A Framework for Implementation-Independent Simulation Models

Jonas Larsson
Mechanical Engineering Systems
Department of Mechanical Engineering
Linköping University, Sweden
jonla@ikp.liu.se

Mathematical modeling and the simulation of complex physical systems is a central activity in engineering. The models are the result of large investments in time and measurement equipment and therefore need to be used as efficiently as possible. If one and the same model can be run in several simulation environments and in other applications, then the model can be simulated today and tomorrow in the most appropriate environment with respect to user background, the engineering domain at hand, and more. Modern equation-based modeling languages provide this functionality in principle since the models are translated into executable code by separate model compilers. These compilers today generate code for standalone simulation using a fixed interface toward a number of numerical solvers. What is missing is a compiler that can generate customized code for various simulation environments, where the generated code interfaces with other models in the environment through the conventional solution of powerports. Each port holds both input and output variables such as force and velocity. A compiler architecture is proposed in this article. Also, an implementation for a subset of the Modelica language is demonstrated in the case of two widely different simulation environment targets. The model ports are adapted initially to the simulation environment characteristics in terms of the variables part of the ports, and an input-output causality is found for which the model can be solved and that suits the environment. The equation system is optimized, and information such as Jacobians, assignments, and more is generated in the XML format. Finally, code generators specific for each environment are employed that create the final executable code.

Keywords: Simulation, DAE, compiler, Modelica, XML, causality, code generation

1. Introduction

Simulation models are often tied to the software used to define the models and cannot be reused in other environments. This is often due to the case that the models are procedural, written in programming languages from which it is difficult to perform a translation to a suitable format. Equation-based modeling languages improve this situation since model compilers are needed that create the code to be executed during simulation. The equation-based models can then be translated into the programming code still used by many conventional simulation environments in defining the models part of their libraries. If the translation is made automatic enough, the equation-based models can be translated on the fly and be experienced as a seamless part of the conventional simulation environment. The user would only need to see the equation-based models.

SIMULATION, Vol. 82, Issue 9, September 2006 563-579

DOI: 10.1177/0037549706071902

Equation-based models offer more advantages over procedural models. The model content is described by equations and is not hidden by details of a programming language. The models can be translated into efficient code without human intervention or expert knowledge. There also often exist object-oriented mechanisms for dealing with complexity. The main drawback is the complexity of the model compilers, which has so far hindered a more widened support for these languages.

The article proposes an open-ended model compiler architecture for equation-based modeling languages that can be adopted for different simulation environments, *or other applications*, with a minimum of effort and that can translate models at any level desired of the model hierarchy without user intervention. An implementation is demonstrated for two widely different applications. Other applications might be code for embedded systems and script code for Web-based applications where dynamics of physical systems is of interest. For standalone simulations, the architecture gives the possibility to test new types of solvers

since most information needed by these is provided. The compiler architecture eliminates the need for creating one model version for each intended application and makes the creation of application-specific compilers a task that can be performed by nonspecialists. This will widen the use of equation-based modeling to areas where it is not used today.

Today, equation-based languages exist for describing both continuous and discrete phenomena. Some of these languages are Modelica [1], Ascend [2], and VHDL-AMS [3]. Support for them is given by a number of integrated software environments for model creation, editing, and numerical simulation. The languages have become increasingly powerful with support for acausal models and object-oriented concepts such as inheritance. In acausal models, no information is given as to what variables to solve for in each equation. Procedural models are the opposite; the equations there have their counterpart in assignments.

Large model libraries are today built up in these languages, where the models have the potential to be accessed by separate tools for modeling and simulation, as is shown in the vision in Figure 1. This improves on the choices available compared to the commonly used integrated modeling and simulation environments. In the vision, various modeling environments such as mechanical CAD and scheme layout software can be used to edit the models of the library. To execute the models in simulation environments or other software, a model compiler first processes the models and generates programming code that is adapted to the target application. The approach is model rather than tool centric and provides a maximum output of the work put into creating a model. The same information is used in all synthesis and analysis.

So what parts exist—and what is missing—to achieve such an approach? Looking at the left part of Figure 1, there exist modeling environments that support equation-based languages. Work [4-6] has been done to integrate modeling environments for mechanical CAD and block diagrams with the Modelica language.

In the area of model compiling, the right-hand side of Figure 1, research is performed within the Open-Source Modelica project [7], where an open-source compiler [8] is developed for the Modelica language. The commercial tool Dymola contains all functionality needed to compile Modelica models into simulation code. Analyzers such as DAEPACK [9] can read Fortran subroutines and generate the input needed by solvers. None of these projects, however, so far provides a straightforward way of customizing the generated code so that it suits a particular simulation environment.

To provide an interface toward numerical solvers, which compute the time response of the models, low-level standards such as DSblock [10] and Simstruct [11] have been defined. There, models are stored in programming languages such as Fortran and C. The compiled models return the information needed, such as the time-derivative vector, by means of call interfaces to the solver used. Dymola outputs DSblock models for simulation primarily in Dymola's internal solver or in Matlab/Simulink. These model standards are, however, final in the sense that they are difficult to translate into other programming languages.

In this article, we define a more generic low-level model representation, to be output by model compilers, which can be further translated by simple means into the programming language demanded by the target application. The DSblock and Simstruct model standards can be generated from this representation as well as other output determined by the interest of the compiler user.

The model compilers today only accept models with defined causality of all interface variables (i.e., the variables that are communicated with surrounding models are determined in terms of which are input and which are output). The assignment of input-output causality and sign is dependent on the target application characteristics, which the compiler needs to be aware of. In this article, we investigate what information is needed about the target application and the models to assign causalities and proper signs to the interfacing variables. It is also shown how these assignments can be made automated.

The Modelica language is used as an example of modern equation-based languages throughout this article. The main ideas are applicable to other modeling languages as well. The language is introduced in the following section, followed by model adaptation, model representations, and finally an implemented open-ended model compiler for a subset of the Modelica language.

2. A Short Introduction to Modelica

The Modelica language is an international effort that unifies and generalizes previous object-oriented modeling languages. It is intended to become a de facto standard. The language has been designed to allow tools to generate efficient simulation code automatically with the main objective of facilitating exchange of models. Modelica has multidomain capabilities, which allow the user to combine electrical, mechanical, and hydraulic component models within the same system model.

A subset of the modeling language Modelica is presented in this section. The subset is the part of Modelica needed to adapt the models to specific simulation environments and is also the part that the implemented model compiler supports. On the equation level, discrete events and arrays are among the features not considered. Arrays are not considered due to implementation complexity. Discrete events put specific demands on the simulation code generated from the models and are outside the scope of this article.

Three of the main classes, or templates, in Modelica are *type*, *connector*, and *model*. A model class contains equations, variables, parameters, constants, and more. In model classes, instances of connector classes, denoted *connectors* in this article, are used as representations of physical connection points through which variables can be exchanged

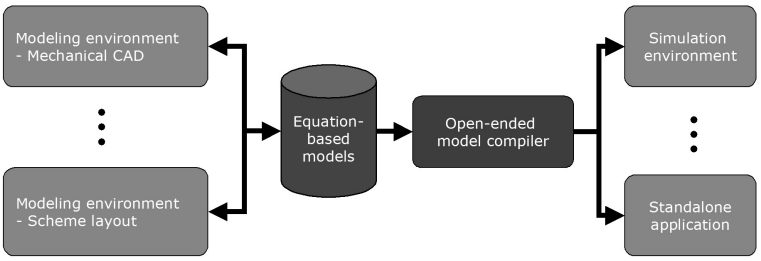


Figure 1. Implementation-independent modeling

with other model instances, denoted *models* in this article.

In the connector class `Pin` below, the variables contained are `v` (voltage) and `i` (current), to be used with models of electrical components. The `flow` keyword is explained later. The word `Real` is the data type of the variable and represents the family of decimal numbers.

```
connector Pin "Pin of an electric comp."
  Voltage v "Potential at the pin";
  flow Current i "Current flowing into";
end Pin;
```

A partial model class, `TwoPin`, can now be defined for an electrical component with two pins. Other model classes can inherit the properties of this class to form complete classes, as is shown in `Resistor`.

```
model TwoPin
  Pin p,n;
  Real i,u;
equation
  0 = p.i + n.i;
  i = p.i;
  u = p.v - n.v;
end TwoPin;

model Resistor
  extends TwoPin;
  parameter Real R "Resistance [Ohm]";
equation
  u = R*i;
end Resistor;
```

In `TwoPin`, instances of `Pin` are created and named `n1` and `n2`. Two internal variables are given the names `i` and `u`, which here are current passing through from the left to the right and voltage difference over the resistor in the mentioned direction. The current in `Pin` is regarded as flowing into the connector from outside the model. Connector variables are accessed by means of dot notation, as is shown in the equations. The variable `i` in the connector `n1` is, for example, referenced as `n1.i`. The model class `Resistor` inherits everything from `TwoPin` through the `extends` keyword and adds one parameter and an equation.

Equations in Modelica can also be specified by using the special `connect` statement. The equation form `connect(c1, c2)` expresses a coupling between the two connectors `c1` and `c2` of two models declared in the current model class. Below, two resistors are connected in a series using the `connect` statement in `TwoResistors`, but first a current source is defined. The connector variable is assigned a negative value since the current preferably flows out from a source, and the connector class defines the current as flowing into the connector.

```
model VoltageSource
  Pin n;
  parameter Real v "Voltage level [V]";
equation
  n.v = v;
end VoltageSource;

model TwoResistors
  Resistor R1,R2;
  VoltageSource VS1,VS2;
equation
  connect(VS1.n, R1.n1);
  connect(R1.n2, R2.n1);
  connect(R2.n2, VS2.n);
end TwoResistors;
```

The connect statements in `TwoResistors` represent equations formed by summing all `flow` connector variables to zero at each connection point and by setting all other connector variables equal—in this case, the potential `v` in each connector. Six equations are thus represented here apart from the explicitly stated ones, two equations for each connection point. In the first connection between the models `VS1` and `R1`, for example, the following equations are generated.

```
VS1.n.i + R1.n1.i = 0
```

```
VS1.n.v = R1.n1.v
```

The `type` classes are used as in programming languages for defining the data type, such as real, integer, or logical for a certain quantity, but in the modeling context also specify, for example, the unit. Two `type` classes suitable for models of electrical components are

```
type Current = Real(unit="A");
type Voltage = Real(unit="V");
```

Instead of specifying a quantity as `Real`, it can thus be specified as `Current` or `Voltage`, which are here `type` classes that inherit from the built-in type `Real`.

2.1 DCmotor Example

A DCmotor will serve as an example throughout the article. The motor model in Figure 2 contains an inductance, a resistor, and an electromechanical component transforming electrical energy to mechanical and vice versa. The components instantiate the `Pin` and `Flange_b` connector classes. The small squares are connectors of the components, whereas the large squares are the connectors of the DCmotor model through which it communicates with other models. The model is thus implemented in the same fashion as the models it has instantiated, such as `Resistor`.

```
type Angle=Real(unit="rad");
type Speed=Real(unit="rad/s");
type Torque=Real(unit="N.m");

connector Flange_b "1D rotational
    flange"
  Angle phi "Abs. rot. angle out of
    flange";
  flow Torque tau "Cut torq. out of
    flange";
end Flange_b;

model DCmotor
  Resistor Res(R=1);
  Inductor Ind(L=1);
  EMF Emf(k=1);
  Pin p "Positive electric pin";
  Pin n "Negative electric pin";
  Flange_b fb "Motor shaft";
equation
  connect(p, Res.p);
  connect(Res.n, Ind.p);
  connect(Ind.n, Emf.p);
  connect(Emf.p, n);
  connect(Emf.flange_b, fb);
end DCmotor;
```

A plot of the motor torque `DCmotor.fb.tau` and voltage `DCmotor.p.v` at the positive pin is depicted in Figure 3. In this simulation, {`DCmotor.p.v, DCmotor.n.v, DCmotor.fb.phi`} are inputs, and {`DCmotor.p.i, DCmotor.n.i, DCmotor.fb.tau`} are outputs. All inputs except `DCmotor.p.v` are set to zero. The dynamics seen is due to the inductance. The output torque is negative due to the sign convention used in the connector class `Flange_b`. We will return to this phenomenon.

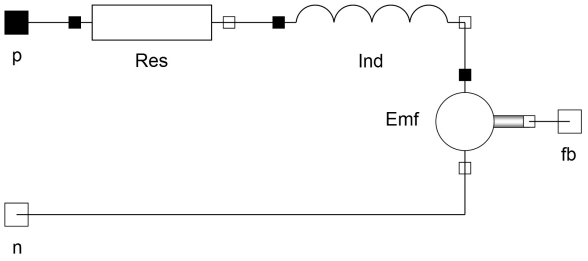


Figure 2. DCmotor model

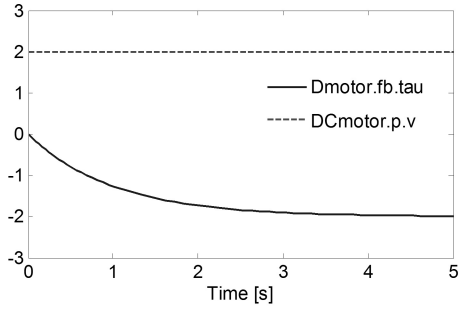


Figure 3. Simulation results of the DCmotor model

3. User Cases

Imagine that the DCmotor model is part of a model library accessed by a number of users, each with their specific simulation environments. One example of such a case is the widespread Modelica standard model library. A natural wish is to be able to insert the model into higher-level models created in these simulation environments, such as Matlab. When changes are made to the equation-based source model, a simple update function could be implemented in which the model in the target environment is generated again. The DCmotor could then be part of hybrid driveline simulations, for example. It will be shown how the model compiler can test various input-output causalities of the interfacing variables of the model and suggest the most

appropriate ones, either automatically or interactively. If several causalities can be applied, then it would be possible to switch the model depending on the boundary conditions of the model in the target environment. The latter feature, however, has not been implemented in this article.

In conventional simulation environments, components are connected through so-called powerports, which correspond to the Modelica connectors. The difference is that the variables passed through the powerports have predefined causalities. As an example, the Hopsan simulation package [12], used mainly for fluid power systems, defines the powerports for electrical and mechanical connections as shown in Table 1.

A sign convention denoted "out of" in the case of current means that a positive current leaves the port and thus the component. A torque with a "into" remark is applied to the port in a direction given by the right-hand rule. The directions of positive variables are shown in Figure 4. The torque is thus applied clockwise, and the shaft rotates counterclockwise if seen from the right for positive values of these variables.

Each powerport type has its counterpart in a powerport that has the opposite causalities and sign conventions for each variable, so that connections can be made where output variables of one component are always connected to input variables of another component.

The C and Z variables of the Hopsan powerports are computed by components that mimic the wave propagation in physical components such as metal rods, electrical wires, and fluid lines. The C variable is the actual wave of, in this case, voltage or torque, and Z is the impedance at the port. The actual voltage and torque at the port connected to such a line element are given by

$$Uel = Cel + Zel \cdot iel, \tag{1}$$

$$Mrot = Crot + Zrot \cdot dThetarot. \tag{2}$$

Since the components of such a simulation are separated in time, there is no need for a central solver; rather, the components in Hopsan compute the state variables on their own. Here, a state is a variable that also exists in differentiated form in the component. The advantage is that all components can be computed in parallel. The method is described in Krus et al. [13] and Auslander [14].

The interesting problem here is that for the `DCmotor` model to work properly in Hopsan, the C and Z variables need to be added to each connector as well as the corresponding equations to the equation set. Also, the rotational velocity is missing in the `DCmotor.fb` connector.

The other target environment treated in this article is CAPSim [15]. It handles simulation of hybrid vehicles in Matlab/Simulink and has a library of batteries, engines, electric motors, and so forth. The causality and sign conventions are as follows, where each port has its counterpart, once again, in a port with inverted causalities and signs. The sign convention needs to be consequent, so in the case of current in the electrical powerport, only "out of" is chosen (see Table 2; Fig. 5).

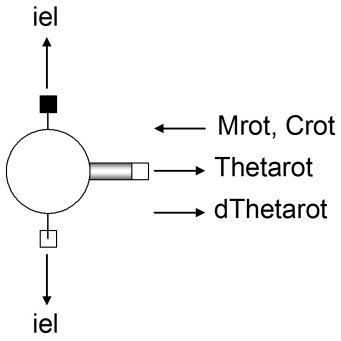


Figure 4. Sign conventions of the simulation package Hopsan applied to the `EMF` component of the `DCmotor` model

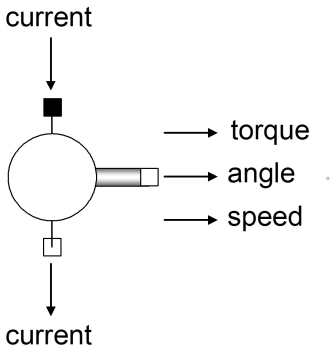


Figure 5. Sign conventions of the simulation package CAPSim applied to the `EMF` component of the `DCmotor` model

The sign of the current depends on port position in CAPSim. Simulink does not make use of the powerport concept in general, but we stick to the powerport concept anyway since it is a way to ensure connections are made that are correct in a physical sense. A model with causal powerports can easily be transformed to a model with a number of separated input and output variables.

4. Compiling Process

Throughout the article, the compiling process of the model is described, with emphasis on the parts that differ from the conventional case.

Table 1. The powerports of Hopsan

Powerport	Variable	State	Unit	Causality	Sign Convention
Electric-QNode	iel	—	A	Output	Out of
	Uel	—	V	Output	—
	Cel	—	V	Input	—
	Zel	—	V/A	Input	—
MechRot-QNode	Mrot	—	Nm	Output	Into
	Thetarot	—	rad	Output	Out of
	dThetarot	Thetarot	rad/s	Output	Out of
	Crot	—	Nm	Input	Into
	Zrot	—	Nm·s/rad	Input	—

Table 2. The powerports of CAPSim

Powerport	Variable	State	Unit	Causality	Sign Convention
Electrical	Current	—	A	Output	(Into) out of
	Voltage	—	V	Input	—
Mechanic	Torque	—	Nm	Output	Out of
	Angle	—	rad	Input	Out of
	Speed	Angle	rad/s	Input	Out of

The conventional compiling process of the Modelica language is now explained. The Modelica source code is first translated into a so-called flat model that contains a flat list of equations and function declarations. Then, the flattened model is read by an optimizer that derives the block lower triangular (BLT) form of the system of equations. Finally, the simulation code is generated. We now discuss each phase in more detail and point out necessary alterations to generate code for the two simulation environments described in the former section.

An overview of the altered compilation process is given here and is described in detail in the following sections. The flattening is followed by a mapping between the outer model connectors and the powerports of the target environment. After a successful mapping, the model is altered to match the powerports. The mapping involves checking whether the equation system can be solved at all. If it can be solved, it is investigated whether the model will pose problems for numerical solvers. An optimization follows as in the conventional case, and possibly equations are added so that states can be computed apart from the normally computed derivatives and algebraic variables. Algebraic variables do not appear in differentiated form in the equation system. The code generation phase is divided into two steps in which Jacobians, assignments, and more are generated first, after which target-specific code generators are applied to this information for each target environment of interest.

4.1 Flattening

The flattened DCmotor model is shown below without the type and connector declarations. The flattening process has been altered here such that the connector declarations have not been flattened. This is to make sure the compiler can perform a comparison of the connectors toward powerports of the target environment. The equations are sorted by components. The last 10 equations are generated from the connect statements. The `der` operator is differentiation with respect to time. The equations are a set of implicit differential algebraic equations (DAEs), which are a mixture of ordinary differential and algebraic equations. Flattening is further described in Fritzson [16].

```
model DCmotor;
  Pin p "Positive electric pin";
  Pin n "Negative electric pin";
  Flange_b fb "Motor shaft'';
  parameter Res.R=1 "Resistance [Ohm]";
  parameter Ind.L=1 "Inductance [H]";
  parameter Emf.k=1 "Torque const. ... ";
  Real Res.v, Res.i, ...;
equation
  Res.R * Res.i = Res.v;
  Res.v = Res.p.v - Res.n.v;
  0.0 = Res.p.i + Res.n.i;
  Res.i = Res.p.i;

  Ind.L * der(Ind.i) = Ind.v;
  Ind.v = Ind.p.v - Ind.n.v;
  0.0 = Ind.p.i + Ind_n_i;
  Ind.i = Ind.p.i;

  Emf.v = Emf.p.v - Emf.n.v;
  0.0 = Emf.p.i + Emf.n.i;
  Emf.i = Emf.p.i;
  Emf.w = der(Emf.flange_b.phi);
```

```
  Emf.k * Emf.w = Emf.v;
  Emf.flange_b.tau = -(Emf.k * Emf.i);

  -fb.tau + Emf.flange_b.tau = 0.0;
  fb.phi = Emf.flange_b.phi;
  -n.i + Emf.n.i = 0.0;
  n.v = Emf.n.v;
  -p.i + Res.p.i = 0.0;
  p.v = Res.p.v;
  Ind.n.i + Emf.p.i = 0.0;
  Ind.n.v = Emf.p.v;
  Res.n.i + Ind.p.i = 0.0;
  Res.n.v = Ind.p.v;
end DCmotor;
```

4.2 Mapping

One may think that a way to adapt the model for usage in Hopsan and CAPSim is to create new models in Modelica that extend `DCmotor` and contain the extra variables and equations needed and where the causality of the interfacing variables is set correctly. This is, however, not possible for a number of reasons that will become apparent. One is that the `der` operator needs to be substituted in some cases, and variables may need to be removed from the connectors. Instead, in this article, we have chosen to adapt the models automatically in the compiling process. This saves work for the end user and hides the substantial complexity of the mapping process.

We assume that the information given in Tables 1 and 2 is available to the model compiler. The process of choosing causalities and signs for all outer connector variables is outlined below and then explained in detail with examples. The first algorithm initially tries to find a powerport for each referenced connector class where each variable of the connector class can be matched with a variable in the powerport in terms of unit. If a powerport is found for every connector class referenced by the model, copies of the connector classes are updated with the information given by the matching powerports, and the model is extended with equations associated with the powerports.

Algorithm 1: Connector-powerport matching

Input: A flattened model with connector class information and information on powerports for the target environment

Output: A modified model with connector variables and that have correct causalities and signs in comparison with the powerports of the target environment

begin
  set *m* = copy of flattened model
    for each free connector *c* of *m* **do**
      set *cc* = copy of class of *c*
      set *connectormatch* = false
      for each powerport *p* **do**
        set *portmatch* = true
        initialize *match* as a vector of size equal to nr of variables in *cc*
        for each variable *v* in *cc* **do**
          if unit of *v* equals unit of a variable *vp* in *p* and *match* has no reference to *vp* **then**
            set *match*(*v*)=*vp*
          else
            portmatch = false
          end if
        end for
      if *portmatch* == true **then**
        connectormatch = true
        set name of *cc* to be "_" + name of *p* + unique id
        complement *cc* with variables of *p* not part of *match*, set causality of all variables, and comment on the new names and sign conv. Change class reference of *c* to the name of *cc*
        for each derivative variable *vd* in *cc* **do**
          set *vs* = state of *vd*
          if *vs* has other causality than *vd* **then** abort compiling process
        end if
        if *vd* is input and is not part of *match* **then** trace from *vs* in *m* until a der operator is found. Replace der expression with *vd*
        else if *vs* is input and is not part of *match* **then** trace from *vd* in *m* until a der operator is found. Replace RHS of der equation with *vd*
        else if not (*vs* is input) and not(both *vs* and *vd* are part of *match*) add equation *vd* = der(*vs*) to *m*
        end if
      end for
      add all equations associated with *p* to *m*where not all variables are defined as inputs
      change sign of all occurrences of connector variables in the equations in the cases where the sign conventions differ
      exit the "for each powerport p" loop
    end if
    end for
    if *connectormatch* == false **then**
      abort compiling process
    end if
  end for
end

When the algorithm above is applied to DCmotor with the default causalities and signs of the CAPSim mapping as input, the following modifications are made:

```
connector _Electric_1 "Replacement for
  Pin"
  input Real v "voltage.-";
  output Real i "current. out of";
end _Electric_1
```

```
connector _Electric_2 "Replacement for
  Pin"
  input Real v "voltage";
  output Real i "current. out of";
end _Electric_2

connector _Mechanic_1 "Repl. for
  Flange_b"
  output Real fb.tau "torque. out of";
  input Real fb.phi "angle. out of";
  input Real fb.speed "speed. out of";
end _Mechanic_1

model DCmotor
  _Electric_1 p "replaced Pin p";
  _Electric_2 n "replaced Pin n";
  _Mechanic_1 fb "replaced Flange_b fb";
  . . .
equation
  . . . Emf.w = fb.speed "replaced
    der(Emf.flange_b.phi)";
  . . .
end DCmotor;
```

The variables of the powerport are matched in the order they appear, meaning that the port can contain several variables with the same unit where the nonmatched variables are added after the unit matching.

In this case, the sign conventions differ for the currents of the original Pin instantiations. The sign convention for the connector variables is found in this article by searching for the keywords "into and out of" in the comments of the connector class variables. In the original DCmotor model, the Pin class had "into" causality for the current, and in the powerport, the current is positive if heading "out of" the connector. All occurrences of `p.i` and `n.i` are thus replaced by `–p.i` and `–n.i`. This is not shown in the resulting model above.

If any powerport variable is the derivative of another and is of output causality, then the state or derivative need not exist in the connector class since it can be reproduced through a differentiation equation. If a derivative is added to the connector class as an input and the state has a match, it is investigated if the der operation on the state variable in the model can be exchanged with the derivative itself. The der expression will otherwise give a value of zero since the state is then an input and thus regarded as constant. Differentiating an input is not a good practice due to numerical reasons and cannot always be achieved anyway. If a state is added and is an input and the derivative has a match, a similar process takes place since otherwise, the component may perform an internal integration on the derivative, which can drift from the connector-supplied state. The der expression will not always be suited for the operations described in this paragraph.

Given a successful completion of algorithm 1, the next step is to evaluate whether the model can be solved. This is done in algorithm 2. If there is no success in algorithm 1, the causalities and signs are inverted for one connector class copy by applying the inverted matched powerport, and algorithms 1 and 2 are applied again. This is performed so that all combinations of connector class causalities are evaluated. For each combination, algorithm 2 will tell whether the model can be solved at all and, if so, whether the numerical solver will experience difficulties. The user or the software can then choose the best combination based on the grade.

Algorithm 2: Evaluation of solvability

Input: A flattened model

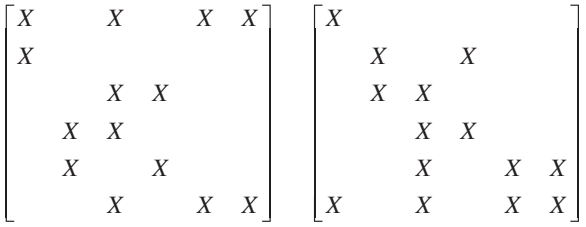
Output: A grade on solvability = failed or a number indicating how many states that need to be made algebraic

begin
  set grade = failed
  set m = flattened model. Perform all operations on m
  perform model simplifications
  try to achieve a zero-free diagonal in the equation system to see if one variable can be assigned to each equation
  if this fails **then** abort evaluation
  transform equation system into block lower triangular form (BLT)
  perform index reduction
  set grade = 1/(1 + number of original states that need to be made algebraic)
end

The operations performed in this algorithm are all well known and are used in conventional compilers for equation-based modeling languages. They are briefly described here anyway to provide the whole picture. A thorough explanation to the operations is given in Bunus and Fritzson [17].

In the simplification stage, one operation performed is to identify and remove simple equations of the form $a = b$ or $a + b = 0$. One of the variables in each of these is substituted in the remaining equations and thus is not part of the unknowns any longer, as described in CAPSim [15], for example. This operation is not performed in this article but normally gives a large reduction of the equation system and thus the time to compile.

To check whether a model can be solved, each unknown variable is assigned one equation each through a number of row permutations in the procedure of maximum transversal. At this stage, a state variable and all its derivatives are regarded as one unknown since they can be derived given each other. The remaining internal variables and outputs (the algebraic variables) are also unknowns. The algorithm is described in detail in Duff and Reid [18], and a Fortran implementation is found in ACM algorithm 575, proposed by Duff and Reid [19]. The matching is perfect if each unknown has been assigned an equation and there are no nonassigned equations left. In the structure incidence matrix of an equation system shown below, the rows are



a) b)

Figure 6. An original and a row-permutated equation system where matrix (b) has a zero-free diagonal

equations, and the columns are unknowns. In Figure 6, the original system is shown in (a) and the row permutated in (b).

If matrix (b) is transformed into a lower triangular form, the unknowns can be computed sequentially one at a time by a forward substitution process. This greatly reduces the computational effort in solving the system but is here applied as a prestage for the index reduction, as will be explained soon. The lower triangular form cannot be achieved in the general case, but there exist algorithms for transforming matrices to the BLT form, depicted in Figure 7, where blocks of equations are solved sequentially. The transformation is performed by symmetric row and column permutations. The preceding transformation to a zero-free diagonal is thus a necessary first step. The algorithm is described by Duff and Reid [20].

To be able to solve the system with respect to the highest-order derivatives and algebraic variables, it may be necessary to differentiate a number of equations. Pantelides's algorithm [21] establishes the minimum number of times each equation has to be differentiated. This situation occurs when, for example, an equation only holds states. Since the equation contains only known variables, the equation does not contribute to the solution of the equation set and needs to be differentiated. One way to maintain an equal number of equations and unknowns is to keep only the last differentiated version of each equation. This results in problems such as drift from the algebraic constraints. The other method is to introduce one new unknown for each equation differentiation by turning states into algebraic variables. This method is used in this article and is explained in Mattsson and Söderlind [22], where BLT partitioning is used as part of the method.

In algorithm 2, the grade is set after the index reduction operation for this particular causality combination. The grades for the `DCmotor` model are shown in Table 3.

The first combination yields no equation differentiation, whereas the second yields seven differentiations. Two original states are made algebraic in the second combination: `Ind.i` and `fb.phi` (the latter is partly due to the inserted der equation from the mapping procedure). The model of

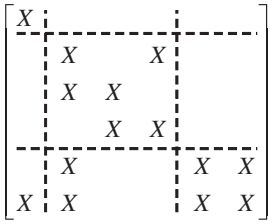


Figure 7. The matrix of Figure 6a in BLT form

Table 3. Grades for all combinations of causality of the connector classes of DCmotor

Causality of _Electric_1, _Electric_2, and _Mechanic_1	Grade
{default, default, default}	1
{default, default, inverted}	1/3
{default, inverted, default}	Failed
{default, inverted, inverted}	Failed
{inverted, default, default}	Failed
{inverted, default, inverted }	Failed
{inverted, inverted, default}	Failed
{inverted, inverted, inverted }	Failed

the second combination is less interesting since some of its dynamics have been eliminated. It is in this case static. All other combinations fail.

5. Deriving an Optimized BLT Form

If a successful assignment of outer-connector variable causalities has been achieved, the compiler now can switch to preparing for the code generation. The equation system is brought into a zero-diagonal form again, where now the unknowns are only the highest-order derivatives.

5.1 Inline Integration

If code is generated at this stage, the states can be computed given the derivatives. But if there is a need to compute the states directly in the generated code, one equation can be added to the system for each state in which the integration is performed numerically at the same time as the state is set as unknown. This also gives the possibility to implement implicit integration (i.e., that present derivatives are part of the numerical integration). An example of the integration formula is the trapezoidal rule, which is used in this article:

$$x_n = x_{n-1} + h\frac{(\dot{x}_n + \dot{x}_{n-1})}{2} \text{ where } h \text{ is the time step.} \quad (3)$$

Since these equations are kept intact with this assignment of unknowns, a time step of zero poses no problems. The equation system can thus be used to compute initial

values for the highest-order derivatives to find consistent start values for all variables. Otherwise, the solution will fail. The technique is covered in detail in Elmqvist, Cellier, and Otter [23] and is applied to distributed simulation in Krus [24].

The BLT algorithm is now applied on the equation system in the same manner, regardless of whether integration formulas have been inserted.

5.2 Tearing

The BLT form is efficient but can be improved drastically by the tearing method [25], in which each block is split into two groups of assignments and implicit equations. The tearing variables are chosen such that the assignments are as many as possible. There exist heuristics for choosing the tearing variables (see [26, 27]) since a complete optimization is not possible, as shown by the same author. Here, the heuristics from Cellier [28] are given as follows:

- Determine the equations, part of the block, containing the largest number of unknown variables.
- For each of these equations, determine the variables referenced by it that are unknown in the largest number of equations.
- For each of these variables, determine how many additional equations can be solved if that variable is assumed known.
- Choose one of those variables as the next tearing variable that allows the largest number of additional equations to be turned into assignment.

In this article, a simple method is used in which it is checked which equations contain variables that have not been computed before the equation. These equations are set as tearing equations, and the variables assigned to them thus become tearing variables. The remaining equations are transformed into assignments.

If the model with the default causality combination output from algorithm 2 is optimized using BLT partitioning and tearing, the result is as below, where inline integration has been employed as well. The blocks are numbered. When solving block 7, the tearing variable `Res.n.v` is given as an initial guess, the assignments are computed, and `Res.n.v` is updated. This process is iterated until the variables have converged satisfactory. The `old_der_Ind.i` variable corresponds to dx_{n-1}/dt in equation (3).

1: Emf.w = fb.w;
2: Emf.v = Emf.k*Emf.w;
3: Emf.n.v = n.v;
4: Emf.p.v = Emf.n.v+Emf.v;
5: Ind.n.v = Emf.p.v;
6: Res.p.v = p.v;
7: Ind.p.v = Res.n.v;
Ind.v = Ind.p.v-Ind.n.v;
der_Ind.i = Ind.v/Ind.L;
Ind.i = (h/2*(der_Ind.i+old_der_Ind.i))
+old.Ind.i;
Ind.p.i = Ind.i;
Res.n.i = -Ind.p.i;
Res.p.i = -Res.n.i;
Res.i = Res.p.i;
Res.v = Res.r*Res.i;
0=Res.v+Res.n.v-Res.p.v "tear: Res.n.v";
8: Ind.n.i = -Ind.p.i;
9: Emf.p.i = -Ind.n.i;
10: Emf.i = Emf.p.i;
11: Emf.n.i = -Emf.p.i;
12: Emf.flange_b.phi = fb.phi;
13: Emf.flange_b.tau = -(Emf.k*Emf.i);
14: p.i = -Res.p.i;
15: n.i = -Emf.n.i;
16: fb.tau = Emf.flange_b.tau;

Without inline integration, the optimized model becomes as below, where the only state is `Ind.i` and its derivative is `der_Ind.i`.

1: Ind.p.i = Ind.i;
2: Res.n.i = -Ind.p.i;
3: Res.p.i = -Res.n.i;
4: Res.i = Res.p.i;
5: Res.v = Res.R*Res.i;
6: Emf.w = fb.w;
7: Emf.v = Emf.k*Emf.w;
8: Emf.n.v = n.v;
9: Emf.p.v = Emf.n.v+Emf.v;
10: Ind.n.v = Emf.p.v;
11: Res.p.v = p.v;
12: Res.n.v = Res.p.v-Res.v;
13: Ind.p.v = Res.n.v;
14: Ind.v = Ind.p.v-Ind.n.v;
15: der_Ind.i = Ind.v*Ind.L;
16: Ind.n.i = -Ind.p.i;
17: Emf.p.i = -Ind.n.i;
18: Emf.i = Emf.p.i;
19: Emf.n.i = -Emf.p.i;
20: Emf.flange_b.phi = fb.phi;
21: Emf.flange_b.tau = -(Emf.k*Emf.i);
22: p.i = -Res.p.i;

```
23: n.i = -Emf.n.i;
24: fb.tau = Emf.flange_b.tau;
```

In the Hopsan case, the torque and currents need to change sign, and one equation is added for each connector due to the wave calculations. A der equation to compute the speed variable is also added.

```
1: fb.phi = Emf.flange_b.phi;
     der_fb.phi = fb.phi;
     der_Emf.flange_b.phi = der_fb.phi;
     Emf.flange_b.phi - old_Emf.flange_b.phi-
     h/2.0*(der_Emf.flange_b.phi+
     old_der_Emf.flange_b.phi) = 0
     "tear: Emf.flange_b.phi";
2: Emf.w = der_Emf.flange_b.phi;
3: Emf.v = Emf.k.*Emf.w;
4: p.v = (p.Zrot*p.i)+p.Crot;
     Res.p.v = p.v;
     Ind.p.v = Res.n.v;
     Emf.p.i = -Ind.n.i;
     Emf.n.i = -Emf.p.i;
     n.i = -(Emf.n.i);
     n.v = (n.Zrot*n.i)+n.Crot;
     Emf.n.v = n.v;
     Emf.p.v = Emf.n.v+Emf.v;
     Ind.n.v = Emf.p.v;
     Ind.v = Ind.p.v-Ind.n.v;
     der_Ind.i = Ind.v/Ind.L;
     Ind.i = h/2*(der_Ind.i+old_der_Ind.i)+ old_Ind.i;
     Ind.p.i = Ind.i;
     Res.n.i = -Ind.p.i;
     Res.p.i = -Res.n.i;
     Res.i = Res.p.i;
     Res.v = Res.R*Res.i;
     (-p.i)-Res.p.i = 0.0 "tear: p.i";
     Res.v+Res.n.v-Res.p.v=0 "tear: Res.n.v";
     Ind.p.i-Ind.n.i = 0 "tear: Ind.n.i"
5: Emf.i = Emf.p.i;
6: Emf.flange_b.tau = -(Emf.k*Emf.i);
7: fb.tau = - Emf.flange_b.tau;
8: fb.w = -(fb.Crot-fb.tau.)/fb.Zrot;
```

6. Code Generation

The code generation process in this article is performed in two stages where, in the second stage, the final programming code is generated, which suits the target simulation environment. This enables the use of plug-ins to the compiler for widening its simulation environment support.

The first stage should produce information that supports the output of the most common operations performed in the model optimization phase, such as index reduction, BLT partitioning, and tearing. The information should also indicate the variable types and corresponding connector and connector class and thus also the powerport and variable name to use in the target environment.

We propose an information model where each equation is given as a residual (RHS = 0) as well as an assignment where possible. The variable to solve for is stated for each equation as well as which block the equation belongs to. For each block, a complete Jacobian is supplied with the partial derivatives of the block residuals with respect to the variables to solve for. These Jacobians can be used not only for solving the complete block by Newton-Raphson methods but also for nonlinear equations part of the iterative solution as a result from tearing.

Tearing equations are indicated. The variables are divided into internal, parameter, constant, input, and output. It is indicated whether the internal and output variables are states, derivatives, or algebraic variables. For inputs and outputs, it is declared which connector they belong to. For each connector, there is a reference to its class, which in turn holds information on the target environment powerport characteristics, such as naming of variables.

What has not been treated so far is the inclusion of external function calls in the equations. This poses no problem given that they are provided by the final code generator plug-in. To improve the convergence of the solution, the partial derivatives with regard to function parameters can be provided as separate functions with a naming convention, such as "dx1_extfunction(x1,x2)," where dx1 stands for the partial derivative with respect to the first argument, and "extfunction" is the name of the original function.

To support discrete events is, however, a more difficult issue and is not discussed in this article. It is a natural future work. The model compiling phase, however, remains much the same since algorithms can be treated as equation sets in the sense that they are dependent on a number of variables to compute the output variables.

7. An Implemented Modelica Compiler

Throughout the article, a model compiler has been employed in which the presented ideas have been implemented by the authors. The compiler is explored in the remaining sections following the data flow of the compiling process. The compiler was mainly implemented to produce this article but has a few solutions that may be interesting to a wider audience. It is straightforward, rather

than efficient, to understand as it is meant to function as a raw model for open-ended model compilers.

A Modelica subset parser outputs the abstract syntax tree (tree-structured grammatical units) in an intermediate XML file [29], which is imported into the XML-based, so-called Modelith.Dynamics (or Modelith for short in this article) language using the XML technology XSLT [30]. Modelith is similar to Modelica.XML [31] but easier to read due to its reduced syntax. It is by no means a competitor to the latter language.

The XML file is operated on in main memory by a Java application. The application makes use of free software for symbolic differentiation, developed by Jens-Uwe Dolinsky at the University of Wismar, and XML structure manipulation. The flattening procedure is performed entirely in XSLT as well as the code generation outside the compiler. Thus, only open-source software is employed throughout the application, and well-known Web technologies are used where possible. This reduces to a minimum the knowledge needed to adapt the model compiler for other source languages or to improve the current implementation.

7.1 XML

XML is a standard for representing relational data and brings text documents to a higher level by introducing a hierarchy where each level contains an identifier, tag, and some text data. XML was originally designed for the exchange of data over the Internet, which is why many possibilities to transform XML data are available. It is a markup language very much like HTML. It is lightweight in comparison with, for example, STEP (ISO 10303), and it supports the definition of schemas (expresses the data model or information structure) for validation of the produced documents. There is a large variety of open- source and license-free tools for performing various operations on XML documents, and support for XML will be available for a long time due to its fundamental nature. To illustrate the most fundamental concept of "marking up" data using XML, the example below shows some XML data. Two *elements* are there, `Modelith` and `model_class`, where `model_class` is contained in `Modelith`. The `model_class` element has an *attribute* with the name *name* and value "`Spring`." This vocabulary is used later in the article.

7.2 Model Import

A parser for a basic subset of Modelica was created using the free parser generator JavaCC [32]. A parser, explained in Levine, Mason, and Brown [33], finds certain units of information in its input. These units are specified in a grammar. Such a grammar is available for Modelica [34]. It was copied, and certain items were omitted where no support was to be given.

For each grammar unit found, an XML element can be output, resulting in a tree-structured result to be further processed by XML tools such as XSLT. The result for a simple example is shown in Figure 9. There, the first two rows are the Modelica source document, and the rest is the produced output from the Modelica parser. The unit implemented by `semi_colon_key`, for example, found that there is the character ";." One of the reasons for creating this intermediate XML file is that new tools are appearing for generating translators using mappings between different XML document types, as in this case from the Modelica grammar into Modelith. As the two languages evolve, the work needed to adjust the translator is not then as extensive as for handwritten translators.

```
<Modelith>
    <model_class name="Spring"/>
</Modelith>
```

Figure 8. XML document sample

```
model a
end a;
- - - - - - - - - - - - - - - - - - - - - - -
<Modelica>
    <model_definition>
        <class_definition>
            <model_key/>
            <IDENT>a</IDENT>
            <class_specifier>
                <end_key/>
                <IDENT>a</IDENT>
            </class_specifier>
        </class_definition>
        <semi_colon_key/>
    </model_definition>
</Modelica>
```

Figure 9. An XML-transformed Modelica grammar document

7.3 Model Representation

On the model input side, the internal model representation should contain everything needed to adapt the models to target environment characteristics. The information needed is model classes with connectors and equations, connector classes, and type classes (to compare the units). The "into" and "out of" keywords mentioned earlier need to be part of the connector class vocabulary. The representation that is described here also contains object-oriented features such as inheritance and model hierarchy. A model compiler only requires a list of equations, but since that is a special case of an object-oriented model structure for equations, such input is automatically supported, and at the same time, the task of flattening such a structure can be taken care of in the model compiler. Flattening is a complex procedure in itself.

In XML, XML-Schema [35] can be used to define what information should go in an XML document and how it should be structured and constrained to be correct. XML-Schema is used here in a graphical form to describe the different parts of Modelith. An XML document in Modelith has a data structure as shown in Figure 10. It has the root element `Modelith`. Then it has three classes, as they are called in Modelica, implemented by the elements `type_class`, `model_class`, and `connector_class` directly under the root. The names of the elements come from the Modelica language. Modelith is, in fact, in the declarative part (equations rather than executable code), mainly a copy of Modelica but with a few changes that make model compiling more straightforward. As in object-oriented programming languages, instances of these classes can be created. The class is used as a kind of template.

In Figure 10, squares indicate XML elements, rounded squares represent choices or sequences, and dashed lines indicate that the element is optional. Under the element, the allowed number of elements is sometimes given. In the figure, the Modelith element must contain one or more elements with the given names. If no number is given, one element/choice/sequence must exist in the document. Plus signs indicate that some information in the figure is hidden. The schema and the pictures have been created in XML Spy [36], which is an editor for XML. An XML document that is valid when compared to the schema above (ignoring hidden information in the schema) could be as shown in Figure 11.

The Modelica class `type`, implemented in Modelith as in Figure 12, is the most fundamental one. Instances—objects of the `type` class, implemented as `type_instance` elements—are used as variables, parameters, and constants. By itself, the class specifies, for example, what unit the instances should use. The `type_class` element has one optional element, as do `model_class` and `connector_class`. The `uses` element tells what class the current class—in this case, `type`—uses as a base for adding further information. This element makes it possible to inherit variables, connectors, and so on from other classes. The optional `change_value` part of `uses` can, for example, be used to change the unit of a variable part of the base class. An example of a type written in both Modelica and the Modelith XML implementation is given in Figure 13.

Instances of the connector class are used as connection points for model instances. Such an instance could be a resistor port or motor shaft, for example. A `connector_class` contains a number of variables implemented by `type_instance` elements. Each such element here has an extra attribute named `prefix` that tells whether the variable should be summed to zero (attribute set to "`flow`") or set equal with respect to corresponding variables in all other connectors connected. An attribute named

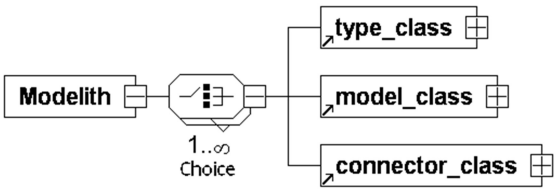


Figure 10. The top element of the Modelith schema

```
<Modelith>
    <model_class/>
    <connector_class/>
    <type_class/>
    <model_class/>
</Modelith>
```

Figure 11. Modelith document sample

`cut_axis_direction` gives information for which cut axis direction the variable is positive. It can have the values "", "`into`," and "`outof`."

The `model` class, implemented in Figure 14, is the central class in the Modelica language. Just like the other classes, it can in Modelith make use of inheritance through the `uses` element. It can contain instances of models through the `model_instance` elements, and connections between them are expressed in `connection`. This way, systems of model instances can be created. The model element also contains equations and type instances. In the example in Figure 15, the use of inheritance and instantiation of a parameter is shown.

All instances—`model_instance`, for example—have the same structure as the class elements such as `type_class` in Figure 12 (i.e., they reference the class they are instances of through the `uses` element). The instances can change the value of any `type_instance` in the referenced class through `change_value`.

The authors have chosen to use a subset of MathML [37] for the equations. Figure 16 shows the schema and Figure 17 contains an example XML document of an equation. The equations are then unfortunately not easy to read, as compared to pure strings, but are much easier to transform and can be used to produce typeset equations for documentation for MathML-enabled browsers. XSLT [30], a language for performing transformation of XML documents into arbitrary text files, is not well suited for string manipulation; it is therefore better to have an XML representation of the equation. This also allows equations to be validated against the schema. User-made functions can be referred to here with any number of arguments. One of the MathML functions supported in this

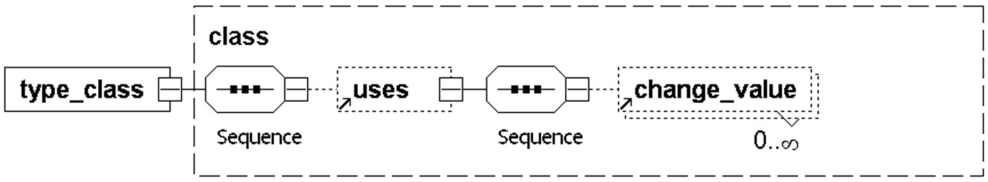


Figure 12. One of the top elements—in this case, `type_class`

```
type Position = Real(unit=”m”);
- - - - - - - - - - - - - - - - - - - - - - - - - -
<type_class name="Position">
  <uses class="Real">
    <change_value type_instance="unit"
                  value="m"/>
  </uses>
</type_class>
```

Figure 13. XML sample document for a single Modelica `type`

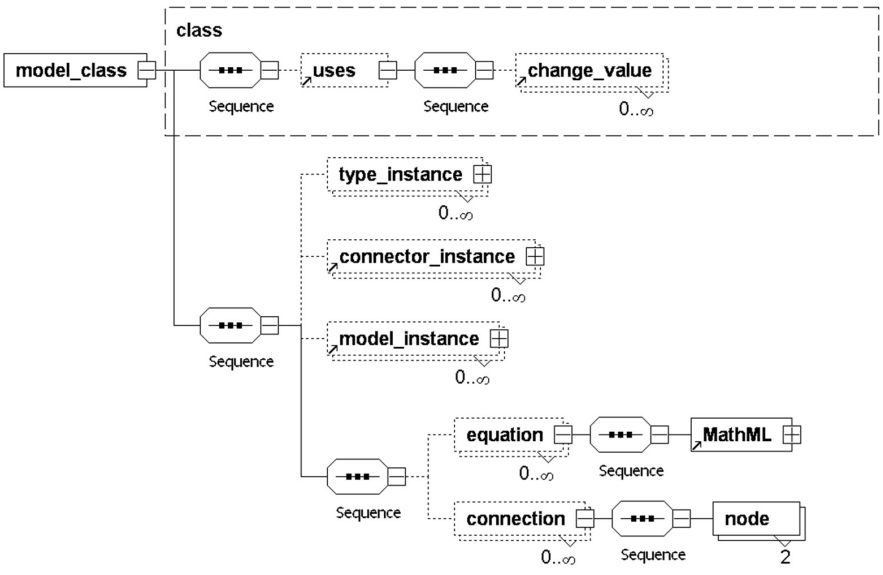


Figure 14. The central class `model_class`, in which all equations are placed

```
model BounceMass extends TwoFlange
  parameter Mass m=10;

- - - - - - - - - - - - - - - - - - - - - - - -
<model_class name="BounceMass">
  <uses class="TwoFlange"/>
  <type_instance name="m" prefix="parameter"
                 value="10">
    <uses class="Mass"/>
  </type_instance>
</model_class>
```

Figure 15. A sample XML document of a single Modelica model

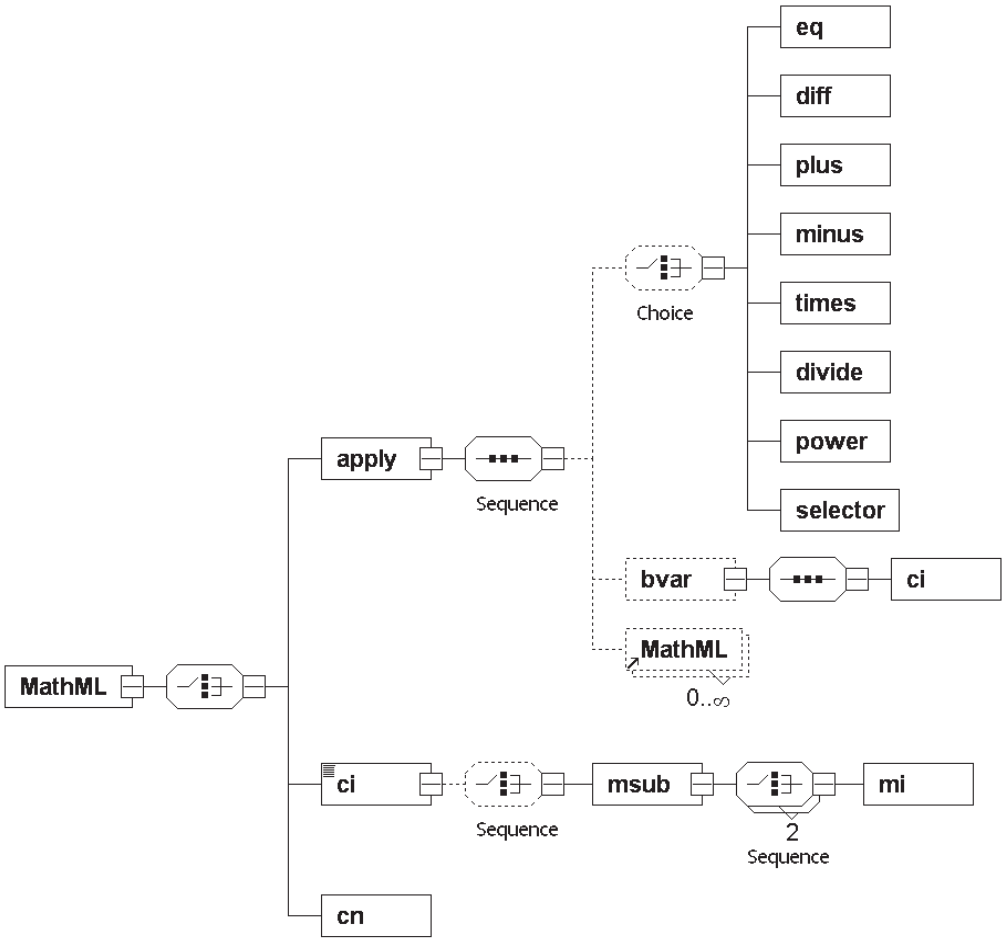


Figure 16. The content of equations and Jacobian elements (i.e., MathML expressions)

```
n1.x = n2.x;
- - - - - - - - - - - - - - - - - - - - - - -
<equation>
  <MathML>
    <apply>
      <eq/>
      <ci>
        <msub>
          <mi>x</mi>
          <mi>n1</mi>
        </msub>
      </ci>
      <apply>
        <minus/>
        <ci>
          <msub>
            <mi>x</mi>
            <mi>n2</mi>
          </msub>
        </ci>
      </apply>
    </apply>
  </MathML>
</equation>
```

Figure 17. Sample XML document of a single equation

schema is `eq`, which is the equality relation. The functions `plus`, `minus`, `times`, `divide`, and `power` need no explanation. The `der` operator in Modelica is implemented here in `diff` and is differentiation with respect to some variable specified in `bvar`, normally set to "`time`." The `selector` function is only used today for the pseudo-code part of the language and specifies a position in a vector or matrix. To call a user-defined (external) function, apply is utilized followed by the function name after which any arguments are given.

To reference the variables part of connector instances, the subscript notation `msub` of MathML is used. The elements `ci` and `cn` contain identifier names and numerical values, respectively.

7.4 Code Generation Prestage Representation

Along the model-compiling process, information is added about causality and so forth for each quantity. To be more precise, a quantity can be a parameter, constant, or input, output, or internal variable. Also, all internal and output variables can be state variables, derivatives, or algebraic, where it is indicated for the state variables whether they should be computed. All equations are assigned to blocks, where each block is a linear or nonlinear equation system. The blocks are to be solved in series, and for each block, a Jacobian is supplied. Each equation has a variable assigned, and if possible, the expression for the assignment of this variable is supplied together with the equation.

```
<equation block="1" solve_for="der_w"
  tear="false">
  <residual>
    m*der_w+(T/L)*x
    <jacobian_element wrt="der_w">
       m
    </jacobian_element>
  </residual>
  <assignment>
    der_w = -(T/L)*x/m
  </assignment>
</equation>
```

Figure 18. Modelith support for solving equation systems

The Jacobian for each block is the partial derivatives of the residuals, formed from the equations, with respect to the variables to compute. The residual and assignment for the variable to solve for are shown for an equation in Figure 18. The equation is part of block 1, and the variable to solve for is "`der_w`", as indicated by the `solve_for` attribute of this `type_instance`. The other two variables x and T are known in this example. The equations are stored as MathML but are shown here as Modelica expressions for clarity.

7.5 Customized Code Generators

The model compiler supports the simulation environments Hopsan and Matlab/Simulink through two code generators, both written in XSLT. Hopsan makes use of bilateral delay lines [13, 14] as a means of separating the various models part of a simulation, in which each model is self-contained with the included solver if needed. The code generator for Hopsan generates Fortran code. The destination characteristics include input-output causality, which is used in Hopsan's model library, and the necessary equations that need to be added for each model to work with the bilateral delay lines.

For Matlab/Simulink, the code generator generates S-functions where the state variables are computed if the user so desires. Both for the Hopsan and Simulink translators, half of the code, about 200 lines, goes into transforming MathML expressions into Fortran or the M-file script language, respectively. Most of the remaining code goes into defining the simulation code interface in terms of variable definitions and so forth. The causality mapping is given for the CAPSim model library mentioned earlier but can be changed in a straightforward manner for other purposes.

8. Conclusions

This article shows how a description of the powerports of a target simulation application, together with a small, final code generator, is enough to create support for generating

executable customized code from equation-based models to a new application. Powerports are the connection points between components in a model and communicate the interfacing variables such as force and velocity. Along the way, it is shown that equation-based models with undefined input-output causalities can be made causal given the powerport information. It is also shown how information about a simulation environment can be used to adapt the equation-based models to fit within the environment in question, for example, by adding appropriate equations to the model. The result is that equation-based model libraries can be used as seamless parts of the procedural model libraries of conventional simulation environments. This will increase the use of equation-based modeling since the user will be able to work only with these and, in the end, result in less procedural modeling for which the models have a shorter life and smaller usage area.

9. References

[1] Fritzson, P., and V. Engelson. 1998. Modelica: A unified object-oriented language for system modeling and simulation. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECOOP'98, Brussels, Belgium.

[2] Piela, P. C., T. G. Epperly, K. M. Westerberg, and A. W. Westerberg. 1991. ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language. *Computers and Chemical Engineering* 15 (1): 53-72.

[3] Christen, E., and K. Bakalar. 1999. VHDL-AMS: A hardware description language for analog and mixed-signal applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 46 (10): 1263-72. Available from http://www.eda.org/vhdl-ams/

[4] Tiller, M., P. Bowles, H. Elmqvist, D. Brück, S. E. Mattson, A. Möller, H. Olsson, and M. Otter. 2000. Detailed vehicle powertrain modeling in Modelica. In *Proceedings of the First International Modelica Workshop*, pp. 169-78.

[5] Bunus, P., V. Engelson, and P. Fritzson. 2000. Mechanical models translation, simulation and visualization in Modelica. In *Proceedings of the First International Modelica Workshop*, pp. 199-208.

[6] Dempsey, M. 2003. Automatic translation of Simulink models into Modelica using Simelica and the AdvancedBlocks library. In *Proceedings of the 3rd International Modelica Workshop*, pp. 115-24.

[7] Fritzson, P., P. Aronsson, P. Bunus, V. Engelsson, and L. Saldamli. 2002. The Open Source Modelica Project. In *Proceedings of the 2nd International Modelica Conference*, pp. 297-306.

[8] Kågedal, D., and P. Fritzson. 1998. Generating a Modelica compiler from natural semantics specification. In *Proceedings of the 1998 Summer Computer Simulation Conference*, Reno, NV.

[9] Tolsma, J. E., and P. I. Barton. 2000. DAEPACK: An open modeling environment for legacy models. *Industrial & Engineering Chemistry Research* 39 (6): 1826-39.

[10] Otter, M., and H. Elmqvist. 1995. The DSblock model interface for exchanging model components. In *Proceedings of the 1995 EUROSIM Conference*, pp. 505-10.

[11] The Mathworks, Inc. 2003. Simulink: Release notes version 5.1. http://www.mathworks.com

[12] Larsson, J., A. Jansson, and P. Krus. 2002. User's guide to Hopsan—an integrated simulation environment. Tech. Rep. LiTH-IKP-R1258, Linköping University, Sweden.

[13] Krus, P., A. Jansson, J.-O. Palmberg, and K. Weddfelt. 1990. Distributed simulation of hydromechanical systems. In *Proceedings of Third Bath International Fluid Power Workshop*, Bath, UK.

[14] Auslander, D. M. 1968. Distributed system simulation with bilateral delay-line models. *Journal of Basic Engineering* 90:195-200.

[15] CAPSim: http://www.capsim.se

[16] Fritzson, P. 2003. *Principles of object-oriented modeling and simulation with Modelica*. New York: Wiley-IEEE.

[17] Bunus, P., and P. Fritzson. 2004. Automated static analysis of equation-based components. *SIMULATION* 80:321-45.

[18] Duff, I. S., and J. K. Reid. 1981. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software* 7:315-30.

[19] Duff, I. S., and J. K. Reid. 1981. Algorithm 575 permutations for a zero-free diagonal. *ACM Transactions on Mathematical Software* 7:387-90.

[20] Duff, I. S., and J. K. Reid. 1978. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software* 4:137-47.

[21] Pantelides, C. 1998. The consistent initialization of differential algebraic systems. *SIAM Journal on Scientific and Statistical Computing* 9:213-31.

[22] Mattsson, S. E., and G. Söderlind. 1993. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing* 14:677-92.

[23] Elmqvist, H., F. Cellier, and M. Otter. 1995. Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. In *Proceedings of the European Simulation Multiconference '95*, Prague, pp. 23-34.

[24] Krus, P. 1996. An automated approach for creating components and sub-systems for simulation of distributed systems." In *Proceedings of the Ninth Bath International Fluid Power Workshop*, Bath, UK.

[25] Elmqvist, H., and M. Otter. 1994. Methods for tearing systems of equations in object-oriented modeling." In *Proceedings of the European Simulation Multiconference*, Barcelona, Spain, pp. 326-32.

[26] Carpanzano, E. 2000. Order reduction of general nonlinear DAE systems by automatic tearing. *Mathematical and Computer Modeling of Dynamical Systems* 6:145-68.

[27] Mah, R. 1990. *Chemical process structures and information flows*. Boston: Butterworths.

[28] Cellier, F. 2002. The loop-breaking algorithm by Tarjan. Material from the course ECE 449/549: "Continuous System Modeling."

[29] World Wide Web Consortium (W3C). 2000. Extensible Markup Language (XML) 1.0 (second edition). http://www.omg.org

[30] World Wide Web Consortium (W3C). 1999. XSL transformations (XSLT) version 1.0. http://www.omg.org

[31] Pop, A., and P. Fritzson. 2003. ModelicaXML: A Modelica XML representation with applications. In *Proceedings of the 3rd Modelica Conference*, Linköping, Sweden, pp. 419-30.

[32] Sun Microsystems. 1998. Java compiler. Technical report.

[33] Levine, J. R., T. Mason, and D. Brown. 1992. *Lex & Yacc*. 2nd ed. Sebastopol, CA: O'Reilly & Associates.

[34] Modelica Association. 2003. Modelica: A unified object-oriented language for physical systems modeling, language specification. http://www.modelica.org

[35] World Wide Web Consortium (W3C). 2001. XML schema part 0: Primer. http://www.omg.org

[36] Kirn, L. 2002. XML integrated development environments: Accelerating XML application development in the enterprise. White paper, Altova, Inc, Beverly, MA.

[37] World Wide Web Consortium (W3C). 2001. Mathematical markup language (MathML) version 2.0. http://www.omg.org

Jonas Larsson *studied at Linköping University in Sweden and obtained his MSc degree in mechanical engineering in 1999. In 2003, he presented his doctoral thesis, titled "Interoperability in Modeling and Simulation," at Linköping University in Sweden, covering coupled simulation of continuous systems. He is currently employed as a research associate at the Division of Mechanical Engineering Systems.*