Differential-algebraic Equations in Process Simulation

Wolfgang Freiseisen, Robert Keber, Wilhelm Medetz,
Petru Pau, Dietmar Stelzmueller

Abstract

Differential-algebraic equations, which are systems of differential equations and algebraic constraints, occur in mathematical models of physical or engineering systems. Specific algorithms for numerical integration of such systems have been designed. Also, software packages dedicated to various domains of physics and engineering have been developed.

We present here the status of the development of a system that consists in a compiler for a special modeling language – Modelica. We also discuss possible directions to use symbolic computation, in order to extend the system to allow automatic generation of Kirchoff laws.

1 Introduction

This paper describes the status of a simulation software that contains a compiler for a modeling language named Modelica, software developed by the institutes RISC and SCCH from Hagenberg, Austria, and company ENGEL from Schwertberg, Austria. We describe the principal mathematical object used in physical models – differential-algebraic equations, and their implementation and numerical integration.

In the current stage of development, the main problem we have to cope with is the automatic generation of equations due to the so-called *flow* variables – variables which, like current in electronic circuits, intervene in Kirchoff laws, or similar physical laws.

In parallel with the development of technology, efficient software tools dedicated to modeling and simulation have been created. It is impossible to design extremely complex engineering systems – like power plants, or space shuttles – without having them first modeled on the computer. The increasing complexity of models requires advanced mathematical machinery. Pure algebraic equations, or differential equations, are not always sufficient. Most of the physical systems have hybrid mathematical description – the so-called *state variables* (which occur in differential equations) are connected by means of algebraic constraints. Such systems are called *systems of differential-algebraic equations* (DAEs).

Scientists began to study DAEs in the early '60s, and in 1971 Gear [1] gives a practical numerical method for integrating certain classes. Perhaps the most widely used algorithm is Linda Petzold's DASSL [3]. Since then, such algorithms have been included in all modeling and simulation packages.

Among the languages that have been designed for modeling we mention only a few: Allan, Dymola, ObjectMath, Omola. In 1996 was initiated the effort to design a new modeling language, Modelica, within an action of the ESPRIT project "Simulation in Europe Basic Research Working Group". The first version used in actual applications was finished in December 1999.

Currently there are two Modelica simulation environments: Dymola from Dynasim, and MathModelica from MathCore.

In 1997 started a cooperation between RISC and company Engel, regarding the possibility to model injection molding machines, produced by the latter. The difficulty came from the high diversity of these machines, which, in fact, are configured individually for each customer. The initial goal of the project, whose name is `VirtMould`, was to provide tools to define easily models of different machines, and graphical interfaces to simulate their functioning.

Modelica was chosen as the language in which the models should be described. Due to the complexity of the models, expressed in the number of variables and equations, it was not possible to use any of the existent Modelica compilers. Therefore a compiler for a subset of Modelica language was implemented, and later extended and enhanced.

The project was developed in C++ – a very good candidate, due to its similarities with Modelica.

Clearly, numerical integrators for DAEs have been included in the system. However, only explicit DAEs were allowed. Moreover, there was not possible to use the so-called *flow* variables (they will be introduced in section 3), thus restricting significantly the area of applicability.

The problem of flow variables is connected to the problem of *DAEs in general form*. Indeed, the equation generated by the existence of flow variables cannot be immediately put in explicit form; sometimes, the DAE system which results cannot be explicitized at all.

All these suggest a symbolic pre-processing of the equations that occur in the model.

In the next section we give a short exposition about DAEs, and a few examples of physical systems that are modeled using DAEs. Section 3 contains a few facts related to Modelica language. In section 4 we discuss the problem of flow variables and implicit DAEs.

2 Differential-algebraic equation: short introduction

In all what follows, x, y, z, X, Y, Z will be functions of variable t,

$$x, y, z, X, Y, Z : R- > Rn, n \geq 1.$$

With f, g, h, F, G, H we denote smooth functions of more than one variable; these functions will be used to define the equations.

We recall that the general form of an ordinary differential equations system (ODE) is:

$$F(t, y, y') = 0,$$

where the Jacobian $\frac{\partial F(t,u,v)}{\partial v}$ is nonsingular.

The general (implicit) form of a differential-algebraic equations system is the same:

$$F(t, y, y') = 0;$$

this time, however, F has singular Jacobian matrix.

There are a few semi-explicit forms of DAEs in the literature:

1.
$$\begin{aligned} y' &= f(t, y, z) \\ 0 &= g(t, y, z). \end{aligned}$$

   Here, z is the vector of *algebraic variables*, and g defines the *algebraic constraints*.

2.
$$\begin{aligned} y' &= f1(y, z) + g1(y, z)u(t), \\ 0 &= f2(y, z) + g2(y, z)u(t). \end{aligned}$$

   Here, u is a known function of t (input variables).

If these two systems are put into implicit form, their Jacobian with respect to the differentiated variables is zero.

It is plausible that a DAE system in semi-explicit form (1) or (2) can be brought to:

$$Y' = F(t, Y)$$

by differentiating and eliminating.

For instance, consider the following system:

$$\begin{aligned} y' &= x, \\ 0 &= y - u(t). \end{aligned}$$

By differentiating the first equation we get $y'' = x'$; from the second equation, we have: $y'' = u''(t)$. Now, if we eliminate y'', we get:

$$\begin{aligned} y' &= x, \\ x' &= u''(t). \end{aligned}$$

Definition. The index of a DAE system is the number of differentiations needed to achieve an ODE system.

The index is considered as a measure of the *singularity* of a DAE system. The symbolic processing that we applied before is called *index reduction*: from a system of index 2 we obtained a system of ODE's, whose index is 0. It may be quite difficult – sometimes impossible – to reduce practically the index of a DAE system.

There is no method for symbolically solving DAE's. Numerical methods should always be applied with care, due to the inherent stiffness of DAE's.

When explicit forms of the equations are known, Euler- or Runge-Kutta-like methods can be applied. For the general case, the best algorithm seems to be DASSL, developed by Linda Petzold [3].

We exemplify the application of DAEs by taking the case of a simple electronic circuit (see Fig. 1).

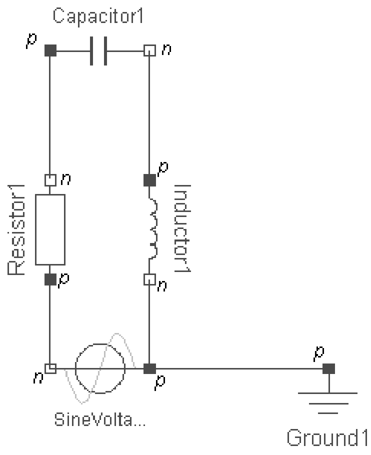

Figure 1: Simple electronic circuit.

The equations corresponding to the components are:

- resistor:
$$R_r * i_r = v_r = v_{p_r} - v_{n_r};$$
- inductor:
$$L_i * i_i' = v_i = v_{p_i} - v_{n_i};$$
- capacitor:
$$C_c * v_c' = i_c;$$
$$v_c = v_{p_c} - v_{n_c};$$
- source:
$$v_s = A * sin(2 * \pi * f * t) = v_{p_s} - v_{n_s};$$
- ground:
$$v_g = 0.$$
- additionally (Kirchoff laws):
$$\begin{aligned} i_s &= -(i_c + i_i + i_r); \\ i_g &= i_c + i_i + i_r + i_s; \\ v_{n_i} &= v_g = v_{p_s}; \\ v_{p_r} &= v_{n_s}; \\ v_{n_r} &= v_{p_c}; \\ v_{n_c} &= v_{p_i}. \end{aligned}$$

Here, i is the notation for current, v for voltage (or voltage drops), R for resistance, L for inductance, and C for capacity. In the formula of the source, A denotes the amplitude of the sine wave, and f the frequency.

Except from the ground, all components have two pins: p (positive) and n (negative), at which voltage can also be measured. Thus, e.g., v_{n_i} denotes the voltage at the negative pin of the inductor.

The capacitor and the inductor introduce differential equations, whereas all other components have algebraic relations among variables.

In the following section we show how this circuit can be modeled using Modelica language.

3 Modelica: a language for modeling

It is not our intention to give here a detailed exposé of Modelica characteristics. We will rather focus on some key elements, necessary to understand the difficulties to implement its compiler.

As presented in the Modelica Tutorial [4],

> Modelica ... is an object-oriented language dedicated to modeling large, complex and heterogeneous physical systems.

It allows high level modeling by composition of components; also, detailed models of components can be created, using the describing equations. Perhaps one of the most important features is that it allows discontinuous variables; the discontinuities are handled by generating *events*.

The feature relevant for our approach is the possibility to introduce communications among components via connections. We make it apparent by giving a Modelica description of the circuit in Figure 1.

There is something common to all components: they have *pins*:

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

connector Pin
   Voltage v;
   flow Current i;
end Pin;
```

A pin is a `connector`, i.e., an object to take part in connections, as we will see.

The keyword **flow** is used to characterize special properties of an object; flow variables are, for instance, the current in electronic circuits, flow rates in piping networks, or forces and torques in mechanical systems. These variables must be taken into account when generating Kirchoff laws or their equivalents.

A component is now something with two pins:

```
partial model OnePort
   Current i;
   Voltage v;
   Pin p;
   Pin n;
equation
   v = p.v - n.v;
   0 = p.i + n.i;
   i = p.i;
end OnePort;
```

`OnePort` is a *partial* model: it cannot participate as it is in larger systems; it can only be *extended*, into a new component.

The equations describe the potential drop and the current law over the component.

Now every component will be a `OnePort`, and will have a few equations on its own:

```
model Capacitor
   extends OnePort;
   parameter Capacitance C=1;
Equation
   i = C*der(v);
end Capacitor;
```

Similarly for the source, resistor and inductor.

As for the ground, it is not derived from `OnePort`:

```
model Ground
  Pin p;
equation
  p.v = 0;
end Ground;
```

Now the connected circuit will be:

```
class LRCSeries
   Ground Ground1;
   Resistor Resistor1;
   Capacitor Capacitor1;
   Inductor Inductor1;
```

```
   SineVoltage SineVoltage1;
equation
   connect(Resistor1.p, SineVoltage1.n) ;
   connect(Inductor1.n, SineVoltage1.p) ;
   connect(Inductor1.p, Capacitor1.n) ;
   connect(Resistor1.n, Capacitor1.p) ;
   connect(SineVoltage1.p, Ground1.p) ;
end LRCSeries;
```

Notice that a Modelica compiler should realize the fact that the `connect` clauses link components that share flow variables, and therefore Kirchoff laws (or their equivalent in other domains of physics) must be generated, in order to obtain a faithful mathematical description of the system.

Also, a closer inspection of the equations of `OnePort` reveals the fact that the equations are not always explicit. The second equation, $0 = p.i + n.i$, cannot be put *a priori* in explicit form, because it is not known from the beginning the order in which the variables are computed or evaluated.

These problems, and other, are discussed in the following section.

4 Implicit equations and flow variables

Normally, a `connect` clause establishes the fact that the variables of the connected objects are common, i.e., they will always have the same value. On the other hand, if flow variables occur in connectors, `connect` clauses must generate some extra equations, always algebraic (polynomial).

Now these equations are in general *implicit*. Since it is not known from the beginning which variables are computed (evaluated) at which moment of the program execution, it is not possible to find the variable that will be explicitized.

A model, even simple, usually has lots of variables: for the circuit discussed along the paper, remember that every component has one or two pins, and every pin has its own current and voltage. The overall number of variables exceeds 20. There should be, therefore, (at least) as many equations. It is not an easy task to write down the system by hand. Clearly, a lot of variables can be immediately eliminated, but this requires *symbolic processing*.

The main problem is to bring the set of equations to a form that can be solved numerically, if possible by keeping them *within* the declarations of the components to which they belong.

One possible strategy is to apply a *flattening* of equations, i.e., to take them from the individual components and create a big system. Every variable will be prefixed (or suffixed) to identify the component from which it comes. Dymola proceeds in this way; here is an excerpt from the C code generated by Dymola from the model of the circuit:

```
Resistor1_v = Resistor1_R*Inductor1_i;
SineVoltage1_signalSource_outPort_signal_1 =
   SineVoltage1_signalSource_p__offset_1
   +(IF LessTime(SineVoltage1_signalSource_p__startTime_1, 0)
     THEN 0
     ELSE
      SineVoltage1_signalSource_p__amplitude_1 *
      sin(6.28318530717959*SineVoltage1_signalSource_p__freqHz_1
      * (Time-SineVoltage1_signalSource_p__startTime_1)
      +SineVoltage1_signalSource_p__phase_1));
Resistor1_n_v =  -(Resistor1_v+
   SineVoltage1_signalSource_outPort_signal_1);
```

```
Inductor1_v = Resistor1_n_v-Capacitor1_v;
Inductor1_der_i =
   divmacro(Inductor1_v,"Inductor1.v",Inductor1_L,"Inductor1.L");
Capacitor1_der_v =
   divmacro(Inductor1_i,"Inductor1.i",Capacitor1_C,"Capacitor1.C");
```

The final system can be relatively easy processed symbolically to eliminate some variables, and to order and explicitize (if possible) the equations. In fact the previous equations are the final form of the system, after Dymola did all necessary reductions and/or explicitizations. Systems in such form can be passed without any problems to numerical integrators.

Another strategy is be to keep the equation local to the components. However, some of these equations are inherently implicit, and their evaluation depends on other components.

In any case, the compiler should keep track of the flow variables, and all variables that are indirectly related to them (they occur in the same equations). The scheme which should be followed is explained in the following:

1. when a `connect` clause of two components with flow variables is detected:
   - there can be more `connect` clauses that affect the same connector: *gather all* `connect` *clauses in which a connector with a flow variable occurs*;
   - *generate the Kirchoff laws corresponding to that set of connections.*
     Example: a junction in an electronic circuit (see Fig. 2.

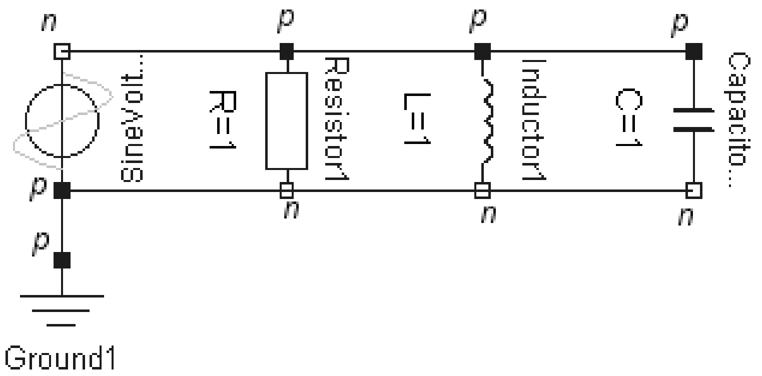

Figure 2: Simple electronic circuit with components connected in parallel.

The negative pin of the source is connected to the positive pins of the resistor, inductor and capacitor:

```
connect(Source1.n, Resistor1.p);
connect(Source1.n, Capacitor1.p);
connect(Source1.n, Inductor1.p);
```

The corresponding Kirchoff law is:

```
0 = Source1.n.i-(Resistor1.p.i+Capacitor1.p.i+Inductor1.p.i);
```

For the positive pin of the source, we have the following equation:

```
0 = Ground1.p.i-(Resistor1.n.i+Capacitor1.n.i+Inductor1.n.i)
    +Source1.p.i;
```

- *isolate all equations of the components that are directly or indirectly related to variables occuring in the equations generated by the connections.*

  At this moment, there is a set S of equations at the level of the connections, and in each component might have remained some equations that are not related to the flow variables, and presumably are in explicit form.

- *explicitize symbolically as many equations in S as possible; let S_e and S_i be the subsets of explicit, respectively implicit equations*;
- *prepare S_e and S_i for numerical integrators.*

We foresee an important advantage of this strategy:

- since equations that are not related to flow variables remain in the components, the system that contains the equations generated by connections and the equations related to flows is smaller than in the flattening case, and thus much easier to process symbolically.

On the other hand, this strategy allows to preserve, in a certain measure, the *granularity* of the model: Indeed, it happens very often that a subsystem of an engineering system contributes to the overall scheme through a well defined set of inputs and outputs, whereas inside, it contains a complex network of components, involving flow parameters as well. In such case, the *flattening* is local to that subsystem; the model of the subsystem is seen as a black box, and the equations remain local.

5 The need for symbolic processing

It is not guaranteed that the set of equations obtained after flattening are in explicit form, or even easily explicitable. In fact, the form of the resulting system can be as general as possible.

There are two issues connected to symbolically processing of that system:

1. The system may be a high index DAE; if this is the case, one cannot ensure the stability of numerical integrators.

   Most of the times, an *index reduction algorithm* is applied to this system. The most commonly used is *Pantelides algorithm for index reduction* (see [2]).

2. Effectively computing the solution of the DAE requires:

   (a) explicitization of the equations; this means solving symbolically the system - or at least those equations for which this is possible.

   (b) preparing the non-explicitable equations for numerical algorithms. Most often, this means computing the Jacobian of the corresponding sub-system.

6 Differential-algebraic equations in VirtMould

The stable version of project VirtMould accepts only DAEs in explicit form. A new version, which will work with DAEs in general form, is currently in the testing phase.

We have restricted the symbolic treatment of DAEs to a low-level equation solving by variable isolation. For more complicated equations, we have used a Mathematica kernel. Our software runs Mathematica in batch mode, and communicates with it via external files. The set of equations and the unknowns are written in a text file that is given as input to Mathematica. The output, collected from another text file, consists in:

- either the input equations in explicit form, if explicitization was possible;
- or the Jacobian of the input equations, if Mathematica was not able to compute a symbolic solution.

Further, for the algebraic (i.e., non-differential) non-explicitable equations a number of numerical solvers are built. For the overall system of DAE's, VirtMould provides Euler and Runge-Kutta integrators.

References

[1] C. W. Gear - *The simultaneous numerical solution of differential-algebraic equations*, IEEE Trans. Circuit Theory, CT-18 (1971) 89-95.

[2] C. Pantelides - *The consistent initialization of differential algebraic systems*, SIAM J. Scientific and Statistical Computing, 9:213-231, 1988.

[3] Linda Petzold - *A description of DASSL: a differential/algebraic system solver*, SANDIA Report SAND82-8637 (1982).

[4] *Modelica™ - A unified object-oriented language for physical systems modeling, Tutorial*, Modelica association, 2000.