Mixed Domain Modeling in Modelica

Christoph Clauss[1] (clauss@eas.iis.fhg.de)
Hilding Elmqvist[3] (Elmqvist@Dynasim.se)
Sven Erik Mattsson[3] (SvenErik@Dynasim.se)
Martin Otter[2] (Martin.Otter@DLR.de)
Peter Schwarz[1] (schwarz@eas.iis.fhg.de)

[1] Fraunhofer Institute for Integrated Circuits, Design Automation Depart. EAS, Dresden, Germany
[2] German Aerospace Center (DLR), Oberpfaffenhofen, Germany
[3] Dynasim AB, Lund, Sweden

Modelica is a language for convenient, component oriented modeling of physical systems. In this article an overview of the language, available libraries, the Dymola simulator and some industrial applications is given. Additionally, a comparison of Modelica with VHDL-AMS is presented.

1. Introduction

Modelica® is a non-proprietary specification of an object-oriented language for modeling of large, complex, and heterogeneous physical systems. It is suited for multi-domain modeling, for example, mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control subsystems, process oriented applications, and generation and distribution of electric power.

Models in Modelica are mathematically described by *differential, algebraic* and *discrete equations*. The set of equations does not need to be manually solved for a particular variable. A Modelica tool will have enough information to decide that automatically. Modelica is designed such that available, structural and symbolic algorithms can be utilized to enable efficient handling of large models having more than hundred thousand equations. Modelica is suited and used for hardware-in-the-loop simulations.

The Modelica design effort was started in September 1996 and the first draft language specification was available after one year. After 19 three-day meetings version 1.3 of the language specification was finished in December 1999. This was the first version used in actual applications. In December 2000, version 1.4, and in January 2002, version 2.0 were released.

In February 2000, a non-profit, non-governmental organization, called Modelica Association, was founded, for the further development, promotion and application of the Modelica language. The association has its seat in Linköping, Sweden and owns and administrates incorporeal rights related to Modelica, including but not limited to trademarks (such as the registered Modelica trademark), the Modelica Language Specification, Modelica Standard Libraries, etc., which should be freely available for the promotion of industrial development and research. More information on Modelica is available from http://www.Modelica.org.

2. Modelica Fundamentals

Modelica supports both high level modeling by composition and detailed library component modeling by equations. Models of standard components are typically available in model libraries. Using a graphical model editor, a model can be defined by drawing a *composition diagram* (also called schematics) by positioning icons that represent the models of the components, drawing connections and giving parameter values in dialogue boxes. Constructs for including graphical annotations in Modelica make icons and composition diagrams portable between different tools.

An example of a composition diagram of a simple motor drive system is shown in Figure 1. The system can be broken up into a set of connected components: an electrical motor, a gearbox, a load and a control system. The textual representation of this Modelica model is (annotations describing the graphical placement of the components and the connection lines are not shown):

```
model MotorDrive
  PID      controller;
  Motor    motor;
  Gearbox  gear    (n=100);
  Inertia  inertia(J=10);
equation
  connect(controller.outPort, motor.inPort);
  connect(controller.inPort2, motor.outPort);
  connect(gear.flange_a     , motor.flange_b);
  connect(gear.flange_b     , inertia.flange_a);
end MotorDrive;
```

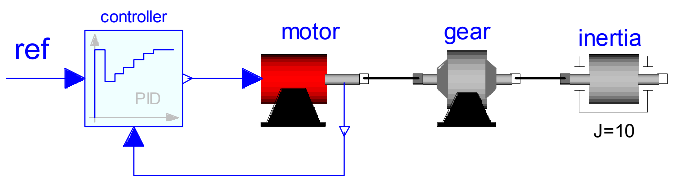


Figure 1: A model of a simple motor drive system.

It is a composite model which specifies the topology of the system to be modeled in terms of components and connections between the components. The statement "`Gearbox gear(n=100);`" declares a component `gear` of model class `Gearbox` and sets the value of the gear ratio, `n`, to `100`. A component model may be a composite model to support hierarchical modeling. The composition diagram of the model class `Motor` is shown in Figure 2. The meaning of connections will be discussed below as well as the description of behavior on the lowest level using mathematical equations.

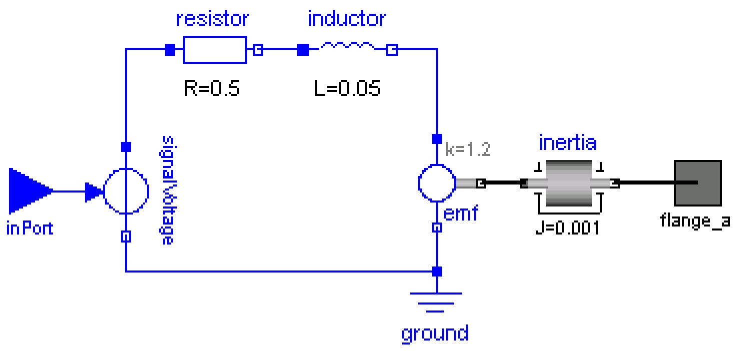


Figure 2: The model Motor.

Physical modeling deals with the specification of relations between physical quantities. For the drive system, quantities such as angle and torque are of interest. Their types are declared in Modelica as

```
type Angle  = Real(quantity="Angle" , unit="rad", displayUnit="deg");
type Torque = Real(quantity="Torque", unit="N.m");
```

where `Real` is a predefined type, which has a set of attributes such as name of quantity, unit of measure, default display unit for input and output, minimum, maximum, nominal and initial value. The Modelica Standard Library, which is an intrinsic part of Modelica, includes about 450 of such type definitions based on ISO 31-1992.

Connections specify interactions between components and are represented graphically as lines between connectors. A connector should contain all quantities needed to describe the interaction. Voltage and current are needed for electrical components. Angle and torque are needed for drive train elements:

```
connector Pin                  connector Flange
  Voltage      v;                Angle       phi;
  flow Current i;                flow Torque tau;
end Pin;                       end Flange;
```

A connection, `connect(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins such that they form one node. This implies two equations, `Pin1.v = Pin2.v` and `Pin1.i + Pin2.i = 0`. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current law saying that the current sums to zero at a node. Similar laws apply to mass flow rates in piping networks and to forces and torques in mechanical systems. The sum-to-zero equations are generated when the prefix **`flow`** is used in the connector declarations. In order to promote compatibility between different libraries, the Modelica Standard Library includes also connector definitions in different domains.

An important feature for building reusable descriptions is to define and reuse **`partial`** models. A common property of many electrical components is that they have two pins. This means that it is useful to define an interface model class `OnePort`, that has two pins, `p` and `n`, and a quantity, `v`, that defines the voltage drop across the component.

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

The equations define common relations between quantities of a simple electrical component. The keyword **partial** indicates that the model is incomplete and cannot be instantiated. To be useful, a constitutive equation must be added. A model for a resistor extends `OnePort` by adding a parameter for the resistance and Ohm's law to define the behavior.

```
model Resistor "Ideal resistor"
  extends OnePort;
  parameter Resistance R;
equation
  R*p.i = v;
end Resistor;
```

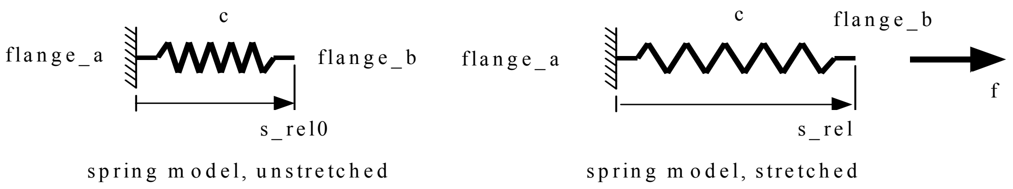
A string between the name of a class and its body is treated as a comment attribute. Tools may display this documentation in special ways. The keyword parameter specifies that the quantity is constant during a simulation experiment, but can change values between experiments.

The most basic Modelica language elements have been presented. Modelica additionally supports mathematical functions, calling of external C and FORTRAN functions, control constructs, hierarchical data structures (record), replaceable components, safe definition of global variables and arrays, utilizing a Matlab like syntax. The elements of arrays may be of the basic data types (Real, Integer, Boolean, String) or in general component models. This allows convenient description of simple, discretized partial differential equations. A unique feature of Modelica is the handling of discontinuous and variable structure components such as relays, switches, bearing friction, clutches, brakes, impact, sampled data systems, automatic gearboxes etc., see, e.g., [5]. Modelica has introduced special language constructs allowing a simulator to introduce efficient handling of events needed in such cases. Special design emphasis was given to synchronization and propagation of events and the possibility to find consistent restarting conditions. Finally, a powerful package concept, similar to the Java package concept, is available to structure large model libraries and to find a component in a file system giving its hierarchical Modelica class name. The complete Modelica specification can be found at the Modelica homepage [7].

3. Modelica and VHDL-AMS

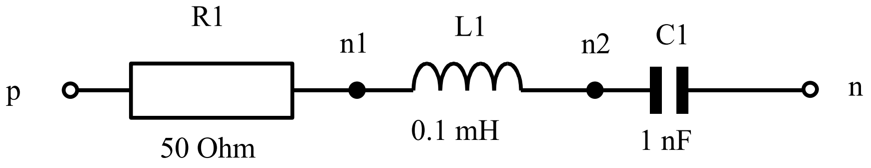
Another well established modeling language for large, complex, and heterogeneous systems is VHDL-AMS (Very high speed integrated circuit Hardware Description Language – Analog and Mixed-Signal extensions) [1,11]. A similar role plays Verilog-AMS but this language will not be considered here. For categorizing both languages, some similarities and differences between VHDL-AMS and Modelica are shown next.

Using a spring model with a linear characteristic a first impression of both languages is given. Both languages clearly distinguish between *interface* description and *internal* structural or behavioral description. This allows the description of complex models by composition of simpler partial models.



```
  -- VHDL-AMS                             // Modelica
  ENTITY spring IS                        model Spring
    GENERIC (s_rel0: real := 0.0,           import SI=Modelica.SIunits;
                    c: real := 1.0);        Flange flange_a, flange_b;
    PORT (TERMINAL flange_a,                parameter Real c(min=0) = 1;
          flange_b: kinematic);             parameter SI.Distance s_rel0 = 0;
  END ENTITY spring;                        SI.Distance s_rel;
                                          equation
  ARCHITECTURE simple OF spring IS              0 = flange_a.f + flange_b.f;
    QUANTITY s_rel ACROSS f                 s_rel = flange_b.s ñ flange_a.s;
       THROUGH flange_b TO flange_a;        flange_b.f = c*(s_relñs_rel0);
  BEGIN                                   end Spring;
    f == c * (s_rel - s_rel0);
  END ARCHTITECTURE simple;
```

The *structural* description with VHDL-AMS is a device oriented (branch oriented) netlist: it is noted which *nodes* and which *component ports* are connected together. In Modelica the connection of pins is described and nodes are not used explicitely.



```
-- VHDL-AMS                                   // Modelica
  ENTITY rlc IS                                 model RLC
    PORT (TERMINAL p, n: ELECTRICAL);             Pin p, n;
  END ENTITY rlc;                                 Resistor  R1 (R=50.0);
                                                  Inductor  L1 (L=0.1e-3);
  ARCHITECTURE structural OF rlc IS               Capacitor C1 (C=1.0e-9);
    TERMINAL n1, n2: ELECTRICAL;                equation
  BEGIN                                           connect (p, R1.p);
    R1: Resistor  GENERIC MAP (R=>50.0)           connect (R1.n, L1.p);
                  PORT MAP (p=>p, n=>n1);         connect (L1.n, C1.p);
    L1: Inductor  GENERIC MAP (L=>0.1E-3)         connect (C1.n, n);
                  PORT MAP (p=>n1, n=>n2);      end RLC;
    C1: Capacitor GENERIC MAP (C=>1.0E-9)
                  PORT MAP (p=>n2, n=>n);
  END ARCHTITECTURE structural;
```

Both languages support multi domain and mixed mode modeling. Designed for use in electronics, VHDL-AMS supports all necessary simulation modes used in electronic applications: DC (direct current, operating point), transient, AC (alternating current, frequency domain), and small-signal noise analysis. Modelica has a slightly different philosophy: Before any operation is carried out, initialization takes place to compute a consistent initial point using all model equations and *additional initial equations* (e.g., setting initial derivatives to zero, if a steady-state start is desired; or setting some states to predefined values and only part of the derivatives to zero, as it is, e.g., needed to initialize a flying aircraft which does never has a steady state if not crashed) to define the initial point uniquely. Based on this initial point *transient analysis* or *linearization* can be carried out. The linearized model may be used by a tool, such as Dymola or Matlab, to perform linear analysis calculations, like eigenvalue, frequency response or noise analysis.

Modelica extensively allows object oriented features like the definition of classes, inheritance and polymorphism, whereas VHDL-AMS uses overloading of operators. A key feature of Modelica is its library concept (package concept) which uses ideas from the Java programming language whereas VHDL-AMS uses the package concept of the programming language Ada.

A VHDL-AMS model description is purely textual and does not include graphical information. The model text may be generated by powerful schematic entry tools in design frameworks. As a consequence, an exchange of VHDL-AMS models between different simulators is portable only on the textual level. In Modelica all details of the graphical layout of a composition diagram on screen and on printer is standardized, see [7], chapter 7 "Annotations for graphical objects". This close integration of the visual representation influences the core language design: For example, it is common in programming languages to require that a variable has to be declared before being used. In a visual environment this does not make much sense because the components on screen and additional declarations, e.g., parameters used for the components, have no particular order. Consequently, Modelica variables may be used in declarations before being declared.

Further differences exist with regard to the analog-digital simulation algorithm:.

VHDL-AMS presupposes the coupling of a discrete event-driven algorithm with a continuous DAE solver, the coupling algorithm is part of the language standard. The analog solver has to determine the actual values of all analog quantities at each event time. The digital signals are calculated by a discrete event-driven algorithm that is part of the VHDL standard. Digital events occur only at time points of a discrete time axis.

In Modelica at all time instants, including events, the whole set of active differential, algebraic and discrete equations is solved as one (algebraic) system of equations, at least conceptually [5]. The advantage is that the synchronization between the continuous and the discrete part is automatically determined at translation time by data flow analysis (= sorting of the equations). This allows, e.g., to figure out non-deterministic behavior before simulation starts. If, e.g., two events accidentally occur at the same time instant, it is still guaranteed that there is a defined evaluation order of the discrete equations (if this is not defined, an error occurs at translation time).. In Modelica, digital events are always time barriers for the continuous integration and hence always slow down the continuous integration. For the formulation of analog-digital conversion both languages have special constructs (VHDL-AMS: 'ramp, 'above, 'slew, break; Modelica: **pre**(..), **edge**(..), **change**(..), **noEvent**(..), if-then-else conditions).

In the design of electronic systems and in the simulation of their environments, VHDL-AMS offers many advantages, e.g., powerful digital and mixed-signal simulators, large model libraries of digital components and complex circuits like microprocessors, and in future the widely used analog SPICE models. Many IC companies require checked VHDL (or Verilog) simulation protocols as part of their quality assurance system. In non-electrical domains, especially with a large part of continuously working subsystems, Modelica offers advantages by its object orientation, handling of complicated modeling situations (e.g., index problems, see section 5 below), the graphic-oriented input language, and many multi-domain libraries.

4. Modelica Libraries

In order that Modelica is useful for *model exchange*, it is important that libraries of the most commonly used components are available, ready to use, and sharable between applications. For this reason, the Modelica Association develops and maintains a growing *Modelica Standard Library*. Furthermore, other people and organizations are developing free and commercial Modelica libraries. For more information and especially for downloading the free libraries, see http://www.Modelica.org/libraries.shtml. Currently, component libraries are available in the following domains:

- About 450 type definitions, such as Angle, Voltage, Inertia.

- Mathematical functions such as sin, cos, ln.
- Continuous and discrete input/output blocks, such as transfer functions, filters, sources.
- Electric and electronic components such as resistor, diode, MOS and BJT transistor.
- 1-dim. translational components such as mass, spring, stop.
- 1-dim. rotational components such as inertia, gearbox, planetary gear, bearing friction, clutch.
- 3-dim. mechanical components such as joints, bodies and 3-dim. springs.
- Hydraulic components, such as pumps, cylinders, valves.
- 1-dim. thermal components, such as heat capacitance and thermal conductor.
- Thermo-fluid flow components, such as pipes with multi-phase flow, heat exchangers.
- Building components, such as boiler, heating pipes, valves, weather conditions, walls.
- Power system components such as generators and lines.
- Vehicle power train components such as driver, engine, automatic gearboxes.
- Flight dynamics components, such as bodies, aerodynamics, engines, atmosphere, wind.
- Petri nets, such as place, simple transition, parallel transition.

Typical composition diagrams in different domains utilizing available libraries are shown in Figure 3.

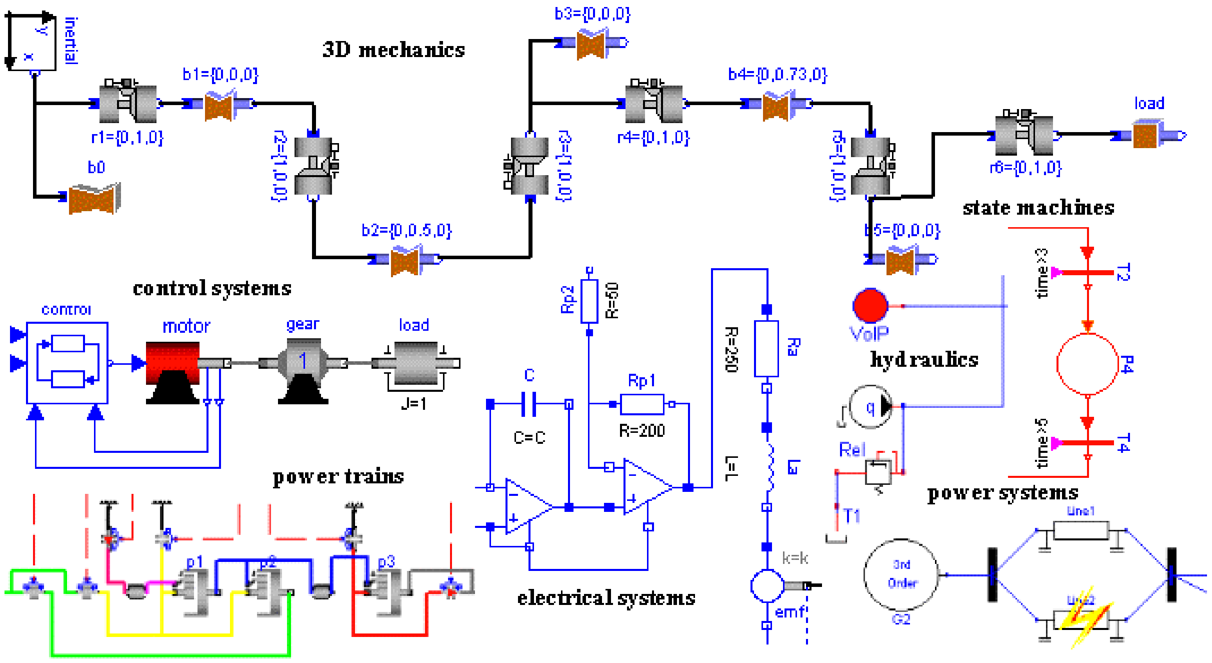


Figure 3: Examples of models built from available Modelica libraries.

Electrical library

In electronics and especially in micro-electronics, the simulator SPICE [4] with its input language is a quasi-standard. SPICE has only a fixed set of device models ("primitives"). These models can be modified by changing their parameters. The transistor models are very sophisticated and are closely integrated into the SPICE simulator. Behavioral modeling by the user is not supported. To model complicated functions, "macromodels" in the form of circuits have to be constructed which use the SPICE primitives as building blocks in a netlist.

To support modeling of electrical subsystems, the Modelica.Electrical.Analog library has been developed. It is a collection of simple electronic device models, independent of SPICE. They are easy to understand, use the object-oriented features of the Modelica language, and can be used as a base of inheritance and as examples for new models. The functionality is chosen in a way that a wide range of simple electric and electronic devices can be modeled. The Modelica.Electrical.Analog library contains at the moment:

- Basic devices like resistor, capacitor, inductor, transformer, gyrator, linear controlled sources, operational amplifiers (opamps).
- Ideal devices like switches (simple, intermediate, commuting), diode, opamp, transformer.

- Transmission lines: distributed RC and RLGC lines, loss-less transmission lines.
- Semiconductors: diodes, MOS and bipolar transistors (Ebers-Moll model).
- Various voltage and current sources.

The numerically critical devices of the library are modeled in a "simulator-friendly" way, e.g. exponential functions are linearized for large argument values to avoid data overflow.

In future, the SPICE models have still to be provided. An interesting approach is presented in [10], where the SPICE models are extracted and translated into Modelica. Another possibility is the call of SPICE models via the foreign function interface. Furthermore, a library of *digital* electronic devices and of composed components, e.g., CMOS building blocks, is under development.

5. The Simulation Problem

In order that the Modelica language and the Modelica libraries can be utilized, a Modelica translator is needed to transform a Modelica model into a form, which can be efficiently simulated in an appropriate simulation environment. The mathematical simulation problem consists of the equations of all components and the equations due to connections. It is a hybrid system of differential-algebraic equations (DAE) and discrete equations. Moreover, idealized elements, such as ideal diodes or Coulomb friction introduce *Boolean unknowns* (e.g., for friction models the Boolean variables describe whether the element is in stuck mode or is sliding). Thus the simulation problem may be a *mixed* system of equations having both *Real* and *Boolean unknowns*. There are no general-purpose solvers for such problems, although there are numerical DAE solvers, which could be used to solve the continuous part.

Modelica was designed such that symbolic transformation algorithms can (and have to) be applied to transform the original set of equations into a form which can be integrated with standard methods. With appropriate symbolic transformation algorithms such as BLT-partitioning (= sorting of equations and variables) and tearing (= "intelligent" variable substitution), it is possible to reduce the number of unknowns visible from the integrators during translation.

When solving an ordinary differential equation (ODE) the problem is to integrate, i.e., to calculate the states when the derivatives are given. Solving a DAE may also include differentiation, i.e., calculate the derivatives of given variables. Such a DAE is said to have high index. It means that the overall number of states of the model is less than the sum of the states of the sub-components. Higher index DAEs are typically obtained because of constraints between models. To support reuse, model components are developed to be “general”. Their behavior is restricted when they are used to build a model and connected to other components. Take as a very simple example, two rotating bodies with inertias J_1 and J_2 connected rigidly to each other.

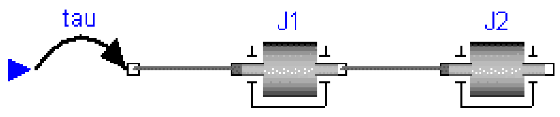


Figure 4: Two rigidly connected rotating bodies.

$$\begin{aligned} \dot{\varphi}_1 &= \omega_1 \\ J_1 \cdot \dot{\omega}_1 &= \tau(t) - \tau_2 \\ \dot{\varphi}_2 &= \omega_2 \\ J_2 \cdot \dot{\omega}_2 &= \tau_2 \\ \varphi_1 &= \varphi_2 \end{aligned}$$

The angles and the velocities of the two bodies should be equal. All four variables cannot be state variables with their own independent start values. The connection equation for the angles, $\varphi_1 = \varphi_2$, must be differentiated twice to get a relation for the accelerations to allow calculation of the reaction torque τ_2. When converting it to ODE form the set of state variables is smaller than the set of differentiated variables.

The reliability of a direct numerical solution is related to the number of differentiations needed to transform the system algebraically into state space form $d\mathbf{x}/dt = f(\mathbf{x},t)$. Today's numerical integration algorithms, such as used by most simulators, can handle systems where equations needed to be at most differentiated *once*. However, reliable direct numerical solutions for non-linear systems are not known if 2 or more differentiations are required (as it is the case in the example above). Furthermore, if mixed continuous and discrete systems are solved, *at event instants* the hybrid DAE must be *initialized*. In this case it is in general not sufficient to just fulfill the original DAE. Instead, also some differentiated equations have to be fulfilled, in order that a consistent initialization is possible. For example, the *model equations* in figure 4 are fulfilled with the initial conditions $\varphi_1(t_0) = d\varphi_1/dt(t_0) = \omega_1(t_0) = d\omega_1/dt(t_0) = \varphi_2(t_0) = 0$; $\tau_2(t_0) = \tau(t_0)$; $d\varphi_2/dt(t_0) = \omega_2(t_0) = 1$; $d\omega_2/dt(t_0) = \tau(t_0)/J_2$.; Without additional information (= the initial conditions have to fulfill also the derivative of the constraint equation $\varphi_1 = \varphi_2$) a numerical method has difficulties to detect that these initial conditions are inconsistent, i.e., the DAE does not have a solution. This indicates that direct numerical solvers have problems at events to determine consistent restart conditions of higher index systems.

Higher index DAEs can be avoided by restricting how components may be connected together and/or include manually differentiated equations in the components for the most common connection structures. The drawback is that (1) physically meaningful component connections may no longer be allowed in the model or (2) unnecessary "stiff" elements have to be introduced in order that a connection becomes possible. For example, the system in Figure 4 has no longer a higher index, if a stiff spring is introduced between the two bodies.

Since most Modelica libraries are designed in a truly object-oriented way, i.e., every meaningful physical connection can also be performed with the corresponding Modelica components, this leads often to higher index systems, especially in the mechanical and thermo-fluid field. Therefore, any reliable Modelica translator must transform the problem before a numerical solution can take place. The standard technique is to (1) use the algorithm of Pantelides [8] to determine how many times each equation has to be differentiated, (2) differentiate the equations analytically, and (3) select which variables to use as state variables (either statically during translation or in more complicated cases dynamically during simulation) with the dummy derivative method of Mattsson and Söderlind [6].

6. The Dymola simulation environment

Dymola [2] from Dynasim (http://www.Dynasim.se) has a Modelica translator which is able to perform all necessary symbolic transformations for large systems (> 100 000 equations) as well as for real time applications. A graphical editor for model editing and browsing, as well as a simulation environment are included (all the model diagrams in the paper are screenshots of Dymola's composition diagram editor).

The traditional approach has been to manually manipulate the model equations to ODE form. Dymola automates all this time-consuming and error-prone work and generates efficient code also for large and complex models. Symbolic processing is a unique feature of Dymola to make simulations efficient. Dymola converts the differential-algebraic system of equations symbolically to state-space forms, i.e. solves for the derivatives. Efficient graph-theoretical algorithms are used to determine which variables to solve for in each equation and to find minimal systems of equations using tearing to be solved simultaneously (algebraic loops). The equations are then, if possible, solved symbolically or code for efficient numeric solution is generated. Discontinuous equations are properly handled by translation to discrete events as required by numerical integration routines.

For high index problems, Dymola deduces automatically which equations to differentiate and differentiates them symbolically. It also selects which variables to use as states. Recall the example of the two rigidly coupled bodies discussed above. Let us make the example a bit more realistic and put a gearbox with fixed gear ratio n between the two bodies. Dymola differentiates the position constraints twice to calculate the reaction torque in the coupling, and it is sufficient to select the angle and velocity of either body as state variables. The constraint leads to a linear system of simultaneous equations involving angular accelerations and torques. The symbolic solution contains a determinant of the form “$J_1 + n^2J_2$”. Dymola thus automatically deduces how inertia is transformed through a gearbox.

Convenient interfaces to Matlab and the popular block diagram simulator SIMULINK exist. For example, a Modelica model can be transformed into a SIMULINK S-function C mex-file, which can be simulated in SIMULINK as an input/output block. This allows to model a plant very conveniently with Modelica (which is tedious or impossible in Simulink for larger systems), utilize it as one input/output block in SIMULINK, realize controllers and filters in SIMULINK and utilize all the analysis and design capabilities of Matlab.

7. Industrial applications with Modelica

7.1 Real-time Simulation of Industrial Robots with Dymola

In Figure 5 an application is shown where a standard robot control panel of the robot manufacturer KUKA is used to control a "virtual" robot, i.e., the "real" robot usually controlled is replaced by a real-time simulation of the robot dynamics. The simulation result is visualized online on a monitor.

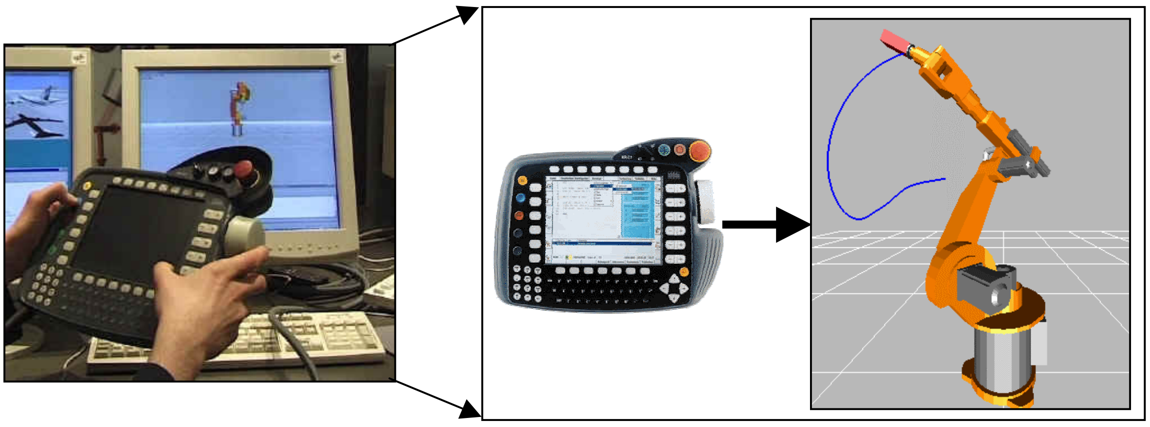


Figure 5: Hannover fair demonstration of robot real-time simulation controlled by KUKA control panel.

The Modelica model of the structurally similar Manutec robot r3 is shown in Figure 6. It is available in the public domain Modelica library as ModelicaAdditions.MultiBody.Examples.Robots.r3.robot.

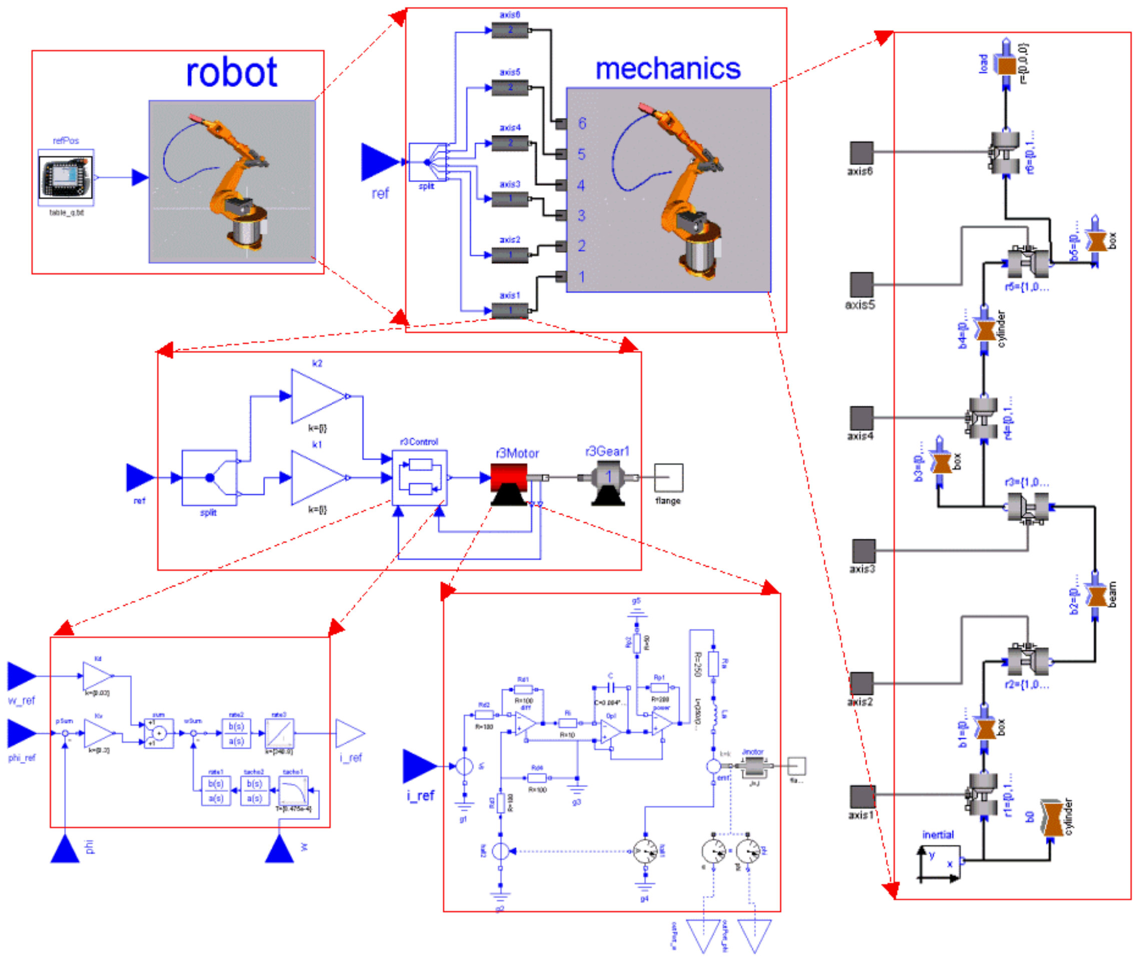


Figure 6: The Manutec r3 robot model.

This Modelica model describes an industrial robot with six degrees of freedom. The model is composed of basic mechanical components such as joints and bars as shown in the right part of Figure 6. At every joint, a drive train is present. Each drive train contains a motor, a gearbox and an actuator as well as a control system. The elasticity of the gears of the first three joints is modelled by one spring for each gearbox. The elasticity of the last three joints is neglected. Figure 6 also shows the model of the motor and the actuator of a joint. This component is defined, most naturally, as an electrical circuit. The control system for one driving axis is defined in block diagram format.

One challenge is real-time simulation of the robot for hardware-in-the-loop testing of robot control hardware and software. For off-line simulations it is interesting to reach the stop time as fast as possible. Stiff solvers may spend a lot of effort to calculate transients but have the ability to speed up once the transients have died out. In real-time simulations it is necessary to produce values at a certain sampling rate and it is the worst case that sets the limit for speed.

The robot contains components with varying model structure, such as Coulomb friction, and systems with slow and fast dynamics, such as the elasticity of gearboxes or controllers (fast dynamics) and 3-dim. mechanics (slow dynamics). The model has 5963 unknowns. After Dymola's elimination of constant and alias variables at translation, 932 nontrivial and time-varying variables remain. For explicit ODE methods there is one linear equation system of size six to solve. It corresponds to the inversion of the mass matrix. Dymola has solved all other equation systems symbolically.

When using explicit Euler, a step size of 0.05 ms has to be selected to achieve stable behavior. The fastest eigenvalues of the linearization of the system are about 7000 in magnitude. Even for a high performance processor the simulation speed would be about two times slower than real time. Typically, the fastest modes are not excited to a degree that it is necessary to resolve them for the intended purpose. In such cases the problem is referred as stiff. The implicit Euler method solves the numerical stability problem and allows larger step sizes to be used. It is the accuracy required that restrict how large step sizes that can be used. Using the implicit Euler method, on the other hand, implies that a nonlinear system of equations needs to be solved at every step. The size of this system is at least as large as the size of the state vector, n. Solving large nonlinear systems of equations in real-time is somewhat problematic because the number of operations is $O(n^3)$ and the number of iterations might vary for different steps. Reducing the size of the nonlinear problem is advantageous. Due to the hybrid nature of the system, the Jacobian of the nonlinear system can change drastically between steps. This makes it difficult to apply methods relying on Jacobian updating.

A simulation using Implicit Euler with a step size of 1 ms runs much slower then real-time and the position errors and velocity errors are too high.

In order to obtain real-time capability special symbolic and numerical model processing and simulation code generation methods have been developed for Dymola. The method of inline integration was introduced to handle such cases. The discretisation formulas of the integration method are combined with the model equations and structural analysis and computer algebra methods are applied on the augmented system of equations. For a robotic model with 66 states, Dymola reduces the size of the nonlinear system to only 6 with no additional local equation systems. The equation systems from discretizing the drive trains and their controllers are linear and Dymola is able to solve them symbolically.

Table 1: Performance of the methods for the robot problem that is simulated for 1 s using a Pentium 4, 1.6 GHz processor.

	Inl. Expl. Euler	**Inl. Impl. Euler**	**Inl. Impl.RK 3**	**Inl. Impl. RK 3**
Step size [ms]	0.05	1	5	10
Pos. error [mm]	0.1	3	0.1	0.4
Vel. error [mm/s]	5	20	5	20
CPU time [s]	1.97	0.16	0.11	0.06

The resulting execution times and maximum position and velocity errors compared to a reference solution calculated using DASSL are shown in Table 1. When judging the errors it may be of interest to know that the robot is of meter size and the maximum speeds are 2-4 m/s. For easy interpretation of the execution times the problem was simulated for one second. It means that if the CPU time is less than one second, the simulation runs faster than real-time. When using explicit Euler the simulation takes 1.97 seconds, i.e. slower than real-time. The solution has good accuracy, but the computational burden is high.

The inlined-implicit methods run all faster than real-time with step size 1 ms. The stability of the inlined implicit Euler is maintained for the stiff system, i.e. a longer step size could be used. However, with larger errors. To achieve desired accuracy, a higher order method such as Runge-Kutta of 3rd order was needed. The inlined implicit RK3 with step size 5 ms gives a solution with the same accuracy only using 0.11 CPU seconds which is only 6% of the effort for the explicit Euler method.

7.2 Hardware-in-the-loop Simulation of Automatic Gearboxes

Hardware-in-the-loop simulation of automatic gearboxes to tune gearshift quality is a major application of Modelica and Dymola for automotive companies. A gearbox is modeled by using predefined components such as planetary gear sets, clutches and brakes, see Figure 7 for a Modelica composition diagram and Figure 8 for an animation view with Dymola's animation engine.

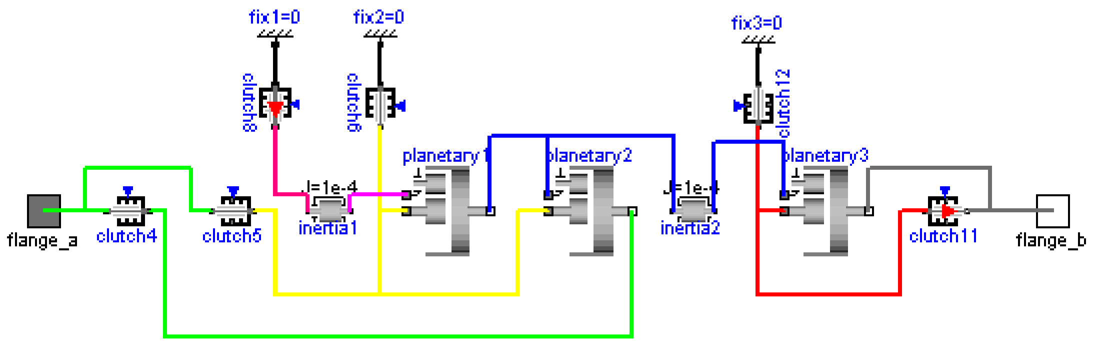


Figure 7: Modelica composition diagram of automatic gearbox model.

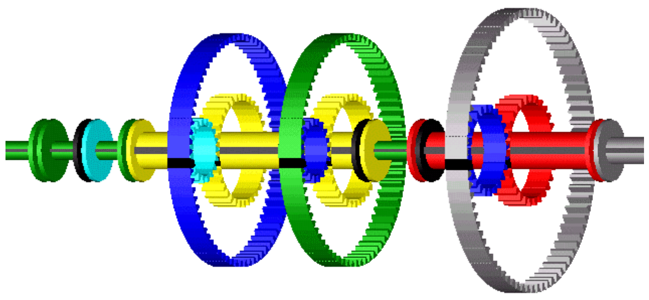


Figure 8: Animation view of automatic gearbox model.

A typical electronic control unit, ECU, which generates the gearshift signals has a sampling time of 5 - 10 ms implying that the simulation needs to produce values in this time frame or faster. A model is built by composing planetary wheel-sets, shafts, clutches, and free-wheels from the Modelica libraries. The resulting mathematical model is a mixed system of Boolean equations and differential-algebraic equations with hundreds of unknown variables. There are no general-purpose solvers for such a problem. There are numerical DAE solvers, which could be used to solve the continuous part, but they are at least 100 times too slow.

The gearbox model has varying structure depending on whether a clutch is sliding or not. It is different combinations of bodies that are rotating together. For a given configuration the simulation problem is much simpler. The traditional approach has been to manually manipulate the equations for each mode of operation and exploit special features. Dymola automates all this work. It takes Dymola less than a minute to translate the Modelica model of the gearbox into efficient simulation code. Already in 1999, a 500 MHz DEC Alpha processor from dSPACE evaluated one Euler step including a possible mode switch in less than 0.18 ms. Today a modern PC processor is 3-4 times faster.

For example, the automatic gearbox ZF4HP22 from ZF Friedrichshafen has six switching elements, which means that there are $2^6 = 64$ possible configurations of the system. It is indeed possible to treat all the 64 cases individually. A major automotive company uses a gearbox with 11 clutches having $2^{11} = 2048$ possible configurations. Fortunately, it turns out that for a typical drive cycle only 10 to 30 modes are active. Dymola includes a facility to find out which variables shall be considered to define a mode. Off-line simulations are run to find out which configurations that are active. These are used in the next translation of the model to generate special code for the algebraic loops of each used combination of clutch locking combinations and for the general case as well.

8. Conclusion

The first useful Modelica environment, Dymola, has been available since beginning of year 2000. Since then an increasing number of modelers is using Modelica both in research and in industrial applications. The development of the Modelica language and of Modelica libraries is still an ongoing effort in order to improve the language and the libraries based on the gained experience. In the future Modelica will probably also be more and more used not only for modeling but also for programming of algorithms, because it turned out that the combination of the Matlab-like array syntax in Modelica functions and the strong typing semantic seems to allow both Matlab-type convenience of algorithm definition and C-type run-time efficiency of the translated algorithm.

9. References

[1] Christen, E.; Bakalar, K.: **A hardware description language for analog and mixed signal applications**. IEEE Trans. CAS-II 46 (1999) 10, 1263-1272.

[2] Dymola: **Dynamic Modeling Laboratory**. Dynasim AB, Lund, Sweden, http://www.Dynasim.se

[3] Elmqvist, H.; Mattsson, S.E.; Olsson, H.: **New methods for hardware-in-the-loop simulation of stiff models**. Modelica'2002, Oberpfaffenhofen, March 18-19, pp. 59-64, 2002 (download from http://www.Modelica.org/Conference2002/papers.shtml)

[4] Johnson, B.; Quarles, T.; Newton, A.R.; Pederson, D.O.; Sangiovanni-Vincentelli, A.: **SPICE3 Version 3e, User´s Manual**. Univ. of California, Berkeley, Ca., 94720, 1991

[5] Mattsson, S.E., Otter, M. Hilding, E.: **Modelica Hybrid Modeling and Efficient Simulation**. 38th IEEE Conference on Decision and Control, CDC'99, Phoenix, Arizona, USA, pp. 3502-3507, Dec 7-10, 1999.

[6] Mattsson S.E.; Söderlind G.: **Index reduction in differential-algebraic equations using dummy derivatives**. SIAM Journal of Scientific and Statistical Computing, Vol. 14, pp. 677-692, 1993.

[7] Modelica Association: **Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 2.0.** (download from http://www.Modelica.org/Documents/ModelicaSpec20.pdf)

[8] Pantelides C.: **The Consistent Initialization of Differential-Algebraic Systems**. SIAM Journal of Scientific and Statistical Computing, pp. 213-231, 1988.

[9] Remelhe, M.A.P.: **Combining Discrete Event Models and Modelica – General Thoughts and a Special Modeling Environment**. Modelica'2002, Oberpfaffenhofen, March 18-19, pp. 203-207, 2002. (download from http://www.Modelica.org/Conference2002/papers.shtml)

[10] Urquia, A.; Dormido, S.: **DC, AC Small-Signal and Transient Analysis of Level 1 N-Channel MOSFET with Modelica**. Modelica'2002, Oberpfaffenhofen, March 18-19, pp. 99-108, 2002. (download from http://www.Modelica.org/Conference2002/papers.shtml)

[11] VHDL-AMS: http://www.vhdl.analog.com (information of the IEEE 1076.1 working group)