Resizable Arrays in Object-Oriented Modeling

Martin Otter[1] Hilding Elmqvist[2]

[1]DLR, Institute of Vehicle Concepts, Germany, martin.otter@dlr.de

[2]Mogram AB, Sweden, hilding.elmqvist@mogram.net

Abstract

The Modelica language (Modelica.org) makes it easy to build large, complex models by allowing the instantiation of reusable component models. Modelica tools typically expand arrays of variables, equations and components and perform symbolic transformations on the scalar elements. This reduces the efficiency of the translation process and makes it difficult to change array dimensions after translation.

This paper describes modest enhancements of standard algorithms to avoid scalarization. As a result, arrays can be resized both after translation and during simulation. The new technique does, however, impose certain restrictions on the way models are written. It is also sketched how to provide more meaningful diagnostics for erroneous models. Several examples demonstrate the new algorithms using the Web App Modiator.

Keywords: Modelica, array equations, compilation

1 Introduction

The Modelica language[1] makes it easy to build large, complex models by allowing the instantiation of reusable component models. Modelica tools typically expand arrays of variables, equations and components and perform symbolic transformations on the scalar elements. This reduces the efficiency of the translation process especially for models with fine discretization of partial differential equations or for handling arrays of parametric data. In many cases, this scalarization can be omitted, which will be explored in this paper.

Furthermore, avoiding scalarization of arrays allows changing array sizes without recompilation. This is important since it's essential to check the fidelity of the discretization for partial differential equations to ensure sufficient accuracy, i.e. to allow quick comparison of simulation results without recompilation with different numbers of segments. The technique also enables changing array dimensions during simulation, i.e., a model modifies its discretization depending on the model's behavior, for example increasing the number of segments when a transient occurs.

This paper outlines the modest enhancements to standard algorithms required for this new approach. A tool can verify whether the model complies with the required restrictions. If so, the new technique is employed, otherwise the current scalarization approach is used. The new approach imposes certain restrictions on models. These restrictions and how to rewrite the model equations to fulfill them are described. The alternative is to scalarize the equations internally, and then try to combine them into array equations and for-loops after the symbolic algorithms have been applied. Such an approach is much more involved.

The new approach and all the examples in this article have been tested with the Web App *Modiator* which supports a subset of Modelica. The current features, restrictions and future plans of Modiator are discussed in the companion paper (Elmqvist and Otter 2025).

Previous work on avoiding scalarization includes the following articles: (Otter and Elmqvist 2017) show that scalarization of array equations is needed for index reduction when using the standard algorithms, e.g., for multibody systems. They scalarize array equations internally and reconstruct arrays after sorting. (Pop et al. 2019) describe a new high-performance frontend to OpenModelica[2] (Fritzson et al. 2020) to convert arrays of component models to array equations and to keep arrays during flattening. (Zimmermann et al. 2020) propose set-based graph algorithms to avoid unrolling of for-loop equations of Modelica models. (Abdelhak et al. 2023,2025) map indices of array variables and equations, as well as iterator values of for-equations to unique scalar indices, apply standard algorithms for matching and sorting on these internally scalarized systems and reconstruct array equations and for-loops afterwards. They tested this approach with an extension to OpenModelica. (Marzorali et al. 2024) propose a matching algorithm for repetitive structures with for-loops over vectors of unknowns and test the implementation as part of the ModelicaCC compiler[3].

2 Model Restrictions

This section explores the restrictions that are needed to be imposed on Modelica authoring to avoid scalarization with the new method and achieve compile times that are independent of array sizes (*O(1) rather than O(n) or worse*).

Restriction 1: *The model must be balanced with regards to non-scalarized variables and equations. The equation*

[1] https://specification.modelica.org/maint/3.6/MLS.html

[2] https://openmodelica.org/

[3] https://github.com/CIFASIS/modelicacc

number of a for loop is the number of equations in the loop.

This restriction is illustrated using the CascadedFirstOrder model from the ScalableTestSuite[4] (Casella 2015):

```
tau*der(x[1]) = u - x[1];
for i in 2:N loop
  tau*der(x[i]) = x[i-1] - x[i];
end for;
```

Equations for tau and u are provided elsewhere. This equation set violates **Restriction 1** and needs to be modified since the number of unknowns is one (der(x)) but there are two equations. It can, for example, be rewritten as:

```
for i in 1:N loop
  tau*der(x[i]) = (if i == 1 then u else x[i-1]) - x[i];
end for;
```

or more compactly using the concatenation operator cat:

```
tau*der(x) = cat(1, {u}, x[1:N-1]) - x;
```

The cat function concatenates the array arguments, in this case along the first dimension. It means that the x vector is shifted upwards (x[N] omitted) and u is inserted as first element. Yet another possibility is to use a reduction expression.

```
tau*der(x) = {(if i == 1 then u else x[i-1]) - x[i] for i in 1:N}
```

Restriction 2: *All elements of an array must be of the same kind (algebraic or differentiated).*

This means that a model such as a variant of the CascadedFirstOrder model from above:

```
x[1] = u;
for i in 2:N loop
  tau*der(x[i]) = x[i-1] - x[i];
end for;
```

violates **Restriction 2** because x[1] is an algebraic variable and variables x[2:N] are variables appearing differentiated. Such a model needs to be rewritten in one of the forms shown above. In similar cases, arrays need to be split in different pieces so that all elements of an array are either algebraic or states.

An equivalent model to the CascadedFirstOrder model can be made with an array of components:

```
model FiltersInSeries
  model Filter
    input Real u;
    output Real x;
    parameter Real tau = 0.1;
  equation
    tau*der(x) = u-x;
  end Filter;
  parameter Integer N = 10 "Number of filters";
  Filter f[N](each x(start=0, fixed=true),
              u=cat(1, {1}, f[1:N-1].x));
end FiltersInSeries;
```

If all components in an array of components have the same causality, all component equations can be sorted together, i.e. that each equation in an array of components can be translated to a for loop over that equation (it is not important in this simple case since there is only one equation in each component).

The outputs f.x[1:N-1] are shifted to form the inputs f.u together with the input 1 to the first filter. Handling acausal connectors is somewhat more complex. See the *Transmission Line* example in section 4.3.

In section 5 more *restrictions* are discussed and there might be still more. Modiator never performs scalarization (also not internally) and therefore *valid Modelica models might be rejected.* The user should be able to understand how to rewrite the model equations from the error messages. The benefit is that generated code never contains scalarized equations or enrolled for-loops, translation is faster and error messages are more compact since the elements of an array are not shown.

3 Assignment without Scalarization

This section gives an overview how to symbolically process array equations so that arrays can be resized after compilation - both for translated Modelica models and Functional Mockup Units (≥ version 3.0)[5]. The details of the new algorithms are presented in **Appendix A**.

3.1 Model Unification

In order to make symbolic processing simpler, the Modelica model is transformed by a model unification process similar to what is done in OpenModelica (Pop et al. 2019):

- For-equations with multiple equations in the body are converted to potentially nested for-equations with *one* equation in the body:
  ```
  for index-expression, … loop
    equation
  end for;
  ```
- Non-constant variables of arrays of components are transformed to array variables with outer indices according to the component array indices.
- Each equation of an array of components is transformed to a for-loop over the equation.

3.2 Structurally Assignable Property

Afterwards, standard symbolic algorithms are used that operate, with some modifications on variable symbols. The following notation is used:

- All variable symbols v_j are collected in a variable vector $\boldsymbol{v}$. A symbol v_j may represent a scalar or an array of one or more dimensions. The first variable v_0 has index $j = 0$ (as it is common in programming languages such as C, C++, Javascript).

[4] https://github.com/casella/ScalableTestSuite

[5] https://fmi-standard.org/docs/3.0.2/

DOI 10.3384/ecp218189

- All equations e_i are collected in an equation vector $\boldsymbol{e}$. An equation e_i may be a scalar or an array equation of one or more dimensions with potentially a for-loop around it, as sketched above. The first equation e_0 has index $i = 0$.
- The relationship between the equations e_i and the variables v_j is defined by a sparse representation of the incidence matrix that is defined as vector $\boldsymbol{G}$, where each entry G_i is a vector consisting of the indices of the variable symbols appearing in equation e_i. $\boldsymbol{G}$ can also be interpreted as a representation of a bi-partite graph.
- The relationship between variable symbols is defined by the variable association vector $\boldsymbol{A}$:
  $A_j = \textbf{if } \dot{v}_j = v_k \textbf{ then } k \textbf{ else } -1.$
- The relationship between the equations is defined by the equation association vector $\boldsymbol{B}$:
  $B_i = \textbf{if } \dot{e}_i = e_k \textbf{ then } k \textbf{ else } -1.$

Additionally, the following *new property* needs to be defined for every variable in every equation:

Definition 1: *A variable v_j is called structurally assignable with respect to an equation e_i, if a complete assignment is possible for all (scalar) elements of v_j with respect to all (scalar) elements of e_i, provided v_j and e_i are expanded to scalars and this property holds, independently of the common dimensions and dimension sizes of v_j and e_i.*

In general, **Definition 2** is difficult to apply. The following sufficient condition is easier to use:

Theorem 1: *A variable v_j is structurally assignable with respect to an equation e_i, if*

- *v_j is a scalar and e_i a scalar equation, or if*
- *v_j is an array and e_i can be expressed as $0 = s \cdot v_j + w$, where s is a scalar expression and w is a vector expression and neither s nor w are a function of v_j.*

Proof: (a) A scalar variable/equation is already expanded and therefore the scalar variable can be assigned. (b) If e_i is expressed as $0 = s \cdot v_j + w$, the k-th element of v_j can be assigned to the k-th element of e_i. ■

The (new) structural assignable property is defined with vector $\boldsymbol{G_a}$, a companion vector of $\boldsymbol{G}$, such that $G_{a,i,j} =$ true, if variable $v_{G_{i,j}}$ is structurally assignable with respect to equation e_i and otherwise $G_{a,i,j} =$ false. Note, every vector $G_{a,i}$ must have at least one true value, since otherwise no variable can be assigned for equation e_i.

The underlying DAE (Differential Algebraic Equations) of a model is defined as:

$$\boldsymbol{e}(\boldsymbol{v}(t), t) \tag{1}$$

where t is time, the independent variable.

The central goal in this section is to provide an algorithm, so that by differentiating equations and variables, appending them to $\boldsymbol{e}$ and $\boldsymbol{v}$, and updating $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{G}, \boldsymbol{G_a}$, a *complete assignment* of the *structurally assignable* highest derivative variables is provided for all highest derivative equations, that is, every highest derivative equation is uniquely assigned to a structurally assignable highest derivative variable:

$$e_i(v_j, t);\ \frac{\partial e_i}{\partial v_k} \text{ is structurally regular for } (B_i = -1,\ A_k = -1, G_{a,i,v_k} = \text{true}) \tag{2}$$

The result of the assignment is reported in vector ***assign***, such that $assign_k = i$, where i is the index of a highest derivative equation, that is $B_i = -1$, and v_k is a structurally assignable highest derivative variable, i.e., $A_k = -1$, and $G_{a,i,v_k} =$ true ($,v_k$ is the index of variable k in $G_{a,i}$). If all variables and equations are scalarized, so $G_{a,i,v_k} = true$ for all the scalarized variables and equations, then there are several (standard) algorithms to derive (2) from DAE (1), see for example (Otter, Elmqvist 2017, section 3.1).

The notation above is clarified with the cascaded first order model in vector notation from section 2, where the equation u = 1 is additionally added:

Table 1: Notation for vectorized CascadedFirstOrder model (T means true, F means False).

e_0	u = 1
e_1	tau*der(x) = cat(1, {u}, x[1:N-1]) - x
$\boldsymbol{B}$	[-1,-1] // no equation appears differentiated
v_0	u
v_1	x
v_2	der(x)
$\boldsymbol{A}$	[-1,2,-1] // the derivative of x is der(x)
G_0	[0] // e_0 is a function of u
G_1	[0,1,2] // e_1 is a function of u, x, der(x)
$G_{a,0}$	[T] // e_0 is structurally assignable for u
$G_{a,1}$	[F, F, T] // e_1 is struct. assignable for der(x)

Using the algorithm given in **Appendix A.1**, a complete assignment (2) for the problem formulation in **Table 1** is derived, yielding the following result:

assign = [0, -1, 1] // v_0 assigned to e_0, v_2 to e_1

This assignment holds for any valid dimension N (for N=1, the model is not valid due to x[1:N-1]). With the algorithms sketched in **Appendix A.4**, the following sorted and solved assignment statements can be derived:

```
u := 1
der(x) := (cat(1, {u}, x[1:N-1]) – x)/tau
```

and corresponding code generated. The translation time is independent of dimension N and N can be changed after translation!

3.3 Differentiation of Equations

In order to derive (2) from DAE (1), equations might need to be differentiated. For scalarized variables and equations several (standard) algorithms are available, see for example (Otter, Elmqvist 2017, section 3.1). These algorithms need to be extended to take the new structurally assignable property $\boldsymbol{G}_a$ into account. In **Appendix A.1**, it is shown how to extend the algorithm of Pantelides (1988) in this respect.

Consider the following simple yet non-trivial example:

Model 1: Model that needs to be differentiated.

```
Model ModelThatNeedsToBeDifferentiated
  Real n[3]; L, e[3]; r[3];
  Real s(start=1, fixed=true);
equation
    L = sqrt(n*n);
    e = 2*n/L;
    r = e*s;
    e*der(r) = der(s) - s;
    n = {1, 2, sin(time)};
end ModelThatNeedsToBeDifferentiated;
```

This model has the following structurally assignable properties:

equations	*structurally assignable*
L = sqrt(n*n);	L
e = 2*n/L;	e, n
r = e*s;	r, s
e*der(r) = der(s) - s;	der(s)
n = {1, 2, sin(time)};	n

With the enhanced Pantelides algorithm of **Appendix A.1** the following complete assignment of differentiated equations can be derived:

assigned	*highest derivative equations*
der(L)	der(L) = 0.5*(n*n)^-0.5*(der(n)*n + n* der(n))
der(e)	der(e) = (2* der(n)*L - 2*n* der(L))/L^2
der(r)	der(r) = der(e)*s + e* der(s)
der(s)	e*der(r) = der(s) - s
der(n)	n = {0,0, cos(time)}

In section **4**, several application examples are given that demonstrate the usefulness of the new approach.

3.4 Improved Error Diagnostics

If a Modelica model is erroneous, in many cases tools provide error messages that make it difficult to figure out the reason for the problem. For example, Modelica tools could provide an error message of the following kind which is quite useless: *The problem is structurally singular. It has 2046 scalar unknowns and 2045 scalar equations*. The error messages have improved over time, but they are often still not helpful enough. In **Appendix A.3**, a new, simple algorithm is sketched to significantly improve error diagnostics. Take for example the following model which contains several errors:

Model 2. Model having several errors.

```
model SeveralErrors
  Real v1,v2,v3,v4,v5,v6,v7,v8,v9,v10,v11;
equation
  der(v1) = -v1;
  der(v2) = -v3;
  v3 = -1;
  v3 = 3 + v2;

  der(v4) = -v4 + v1;
  der(v5) = -v5 - v4;
  v5 = 2;

  der(v6) = -v6 + v7;
  der(v7) = -v8 + v9;
  0 = v7 + v8 + v9;
end SeveralErrors;
```

Modiator provides the following error message.

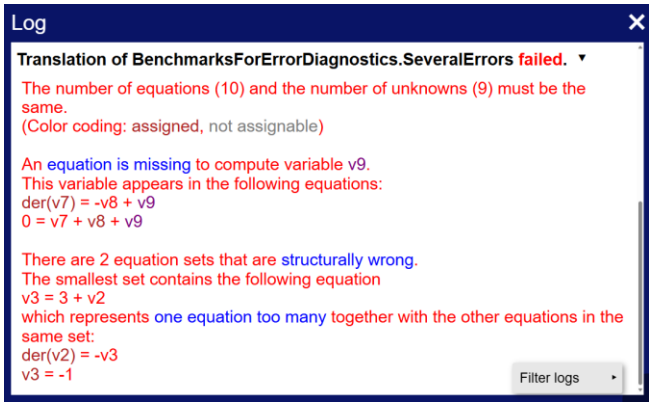


Figure 1. Modiator error message for **Model 2**.

The first message states, that one equation is missing to compute variable v9 and the equations are listed in which this variable appears. All of these equations are already assigned, so it is impossible to assign for v9. Most likely, just an equation for v9 is missing and should be added. Modiator lists these types of error messages first, because little output is expected, since a variable typically only appears in a few equations.

The second message states that two equation sets are wrong. Since many equations might be involved, more information is only given for the set with the smallest number of equations. For this set, first the equation is listed that makes the following set of equations overdetermined. This equation cannot get an assignment because there is one equation too many in this set. Afterwards, all the other (already assigned) equations are listed together with their assignment.

Note, the output of the error messages is not unique and a different initial ordering of the equations can display a different message (e.g. instead of v9, variable v8 could be listed). However, the important point is that the error message either points to *one* problematic *variable* or to *one* problematic *equation*, to help the user identifying the issue.

DOI
10.3384/ecp218189

4 Application Examples

4.1 Heat Exchanger

The ScalableTestSuite and LargeTestSuite[6] of (Casella, 2015) contain cocurrent heat exchanger models with different numbers of segments from 10 to 81920. The compilation time in Modiator is essentially independent of n: about 11 milliseconds. The simulation results for n=10 are shown in Figure 1, and n can be changed after compilation.

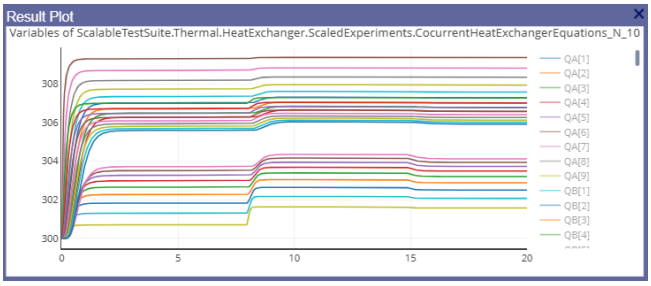


Figure 2. Simulation results for the cocurrent heat exchanger model with 10 segments from the ScalableTestSuite.

A small modification of the model was needed. The equations:

```
TA[1] = if time < 8 then 300 else 301;
TB[1] = 310;
for i in 2:N loop
  TA[i] = TAtilde[i - 1];
  TB[i] = TBtilde[i - 1];
end for;
```

were replaced by:

```
TA = cat(1, {if time < 8 then 300 else 301}, TAtilde)
TB = cat(1, {310}, TBtilde)
```

in order to comply with **Restriction 1**.

4.2 Shock Waves

Capturing shock waves typically needs a large number of grid points, i.e. benefits substantially by using the described array preserving technique. Consider the water hammer model, **Model 3**, using only array equations. The valve at the end of the pipe closes at time=0.01 inducing a shock wave. The number of grid points n can be changed after compilation.

Model 3: Modelica Water Hammer model capturing shock waves from Clément Coïc.

```
model WaterHammer
  parameter Integer n=10;
  parameter Real L=100 "Length of the pipe (m)";
  parameter Real A=3.1415*D^2/4 "Cross-sect. area of pipe (m^2)";
  parameter Real rho=1000 "Density of water (kg/m^3)";
  parameter Real B=21e8 "Bulk modulus of water (Pa)";
  final parameter Real c=sqrt(B/rho) "Speed of sound in water (m/s)";
  parameter Real tClose=0.01 "Time at which the valve closes (s)";
  parameter Real Q0=0.0002 "Initial flow rate (m^3/s)";
  final parameter Real dx=L / n "Length of each pipe segment (m)";
  final parameter Real dt=dx / c "Time step for stability (s)";
  parameter Real f=0.05 "Darcy-Weisbach friction factor (dim.less)";
  parameter Real D=0.05 "Pipe diameter (m)";
  Real p[n+1](start=fill(3e5, n+1), fixed=true) "Press. at each node (Pa)";
  Real Q[n] "Flow rate in each segment (m^3/s)";
  Real V[n](start=fill(0.1, n), fixed=true) "Speed of flow (m/s)";
  Real Qend "Flow rate at the end of the pipe (m^3/s)";
  Real QPlusEnds[n+2];
  Real valveOpen "1 if valve is open, 0 if closed";
equation
  Q = V*A;
  valveOpen = if time < tClose then 1 else 0;
  Qend = Q0 * valveOpen;
  der(V) = -(p[2:n+1] - p[1:n]) / (rho * dx) - (f / (2 * D)) .* V .* abs(V);
  QPlusEnds = cat(1, {Q[1]}, Q, {Qend});
  der(p) = B*(-(QPlusEnds[2:n+2] - QPlusEnds[1:n+1]) / (A * dx));
 annotation(experiment(StopTime=0.3));
end WaterHammer;
```

Figure 3 plots the pressures against time.

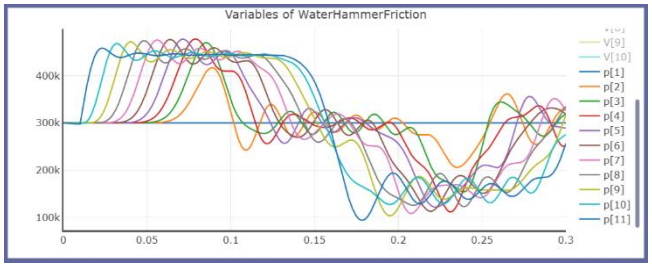


Figure 3. Pressures of WaterHammer model.

Such time plots do not convey the wave behavior very well. It is better to show the pressure profile over the pipe at different times, **Figure 4**. Spline interpolation is used to smooth the curve over the 51 spatial points (n=50). The pressure wave traverses from right to left. After about 72 milliseconds, the wave starts returning (green curve):

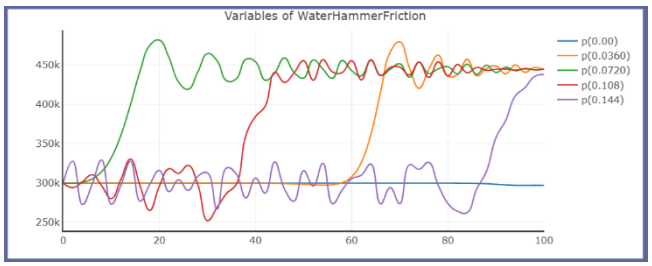


Figure 4. Spatial plot of pressures of WaterHammer model.

4.3 Transmission Line

Consider the transmission line element of **Figure 5**. It is a regular structure of connected electrical components (Resistor, Inductor, Conductor and Capacitor). Such a device can be modeled in Modelica using arrays of components and a for-loop for connections. A voltage source is connected to the first segment and a resistance load to the last segment:

[6] https://github.com/casella/ScalableTestSuite

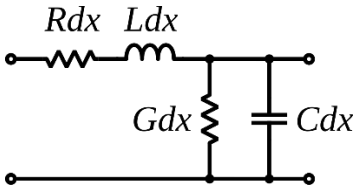


Figure 5. Image of transmission line element from Wikimedia[7]

Model 4: TransmissionLine
modeled with arrays of components.

```
model TransmissionLine
  import Modelica.Electrical.Analog.Basic;
  model TransmissionLineSegment
    parameter Real dx = 1 "Length of the segment in meters";
    parameter Real resistancePerMeter = 0.01;
    parameter Real inductancePerMeter = 0.001;
    parameter Real capacitancePerMeter = 1e-9;
    parameter Real conductancePerMeter = 1e-6;

    Modelica.Electrical.Analog.Interfaces.Pin p;
    Modelica.Electrical.Analog.Interfaces.Pin n;

    Basic.Resistor r(R=resistancePerMeter*dx);
    Basic.Inductor l(L=inductancePerMeter*dx);
    Basic.Capacitor c(C=capacitancePerMeter*dx);
    Basic.Conductor g(G=conductancePerMeter*dx);
    Basic.Ground ground;
  equation
    connect(p, r.p);
    connect(l.n, g.p);
    connect(g.p, c.p);
    connect(c.p, n);
    connect(r.n, l.p);
    connect(g.n, c.n);
    connect(c.n, ground.p);
  end TransmissionLineSegment;

  parameter Integer N = 10 "Number of segments";
  parameter Real length = 1000 "Length of in meters";

  TransmissionLineSegment segments[N](dx=fill(length/N, N));

  Basic.Ground ground;
  Basic.Resistor load(R=100);
  Modelica.Electrical.Analog.Sources.ConstantVoltage
      source(V=10);
equation
  // Connect the segments in series
  for i in 1:N-1 loop
    connect(segments[i].n, segments[i+1].p);
  end for;

  // Connect the first segment to the source
  connect(source.p, segments[1].p);
  connect(source.n, ground.p);

  // Connect the last segment to the load
  connect(segments[N].n, load.p);
  connect(load.n, ground.p);
end TransmissionLine;
```

In order to avoid recompilation of this model when N is changed, the connect statements are manually expanded. Since segments[N].n is connected to a capacitor having a state, the voltages are copied to the right (n to p) with the first element coming from source.p via a node nSource (since source.p needs to be connected, otherwise source.p.i would be zero). The currents are copied from p to n with the last current coming from load.p.i via the node nLoad.

```
segments.p.v = cat(1, {nSource.v}, segments.n.v[1:N-1]);
segments.n.i = -cat(1, segments.p.i[2:10], {nLoad.i});

nSource.i = segments.p.i[1];
nLoad.v = segments.n.v[N];

connect(source.p, nSource);
connect(source.n, ground.p);

connect(load.p, nLoad);
connect(load.n, ground.p);
```

The compilation time is about 50 milliseconds independent of n. The resulting voltages over the capacitors are shown in **Figure 6**.

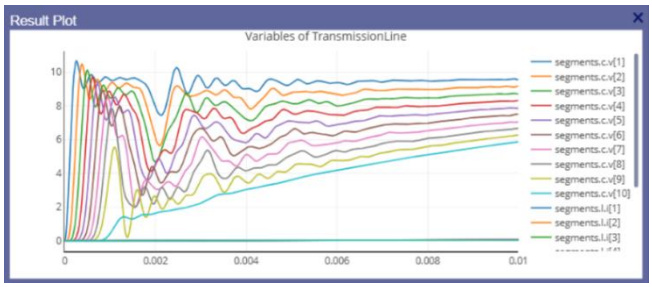


Figure 6. Capacitor voltages.

It should be noted that enabling changes to the discretization after compilation relied on certain specific properties of the model. The capacitor to the right of the TransmissionLineSegment enabled simultaneous copying of all voltages to the right in one array equation. Correspondingly due to the inductor to the left, all currents could be copied simultaneously in one array equation to the left.

4.4 Adaptive Grid

An important advantage of not recompiling when array dimensions change is that a model can refine itself, as shown in the example of this section.

An insulated rod model is shown below and a plot of a simulation in **Figure 7**. It changes the number of segments at time=2000 and time=4000. The changes could also be triggered by detecting some transient behavior.

[7] https://commons.wikimedia.org/wiki/File:Transmission_line_element.svg

DOI
10.3384/ecp218189

Model 5: Insulated rod model with adaptive grid.

```
model InsulatedRod_gradient "Insulated rod with gradient"
  import SI = Modelica.Units.SI;
  parameter SI.Length L=1 "lenght of rod";
  parameter SI.Area A=0.0004 "area of rod";
  parameter SI.Density rho=7500 "density of rod material";
  parameter SI.ThermalConductivity lambda=74
                              "thermal conductivity of material";
  parameter SI.SpecificHeatCapacity c=450
                              "specifc heat capacity";
  parameter SI.Temperature T0=293 "initial temperature";
  parameter Integer nT(min=2)=4 "number of inner nodes";
  SI.Temperature T[nT](start={300, 297.5, 295, 292.5},
              each fixed=true) "temperatures at inner nodes";
  SI.HeatFlowRate Q_flow[nT+1];
  Real dx;
  Real k1;
  Real k2;
  Real k3;
  Real port_a_Q_flow;
  Real port_b_Q_flow;
  SI.Temperature port_a_T;
  SI.Temperature port_b_T;
equation
  dx = L/nT;
  k1 = lambda*A/dx;
  k2 = rho*c*A*dx;
  k3 = 2*k1;

  // Connection equations
  port_a_T = if time<2000 then 300 else 320;
  port_b_Q_flow = 0;

  // Acausal part (assignment depends on connection)
  port_a_Q_flow = k3*(port_a_T - T[1]);
  port_b_Q_flow = k3*(T[nT] - port_b_T);

  // Causal part (assignment does not depend on connection)
  Q_flow = cat(1, {port_a_Q_flow}, k1*(T[1:nT-1] - T[2:nT]),
                  {port_b_Q_flow});
  der(T) = (Q_flow[1:nT] - Q_flow[2:nT+1])/k2;
  when time > 2000 then
    ast.setParameter("nT", 8);
    ast.setStart("T", interpolate(8, T));
  end when;
  when time > 4000 then
    ast.setParameter("nT", 6);
    ast.setStart("T", interpolate(6, T));
  end when;
  annotation(experiment(StopTime=5000));
end InsulatedRod_gradient;
```

A new function ast.setParameter is used to change the grid resolution. Since the state vector changes, new start values for the states are given using ast.setStart. A function *interpolate* is interpolating the temperatures from 4 points to 8 and back to 6. This function considers how the grid points are distributed along the staggered grid. This explains why the curves have discontinuities since T[i] refers to different positions along the rod when nT changes.

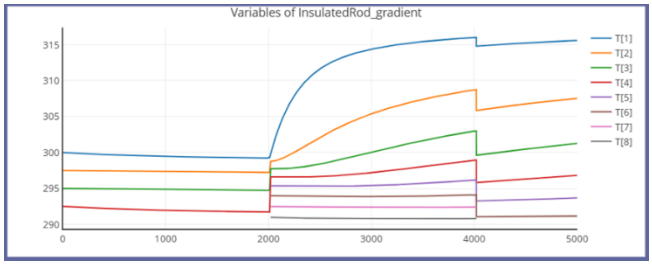


Figure 7. Simulation results of the temperature nodes along an insulated rod. At time=2000 and time=4000 the number of segments is changing from 4 to 8 to 6.

5 Reformulation of Array Equations

In this section, valid Modelica models are presented that are rejected by the algorithm of **Appendix A.1**. It is shown how to reformulate the equations in order that code can be generated.

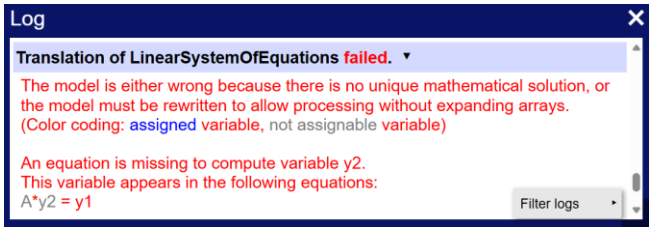
5.1 Linear Equation Systems

Consider the following valid Modelica model that contains a linear system of equations

Model 6: Model with a linear system of equations.

```
model ModelWithLinearSystemOfEquations
  parameter Real n[2] = {1,2};
  parameter Real A[2,2]= [1,2;3,4];
 Real y1[2];
 Real y2[2];
equation
  y1 = n*time;
  A*y2 = y1;
end ModelWithLinearSystemOfEquations;
```

Modiator prints the following error message:



The listed equation must be reformulated, since (a) an equation is missing to compute variable y2, (b) variable y2 appears only in this equation and no variable is assigned to this equation. Note, that y2 is not structurally assignable according to **Theorem 1**. The remedy is to explicitly solve the linear equation system in the model:

```
equation
  y1 = n*time;
  y2 = Modelica.Math.Matrices.solve(A,y1);
```

5.2 Models That Must Be Scalarized

There are various kinds of Modelica models that can only be processed if array variables and equations are scalarized in the generated code. A simple example is shown in the following model:

Model 7: Model where array variables and equations need to be scalarized.

```
model ModelThatRequiresScalarization
  parameter Real n[:]={1,2,3};
  Real s(start=1, fixed=true);
  Real r[3];
equation
    r = n*s;
    der(r)*n = -s;
end ModelThatRequiresScalarization;
```

Modiator prints the following error message:

Translation of ModelThatRequiresScalarization failed.

Equation differentiated 10 times which is too much (most likely processing would be successful if arrays would be expanded, which is not done):
der(r)*n = -s

So, the error message states that the listed equation is differentiated too often and that the likely reason is that array variables in this equation must be scalarized, which is not done in Modiator. If the array variables and equations are scalarized, the following assignment is possible:

assigned	*highest derivative equations*
der(s)	der(r[1]) = n[1]*der(s)
der(r[2])	der(r[2]) = n[2]*der(s)
der(r[3])	der(r[3]) = n[3]*der(s)
der(r[1])	der(r[1])*n[1]+der(r[2])*n[2]+der(r[3])*n[3]=-s

However, it is not possible to convert the scalarized variables and equations back into array variables and equations. One remedy is to split variable r1 and the first equation in two pieces:

Model 8: Reformulation of **Model 7** in order that array variables and equations can be kept without expansion in the generated code.

```
model ChangedModelThatRequiredScalarization
  parameter Real n1 = 1;
  parameter Real n2[:] = {2,3};
  Real s(start=1, fixed=true);
  Real r1;
  Real r2[size(n2,1)];
equation
    r1 = n1*s;
    r2 = n2*s;
    der(r1)*n1 + der(r2)*n2 = -s;
end ChangedModelThatRequiredScalarization;
```

An assignment is now possible without expanding array variables and equations:

assigned	*highest derivative equations*
der(s)	der(r1) = n1*der(s)
der(r2)	der(r2) = n2*der(s)
der(r1)	der(r1)*n1+der(r2)*n2 = -s

5.3 Mechanical Systems

In the following simple model, a mass is sliding along a given direction in 3D space:

Model 9: 3D sliding mass model that requires scalarization.

```
model SlidingMass3DrequiringScalarization
  parameter Real m=1 "Mass of body";
  parameter Real n[3] = {1,0,1}/sqrt(2) "Sliding direction";
  parameter Real g[3] = {0,0,-9.81} "Gravity acceleration";
  Real r[3] "Position";
  Real v[3] "Velocity";
  Real f[3] "Constraint force of prismatic joint";
  Real s "Generalized coordinate of prismatic joint";
equation
  r = n*s;
  v = der(r);
  m*(der(v)-g) = f;
  0 = n*f;
end SlidingMass3DrequiringScalarization;
```

Modiator prints the following error message:

Translation of SlidingMass3DthatRequiresScalarization failed.

The model cannot be processed without exanding arrays because the following equation with arrays does not have at least one unknown variable that appears linearly in the form scalar*variableSymbol:
0 = n*f

The reason is that the *scalar* equation 0 = n*f contains only *array* variables and therefore none of these variables is structurally assignable.

In (Elmqvist and Otter 2017, section 3), it is shown that array variables and equations of mechanical systems must be at least internally expanded in order that processing is possible. After symbolic processing, it is usually possible to recover the non-expanded forms. Since the algorithm in **Appendix A.1** does not expand array variables and equations, *existing* 3D mechanical models, such as those from the Modelica.Mechanics.MultiBody[8] library, cannot be processed by Modiator. For tree-structured mechanical systems it is easy to fix this issue in the following way:

A tree-structured mechanical system with $\boldsymbol{q}$ the vector of generalized minimal coordinates (= generalized coordinates in the joints) and $\boldsymbol{u}$ the generalized forces (= generalized forces in the joints) is described by the following equation:

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{u}; \quad \boldsymbol{M} = \boldsymbol{M}^T \geq 0 \qquad (3)$$

In object-oriented modeling, where bodies, joints and other objects can be connected together in a nearly arbitrary fashion, it is easy to define a system where the symmetric mass matrix $\boldsymbol{M}$ is positive semi-definite (that is, it is singular) instead of positive definite as postulated by mechanical principles. Examples:

- A non-zero mass and a *zero inertia-matrix* is defined for a body, and the connection structure allows rotations of the body.

[8] https://doc.modelica.org/Modelica%204.0.0/Resources/helpDymola/Modelica_Mechanics_MultiBody.html

DOI 10.3384/ecp218189

- A chain of mechanical objects ends with a joint and not with a body.
- More as 3 revolute joints are connected directly together, without having a body between the joints.

Let's subtract a small term $\varepsilon_m \ddot{\boldsymbol{q}}$ from the generalized forces $\boldsymbol{u}$, where the scalar $\varepsilon_m \geq 0$:

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{u} - \varepsilon_m \ddot{\boldsymbol{q}} \quad (4)$$

or

$$\big(\varepsilon_m \boldsymbol{I} + \boldsymbol{M}(\boldsymbol{q})\big)\ddot{\boldsymbol{q}} + \boldsymbol{h}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{u} \quad (5)$$

where $\boldsymbol{I}$ is the unit-matrix. The new mass matrix is positive definite (that is, regular) if $\varepsilon_m > 0$. This follows from its positive quadratic form:

$$\boldsymbol{x}^T(\varepsilon_m \boldsymbol{I} + \boldsymbol{M})\boldsymbol{x} = \varepsilon_m \boldsymbol{x}^T\boldsymbol{x} + \boldsymbol{x}^T\boldsymbol{M}\boldsymbol{x} > \boldsymbol{0}, \text{if } |\boldsymbol{x}| > \boldsymbol{0} \quad (6)$$

since $\varepsilon_m \boldsymbol{x}^T\boldsymbol{x} > \boldsymbol{0}$ and $\boldsymbol{x}^T\boldsymbol{M}\boldsymbol{x} \geq \boldsymbol{0}$. Therefore, the mentioned issues can no longer occur if $\varepsilon_m > 0$ and a mechanical model in Modelica becomes more reliable against user errors. Note, the original multibody system is unchanged, if $\varepsilon_m = 0$. Processing with unexpanded arrays becomes possible by adding a corresponding term to the joints of a Modelica model, as shown exemplarily by the following modified version of the sliding mass model:

Model 10: Changed **Model 9** in order that assignment is possible without expanding arrays.

```
model SlidingMass3D
  ...
  parameter Real eps_m(min=0) = 0 "Small mass in joint";
equation
  r = n*s;
  v = der(r);
  f = m*(der(v)-g);
  sd = der(s);
  0 = n*f + eps_m*der(sd); // der(sd) structurally assignable
end SlidingMass3D;
```

The last (scalar) equation is now structurally assignable for the scalar der(sd). With the algorithm of **Appendix A.1** the following assignment can be derived:

assigned	*highest derivative equations*
der(der(r))	der(der(r)) = n* der(der(s))
der(v)	der(v) = der(der(r))
f	f = m*(der(v)-g)
der(der(s))	der(sd) = der(der(s))
der(sd)	0 = n*f + eps_m*der(sd)

When sorting these highest derivative equations, an algebraic loop is identified that corresponds to (5). The equations can be, for example, transformed in the following linear equation in one unknown:

```
input: der(sd)  // tearing variable
output: residue
der(der(s)) := der(sd)
der(der(r)) := n*der(der(s))
der(v) = der(der(r))
f := m*(der(v)-g)
residue := n*f + eps_m*der(sd)
```

Inserting all terms in the last equation and setting residue=0, results in one linear equation with der(sd) as unknown:

```
(eps_m + m*(n*n))*der(sd) = m*(n*g)
```

Note, if arrays are no longer expanded for mechanical systems, the number of modes of *analytically described elastic bodies*, such as *beams* or *plates*, can be changed after translation!

6 Conclusions

The Modelica Web-App Modiator (Elmqvist and Otter, 2025) does not expand arrays of equations, for loops or arrays of components in order to speed up translation and avoid recompilation when array changes are made. The needed restrictions have been described, translation techniques have been outlined, and several presented model examples show the benefits of this new method. As a result, the Modiator Web-App has compilation times independent of array sizes (O(1) instead of O(n) or worse) and arrays can be resized after compilation.

Special emphasis has been put on enabling index reduction without scalarization for models described by array equations. The developed extensions of the Pantelides algorithm are utilized in Modiator and allow index reduction directly on array equations without expanding arrays (also not internally). The algorithm is described and the proofs of its properties are given. The needed changes to the symbolic processing of a Modelica tool are modest and reasonable diagnostics can be given if index reduction and/or assignment fails by pointing to the equations and arrays that need to be changed, in order that processing is possible without expanding array equations and arrays.

The enhanced algorithms presented in the **Appendix**, together with tests for some of the models in this paper, are also provided in Javascript format in file *assign-highest-derivatives-test.html* in the accompanying zip-file. *Dragging this file into a web browser will execute the tests* and display the test results in the web browser window.

Acknowledgements

The authors want to thank Clément Coïc for providing the water hammer model in **Section 4.2**.

This work was partially funded for the first author by the German Federal Ministry of Education and Research (BMBF, grant number 01IS23062C) within the European ITEA4 research project OpenSCALING[9].

[9] https://itea4.org/project/openscaling.html

References

Abdelhak, Karim, Francesco Casella and Bernhard Bachmann (2023). “Pseudo Array Causalization”. In: *Proceedings of the 15th International Modelica Conference*, DOI: 10.3384/ecp204177.

Abdelhak, Karim, and Bernhard Bachmann (2025). “Constant Time Causalization using Resizeable Arrays”. In: *Proceedings of the 16th International Modelica Conference*, Lucerne, Switzerland.

Casella, Francesco (2015). “Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives”. In: Proceedings of the *11th International Modelica Conference*, DOI: 10.3384/ecp15118459.

Elmqvist, Hilding and Martin Otter (2025): “Modiator, a Web App for Modelica Simulation“. In: *Proceedings of* the *16th International Modelica & FMI Conference*, Lucerne.

Fritzson, Peter et al. (2020). “The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development”. In: *Modeling, Identification and Control*. 41(4), pp. 241-285. DOI 10.4173/mic.2020.4.1

Marzorati, Denise, Joaquin Fernandez and Ernesto Kofmann (2024): “Efficient Matching in Large DAE Models”. In: *ACM Transactions on Mathematical Software*, Volume 50, Issue 4, Article No. 18, pp. 1-25. DOI: 10.1145/3674831

Mattsson, Sven-Erik and Gustaf Söderlind (1993). “Index Reduction in Differential-Algebraic Equations using Dummy Derivatives”. In: *SIAM Journal of Scientific Computing*. 14(3), pp. 677-692.

Modelica Association (2023). *Modelica – A Unified Object-Oriented Language for Systems Modeling, Language Specification, Version 3.6*. URL: https://specification.modelica.org/maint/3.6/MLS.html.

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

Pantelides, Constantinos C. (1988). “The Consistent Initialization of Differential-Algebraic Systems”. In: *SIAM Journal on Scientific and Statistical Computing* 9.2, pp. 213–231. DOI: 10.1137/0909014

Pop, Adrian et al. (2019). “A New OpenModelica Compiler High Performance Frontend”. In: Proceedings of the *13th International Modelica Conference*, pp. 689–698. DOI: 10.3384/ecp19157689

Tarjan, Robert (1972): "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1(2), pp. 146-160. DOI: 10.1137/0201010

Zimmermann, Pablo, Joaquín Fernández, and Ernesto Kofman (2020). “Set-Based Graph Methods for Fast Equation Sorting in Large DAE Systems”. In: *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. EOOLT’2019*. Berlin, Germany, Association for Computing Machinery. DOI: 10.1145/3365984.3365991

Appendix A

In this appendix, the enhancements of standard algorithms are sketched to find a complete assignment (2) for a DAE (1) without expanding array variables and equations.

A.1 Assign Highest Derivatives

Starting point is the algorithm of Pantelides (1988). The modified algorithm is provided in Javascript format (which is nearly the same as C/C++ format here) in the following listing. The yellow parts are deviations from (Pantelides, 1988) and are discussed below:

Listing 1: Function assignHighestDerivatives in Javascript format to determine highest derivative equations that have a complete assignment with respect to *structurally assignable* highest derivative variables. Yellow parts are deviations from (Pantelides, 1988).

```
a1  function assignEquation(i,
                            G,Ga,assign,vColor,eColor,vInspect){
      const Gi = G[i]
      const Gai = Ga[i]
      eColor[i] = true
      for (let j=0; j<Gi.length; j++) {
        const v = Gi[j]
        if(vInspect[v] === -1 && assign[v] === -1 && Gai[j]){
          assign[v] = i;
          return true}}
      for (let j=0; j<Gi.length; j++) {
        const v = Gi[j]
        if (vInspect[v] === -1 && assign[v] > -1 && !vColor[v]
                                                 && Gai[j])  {
          vColor[v] = true
          if (assignEquation(assign[v],
                             G,Ga,assign,vColor,eColor,vInspect)) {
            assign[v] = i;
            return true}}
      }
      return false
    }
a2  export function assignHighestDerivatives(G,Ga,A) {
      let nA  = A.length
      let nB  = G.length
      const assign = Array(nA).fill(-1)
      const B      = Array(nB).fill(-1)
      const eColor = Array(nB)
      const vColor = Array(nA)
a3    function colorEquation(i) {
        eColor[i] = true
        for (const v of G[i]) {
          if (A[v] === -1 && assign[v] > -1 && !vColor[v]
                          && B[assign[v]] === -1) {
            vColor[v] = true
            colorEquation(assign[v])}
        } // end for
      }
a4    const nB_initial = nB
      for (let k = 0; k < nB_initial; k++) {
        let i = k
        while (true) {
          eColor.fill(false)
```

```
    vColor.fill(false)
    if (assignEquation(i,G,Ga,assign,vColor,eColor,A)) break
a5  for (let j=0; j<nB; j++) {
      if (eColor[j] && Ga[j].some(val=>val===false)){
        colorEquation(j)}}}
a6  for (let j=0; j<nB; j++) {
      if (eColor[j]) {
        nB++; B.push(-1); B[j]=nB-1;
        for (const v of G[j]) {
          if (A[v] === -1) {
            nA++; A.push(-1); A[v]=nA-1; assign.push(-1)}}
        G.push( Gj.map(m => A[m]) )
        Ga.push( Ga[j] )}}
    for (let j = 0; j < nA; j++) {
      if (vColor[j]) assign[A[j]] = B[assign[j]]}
a7  i = B[i]
    eColor.length = nB
    vColor.length = nA
  } // end while
a8 } // end for
   return [assign,B]}
```

Utility function assignEquation(i,...), see code block (a1), is basically function AUGMENTPATH of (Pantelides, 1988). The function inspects all variables v of equation i that are *structurally assignable* and have vInspect[v] = -1. It returns true if a v is assigned to equation i and otherwise it returns false. If false is returned, the following property holds:

Theorem 2: *If* assignEquation *returns with* false, *the equations with* eColor[k] = true *are minimally structurally singular with respect to the variables with* vColor[k] = true *(these variables are structurally assignable with respect to the assigned equations).*

Proof: Follows from the proof of Lemma 3.3 of (Pantelides, 1988), with the only difference that a subset of the variables is inspected (variables that are structurally assignable and have vInspect[v] = -1, i.e. are highest derivative variables since A is passed to the function as vInspect). ■

This property means that the colored equations are the smallest subset that need to be differentiated. This subset has more equations than variables and therefore differentiating this subset will introduce more new equations as it will introduce more new variables.

The core function assignHighestDerivatives, see code blocks (a2-a9), inspects all equations in sequence. For every equation, function assignEquation is called with vInspect = A, that is only highest derivative variables are inspected, see last statement in code block (a4). If the function returns true, the next equation is inspected. Otherwise, a while loop iterates over the function call, until an assignment is found. If assignEquation returns false, the colored set of equations needs to be differentiated.

Colored equations that have at least one highest derivative variable that is *not* structurally assignable are inspected by calling the function colorEquation on them, see code blocks (a5,a3). This function colors additional equations that need to be differentiated. Assume for example, that equation j colored by assignEquation has a highest derivative variable v that is not structurally assignable with respect to equation j, is already assigned, is not colored and the equation to which v is assigned is a highest derivative equation. If equation j is differentiated, then v is also differentiated, but without differentiating the highest derivative equation assign[v], because this equation was not colored by assignEquation. The construction of code blocks (a5,a3) ensures, that these assigned equations are also colored and are thus differentiated. Note, this additional equation coloring appears, for example, in **Model 1** from section 3.3.

In code block (a6), the data structures G,Ga,A,B,assign are appended with new entries that include the incidences of the equations and the highest derivative variables that will appear when the equations are differentiated. This part is nearly identical to code block (3b-5) of ALGORITHM 4.1 of (Pantelides, 1988). The essential difference is that in (3b-5) of (Pantelides, 1988) first all colored variables are appended to A before G,B are updated. However, this is not possible here, because a colored equation j may have a variable v that is not structurally assignable and that is not assigned. When differentiating equation j, v is differentiated as well, but the differentiated v may not yet exist because it was not colored. For this reason, A is updated when the incidence of the differentiated equation j is appended to G. The remaining part of the code is identical to ALGORITHM 4.1 of (Pantelides, 1988). Function assignHighestDerivatives returns the newly constructed arrays assign,B via the return statement, and the updated arrays G,Ga,A via the argument list.

A.2 Assign Extended System

Since assignHighestDerivatives has a while loop, the question arises whether this loop will terminate for all inspected equations. In (Pantelides, 1988) an *extended* system $\boldsymbol{EG} = \begin{bmatrix} \boldsymbol{G} \\ \boldsymbol{G}_h \end{bmatrix}$ is analyzed consisting of $\boldsymbol{G}$ and of the incidence matrix $\boldsymbol{G}_h$ of additional equations $0 = h_i(v_i, \dot{v}_i)$ for every variable v_i appearing differentiated. A corresponding matrix $\boldsymbol{EG_a} = \begin{bmatrix} \boldsymbol{G}_a \\ \boldsymbol{G}_{h,a} \end{bmatrix}$ defines whether the elements of $\boldsymbol{EG}$ are structurally assignable or not. Hereby, all elements of $\boldsymbol{G}_{h,a}$ representing the new equations h_i are true.

Theorem 3: *If* $\boldsymbol{G}_a$ *has only true values then the* while *loop in function* assignHighestDerivatives *terminates for all inspected equations after a finite number of iterations if and only if the extended system* $\boldsymbol{EG}$ *is structurally nonsingular. In such a case, the highest derivative equations of* $\boldsymbol{G}$ *have a complete assignment with respect to the highest derivative variables. If* $\boldsymbol{EG}$ *is structurally singular, then the underlying DAE (1) has no solution since it is impossible to find a set of consistent initial conditions.*

Proof: This is Theorem 4.2 of (Pantelides, 1988) formulated with the notation of this paper. ■

Unfortunately, this strong property does no longer hold, if one or more elements of $\boldsymbol{G}_a$ are false, that is, not all variables are structurally assignable. This is demonstrated with the simple example of **Model 7** of section 3.3. A complete assignment of the extended system $\boldsymbol{EG}$ with respect to all structurally assignable variables is possible:

assigned	*equations of extended system*
r	r = n*s
s	der(r)*n = -s
der(r)	0 = h(r, der(r))

Analyzing the equations of **Model 7** with function assignHighestDerivatives results in the following sequence of assignments: Function assignEquation returns false for r = n*s because r is not a highest derivative variable and s is not structurally assignable. Therefore, this equation is differentiated leading to the new highest derivative equations:

assigned	*highest derivative equations*
der(r)	der(r) = n*der(s)
---	der(r)*n = -s

Function assignEquation returns false for der(r)*n = -s because der(r) is not structurally assignable and s is no highest derivative variable. Differentiating this equation and additionally the first equation due to code block (a5), results in the following new highest derivative equations:

assigned	*highest derivative equations*
der(der(r))	der(der(r)) = n*der(der(s))
---	der(der(r))*n = - der(s)

A similar situation as in the previous step occurs: The second equation needs to be differentiated, because der(der(r)) is not structurally assignable and der(s) is no highest derivative variables. Also, the first equation is differentiated due to code block (a5). As a result, an infinite number of differentiations occurs.

If instead, variables r and n are expanded, the highest derivative equations are identified and assigned as shown in section 3.3. As can be seen, it is not possible for this model to have an assignment of structurally assignable variables without expanding array variables and array equations. ■

The consequence is that the number of iterations of the while loop in function assignHighestDerivatives must be explicitly limited and when this limit is reached, the function either returns with an error or the model is processed again with all arrays expanded. Assume that the following assumption holds (which is checked before function assignHighestDerivatives is called):

Assumption 1: *The number of equations e_i of the underlying DAE (1) is identical to the number of highest derivative variables, so variables v_j that have $A_j = -1$.*

then the following theorem states the property of the return result of function *assignHighestDerivative*:

Theorem 4: *The highest derivative equations of $\boldsymbol{G}$ have a complete assignment with respect to their structurally assignable highest derivative variables, if Assumption 1 holds and the while-loop in function assignHighestDerivatives terminates after a finite number of iterations for all inspected equations.*

Proof: If the function returns successfully, then all underlying equations have been inspected once, due to the for-loop of code block (a4). Since every equation gets an assignment for a structurally assignable highest derivative variable that is not yet assigned, due to code block (a1), all equations are assigned for highest derivative variables that are structurally assignable. This assignment does not influence already assigned equations, because otherwise these equations would have been colored and then differentiated together with the not-yet assigned equation and their assigned variables would have been differentiated correspondingly. Due to **Assumption 1**, all highest derivative variables or derivatives of them are assigned because the same number of equations get an assignment. ■

If the *while*-loop does not terminate, then either the DAE (1) has no unique solution, or a complete assignment with respect to *expanded* arrays is possible.

In order that meaningful diagnostics can be given, the following theorem is utilized:

Theorem 5: *If the extended System $\boldsymbol{EG}$ is structurally singular with respect to the structurally assignable variables, the while-loop does not terminate for at least one of the inspected equations.*

Proof: Follows from the proof of Theorem 4.2 in (Pantelides, 1988), Part B until "the algorithm will keep differentiating until infinitum". ■

Note, if the extended System **EG** is structurally singular with respect to the structurally assignable variables, it is still possible that the DAE is solvable if the array variables and array equations are expanded. This can be easily seen from **Model 6** in section 5.1. The extended System **EG**

assigned	*equations of extended system*
y1	y1 = n*time
--	A*y2 = y1

is *structurally singular* with respect to the structurally assignable variables because the first equation can be assigned to y1 and the second equation can be also assigned to y1 but not to y2, because y2 is not structurally assignable according to **Theorem 1**. Therefore, the while loop does not terminate, but this can be detected beforehand. If arrays are expanded, the elements of the second equation can be assigned to the elements of y2 and the extended system is *structurally regular*. ■

DOI 10.3384/ecp218189

Listing 2 shows the code for function assignExtendedSystem that builds and checks the extended system for structurally assignable variables.

Listing 2: Function assignExtendedSystem in Javascript format to determine a complete assignment of the extended system with respect to the *structurally assignable* variables. Yellow parts are deviations from (Pantelides, 1988).

```
b1  export function assignExtendedSystem(G,Ga,A) {
      // Build extended system
      const EG  = [...G]
      const EGa = [...Ga]
      const nA  = A.length
      for (let i=0; i<nA; i++) {
        const Ai = A[i];
        if (A[i] > -1) {
          EG.push([i, A[i]]); // h-equation
          EGa.push([true, true])}
      }
b2    // Assignment of extended system
      const assign   = Array(nA).fill(-1)
      const vColor   = Array(nA)
      const eColor   = Array(EG.length)
      const vInspect = Array(nA).fill(-1)  // inspect all variab.
      const unassignedEquations = []
      for (let i=0; i<nA; i++){
        eColor.fill(false)
        vColor.fill(false)
        if ( !assignEquation(i,EG,EGa,
            assign,vColor,eColor,vInspect) && i<G.length){
          unassignedEquations.push(i)}
      }
b3    // Collect unassigned variables
      const unassignedVariables = []
      for (let j=0; j<assign.length; j++) {
        if (assign[j] === -1) unassignedVariables.push(j)
      }
      return [unassignedEquations, unassignedVariables]
    }
```

In code block (b1) the extended system for G and Ga is constructed. In code block (b2), function assignEquation is called for all equations of the extended system and for *all* of its variables (due to vInspect = Array(nA).fill(-1)). If assignEquation returns false, and the inspected equation is no h-equation, it is pushed on stack unassignedEquations. (beause h-equations should not be reported to the user and if a h-equation is not assigned, there is also an unassigned variable). In code block (b3) the unassigned variables are collected in stack unassignedVariables. Both stacks are returned. If one of them is not empty, the complete assignment of the extended system failed and the unassigned equations and unassigned variables are returned and reported to the user.

The practical application of the theorems requires additionally the following small extension:

Function assignHighestDerivatives(G, Ga, A, maxDer =-1) gets an additional optional argument that defines the maximum number of allowed differentiations for an equation (if maxDer =-1, the number of differentiations is not limited) . If this limit is reached, the function terminates. It returns [assign,B,eqDiffTooMuch]. If the maximum number of differentiations is reached, eqDiffTooMuch is the equation of the underlying DAE that was differentiated too often. This equation can be reported in the error message.

A.3 Improved Error Diagnostics

The information for the error messages presented in section 3.4, is deduced from assignExtendedSystem(…) of **Listing 2** by extending this function a bit:

- The function already collects all unassigned variables. For every unassigned variable, all equations are inspected where this variable appears and then these equations are printed together with their assignments.
- The function also collects all unassigned equations which is not very useful. Instead, when no assignment can be found, *all colored equations* are collected together because they constitute the set of equations that, together with the unassigned equation, creates an overdetermined system. This combined set is stored. If more overdetermined sets of equations appear, only the one with the smallest number of equations is kept.

A.4 Further Symbolic Processing

A tool could call the presented functions in the following order:

1. Call assignExtendedSystem. If the function returns with unassigned equations and/or unassigned variables either print an error message showing the unassigned equations/variables ("*Either DAE is not solvable or processing not possible without expanding arrays, which is not supported*") or expand all arrays and equations and perform the standard processing of the tool for expanded equations/arrays.
2. If assignExtendedSystem returns successfully, call assignHighestDerivatives. If the function terminates because the maximum number of differentiations reached, either print an error message showing the equation that was differentiated too often ("*Equation eqDiffTooMuch differentiated until the limit of maxDer differentiations reached. Either DAE is not solvable or processing is not possible without expanding arrays, which is not supported.*") or expand all arrays and equations and perform the standard processing of the tool for expanded equations/arrays.
3. If assignHighestDerivatives returns successfully, a full assignment of the highest derivative equations with respect to the structurally assignable highest derivative variables was found and standard processing continues with only symbol information (so with unexpanded equations/arrays).

After the complete assignment of the highest derivative equations with respect to the structurally assignable highest derivative variables is determined with function

assignHighestDerivatives the standard approach of object-oriented modeling can be used for further processing and code generation by using basically only the symbol information (without expanding arrays):

1. *Analytically differentiating the equations*
   The equations are analytically differentiated based on the information provided in $\boldsymbol{G}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{assign}$.
2. *Equation sorting*
   $\boldsymbol{G}$ and ***assign*** of the highest derivative variables $\boldsymbol{A}$ with $A_k = -1$ define a directed graph where the nodes are the highest derivative equations together with the assigned highest derivative variables. The edges are defined by the rows of the incidence matrix $\boldsymbol{G}$ that correspond to the highest derivative equations. With the algorithm of (Tarjan, 1972) the strong components (algebraic loops) in the directed graph are determined and the equations are sorted.
3. *State selection*
   From the sorted highest derivative equations, the constraint equations are determined by the algorithm of (Mattsson and Söderlind, 1993) based on the highest derivative equation systems determined in step 2. From this information states are statically and/or dynamically selected.
4. *Code generation*
   Code is generated. Arrays with small, fixed dimensions and corresponding array equations with small, fixed dimensions might be expanded. If arrays are kept unexpanded, then the generated code must support array operations where dimension sizes might be only known at run-time and can be changed after translation to object code. Typically, Modelica tools support this already, because code generation for Modelica functions need a similar infrastructure to operate on arrays.

DOI 10.3384/ecp218189