Innovations for Future Modelica

Hilding Elmqvist[1]   Toivo Henningsson[2]   Martin Otter[3]

[1]Mogram AB, Magle Lilla Kyrkogata 24, 223 51 Lund, Sweden, `Hilding.Elmqvist@Mogram.net`
[2]Lund, Sweden, `toivo.h.h@gmail.com`
[3]Institute of System Dynamics and Control, DLR, Oberpfaffenhofen, Germany, `Martin.Otter@dlr.com`

Abstract

This paper discusses language innovations for future Modelica versions, on the one hand for generally applicable language elements, and on the other hand to improve modeling of multibody systems with contacts, and media modeling. In a companion paper new algorithms are proposed to handle much larger models than can be treated today. All these innovations are developed and evaluated with the experimental modeling and simulation environment Modia. Modia is based on Julia, a powerful programming language with strong focus on scientific computing, meta-programming and just-in-time compilation that allows very fast development. The modeling language is directly defined and implemented with Julia's meta-programming constructs and is designed tightly together with the symbolic and numeric algorithms. This approach is very well suited for innovation and experimenting with evolutions of modeling capabilities in Modelica.

Keywords: Modelica, Modia, Julia, modeling, simulation

1 Introduction

The objective is developing and testing innovations for future Modelica versions with reasonable effort both from a language point of view as well as for new symbolic and numeric algorithms that are tightly designed together with the language elements. To achieve this goal, an experimental modeling and simulation environment called Modia is under development. Modia uses a Modelica-like language. It shall be both simpler and more powerful than Modelica 3.3 (*Modelica Association, 2014*) and takes into account the experience gained with Modelica in the last 20 years.

New algorithms have been already developed and test-implemented in Modia and are described in the companion paper (*Otter and Elmqvist, 2017*). For example, arrays defined in a model stay as arrays in the generated code, even if (array) equations need to be differentiated. This is a pre-requisite to handle much larger models than what can be treated with current Modelica tools.

In addition to equations, Modelica has a function concept for procedural programming of tasks, such as table look-up, media calculations and control system implementations. The function part of Modelica is, however, not rich enough. There are no advanced data structures such as union types, no matching construct. Type inference is missing with the implication that there are presently separate blocks for adding Reals, Integers and Complex numbers. The evolution of Modelica has slowed down since it's a too large task to make a full algorithmic language. Instead of inventing all such features, it makes sense to use another language as a base.

Julia (*Bezanson, et al., 2017*) is a very promising language design effort with focus on scientific computing and has many of the properties needed to complement the equational style for modeling. Julia also allows definition of real equations (expression = expression). Furthermore, advanced meta-programming features are available which are suitable for symbolic treatment of equations before just-in-time compilation.

Julia allows developing a modeling language together with a public reference implementation so that language features and symbolic/numeric algorithms are designed tightly together. Native Julia functions are used in models and equations use Julia syntax.

Examples of other research oriented language designs for modeling are: SOL (*Zimmer, 2010*), Hydra (*Giorgidze and Nilsson, 2009*) and Modelyze (*Broman and Siek, 2012*). There is also one experimental simulation package for Julia called Sims (*Short, 2012*). Sims does not make any structural and symbolic processing though, but has event handling. It is based on ideas from Modelyze and Hydra.

This paper introduces major language constructs of Modia and proposes new language features for future Modelica versions. Other aspects of Modia and its implementation are given in (*Elmqvist, et al., 2016*). Modia is available from https://github.com/ModiaSim.

2 Modia Language Design

2.1 Model with differential equations

Modia is a domain specific language extension of Julia by means of structured macros, that is, the Julia parser is used to parse Modia models.

A simple first order example model is shown below:

```
@model FirstOrder begin
  x = Variable(start=1)
  T = Parameter(0.5, "Time constant")
  U = 2.0
@equations begin
  T*der(x) + x = u
  end
end
```

@model is a call to the Modia macro called **model**. The first part after **begin** is used for variable and component declarations by means of calling constructors. The second part inside the **@equations** macro contains differential and algebraic equations as well as connections. # starts a Julia comment. Semicolons can be omitted in Julia.

The constructor `Variable` is used to declare `x` with a start value of 1. In general it constructs instances of ordinary variable types and arrays of those. It is a Julia composite type which in addition to its value also allows specifying type, min, max, variability, start value, info, etc. The constructor `Parameter` is a specialization of the Variable constructor which sets the variability to parameter, that is, a quantity that is changeable before simulation starts but constant during simulation. There is also a special short hand notation to define parameters by just giving a default value. This notation is used to define the parameter `u`. The operator der() denotes the time derivative of its argument.

The corresponding Modelica model is:

```
model FirstOrder
  Real x(start=1);
  parameter Real T=0.5 "Time constant";
  parameter Real u = 2.0;
equation
  T*der(x) + x = u;
end FirstOrder;
```

Modia uses the Julia way to declare variables with constructor calls. The benefit with respect to current Modelica is a simpler syntax since value, variability, info, etc. are all given in the constructor calls. This allows to easily extending the language with new attributes/properties in the future.

2.2 Coupled models

In order to couple models, the interfaces need to be defined. For simplicity of the language and its implementation, this is currently described as a **@model** (and might be improved in the future by a dedicated **@connector** macro):

```
@model Pin begin
  v = Float()
  i = Float(flow=true)
end
```

`Float` is a specialization of `Variable` with fixed type `Float64`. The flow variable, i, is marked with an attribute `flow=true`. Such a Pin can be used to define the terminals `p` and `n` of an electrical resistor:

```
@model Resistor begin
  p = Pin()
  n = Pin()
  v = Float()
  i = Float()
  R = Parameter(info="Resistance")
@equations begin
  v = p.v - n.v # Voltage drop
  0 = p.i + n.i # KCL within component
  i = p.i
  R*i = v # Ohm's law
  end
end
```

An electrical component library has been developed containing also Capacitor, Inductor, VoltageSource, etc. A low-pass filter can then be defined as a set of connected components:

```
@model LPfilter begin
  R = Resistor(R=100)
  C = Capacitor(C=0.001)
  V = ConstantVoltage(V=10)
@equations begin
  connect(V.p, R.p)
  connect(R.n, C.p)
  connect(C.n, V.n)
  end
end
```

The function `connect` has the same meaning as in Modelica. Note, that no ground component is needed because the missing ground can be automatically handled with a new algorithm described in (*Otter and Elmqvist, 2017*). Modia is used to evaluate whether this simplification is reliable and transparent for the user. The diagram of a corresponding Modelica model is shown in Figure 1.

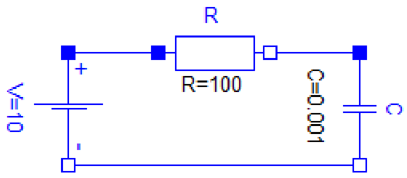

Figure 1. Low pass filter (without ground object)

2.3 Inheritance

There are several electrical components that share the property of having two Pins. Such components are called OnePorts. Similarly to Modelica, it is possible to describe the common properties once and inherit them. The common properties are:

```
@model OnePort begin
  p = Pin()
  n = Pin()
  v = Float()
  i = Float()
@equations begin
  v = p.v - n.v # Voltage drop
  0 = p.i + n.i # KCL within component
  i = p.i
  end
end
```

The Resistor model can then be simplified:

```
@model Resistor begin
```

```
  @extends OnePort()
  @inherits i, v
  R = Parameter(info="Resistance")
@equations begin
  R*i = v # Ohm's law
  end
end
```

The @**extends** macro incorporates all declarations and all equations from `OnePort`. The `OnePort` variables can be accessed by `this.v` and `this.i` in the equations of the `Resistor`. The @**inherits** macro enables to directly use variables `i` and `v`.

2.4 Type and size inference

The Modelica Standard Library (*Modelica Association, 2016*) contains similar models operating on different data types. One example is switches, which based on a Boolean signal select between two Real, two Booleans, or two Complex numbers. There is a desire in the Modelica community to unify this situation by means of type inference.

To experiment with type and size inference, such a feature is included in Modia: Variable constructors do not need to specify type and size. Types and sizes can be inferred from the environment of a model or start values provided, either initial conditions for states or approximate start values for algebraic constraints.

A generic switch that can be applied to matrices and strings as well, can be then defined as:

```
@model Switch begin
  sw = Boolean()
  u1 = Variable()
  u2 = Variable()
  y = Variable()
@equations begin
  y = if sw; u1 else u2 end
  end
end
```

2.5 Variable Declarations

There are, however, cases when size and type inference based on start values is not natural, for example, when algebraic equations form a linear system of real simultaneous equations. In such a case, the solution is independent of any start value and only size needs to be given.

It is possible to provide type information in variable declarations using the type parameter `T` in the `Variable` constructor or its short version `Var`:

```
v1 = Var(T=Float64)
```

It is unspecified if the variable v1 is a scalar or array of `Float64`. It is possible to provide information that a variable is of array type with a certain number of dimensions:

```
array = Var(T=Array{Float64,1})
matrix = Var(T=Array{Float64,2})
```

The size can be fixed using the size attribute:

```
scalar = Var(T=Float64, size=())
array3 = Var(T=Float64, size=(3,))
matrix3x3 = Var(T=Float64, size=(3,3))
```

The size is given with the tuple constructor according to the result of the Julia `size` function. Empty tuple, (), means scalar. A vector size is given as a tuple with the size. Such a tuple with one element needs a comma to distinguish it from an expression within parenthesis. When the size attribute is given, T denotes the array element type.

There is also a FixedSizeArrays module for Julia (*Danish, 2014*) which gives faster code since stack allocation is possible and garbage collection avoided. The corresponding Modia declarations are then:

```
fixedArray3 = Var(T=Vec{3,Float64})
fixedMatrix3x3 = Var(T=Mat{3,3,Float64})
```

SI units can be given using the Julia SIUnits module (*Fisher, 2013*). It has predefined types such as: `Meter, KiloGram, Second, Ampere, Kelvin, Mole, Candela, Radian, Steradian, Joule, Coulomb, Volt, Farad, Newton, Ohm, Siemens, Hertz, Watt, Pascal`. A Modia variable with unit `Volt` is declared as:

```
v2 = Var(T=Volt)
```

There is an option in the SIUnits module to use units with shorter names (`m, kg`, etc) (and * is not needed between literal and identifier), for example:

```
m=2.5kg
length=5m
```

However, this feature is not useful since unit `m` would then be in the same name space as the variable `m`. Investigations are being made to allow a local scope for units after literals using a syntax with [ ]:

```
m=2.5[kg]
length=5[m]
```

2.6 Type Declarations

To avoid repeatedly typing type and size information, it's possible to define alternative variable constructors outside the @**model** macro:

```
Float3(; args...) = Var(T=Float64,
  size=(3,); args...)
Voltage(; args...) = Var(T=Volt;
  args...)
```

The notation "`; arg...` " denotes a list of keyword arguments which are just passed to the `Variable` constructor using the same notation. This means that a 3-vector with start attribute and a three-phase `Voltage` variable can be declared as:

```
v3 = Float3(start=zeros(3))
v4 = Voltage(size=(3,), start=[220.0,
  220.0, 220.0]Volt)
```

2.7 Redeclaration of submodels

With the "replaceable" language element, Modelica has a powerful concept to exchange submodels on a lower level. However, it is complicated to understand and difficult to implement for tools. Furthermore, it is not powerful enough for certain applications, because redeclarations cannot be controlled by variables and they must be planned in advanced, because only models can be replaced that are marked to be **replaceable**.

A simpler and more powerful concept for redeclarations has been tested in Modia and follows naturally from the constructor style of declaration using expressions, as shown in the following example:

```
MotorModels = [Motor100KW,
               Motor200KW,
               Motor250KW] # Modia models
selectedMotor = motorConfig(  ) # Int

@model HybridCar begin
  @extends BaseHybridCar(
    motor = MotorModels[selectedMotor](),
    gear  = if gearOption1; Gear1(i=4)
            else Gear2(i=5) end)
end
```

In model `BaseHybridCar` every submodel can be replaced without being marked. In particular new motor and gearbox models are provided. The motor model is selected from an array of Modia models via an integer. The gearbox model is selected based on a logical condition. Such flexible types of redeclarations cannot be formulated in Modelica 3.3.

2.8 Multi-mode Modeling

Several attempts have been made to generalize the semantics of Modelica to allow mode changes, for example (*Mattsson, et al., 2015*). However, only a limited classes of problems could be handled. One reason is the imposed restriction that the equations are only processed once, code is generated and this code should hold for all mode changes. There are academic simulation prototypes that dynamically process and switch equations during run-time, such as (*Zimmer, 2010*; *Höger, 2014*). The question is how to incorporate such ideas in to Modelica and Modelica tools with the goal to solve real-world industrial problems.

First investigations have been carried out in Modia to experiment with changing model structure. Consider the model of an electrical motor with a load in Figure 2. The shaft between motor and load breaks at a certain time.

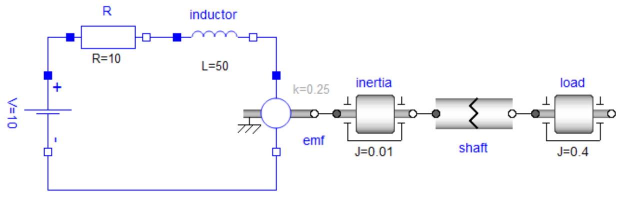

Figure 2. Electrical motor, load and breaking shaft.

The breaking shaft can be modelled as follows using conditional equations:

```
@model BreakingShaft begin
  flange1 = Flange()
  flange2 = Flange()
  broken = Boolean()
@equations begin
    if broken
      flange1.tau = 0
      flange2.tau = 0
    else
      flange1.w = flange2.w
      flange1.tau + flange2.tau = 0
    end
  end
end
```

Figure 3 shows the angular speeds of the two inertias when the shaft breaks at time = 100.

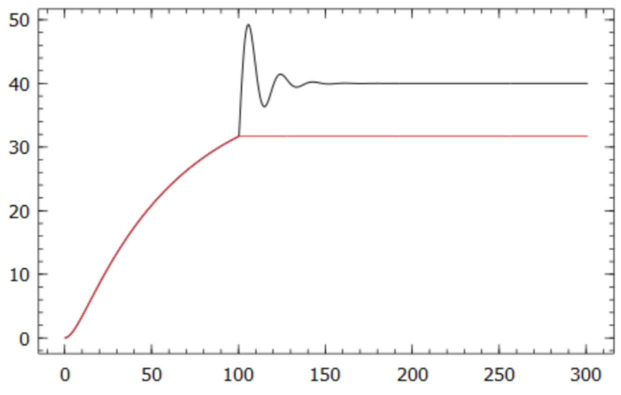

Figure 3. Angular speeds of inertias

The set of model equations and the DAE index is changing when the shaft breaks. The Modia environment makes new symbolic transformations and just-in-time compilation for each mode of the system. The final results of variables before an event is used as initial conditions after the event.

Mode changes with conditional equations might introduces inconsistent initial conditions causing Dirac impulses to occur. This more general problem is treated in (*Benveniste, et al., 2017*).

2.9 Other features

Other features, such as type and size inference, time events, synchronous controllers, state events, multi domain models are exemplified in (*Elmqvist, et al., 2016*). There are also ongoing development and experimentation regarding nested simulations, etc.

3 Model Examples

3.1 Multibody modeling

Multibody models uses vector and matrix equations. Below, a tiny multibody library is defined with similarities to package MultiBody of the Modelica Standard Library

(*Modelica Association, 2016*). First, convenient Variable constructors involving SIUnits are defined.

```
Position(; args...) =
  Var(T=Meter; size=(), args...)
Velocity(; args...) =
  Var(T=Meter/Second; size=(), args...)
Acceleration(; args...) =
  Var(T=Meter/Second^2; size=(), args...)
Angle(; args...) =
  Var(T=Radian; size=(), args...)
AngularVelocity(; args...) =
  Var(T=Radian/Second; size=(), args...)
AngularAcceleration(; args...) =
  Var(T=Radian/Second^2; size=(),
  args...)
Force(; args...) =
  Var(T=Newton; size=(), args...)
Torque(; args...) =
  Var(T=Newton*Meter; size=(), args...)
Mass(; args...) =
  Var(T=KiloGram; size=(), min=0,
  args...)
```

Based on these scalar Variable constructors, vector and matrix constructors can be defined.

```
Axis3(; args...) =
  Var(T=SIPrefix; size=(3,), args...)
Position3(; args...) =
  Position(size=(3,); args...)
Velocity3(; args...) =
  Velocity(size=(3,); args...)
Acceleration3(; args...) =
  Acceleration(size=(3,); args...)
Rotation3(; args...) =
  Var(T=SIPrefix; size=(3,3),
  property=rotationGroup3D, args...)
AngularVelocity3(; args...) =
  AngularVelocity(size=(3,); args...)
AngularAcceleration3(; args...) =
  AngularAcceleration(size=(3,); args...)
Force3(; args...) =
  Force(size=(3,); args...)
Torque3(; args...) =
  Torque(size=(3,); args...)
Inertia3(; args...) =
  Var(T=KiloGram*Meter*Meter, size=(3,3);
  property=symmetric, args...)
```

It should be noted that Rotation3, the type for rotation matrices, has a special property:

```
property=rotationGroup3D
```

In particular this means that the element declared in this way is a 3x3 rotation matrix that has 9 elements with 6 implicit constraints between them. In case kinematic loops are present, this property of rotation matrices would lead to redundant constraint equations that are difficult to handle. As discussed in (*Elmqvist and Mattsson, 2016*), a tool can, however, automatically remove this redundancy of a kinematic loop in a pre-processing step, provided the rotation matrices are marked, as done above. Compared to current Modelica, the benefit is that no special operators Connections.branch/.root/.isRoot etc are needed anymore. Note, these operators are awkward, difficult to understand and it is easy to make mistakes.

Other properties can be defined as well. In particular, the Inertia3 constructor specifies the matrix to be symmetric. This can enable better user interface for setting parameters.

The coupling semantics is defined by Frames.

```
@model Frame begin
  r_0 = Position3()
  R = Rotation3()
  f = Force3(flow=true)
  t = Torque3(flow=true)
end
```

A Prismatic joint has two Frames. Axis of translation is given by a vector parameter n.

```
@model Prismatic begin
  n = Axis3(value=[1,0,0],
    variability=parameter)

  frame_a = Frame()
  frame_b = Frame()

  s = Position(start=0*Meter)
  v = Velocity(start=0*Meter/Second)
  a = Acceleration()
  f = Force()
@equations begin
  v = der(s)
  a = der(v)

  frame_b.r_0 = frame_a.r_0 +
    frame_a.R'*(n*s)
  frame_b.R = frame_a.R
  frame_a.f = -frame_b.f
  frame_a.t + frame_b.t =
    cross(n*s, frame_b.f)

  # d'Alemberts principle
  f = -dot(n, frame_b.f)
  f = 0*Newton # Not driven
  end
end
```

A Revolute joint has similar structure.

```
@model Revolute begin
  n = Axis3(value=[0,1,0],
    variability=parameter)

  frame_a = Frame()
  frame_b = Frame()

  phi = Angle(start=0)
  w = AngularVelocity(start=0)
  a = AngularAcceleration()
  tau = Torque()
  R_rel =  Rotation3()
@equations begin
  R_rel = n*n' + (eye(3) - n*n')*cos(phi)
    - skew(n)*sin(phi)

  w = der(phi)
  a = der(w)

  frame_b.r_0 = frame_a.r_0
  frame_b.R = R_rel*frame_a.R
  frame_a.f = -R_rel'*frame_b.f
  frame_a.t = - R_rel'*frame_b.t

  # d'Alemberts principle
  tau = -dot(n, frame_b.t)
  tau = 0*Newton*Meter # Not driven
  end
end
```

The skew function is defined as:

```
skew(x) = [    0 -x[3]  x[2];
             x[3]     0 -x[1];
            -x[2]  x[1]     0]
```

Gravity is defined by the following function:

```
gravityAcceleration(r) =
  9.81*[0,-1,0]*Meter/Second^2
```

A Body has one Frame. The parameter r_CM gives the vector from the frame to center of mass.

```
@model Body begin
  r_CM = Position3(variability=parameter)
  m = Mass(variability=parameter)
  I = Inertia3(variability=parameter)

  frame = Frame()

  r_0 = Position3()
  R = Rotation3()
  v_0 = Velocity3()
  a_0 = Acceleration3()
  w_a = AngularVelocity3()
  z_a = AngularAcceleration3()
  g_0 = Acceleration3()
  W = Var(T=Float64, size=(3,3))
@equations begin
  r_0 = frame.r_0
  R = frame.R
  g_0 = gravityAcceleration(r_0 + R'*r_CM)

  # Translational kinematic differential
  # equations
  v_0 = der(r_0)
  a_0 = der(v_0)

  # Rotational kinematic differential
  # equations
  W = der(R)*transpose(R)
  w_a = [W[3,2], W[1,3], W[2,1]]
  z_a = der(w_a)

  # Newton/Euler equations
  frame.f = m*(R*(a_0 - g_0) +
    cross(z_a, r_CM) + cross(w_a,
    cross(w_a, r_CM)))
  frame.t = I*z_a + (cross(w_a, I*w_a) +
    cross(r_CM, frame.f))
  end
end
```

The coordinate systems must be fixed for multibody dynamics. This is done by using a World object:

```
@model World begin
  frame = Frame()
@equations begin
  frame.r_0 = zeros(3)*Meter
  frame.R = eye(3,3)
  end
end
```

A simple sliding mass model is shown below:

```
@model TranslationalBody begin
  world = World()
  j = Prismatic(n=[1,1,1]/sqrt(3),
    v = Velocity(start=1*Meter/Second))
  body = Body(r_CM=[0.5,0,0]*Meter,
    m=1.0*KiloGram,
    I=1e-3*eye(3)*KiloGram*Meter^2 )
@equations begin
  connect(world.frame, j.frame_a)
  connect(j.frame_b, body.frame)
  end
end
```

3.2 Functions and data structures

One of the reasons for developing Modia on top of Julia is to have direct access to Julia algorithmic features, i.e. much more powerful functions and data structures than available in current Modelica.

One of the limitations of current Modelica is a convenient way of handling collisions of many objects for DEM (Discrete Element Modeling). The problem is that there are n*(n-1)/2 potential contacts possible for n objects. The user can of course not explicitly make these connections.

One approach is that each object registers its position. After that, the forces between each pair of objects in contact are calculated. Then each object retrieves the sum of the forces acting on the object. This force is used in the equations of motion. In (*Elmqvist et al., 2015*), the information about each object and the above calculations are handled in C/C++. A problem is that there is no convenient method in current Modelica to make sure all objects have registered their position before forces are extracted. An elaborate scheme involving inner/outer construct together with flow variables was used.

An experimental feature has been included in Modia to solve this problem. The built-in operator `allInstances(v)` creates a vector of all the variables v within all instances of the class where v is declared. It can be seen as a specialization of the proposed Modelica array constructor: [c.v **for** c **in class** Class], (*Elmqvist, et al., 2015b*). This construct did not make it into Modelica 3.4 due to concerns about self-reference and mutual recursive loops. The `allInstances` operator is referring to the class where it's used but has a more restricted semantics.

Consider modeling a set of spherical balls moving on a plane. We will assume the same radius for simplicity and a force law of a spring-damper during contact. A Modia model is shown below.

```
@model Ball begin
  r = Var()
  v = Var()
  f = Var()
  m = 1.0
@equations begin
  der(r) = v
  m*der(v) = f
  f = getForce(r, v, allInstances(r),
    allInstances(v), (r,v) -> (k*r + d*v))
  end
end
```

The force is dependent on the position and velocity of all Balls, that is, the `allInstances` operator is used on both `r` and `v`. The force law is provided as an anonymous function: `(r,v) -> (k*r + d*v)`.

A set of Balls can easily be modelled by just instantiation. The contact handling is automatic:

```
@model Balls begin
  b1 = Ball(r = Var(start=[0.0,2]),
            v = Var(start=[1,0]))
  b2 = Ball(r = Var(start=[0.5,2]),
            v = Var(start=[-1,0]))
  b3 = Ball(r = Var(start=[1.0,2]),
            v = Var(start=[0,0]))
end
```

In this case with three balls, the operator `allInstances(r)` expands to `[b1.r, b2.r, b3.r]`.

The force contributions from all other balls are calculated according to the spring-damper model by function `getForce`:

```
const k=10000
const d=100
const radius=0.05

function getForce(r, v, positions,
  velocities, contactLaw)
  force = zeros(2)
  for i in 1:length(positions)
    pos = positions[i]
    vel = velocities[i]
    if r != pos
      delta = r - pos
      deltaV = v - vel
      f = if norm(delta) < 2*radius;
        -contactLaw((norm(delta)-
          2*radius)*delta/norm(delta),
          deltaV) else
        zeros(2) end
      force += f
    end
  end
  return force
end
```

The described technique opens up the possibility for further important optimizations. In order to avoid O(n^2) complexity when deciding which objects that are in contact, space partitioning by quad-trees or oct-trees can be used, see (*Elmqvist et al., 2015*). This requires recursive data structures that are available in Julia.

3.3 Media Modelling

The Modelica.Media library within the Modelica Standard Library[1] provides a large set of packages and functions to compute media properties of one and two-phase media dedicated for simulation. Although the Media library is powerful, it has conceptual limitations for the modeling of media with multiple substances that have multiple phases. Furthermore, the details of the library are difficult to understand and difficult to support by Modelica tools due to the extensive use of replaceable packages and functions. There have been several attempts to simplify the approach and making media modeling more powerful.

Julia allows a fresh view on this difficult topic and it seems that *multiple dispatch* and other Julia features allow a surprisingly simple way to model complex media: Following the Modelica.Media library design, a medium has the following orthogonal properties:

1. *Medium states* that define the independent variables of the medium. A medium may have different types of independent variables. For example, it might have as independent variables pressure p and temperature T or pressure p and specific enthalpy h. In Julia they would be described as types.
2. *Medium constant data* that defines constants for every instance of a specific medium. For example a simple medium may have a constant d_const for the mean density. In Julia constant data would be described as constants in a module.
3. *Medium immutable data* that defines constants specific to an instance of a specific medium that cannot be changed once the medium is instantiated. Typically reference points such as h_offset may have a default value, but might be changed for particular medium instances. In Julia such data would be described as immutable types.
4. *Medium functions* that define properties of a medium as function of the medium constant and immutable data and the medium states. For example `density(medium,state)` computes the density for a medium using the given state description.

Below is a sketch of a new Media library design:

```
module Media  # Interface of media models
  # Possible medium states
  type State_pT
    p::Float64
    T::Float64
  end

  type State_ph
    p::Float64
    h::Float64
  end

  # Possible medium functions
  density(medium,state)=error(..)
```

[1] http://doc.modelica.org/om/Modelica.Media.html

```
  specificEnthalpy(medium,state)=error(..)
  setState_pT(medium,p,T)=error(..)
  setState_ph(medium,p,h)=error(..)
  ...
end
```

In a generic module Media, the supported medium states and the supported medium functions are collected. The default implementation of the functions for every medium are error messages. However, also concrete functions could be added here that hold for every medium.

A specific medium is implemented with a Julia module, as shown here for a simple water model:

```
module SimpleWater
  import Media

  # Constants of medium
  const cp_const = 4184.0
  const cv_const = 4184.0
  const d_const  = 995.586
  const T0       = 273.15

  # Variables specific to an instance
  immutable Medium
    h_offset::Float64
    Medium(;h_offset=0.0) = new(h_offset)
  end

  # Functions of medium
  Media.density(
    m::Medium,
    state::Media.State_pT) = d_const
  Media.specificEnthalpy(
    m::Medium,
    state::Media.State_pT) =
    cp_const*(state.T - T0) + m.h_offset
  Media.setState_pT(m::Medium, p, T) =
    Media.State_pT(p,T)
  Media.setState_ph(m::Medium, p, h) =
    Media.State_pT(p,
      T0+(h-m.h_offset)/cp_const)
  ...
end
```

In a Modia model Julia data structures and functions can be used. As a result, it is possible to instantiate a medium model at some place with

```
medium1=SimpleWater.Medium()
medium2=SimpleWater.Medium(h_offset=10.0)
```

and then propagate this medium through all connected fluid component models:

```
@model FluidPort begin
  # contains medium, p, h, ...
end
...
port = FluidPort()
...
port.medium = SimpleWater.Medium()
```

Inside a component model, medium properties are computed. The implementation of such a component model neither knows which concrete medium model is used, nor which independent states the medium has, so the component model can be used for all media that provide an implementation of the used functions:

```
state = setState_ph(port.medium,
                    port.p,
                    port.h)
d = density(medium,state)
h = specificEnthalpy(medium,state)
```

Julia selects the concrete functions to be called based on the *medium* type *and* the *state* type. This is the key innovation that makes media modeling suddenly so simple: a function is (statically) selected based on the types of *several* arguments.

4 Implementation

The Modia implementation is made in Julia which provides meta-programming capabilities which are suitable for symbolic treatment of the equations.

4.1 Meta-programming in Julia

Languages such as Modelica and Modia require symbolic transformations of equations into executable code. A mathematical expression is conveniently represented by an AST (abstract syntax tree). The Julia language (*Bezanson, et al., 2017*) allows creation of "quoted" expressions encapsulated as, ″`:(…)`″.

```
julia> equ = :(0 = x + 2y)
:(0 = x + 2y)
```

Such an expression is stored as an AST. The AST can be shown by using a built-in function, `dump()`:

```
julia> dump(equ)
Expr
  head: Symbol =
  args: Array(Any,(2,))
    1: Int64 0
    2: Expr
      head: Symbol call
      args: Array(Any,(3,))
        1: Symbol +
        2: Symbol x
        3: Expr
          head: Symbol call
          args: Array(Any,(3,))
          typ: Any
      typ: Any
  typ: Any
```

equ is of type Expr which has three fields: head, args and typ. equ.head is the Symbol = representing the equality of the two expressions of the equation. The right hand side is the sum of two expressions: x and 2y. The operator + is represented as a function call: equ.args[2].head. Which function to call is defined in equ.args[2].args[1]. The operands of the + operator are equ.args[2].args[2] and equ.args[2].args[3].

A new AST can be built using the Expr constructor. For example, solving an equation of the form:

```
0 = x + expression
```

can be done as follows:

```
julia> solved = Expr(:(=),
  equ.args[2].args[2], Expr(:call, :-,
  equ.args[2].args[3]))
:(x = -(2y))
```

It is also possible to create a quoted expression referring to parts of `equ` by the use of "interpolation", `$( )`.

```
julia> solved = :($(equ.args[2].args[2]) =
  - $(equ.args[2].args[3]))
:(x = -(2y))
```

The result is presented as a quoted expression. By assigning the variable y, it's possible to calculate x using the `eval` function on the AST `solved`:

```
julia> y = 10
10
julia> eval(solved)
-20
julia> @show x
x = -20
```

4.2 Symbolic Transformations of Modia Models

The following list shows some of the structural and symbolic transformations which are performed by the Modia implementation:

- Instantiation
- Flattening
- Alias elimination
- Type and size inference
- Removal of singularities
- Index reduction and BLT of array equations
- Symbolic differentiation of matrix equations
- Symbolic solution of matrix equations
- Partial state selection and tearing
- Transformation to a special index one DAE
- Determining sparseness structure of Jacobian

Modia supports *type and size inference*, that is, the Variable constructor does not need to specify type and size. However, Pantelides algorithm and removal of singularities require that types and sizes of variables and equations are known. Types and sizes are inferred from the start values provided and by propagation. The left and right hand sides of equations are evaluated with given start values and the type and size inference of Julia is used to determine the size and types of variables and equations.

There are useful application models where structural symbolic algorithms fail and may lead to strange error messages during symbolic processing or to run-time errors. For example, if an electrical circuit is not grounded, the potentials of the electrical Pins can float, that is, the system equations are underdetermined. On the other hand, the equations are overdetermined regarding currents. Such *singularities* needs to be removed before further structural processing. Details of such a technique is described in the companion paper (*Otter and Elmqvist, 2017*).

The Pantelides algorithm and other structural index reduction algorithms are designed for scalar variables and equations. So Modelica tools typically symbolically expand array equations into a set of scalar equations involving the variable elements. This is not feasible if large array equations are used, for example, for flexible bodies or other discretized partial differential equations. Generalizations of BLT and Pantelides algorithms to directly handle *array equations* can be found in (*Otter and Elmqvist, 2017*).

Pantelides algorithm determines which array equations that needs to be differentiated. Special care are needed when performing *symbolic operations on array and matrix equations* since matrix multiplication is not commutative. Solving for unknowns are done by a set of rewrite rules. As an example, the right division operator, `/`, or the left division operator, `\`, is used depending on whether the unknown is on the right or left side of a multiplication operator. Special rules can be used for rotation matrices to replace division by multiplication with the transpose of the rotation matrix.

4.3 Numeric Solution of Modia Models

Numeric treatment and transformation of the resulting differential algebraic array equations to index one form is described in the companion paper (*Otter and Elmqvist, 2017*).

5 Outlook

The Modia experimental language gives new possibilities for creation of new innovative language elements and algorithms to model and simulate more complex models than is possible in current Modelica.

The suggested innovations of the companion paper (*Otter and Elmqvist, 2017*) can be directly utilized in current Modelica tools. A change in the Modelica language is not needed for them. Part of the proposed innovations in this paper for new language elements, such as type inference, marking of rotational matrices in combination with new algorithms, or the allInstances(..) operator, could be included in a fully backwards compatible form in a future Modelica 3.x version.

The use of native Julia for the algorithmic part would simplify the Modelica effort considerably since Modelica does not need to be extended with new features in functions. This means that evolution of Modelica could be focused on the equational modeling aspects.

Contributions to Modia for language design and for improved symbolic and numeric algorithms are welcome.

References

Benveniste A., Caillaud B., Elmqvist H., Ghorbal K., Otter M., and Pouzet M. (2017): *Multi-Mode DAE Models - Challenges, Theory and Implementation*. Lecture Notes on Computer Science, submitted for review.

Bezanson J., Edelman A., Karpinski S. and Shah V.B. (2017): *Julia: A Fresh Approach to Numerical Computing*. SIAM Review, Vol. 59, No. 1, pp. 65-98. http://julialang.org/publications/julia-fresh-approach-BEKS.pdf; see also: http://julialang.org/

Broman D., Siek J. G. (2012): *Modelyze: a Gradually Typed Host Language for Embedding Equation-Based Modeling Languages*, University of California at Berkeley, No. UCB/EECS-2012-173, www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-173.html.

Danish D. (2014): *FixedSizeArrays*, https://github.com/SimonDanisch/FixedSizeArrays.jl

Elmqvist H., Goteman A., Roxling V., Ghandriz T. (2015): *Generic Modelica Framework for MultiBody Contacts and Discrete Element Method*. Proceedings 11th International Modelica Conference, Versailles. http://www.ep.liu.se/ecp/118/046/ecp15118427.pdf

Elmqvist H., Olsson H., Otter M. (2015b): *Constructs for Meta Properties Modeling in Modelica*. Proceedings 11th International Modelica Conference, Versailles. http://www.ep.liu.se/ecp/118/026/ecp15118245.pdf

Elmqvist H. and Mattsson S.E. (2016): *Exploiting Model Graph Analysis for Simplified Modeling and Improved Diagnostics*. Proceedings EOOLT '16, April 18, Milano, Italy.

Elmqvist J., Henningsson T. and Otter M. (2016): *System Modeling and Programming in a Unified Environment based on Julia*. Proceedings of ISoLA 2016 Conference Oct. 10-14, T. Margaria and B. Steffen (Eds.), Part II, LNCS 9953, pp. 198-217.

Fisher K. (2013): *SIUnits*. https://github.com/Keno/SIUnits.jl

Giorgidze G., Nilsson H. (2009): *Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems*. In Proceedings of the 7th International Modelica Conference, pages 208–218, Como, Italy. http://www.ep.liu.se/ecp/043/022/ecp09430137.pdf.

Höger C.: *Dynamic structural analysis for DAEs*. In Proceedings of the 2014 SCS Summer Simulation Multiconference, 2014.

Mattsson S.E, Otter M., and Elmqvist H. (2015): *Multi-Mode DAE Systems with Varying Index*. Proceedings 11th International Modelica Conference, Versailles. http://www.ep.liu.se/ecp/118/009/ecp1511889.pdf

Modelica Association (2014): *The Modelica Language Specification, Version 3.3 Revision 1*, https://www.modelica.org/documents/ModelicaSpec33Revision1.pdf

Modelica Association (2016): *The Modelica Standard Library, Version 3.3.2*, https://github.com/modelica/Modelica

Otter M., and Elmqvist H. (2017): *Transformation of Differential Algebraic Array Equations to Index One Form.* Modelica Conference 2017, Prague, May 15-17.

Short T. (2012): *Sims - A Julia package for equation-based modeling and simulations.* https://github.com/tshort/Sims.jl.

Zimmer D. (2010): *Equation-Based Modeling of Variable Structure Systems*. PhD Dissertation, ETH Zürich. http://e-collection.library.ethz.ch/eserv/eth:1512/eth-1512-02.pdf.