  load = Map(J = 19.5 ± 4.0),
  axis = Map(c = 8.0 ± 0.8),
  angle_error = :(
    axis.axisControlBus.angle_ref-load.phi)
)
```

Using the Measurements.jl package is attractive because, with very small effort, uncertainties of many variables can be defined and the propagated uncertainties of all variables computed in a reasonably efficient way. The drawbacks of this approach are that only normal distributions are supported and that uncertainty propagation is performed with linear theory, based on the analytic derivatives of all expressions. This means that larger parameter uncertainties will not be properly described by this approach and if the model contains discontinuous changes then the calculated standard deviations might be questionable.

### 5.4 Monte Carlo Simulation

**Monte Carlo Simulation**

Monte Carlo Simulation is a standard technique to evaluate a model with respect to uncertain parameters and initial values by randomly generating parameter and initial values with respect to given distributions and perform simulations for every randomly selected value set. The Julia package MonteCarloMeasurements.jl\(^\text{17}\) (Carlson 2020) provides a variant via the nonlin-

\(^\text{13}\)https://github.com/JuliaMath/DoubleFloats.jl
\(^\text{14}\)https://gmplib.org/
\(^\text{15}\)https://www.mpfr.org/
\(^\text{16}\)https://github.com/JuliaPhysics/Measurements.jl

\(^\text{17}\)https://github.com/baggepinnen/MonteCarloMeasurements.jl
ear propagation of arbitrary multivariate distributions by means of method overloading. This approach is attractive because the setup and usage is very simple, provided the underlying simulation environment is prepared to operate on any floating point type, as it is the case for Modia. A simple example of this package is given in Listing 16:


```julia
using MonteCarloMeasurements, Distributions
uniform(vmin, vmax) = StaticParticles(5,
    Distributions.Uniform(vmin, vmax))

v1 = uniform(2.0, 3.0) # v1.particles = 
    # [2.7, 2.9, 2.3, 2.5, 2.1]

v2 = uniform(5.1, 8.3)

v3 = v1 + v2 # v3.particles = 
    # [8.76, 10.24, 10.28, 9.2, 7.52]
```

Function `uniform` defines a uniform distribution between a minimum and maximum value. It generates five random values according to the given distribution and stores these five values internally in a vector. The five random values are just for illustration in this example. For realistic computations, typically several thousand random values are generated. The Julia package Distributions.jl (Besançon et al. 2019) provides a large set of probability distributions and functions operating on them and can be used to generate a large variety of distributions for MonteCarloMeasurements.jl.

The package overloads all operations for floating point numbers by replacing for example the addition of two scalars by the addition of the two vectors, in which the randomly generated values are stored. Furthermore, the package is implemented to make effective use of SIMD instructions available on modern processors. There is also support for computation on GPUs. To handle some corner cases, a few instructions in Modia had to be adapted to make Modia work with this package. In Listing 17, this type of Monte Carlo Simulation is applied on the `OneAxis` model, again, with uncertainties for the load inertia and the gear stiffness $c$ using 100 samples per distribution:

Listing 17. OneAxis model with MonteCarloMeasurements.jl.

```julia
using Modia,
using MonteCarloMeasurements, Distributions
using ModiaPlot
uniform(vmin, vmax) = StaticParticles(100,
    Distributions.Uniform(vmin, vmax))

OneAxis3 = OneAxis | Model(
    load = Map(J = uniform(11.5, 27.5)),
    axis = Map(c = uniform(6.6, 9.6)),
    angle_error = : (axis.axisControlBus.angle_ref_load.phi)
)

oneAxis3 = @instantiateModel(OneAxis3,
    FloatType=StaticParticles{Float64,100})
simulate!(oneAxis3, Tsit5(), stopTime=0.3)
plot(oneAxis3, "angle_error")
```

Function plot displays the nominal value as thicker line and the particles of the corresponding variable as transparent, thin lines.

Usage of the MonteCarloMeasurements.jl package is attractive because, with very small effort, uncertainties of a reasonable amount of variables with large uncertainties can be defined for a large variety of probability distributions and the nonlinear propagation of these distributions is computed in an efficient way.

### 5.5 Linearization

An instantiated Modia model can be linearized:

```julia
using Modia
oneAxis = @instantiateModel(OneAxis)
(A0, x0) = linearize!(oneAxis)
(A1, x1) = linearize!(oneAxis, Tsit5(),
    stopTime=1.0, analytic=true)
xNames = get_xNames(oneAxis)
```

The first `linearize!` call initializes model `oneAxis`, computes the Jacobian of the state derivatives with respect to the states $x$ numerically with a central finite difference approximation using Julia package FiniteDiff.jl, and returns the Jacobian as matrix $A0$ together with the initial state vector $x0$ computed during initialization. The nonlinear Modia model is hereby approximated at the initial state with the linear differential equation system $\Delta x = A0 \Delta x \approx x0 + \Delta x$. Linearization is performed with respect to the floating point type as defined by FloatType. If `FloatType = Measurement(Float64)`, the elements of matrix $A$ are of this type, that is, contain uncertainties. Further processing is possible, especially with Julia package ControlSystems.jl that also supports linear systems with uncertainties. In the near future, instantiation and linearization will be also supported with respect to top-level inputs and outputs of a Modia model.

The second `linearize!` call simulates the model with method `Tsit5` until the stop time, linearizes the system analytically using Julia package ForwardDiff.jl (Revels, Lubin, and Papamarkou 2016) and returns the Jacobian as matrix $A1$ together with the state vector $x1$ at the stop time. Analytic differentiation may not always work. For example, an error is currently triggered if `FloatType =

---

18https://github.com/JuliaStats/Distributions.jl
19Single Instruction, Multiple Data (= computers with multiple processing elements performing the same operation on multiple data points simultaneously).
20https://github.com/JuliaDiff/FiniteDiff.jl
21https://github.com/JuliaControl/ControlSystems.jl
22https://github.com/JuliaDiff/ForwardDiff.jl
Measurement{Float64} is used. Furthermore, analytic linearization takes some time, because the complete model is analytically differentiated, code generated and compiled.

Function call get_xNames(instantiatedModel) returns the names of the state vector x as a vector of strings. This allows to interpret further operations on the linearized system with respect to the nonlinear Modia model.

6 Modia with 3D models

6.1 Domain specific algorithms

Equation based modeling, for example with the Modelica language (Modelica Association 2021), maps a hierarchical model to a set of equations and transforms these equations appropriately. The drawback of this approach is that if a model contains, say, N instances of a body, the equations of the body are present N-times in the generated code. As a result, this approach does not scale for large models because the code size grows at least proportionally with the number of instances (and their number of equations) and therefore inherently limits the size of models that can be practically handled.

Traditional, domain-specific software, such as an electrical circuit simulator or a multibody program\textsuperscript{23}, have a completely different architecture: The equations of a component, say of a body, are available in a few variants and every variant is hard-coded in a function. If N bodies with the same variant are used, then the corresponding function is called N times, and the equations of the body are present only once. Therefore, the code size is independent from the size of the model. Additionally, special algorithms can be used, for example, to treat 3-dimensional rotations specially, handle over-determinism in planar kinematic loops, or use special sparse matrix methods, such as an O(n) multibody algorithm. The drawback of this approach is that the introduction of new component types or the combination of sub-models of different domains is orders of magnitude more difficult for a user to define than with equation based modeling.

In Modia, a new technique is utilized to combine the advantages of both approaches in a generic way, i.e., to combine equation-based modeling with domain-specific software. This approach is sketched and demonstrated in this section by combining equation based modeling with a multibody program\textsuperscript{24}.

In the simplest case, a multibody program for tree-structured systems basically has the following structure in pseudo-code notation, where \(q\) is the vector of generalized joint coordinates (for example angles of revolute joints) and \(v\) is the vector of generalized joint velocities:

\[
\begin{align*}
\text{mbs} &= \text{readAndInit("fileName")} \\
\text{q} &= \text{v} \\
\text{v} &= h(\text{mbs}, q, \text{v})
\end{align*}
\]

The multibody system is defined on file. This file is read with function readAndInit(...) that returns an object reference mbs (so basically a pointer) of an internal data structure that allows fast evaluation of function h (6) which computes the derivative of v. In the following it is shown how three basic issues are solved in Modia:

\[
\text{axle} = \text{RobotR3.Utilities.AxisType2)}
\]

- Replacing the definition of the multibody system on file by a definition with the Modia language.
- Handle object references, such as mbs, in the Modia language.
- Handle state constraints, DAE index reduction and systems of equations that might occur when combining a multibody system with equation based models, for example when a drive train without gear elasticity (say, RobotR3.Utilities.AxisType2) is connected to a flange of a revolute joint.

6.2 Modia3D

Modia3D\textsuperscript{25} (Neumayr and Otter 2018; Neumayr and Otter 2019a) is a Julia package that implements basically a multibody program, so targeted for solvers with adaptive step-size to compute results close to real physics, and combines this with the generic component-based design pattern of modern game engines. This allows a very flexible definition of 3D systems of any kind. Hereby, a coordinate system located in 3D is used as a primitive that has a container with optional components (such as geometry, visualization, dynamics, collision properties, light, camera, sound, etc.), see for example (Nystrom 2014)\textsuperscript{26}, Unity\textsuperscript{27}, Unreal Engine\textsuperscript{28}, three.js\textsuperscript{29}.

From a user’s point of view, Modia3D provides a set of predefined model components (= constructor functions that generates dictionaries). The core component is Object3D that defines a coordinate system moving in 3D together with associated, optional features, see Figure 4. An Object3D is described relative to an optional parent Object3D by vector translation that defines the coordinates of the Object3D in its parent system and by vector rotation that defines three rotation angles to rotate the parent system into the Object3D system.

An example of a simple pendulum with damping in its joint is shown in Listing 18. The Object3D object that has feature Scene is the root of all other Object3Ds and defines a global inertial system. It is called world in the example. Object3D body defines a solid part that has a mass of 1 kg. On the body an Object 3D axle is defined that is translated 0.5 m along the negative x-axis of the body. Finally, a RevoluteWithFlange joint, that is a revolute joint with a flange, constrains the motion of axle with respect to world so that axle can only rotate around its z-axis.

\[
\text{https://uwwaterloo.ca/motion-research-group/multibody-system-dynamics-international-research-activities}
\]

\[
\text{https://github.com/ModiaSim/Modia3D.jl}
\]

\[
\text{http://gameprogrammingpatterns.com/component.html}
\]

\[
\text{https://docs.unrealengine.com/Manual/GameObjects.html}
\]

\[
\text{https://docs.unrealengine.com/en-us/Engine/Components}
\]

\[
\text{https://threejs.org/docs/index.html#api/core/Object3D}
\]
Listing 18. Simple pendulum with damping in its joint.

Pendulum = Model(
    # Multibody components
    world = Object3D(feature = Scene()),
    body = Object3D(feature = Solid(
        massProperties = MassProperties(mass = 1.0)));
    axle = Object3D(parent = :body,
        translation = [-0.5, 0.0, 0.0]),
    rev = RevoluteWithFlange(axis=3,
        obj1=:world, obj2=:axis),

    # Equation based components
    damper = Damper | Map(d=100.0),
    fixed = Fixed,
    equations = :
      connect(damper.flange_b, rev.flange),
      connect(damper.flange_a, fixed.flange)
)

pendulum = @instantiateModel(
  buildModia3D(Pendulum),
  unitless=true)
simulate!(pendulum, stopTime=3.0)
plot(pendulum, "rev.phi")

The remaining elements of the Pendulum use predefined Models of a small Modia library that corresponds to the Modella.Mechanics.Rotational library. In particular a rotational 1D Damper is connected to a fixed point on one side and on the other side it is connected to the flange of the revolute joint.

Before calling @instantiateModel(.,. the special function buildModia3D must be called on the model to introduce a few equations to the model that depend on the used multibody components (details are given below). Additionally, option unitless=true has to be given temporarily, because units are not yet fully supported in Modia3D. The instantiated model can be simulated and variables plotted as before.

Feature Visual introduces objects for 3D animation by defining shapes like box, sphere, cylinder, 3D meshes on file in different formats (3ds, dxf, obj, stl), text, grids, or coordinate systems. A visualMaterial defines its visual properties. In Listing 19 an Object3D with a half transparent light blue cylinder with radius 0.01 and height 0.12 is created. For further details on visual objects see (Neumayr and Otter 2018, Section 2.2).

Listing 19. A visual Object3D with a light blue cylinder.

cylinder = Object3D(
    feature = Visual(
        shape = Cylinder(
            diameter = 0.01, length = 0.12),
        visualMaterial = VisualMaterial(
            color = "LightBlue",
            transparency = 0.5))))

Feature Solid defines solid bodies. Argument shape accepts primitive shapes and 3D meshes on file in obj-format. There are several ways for defining massProperties, including mass, center of mass, and inertia tensor of the solid: The default setting computes the mass properties from the density defined with the optional solidMaterial keyword and from the shape of shape. A solid object is considered in collision situations if keyword collision is set to true. Further-

more, it is possible to use keyword collisionMaterial to define properties of the collision behaviour, for example sliding friction coefficient and coefficient of restitution (Neumayr and Otter 2019b). In Listing 20 one link of a KUKA YouBot robot is defined as a solid 3D mesh with collision properties and its mass properties are computed from shape geometry and mass.

Listing 20. A solid Object3D that is allowed to collide and mass properties computed from shape geometry and mass.

body = Object3D(
    feature = Solid(
        shape = FileMesh(file =
            "YouBot/arm_joint_2.obj"),
        massProperties = MassPropertiesFromShapeAndMass(
            mass = 1.318),
        collision = true)
)

Feature Scene provides many options. A few are shown in Listing 21: With keyword gravity a uniform gravity field is defined pointing in negative z-direction. Only if enableContactDetection is set to true, collision handling is performed for all solid Object3Ds with enabled collision option.

Listing 21. World Object3D with scene defining a uniform gravity field pointing in negative z-direction and enabled contact detection.

world = Object3D(
    feature = Scene(
        gravity = UniformGravityField(
            g = 9.81, n = [0.0,-1]),
        enableContactDetection = true)
)

For modeling of freely moving bodies, without any kinematic constraint, Modia3D provides a FreeMotion joint with six degrees of freedom. It describes the orientation of Object3Ds by Tait-Byran (or Cardan) angles with rotation sequence x-y-z with respect to its parent Object3D, which is usually world. An advantage of this approach is that the state variables are illustrative for the user. Furthermore, in contrast to quaternions, Tait-Byran angles do not introduce nonlinear algebraic constraints and are directly defined in ODE form. If the main rotation is approximately around one axis (frequently given in technical applications) Tait-Byran angles behave nearly linear, so that integrators with adaptive step size selection can use larger steps compared to a description with quaternions.

However, a significant drawback of Tait-Byran angles is gimbal lock: When the second rotation about the local y-axis is

$$\alpha_2 = 90^\circ + n \cdot 180^\circ \quad n \in \mathbb{Z},$$

(7)
x- and z-axes are parallel. In this configuration only the sum of the first and third angle is unique. As a consequence, the model equations become singular and simulation fails.

In Modia3D this situation is avoided by adaptive rotation sequence handling. For this, the second angle is monitored by a zero crossing function which stops time integration, if the absolute difference between its value and the gimbal lock condition (7) falls below a certain limit. Before restart, the FreeMotion joint is switched to the alternative rotation sequence x-z-y such that its relative orientation remains unchanged. Since x- and z- axes are nearly parallel, the new second Tait-Byran angle about
the local z-axis obviously is far away from a gimbal lock configuration. After restart the same procedure is applied where the second angle about z is monitored and a switch back to rotation sequence x-y-z is triggered if the gimbal lock limit is reached again.

### 6.3 Modia3D implementation

In this section a sketch is given how the internal Modia3D structs and functions are interfaced with a Modia Model:

All Modia3D models, with exception of the joints, are defined in the following way:

```python
Listing 22. Modia3D interface definition.
Object3D(; kwargs...) = Par(;
    _constructor =
        (Modia3D.Composition.Object3D),
        kwargs...)
Visual(; kwargs...) = Par;
    _constructor =
        (Modia3D.Shapes.Visual),
        kwargs...)
Solid(; kwargs...) = Par;
    _constructor =
        (Modia3D.Shapes.Solid),
        kwargs...)
```

Par (see Listing 1) is a special Var provided by Modia and states that all keyword arguments are treated as parameters. The definition `Object3D(; kwargs...) = Par(;)` defines a function `Object3D` that has an arbitrary number of keyword arguments and calls Modia constructor `Par` with these keyword arguments, together with `_constructor = (Modia3D.Composition.Object3D)`. Therefore, calling function `Object3D` returns a dictionary containing these keyword arguments together with the `_constructor` keyword argument.

For the code generation, Modia processes certain keywords of a parameter, for example, to access the parameter value in the generated Julia function. Other keywords, such as the Modia3D keywords, are ignored for the code generation. Before simulation starts, all parameters are evaluated. This means that the hierarchical dictionary of the parameter definitions are traversed recursively and parameter expressions and propagated parameters are evaluated. Furthermore, whenever a `constructor` key is found, the corresponding constructor is called with the keyword arguments defined in that dictionary, the generated instance is compiled, and the value of the corresponding key (which was previously a dictionary) is replaced with a reference to the instance.

For example, `body = Object3D(...)` triggers a call of the constructor of the mutable struct `Object3D` in module `Modia3D.Composition` with the feature as keyword argument and the returned instance is used as `value` for key `body`. Some arguments of Modia3D components reference other Modia3D components, for example `axle = Object3D(parent = :body, ...)`. Since `:body` is a Julia Symbol, upper hierarchies of the hierarchical parameter dictionary are searched for a key corresponding to this symbol. Once found, the symbol used in `parent` is replaced by the `value` of key `body`, so by the reference to the body instance. After the parameter evaluation, the complete Modia3D data structure of this model is instantiated and available in the dictionary of evaluated parameters. Note, all this is generic and not specific to Modia3D.

In order that state constraints can be defined and index reduction performed, the interface to the Modia3D functionality is designed to define differential equations only on the Modia side. Since all Modia3D states are a subset of the generalized joint coordinates, part of the joint definition is done with the Modia language. For example, the definition of the `RevoluteWithFlange` joint is shown in Listing 23.

#### Listing 23. Modia definition of a revolute joint with a flange.

```python
RevoluteWithFlange(; obj1, obj2, axis=3, phi=Var(init=0.0), w=Var(init=0.0),
    canCollide=true) = Model;
    _constructor =
        (Modia3D.Shapes.Revolute),
        _jointType = :RevoluteWithFlange),

obj1 = Par(value = obj1),
obj2 = Par(value = obj2),
axis = Par(value = axis),
canCollide = Par(value = canCollide),
flange = Flange, # defined in Rotational
phi = phi,
w = w,

equations = :[
    phi = flange.phi
    w = der(phi)]
```

This definition contains a `constructor` variant, where only parts of the elements are parameters (defined with `Par`) and parts of the elements are Modia variables and equations. Only elements `constructor`, `obj1`, `obj2`, `axis` are included in the parameter data structure. All other elements, especially `equations`, are processed in the usual way by Modia.

Function `buildModia3D(model)` (see Listing 18, recursively traverses model, so a hierarchical dictionary, and collects all information about the used joints (identified by `_constructor = Par(..., _jointType=xx)` together with the path name of this joint). Based on this information, the code from Listing 24 is merged to the model.


```python
model | Model(_id = rand(Int),
equations = :[
    _mbs1 = initJoints!(_id, instantiatedModel,
        $ndofTotal, time)
    _mbs2 = setJointStates!(_mbs1,
        ($jointStates...))
    $jointForces = getJointForces!(_mbs2,
        $jointAccelerations...)
]
```

The value of a Julia expression preceded by `$` is inserted in the quoted expression, in this case, the ast of `equations` (see the result in Listing 25). Variable `_id` is a random number to provide a unique identification (details are given below). The `getDerivatives` function generated by Modia for the Pendulum example of Listing 18 is shown in Listing 25.

When the statement

```
_mbs1 = initJoints!(p[:_id],_m,1,_time)
```

is executed, the _id parameter is a dictionary with a unique identifier. The generated code contains `getDerivatives` function with the arguments `p[:_id]`, `_m`, `1`, and `time`.
Listing 25. The `getDerivatives` function of the Pendulum example of Listing 18.

```plaintext
function getDerivatives(_der_x, _x, _m, _time)
    _p = _m.evaluatedParameters
    _leq = nothing
    var"rev.phi" = _x[1]
    var"rev.w" = _x[2]
    _mbs1 = initJoints!(_p[:_id], _m, _time)
    _mbs2 = setJointStates!(_mbs1, var"rev.phi", var"rev.w")
    var"damper.tau" = (_p[:damper])[:d] * 
                      SVector(-var"damper.tau")
    _leq = nothing
    while leqIteration(_leq, <more arg.>)
        var"der(rev.w)" = _leq.x[1]
        append!(SVector(_leq.residuals), 
                getJointForces!(_mbs2, _leq, var"der(rev.w)") - 
                var"der(rev.w)"
        )
        _leq = nothing
    end
    _der_x[1] = var"der(rev.phi)"
    _der_x[2] = var"der(rev.w)"
    ...
```

is called the first time, it traverses the evaluated parameters dictionary `_m.evaluatedParameters` until parameter `:_id` with the provided value `(_p[::_id])` is found. Afterwards, it inspects all Object3D instances in this subtree and checks that they are correctly defined, for example that exactly one of them has a Scene and that all Object3Ds are directly or indirectly connected to this object. Finally, an internal data structure is instantiated in which all needed information is stored, in particular references to the root Object3D world, and to the Scene. A reference to this data structure is stored in the dictionary `_m.userObjects[::_id]` by using `_id` as key. In all subsequent calls, this data structure is retrieved by accessing the dictionary. A reference to this data structure is returned as `_mbs1`. The statement

```
_mbs2 = setJointStates!(_mbs1, ...) 
```

copies the joint states into the internal multibody data structure. Function `getJointForces!(_mbs2, ...)` computes the generalized joint forces from the generalized joint accelerations (here: `der(rev.w)`). Since the generalized joint accelerations are unknowns, this function call has to be inverted. This is performed by treating the function call as a residual equation of a linear equation system and the technique sketched in section 4 is used to solve this linear equation system with the while loop in Listing 25. If further equations are defined in the model as function of the unknown joint accelerations, for example inertias and ideal gear boxes connected to a flange of a joint, then these equations show also up in the body of the while-loop.

Note, the essential part of the multibody-related code - the three function calls - is independent of the size of the multibody system. However, all states, derivatives of states and generalized forces of the joints are present in function `getDerivatives`, so the code size is linearly dependent on the degrees of freedom of the multibody system. But this code size is two to three orders of magnitude smaller, as a corresponding code of a Modelica multibody model.

6.4 Animation

Modia3D provides a generic interface to visualize simulation results with various 3D renderers:

- Both, the free community as well as the professional edition\(^{30}\) of the DLR Visualization library\(^{31}\) (Bellmann 2009; Hellerer, Bellmann, and Schlegel 2014) are supported that provide rendering during simulation and generation of videos in different formats at the end of the simulation.

- Another option is the automatic generation of a three.js JSON file at the end of the simulation. This file can be imported in the three.js editor\(^{32}\) that allows flexible inspection of the animation and provides several ways for rendering the scene with different cameras and light options. Furthermore, the animation can be exported in the standard file format glTF\(^{33}\) or its binary `glb` version for which many viewers are available. The initial configuration can also be exported in `obj`, `ply` or `stl` format.

- Moreover, an interesting feature of Microsoft Office 2019 (e.g. Word or PowerPoint) is the importing and rendering of these file formats. While for Office 2019 only a static rendering is possible, the latest Office 365 Subscription also supports playing the animation sequence.

6.5 Example: YouBot robot

The KUKA YouBot robot is a mobile robot with a 5 degree-of-freedom arm that was manufactured by KUKA in the years 2010-2016. This robot is an attractive benchmark because a lot of data, such as CAD drawings, visualization files, and solid data is freely available from the youbot-store\(^{34}\). The YouBot robot is modeled with Modia in a similar way as the Pendulum example, see Listing 18. In Figure 5, one Youbot is handing over a ball to another Youbot. The animation is stored in a JSON file, imported into three.js, exported in glb format and can be viewed with any glb viewer, such as the Windows 3D viewer included in Windows 10. The video of this example is available in the Modia3D tutorial\(^{35}\).

7 Benchmarks

In order to evaluate the efficiency of Modia translations and simulations, the recursively defined benchmark model of figure 6 is used. It consists of a tree of solid wooden boxes and wooden spheres that are connected together with revolute joints in the form of a mobile. In every joint damping is present defined with Damper and Fixed components that are connected to the flange of the respective joint. With parameter `depth` the depth of the mobile model is defined and the Modia model\(^{36}\) is recursively constructed. This model represents a reasonable mix of Modia language and of multibody components. The essential Modia code parts are shown in Listing 26.

---

\(^{30}\)https://visualization.lrz.de/

\(^{31}\)http://www.systemcontrolinnovationlab.de/the-dlr-visualization-library/

\(^{32}\)https://three.js.org/editor/

\(^{33}\)https://www.khronos.org/glTF

\(^{34}\)http://www.youbot-store.com/wiki/index.php/YouBot_3D_Model

\(^{35}\)https://modiasim.github.io/Modia3D.jl/resources/videos/-YouBotsGripping.mp4

\(^{36}\)Modia3D/test/Profiling/Mobile.jl
Since a Modia model is basically a hierarchical dictionary, it can be constructed with the full power of the Julia programming language, and in particular with a recursive function. Essential properties of the benchmark model are summarized in Table 1. For various depths, simulations have been carried out for 5s with 500 communication points and a relative tolerance of $10^{-4}$. All parameters are evaluated and animation and plotting is switched off. Equivalent Modelica models have also been constructed and simulations performed with OpenModelica and two commercial Modelica tools. Comparing Modelica simulation tools with Modia can only be done very roughly, because the Modelica tools provide different timing information, or some timing information is not available and needs to be estimated with a stopwatch. The timing given for Modia is the time to execute \@instantiate(...) in Table 2 and simulate!(...) in Table 3.

Timings until simulation starts are given in Table 2 (column 2 gives absolute time and columns 3-5 timing factors relative to column 2).

The standard approach in Modia, M-ODE, provides the multi-body equations in the form joint-forces = f1(joint-accelerations), so given the generalized accelerations in the joints, the generalized joint forces are computed, see subsection 6.3. This single equation can be combined with additional equations, for example an inertia can be attached directly to a revolute flange and then an additional equation is added that is a function of the joint acceleration. Also, a joint can be rheonomic, so the movement given. In all these cases, a linear system of equations is generated where the generalized joint accelerations, or angular accelerations in attached drive trains or generalized forces of rheonomic joints are the unknowns. So, this approach is general and allows to handle all cases that can appear in Modelica models.

The experimental approach M-DAE, provides instead the multi-body equations in the form joint-accelerations = f2(joint-forces) and this function solves internally a linear system of equations over function f1(...). An error is raised if this function call appears in a system of equations. The effect is that, for example, it is no longer possible to attach an inertia of a drive

---

**Figure 5.** One YouBot handing over a ball to another Youbot (animation file in glb format, visualized with Windows 3D Viewer).

**Listing 26.** Recursively defined benchmark model mobile.

```julia
function createMobile(depth)
    if depth == 1
        Model(
            rod = Rod,
            sphere = Object3D(
                parent=:,(rod.frame0), ...)
    else
        Model(
            rod = Rod,
            bar = Bar | Map(L=barLength(depth)),
            sub1 = createMobile(depth-1),
            sub2 = createMobile(depth-1),
            rev0 = RevoluteWithDamping(
                obj1=:,(rod.frame2),
                obj2=:,(bar.frame0)),
            rev1 = RevoluteWithDamping(
                obj1=:,(bar.frame1),
                obj2=:,(sub1.rod.frame1)),
            rev2 = RevoluteWithDamping(
                obj1=:,(bar.frame2),
                obj2=:,(sub2.rod.frame1))
        end
    end

    mobile = Model(
        world = Object3D(feature=Scene(..)),
        top = createMobile(8),  # depth = 8
        rev0 = RevoluteWithDamping(
            obj1=:world,
            obj2=:,(top.rod.frame1),
            phi_start=0.2))
end
```

---

**Table 1. Properties of the mobile benchmark.**

<table>
<thead>
<tr>
<th>depth</th>
<th>#states</th>
<th>#unknowns</th>
<th>#solids</th>
<th>#joints</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>603</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>1140</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>3590</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>44</td>
<td>7800</td>
<td>53</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>92</td>
<td>16300</td>
<td>109</td>
<td>46</td>
</tr>
<tr>
<td>6</td>
<td>188</td>
<td>33300</td>
<td>221</td>
<td>94</td>
</tr>
<tr>
<td>7</td>
<td>380</td>
<td>67000</td>
<td>445</td>
<td>190</td>
</tr>
<tr>
<td>8</td>
<td>764</td>
<td>135000</td>
<td>893</td>
<td>373</td>
</tr>
<tr>
<td>9</td>
<td>1532</td>
<td>271000</td>
<td>1790</td>
<td>766</td>
</tr>
<tr>
<td>10</td>
<td>3068</td>
<td>543000</td>
<td>3580</td>
<td>1534</td>
</tr>
</tbody>
</table>

---

37https://www.openmodelica.org/
Table 2. Time-to-start-simulate including reading of model, symbolic transformations and 
Modia: Generation of one Julia function, eval(...) of this function and executing this function twice.

Modelica tools: Generation of many C-Code functions, compilation, generation of executable and starting the executable.
M-DAE: Modia code: joint-accelerations = f\{\text{forces}\}
M-ODE: Modia code: joint-forces = f\{\text{accelerations}\}

<table>
<thead>
<tr>
<th>tool</th>
<th>OpenModelica 1.17.0</th>
<th>tool 1</th>
<th>tool 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>M-ODE</td>
<td>15s</td>
<td>× 3</td>
<td>× 15 × 120</td>
</tr>
<tr>
<td>M-ODE</td>
<td>4s</td>
<td>× 4.5</td>
<td>× 10 × 50</td>
</tr>
<tr>
<td>M-DAE</td>
<td>15s</td>
<td>× 3.5</td>
<td>× 4.3 × 90</td>
</tr>
<tr>
<td>M-DAE</td>
<td>50s</td>
<td>× 5</td>
<td>× 2.3</td>
</tr>
<tr>
<td>M-DAE</td>
<td>204s</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 3. Simulation times for mobile benchmark.

M-ODE and the Modelica tools use Sundials CVODE and solve a linear equation system in the model.
M-ODE: Modia with Sundials.IDA() and residual algorithm.

<table>
<thead>
<tr>
<th>depth</th>
<th>M-ODE</th>
<th>tool 1</th>
<th>tool 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.1s</td>
<td>× 30</td>
<td>× 30 × 240</td>
</tr>
<tr>
<td>7</td>
<td>0.3s</td>
<td>× 80</td>
<td>× 80 × 110</td>
</tr>
<tr>
<td>8</td>
<td>1.8s</td>
<td>× 100</td>
<td>× 180 × 120</td>
</tr>
<tr>
<td>9</td>
<td>10.5s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>10</td>
<td>55s</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

train directly to a joint flange (a drive train must be attached with a compliant shaft). Also rheonomic joints cannot be used. The benefit of M-DAE is that the symbolic engine does not longer see an algebraic loop but only sortable statements, so the symbolic processing is faster and the generated code is smaller. Note, an alternative would be to use an O(n) algorithm, (Featherstone 1983) or (Brandl, Johanni, and Otter 1986), to compute the accelerations. The drawback would be, that the same restrictions as for M-DAE hold, e.g., it would not be possible to connect a 1D inertia to a joint flange.

As sketched at the end of section 4, when a DAE solver is used, all linear equation systems that exceed a given size are solved from the DAE solver during integration, provided all the unknowns of the linear equation system are a subset of the DAE state derivatives. The linear equation systems present for M-ODE and M-DAE fulfill the pre-requisites of this approach. The effect is that during integration no linear equation system is solved, but just the residuals of the linear equation system are computed.

For depth ≤ 5, the Time-to-start-simulate of the Modelica tools is at least several seconds and is much longer as the actual simulation run, whereas the Modia simulation (both M-ODE and M-DAE) starts nearly immediately. For depth ≤ 8, Time-to-start-simulate is below 15s for M-DAE and a factor of 4-120 larger for the Modelica tools.

For depth > 10, Time-to-start-simulate is no longer reasonable for M-DAE (and any other of the evaluated tools). The reason is that the generated Julia function becomes larger than a few thousand lines of code and then the quadratic increase of the Julia compilation time becomes dominant and limits the practical usage. For depth = 10, the compilation of the generated Julia code takes 174 seconds whereas the symbolic treatment of the model takes only 30 seconds. This barrier can be removed, because the generated Julia code can still be made more compact and also the technique of the Modelica tools can be used to split the computation in several functions.

The timings for the simulation runs are sketched in Table 3. As ODE integrator CVODE and as DAE integrator IDA from the Sundials suite (Hindmarsh et al. 2005) is used. Modia utilizes these solvers via Sundials.jl (Rackauckas and Nie 2017a). As can be seen, the simulation time of M-DAE is 1-2 orders of magnitude smaller than with the other solutions. The reason is that M-ODE, tool 1 and tool 2 solve a large, dense linear system of equations in every model evaluation, whereas M-DAE just computes the residuals of this equation system. The CVODE and IDA integrators calculate and factorize the system Jacobian for the benchmark simulation only about 10-20 times during one simulation run. This is just a small fraction of the linear equation systems that are solved inside every model evaluation of M-ODE, tool 1 and tool 2.

8 Conclusion and Outlook

The paper outlines a path for utilization of available Modelica models in modern tools based on Julia, at the same time allowing integration of domain specialized models, such as multibody models, coded in Julia. In addition, the Modia language and new symbolic and numerical treatment of DAEs provide an experimental platform for developing new modeling capabilities. To make this path feasible, the translator from Modelica to Modia needs to be extended and be able to invoke domain specializations for multibody, fluid, media, etc. fully automatically.

It has been shown how simulation with uncertainties can be performed efficiently. A natural next step is to use these solver capabilities in the context of optimization and machine learning for surrogate models for speeding up simulations utilizing available packages from the Julia eco-system.

The Modia prototype handles benchmark models consisting of large multibody systems together with equation-based components much more efficiently as the examined Modelica tools - both for startup/compilation time as well as for simulation speed.

References


