
Freiseisen W., Keber R., Medetz W., Pau P., Stelzmueller D.:
**Using Modelica for Testing Embedded Systems**
2nd International Modelica Conference, Proceedings, pp. 195-201

Paper presented at the 2nd International Modelica Conference, March 18-19, 2002,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Oberpfaffenhofen, Germany.

All papers of this workshop can be downloaded from
*http://www.Modelica.org/Conference2002/papers.shtml*

# Using Modelica for Testing Embedded Systems

*Wolfgang Freiseisen, Robert Keber, Wihelm Medetz,*
*Petru Pau, Dietmar Stelzmueller*
[wfreisei,rkeber,ppau]@risc.uni-linz.ac.at
[wilhelm.medetz,dietmar.stelzmueller]@scch.at

## Abstract
In this paper, we give an overview of a simulation environment based on Modelica, dedicated to testing PLC programs. The main components of the system are a compiler of a Modelica subset and a runtime environment, which provides the necessary tools for simulating the evolution of models.

## Introduction

Due to the high complexity of embedded software systems, it is more and more desirable to provide programmable logic control (PLC) programmers with *virtual test environments*. Usually, tests of the complete software, including PLC programs, high-level task control and human-machine interface (HMI) visualization, can only be performed when the mechanical environment, which is controlled by the software, has been finished.

This is the reason why a *simulation environment* based on Modelica has been developed. We describe in this paper a simulation system that handles models written in Modelica and uses C++ as intermediate language. Developed in the frame of the project VirtMould, supported by RISC institute and company ENGEL from Schwertberg, Austria, the compiler was meant to provide a tool for simulating *injection-molding machines*. The compiler accepts only a subset of Modelica language: this subset suffices for obtaining a model that simulates faithfully an injection-molding machine.

In order to minimize the necessary simulation modelling time, Modelica descriptions of many components are generated automatically from CAD models. Models are translated first to XML, the resulting files containing, in fact, the syntactic structure of the Modelica programs. The XML files are further parsed and provide the input for a pushdown automaton, which creates the internal data structures that store the essential content of the future C++ classes.

This C++ code is compiled and linked to specific simulation libraries; the resulting software component – a dynamic-link library or a static library – can be linked to external tools for visualization or process simulation.

In fact, the runtime environment is interfaced with a simulation of the embedded software environment. It allows an almost real time execution and simulation of the embedded software. In addition, an open, OPC (OLE for process control) based interface for visualization and monitoring of the process is provided.

As we have already mentioned, the stable version of our product accepts only a subset of Modelica language. Thus, among the restricted classes, only blocks are currently translated, and the syntax is restricted. A version that can handle Modelica models is currently in the testing phase.

The paper is structured as follows: We begin by giving a short overview of the project. The architecture is detailed in Section 2. In Section 3 we present the main features of the C++ generated code. Section 4 describes the interface provided by the software component obtained after compiling the model. In Section 6 we give some examples and snapshots of the visual interfaces.

## 1. Project overview

ENGEL is a leading manufacturer of injection moulding machines, producing and selling integrated flexible manufacturing cells. A typical manufacturing cell consists of:

- an injection-moulding machine, whose individual components are selected from a wide variety of available product features;
- a handling system built upon a free programmable robot.

The whole manufacturing cell is controlled by an *embedded software system* that integrates an IEC 1131 based PLC, a high-level task-coordination language, a Java-based HMI and communication components for manufacturing execution system (MES) integration.

The software engineers and service technicians should have the possibility to perform software tests *offline* on their desktop with a "virtual injection-moulding machine", whenever a new PLC is produced. In order to achieve a high user acceptance, a *test environment* (see [6] for a description of the main features of such software components) must be closely integrated with the development environment used by the PLC programmers; also, the PLC programmer should not be bothered with building simulation models for his specific target machine. Therefore the simulation models used for testing must be automatically generated from CAD designs.

The main goal is to increase software quality through simulation-based testing without increasing the

time spent for testing. In addition to this, VirtMould should also be applicable for other application domains, like *customer support offline diagnosis*, *computer based training* or *sales support*.

# 2. System architecture

**VirtMould** environment contains four major components:

- automatic generation of a simulation model;
- programming environment;
- runtime environment;
- visualization.

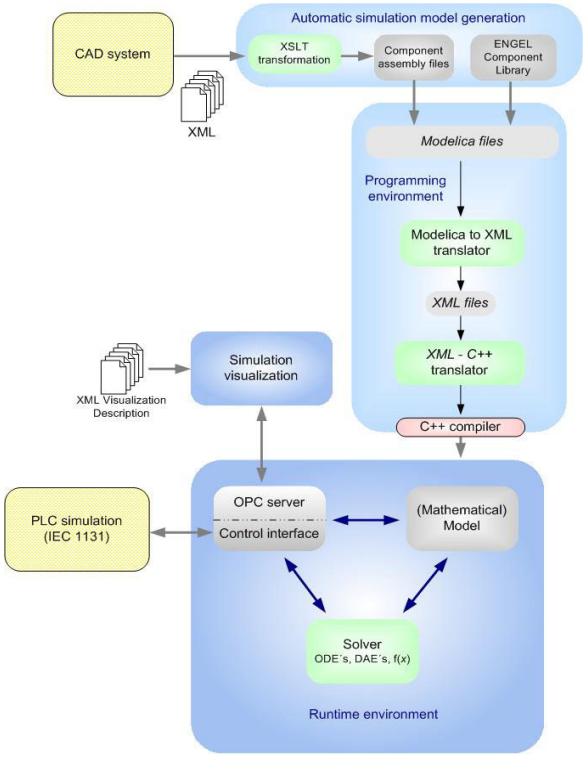


Figure 1: System architecture of the VirtMould environment.

### Automatic generation of the simulation model

In order to minimize the effort for building simulation models, XML-based tools have been developed for translating CAD files (containing electrical, hydraulic, etc. components), together with information about product configuration, to Modelica. These tools work with *XLST transformations* based on the XML files exported from CAD, and generate *component assembly files*.

A library containing the *mechatronic blocks* and *handling system components* of the injection-moulding machine has been developed and is continuously improved and extended (*ENGEL Component Library*). The goal has not been to achieve the highest possible simulation accuracy, but rather to provide the accuracy required for software testing, together with a high degree of flexibility and fast simulation execution. This library also contains components for interactive test manipulation, interaction with the panel of the virtual machine, and simulation of machine failures.

### Programming Environment

A compiler for a Modelica subset has been implemented. The focus of this compiler was to provide an easy integration into the overall system architecture and to allow the efficient simulation of discrete events. This is done in two phases: The Modelica - *XML translator* parses the Modelica files and generates a XML representation, and the *XML - C++ translator* generates C++ class files. Details on the Modelica compiler are presented in Section 3.

### Runtime Environment

The runtime environment is loaded by the PLC program simulation and contains some major components:

- the *model* is the C++ collection of classes corresponding to the Modelica program;
- the *solver* is the C++ framework for computing trajectories of the variables of the model;
- there are two kinds of interfaces for controlling the simulation and for data exchange:
  - the *control interface* is a COM in-process DLL interface, which has been chosen for coupling the PLC program simulation with the Modelica simulation runtime environment. The PLC program simulation allows program execution in either *soft real-time* or *virtual simulation time* mode.
  - for accessing the simulating state we have added an *OPC server*. A more detailed description is given in Section 4.

### Visualization

In addition to using third party visualization tools, a built-in configurable user interface allows the design of *virtual hardware panels*. The user has the possibility to:

- inspect all model variables in a hierarchical manner;
- monitor some variables by drawing their trajectories;
- change the values of some variables, provided that those variables permit this kind of user interaction.

The user interface can be configured through an XML file.

# 3. Short description of the Modelica compiler

The compilation of a Modelica model comprises three steps. After each step, one or more files of specific types are generated. Thus:

1. The model is parsed and the syntactic structure is stored in an XML file. In the same format (XML) are stored the libraries.
2. One or more XML files are taken as input by a program that produces C++ code. This code mirrors the Modelica code, a Modelica

class having a C++ counterpart, which is a C++ class.
3. The C++ file is compiled, and the object is linked with some libraries, which provide the framework for solving the model, i.e., generating a simulation, and for interfacing with other software tools.

In the following we focus on the characteristics of the C++ code.

### Base classes

C++ as intermediate code was a normal choice, due to the object-oriented philosophy of Modelica (see Modelica tutorial [1]). A Modelica class is translated into a C++ class. Among the restricted Modelica classes, our system accepts **type**, **block**, **connector** and **package**. Models are accepted with some restrictions.

To each component of a Modelica class corresponds a member data in its C++ image. According to the specifics of the Modelica class, a number of other members are included in the C++ code, like, e.g., one additional member for each differentiated variable.

A Modelica block and its C++ image are shown in Figure 1. We have kept in the C++ code only the member declarations and function definitions that correspond directly to their Modelica counterparts.

The C++ correspondent of a Modelica **package** is derived from a special class, which has no method with equations but can contain a number of inner classes.

The **connector**s provide a set of template member functions, in order to allow connections with components of various types.

We must emphasize that the granularity of Modelica code is preserved: The models are described hierarchically in Modelica, by giving the mathematical description of components and connecting them in composite objects; this structure is transmitted to the C++ code.

### Equations

The *equations* that describe the behavior of a component generate a few member functions. Thus:

1. *discrete* equations are collected in a member function that is executed only when special events occur;
2. *differential* equations go to another member function;
3. the remaining equations are divided in two groups, depending on the relation of their variables with differential equations. Thus, we call *dynamic* these equations that are related to differential equations, and
4. *independent algebraic* the other ones.

These methods are easily generated for blocks – provided that the Modelica code contains all equations in explicit form. If *flow* variables occur and Kirchoff laws have to be generated, implicit equations should be added and the translation requires further processing. Moreover, for equations that cannot be explicitized (like transcendental equations, or polynomial of high degree), numerical solvers are required. These features will be covered in a future version of our system.

### Events

The set of discrete equations is solved only when some *events* occur. During the simulation, an event queue is maintained and used for triggering the evaluation of these equations.

Changes of values of discrete variables usually generate events; also, the special functions sample and edge are event-generators. On the other hand, events can be generated *externally*, by the interaction of the user with the visual interface: Recall that the output of our system is a *software component* that can be embedded or attached to other software tools; this component communicates with the environment in two ways, that is, it exposes names, types and values of variables, and it accepts, with some restrictions, modifications of these variables.

### Simulation

A *simulation* consists in the computation of trajectories of variables during a specific time period. In fact, a finite set of points on these trajectories is computed, for values of time discretized by a *time step* whose value is set externally.

For solving differential-algebraic equations we have implemented a few numerical integrators: Euler, Runge-Kutta, Runge-Kutta with variable step size (see [4], [5] for detailed descriptions of these methods). In order to ensure a higher stability, the integrators have an *internal time step* (fixed or variable), which is usually much smaller than the external. The differential and dynamic equations are evaluated after each internal time step, whereas the independent algebraic equations are evaluated after the external time step.

Basically, the simulation is performed following the same rules as described in the Modelica language specifications (see [2], Chapter 4): The integrator solves numerically the equations between two events. When an event occurs, the set of discrete equations is solved and any change of the values of the discrete variables can influence the set of differential and algebraic equations.

Ordering their enclosing blocks gives the order in which the equations are evaluated, so that every variable that occurs in the right-hand side of an equation has already been given a value before the equation is evaluated.

```
block Sine
    parameter Real amplitude[:]={1.};
    parameter SIunits.Frequency freqHz[:]={1.};
    parameter SIunits.Angle phase[:]={0.};
    parameter Real offset[:]={0.};
    parameter SIunits.Time startTime[:]={0.};

    extends Interfaces.MO(final nout=max([size(amplitude, 1); size(freqHz, 1); size(phase, 1);
                                          size(offset, 1); size(startTime, 1)]));
protected
    constant Real pi=Modelica.Constants.pi;
    parameter Real p_amplitude[nout]=(if size(amplitude, 1) == 1 then ones(nout)*amplitude[1] else amplitude);
    parameter Real p_freqHz[nout]=(if size(freqHz, 1) == 1 then ones(nout)*freqHz[1] else freqHz);
    parameter Real p_phase[nout]=(if size(phase, 1) == 1 then ones(nout)*phase[1] else phase);
    parameter Real p_offset[nout]=(if size(offset, 1) == 1 then ones(nout)*offset[1] else offset);
    parameter SIunits.Time p_startTime[nout]=(if size(startTime, 1) == 1 then ones(nout)*startTime[1] else startTime);
equation
    for i in 1:nout loop
        y[i] = p_offset[i] + (if time < p_startTime[i] then 0.
                    else p_amplitude[i]*Modelica.Math.sin(2*pi*p_freqHz[i]*(time - p_startTime[i]) + p_phase[i]));
    end for;
end Sine;
```

```
class Sine : public Interfaces::MO {
// Variables ...
public:
    ParamVectorN<Real>                  amplitude;
    ParamVectorN<SIunits::Frequency>    freqHz;
    ParamVectorN<SIunits::Angle>        phase;
    ParamVectorN<Real>                  offset;
    ParamVectorN<SIunits::Time>         startTime;
// protected members
    ConstScalar<Real>           pi;
    ParamVectorN<Real>          p_amplitude;
    ParamVectorN<Real>          p_freqHz;
    ParamVectorN<Real>          p_phase;
    ParamVectorN<Real>          p_offset;
    ParamVectorN<SIunits::Time>     p_startTime;
// methods
public:
    Sine(){     // Default Constructor....
        // ... initialization of parameters with constant vals
        // ...
    }
    void resize(){   // resizing vector components
        p_amplitude.resize(nout);
        p_freqHz.resize(nout);
        p_phase.resize(nout);
        p_offset.resize(nout);
        p_startTime.resize(nout);
    }
    void init() {
        // initializing parameters with non-constant expr.
        //
        // initializing nout
        nout.parameter(max ( size ( amplitude,1 ) ,
                size ( freqHz,1 ) ,size ( phase,1 ) ,
                size ( offset,1 ) ,size ( startTime,1 ) ), 1);
        MO::init();
        //...
        resize();
    }
```

```
    void start(){     // Settings for start
        MO::start();
    }

// ...

    void equation_dyn(const Time &time){
        // Dynamic Equations for  this Block
        MO::equation_dyn(time);
        {
            Integer _initialCond;
            Integer _finalCond;
            _initialCond = 1;
            _finalCond   =  nout;
            for (int i=_initialCond;i<=_finalCond;i++){
                Real _if5;
                    if (TimeLt(time, p_startTime [ i - 1 ] )){
                        _if5 = 0.;
                    } else {
                        _if5 = p_amplitude [ i - 1 ] *
                            Modelica->Math->sin (
                             2*pi*p_freqHz [ i - 1 ] *
                             ( time-p_startTime [ i - 1 ]  ) +
                             p_phase [ i - 1 ]  ) ;
                }
                y [ i - 1 ]  = p_offset [ i - 1 ] + ( _if5 ) ;
            }
        }
        return;
    }

// ...

};   // end class: Sine
```

Figure 2: A Modelica block and its C++ translation.

## 4. Communication with other programs

The runtime system contains two interfaces for connections with other programs:

- a proprietary COM interface for communicating with the PLC program simulation;
- an open OPC-based interface for communication with visualization tools.

After compiling the initial `Modelica` text, a *library* (`.lib` file) or a *dynamic-link library* (`.dll` file) is generated. This library encapsulates both the model description and the integration algorithms. Among the possible services provided by this library we mention: start/stop a simulation process, transmit the names, types and values of all the internal variables of the model, at every time moment during the simulation run, and accept new values for some of the variables.

Interaction with other programs is realized through a common interface: we have used the COM (*Common Object Mode*) technique. A COM interface component for loading and unloading simulations, calculating time steps and exchanging data with the simulation has been designed.

For visualizing the results of the simulation an OPC (*OLE for process control*, see [3] for more information) server has been developed and implemented. The simulation results are provided via OPC and these values can be visualized with standard OPC client tools. OPC is the most commonly used standard for inter-process communication in the area of manufacturing automation.

The OPC specification is a non-proprietary technical specification that defines a set of standard interfaces based upon Microsoft OLE/COM technology. An OPC standard interface makes possible interoperability between automation/control applications, field systems/devices and business/office applications.

Traditionally, each software or application developer was required to write a custom interface, or server/driver, to exchange data with hardware field devices. OPC eliminates this requirement by defining a common, high performance interface that permits this work to be done once, and then easily reused by HMI, SCADA (Supervisory Control and Data Acquisition), control and custom applications.

The advantage of using the OPC Data Access Specification is that it provides a hierarchically structured namespace that can be directly used to map `Modelica` variables. Clients can then retrieve OPC items (i.e. `Modelica` variables) either synchronously or asynchronously. OPC also provides possibilities to specify the desired update rates for items and to browse the available item name space. The OPC server defines the access status (read, write) for each item together with additional descriptive information. As OPC is implemented by a COM object, the inter-process communication can be realized either as an highly efficient in-process communication or as distributable (DCOM) out-of-process communication. Also several clients can communicate in parallel with one OPC server.

## 5. Usage and examples

The following figure shows a standard ENGEL injection moulding machine. The ENGEL HL is a highly accurate, fast and energy-saving injection moulding machine in tiebarless design for use in the range from 200 to 6,000 kN clamping force.


Figure 3: An injection moulding machine.

After a thorough analysis of the machine structure, a simulation model written in `Modelica` has been developed, which should describe its components with an appropriate accuracy for testing its general behavior. As we have already mentioned, Modelica description of many components has been automatically generated from CAD specifications.

An excerpt of the main `Modelica` model of the injection model machine is given in the following. In the instantiation sector of this model all the functional units of the machine are defined.

```
block InjectMoldMachine
    "Tiebarless Injection Molding Machine"
  Lib.IMM.FunctionalUnits.MainPowerSupply   MainPower;
  Lib.IMM.FunctionalUnits.ControlVoltages   ContrVolt;
  Lib.IMM.FunctionalUnits.EmergencyOff      EmergOff;
  Lib.IMM.FunctionalUnits.FilterMotor       FilterMotor1;
  Lib.IMM.FunctionalUnits.Motor             Motor1;
  Lib.IMM.FunctionalUnits.SafetyGateMoldFront SGMoldFront;
  Lib.IMM.FunctionalUnits.SafetyDoor        SafetyDoor;
  Lib.IMM.FunctionalUnits.SafetyGateInject  SGInject;
  Lib.IMM.FunctionalUnits.Heating
        Heat1 (startTemp = 30.);
  Lib.IMM.FunctionalUnits.TraverseCooling   Cool1;
  Lib.IMM.FunctionalUnits.PumpBlock         Pumps;
  Lib.IMM.FunctionalUnits.Ejector
        Ejector1 (startPos = 0.3);
  Lib.IMM.FunctionalUnits.Core
        Core1 (startPos = 10.),
        Core2 (startPos = 10.);
  Lib.IMM.FunctionalUnits.InjectionUnit
        InjUnit1 (startPos = 5.);
  Lib.IMM.FunctionalUnits.Mold
        Mold1 (startPos = 3.);
  Lib.IMM.FunctionalUnits.InjectionPlasticize
        InjPlast1 (startPos = 1.);
  Lib.VMLib.Electrics.AlarmLamp             Alarm;
  PLC_Interface                             PLC_IO;
  ButtonBlock                               Buttons;
equation
  // input connections for "SafetyDoor"
   connect(ContrVolt.VAC24, SafetyDoor.VAC24);
   connect(ContrVolt.VE24, SafetyDoor.VE24);
   connect(Buttons.SafetyDoor.outPort, SafetyDoor.HandleGate);
   connect(Buttons.QuitKey.outPort, SafetyDoor.QuitKey);
   //. . .
end InjectMoldMachine;
```

The simulation model contains all components necessary for characterizing the behavior of the machine, including its electrical parts (e.g. power supply with different control voltages), hydraulic movements (e.g. ejector, mold) and interaction with the user (e.g. open/close safety gate).

Currently, the typical testing environment consists of the following windows:

- HMI of the injection moulding machine; this graphical object has the appearance and functionalities of the touch screen of the real machine;
- Graph window: for visualizing simulation results of selected variables;
- Control panel: buttons and switches for opening/closing safety gates, moving hydraulic units, etc.; this object is a faithful copy of the control panel of the injection moulding machine;
- PLC simulation window: gives internal information of the PLC runtime environment.

These components can be seen in Figure 4.

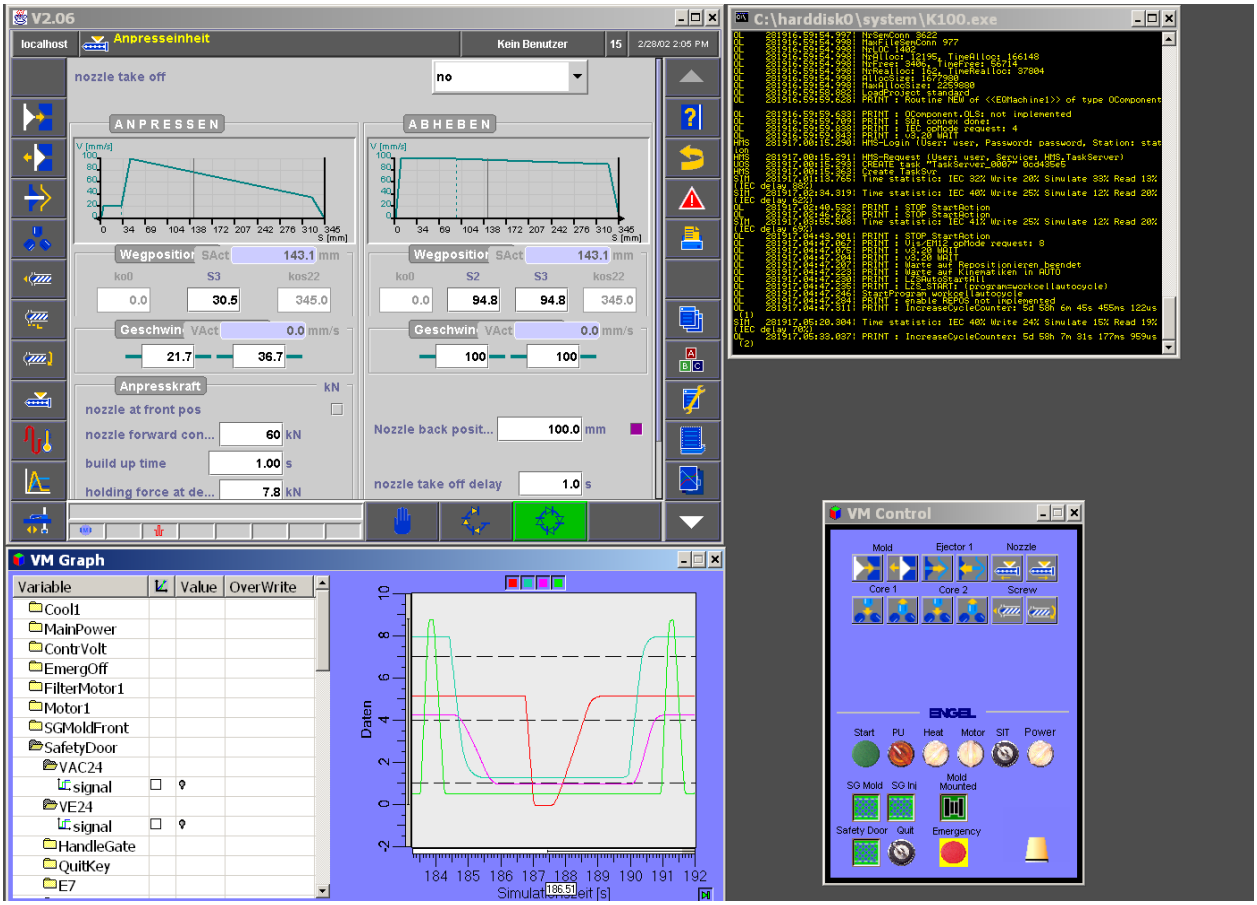


Figure 4: Screenshot from a simulation session.

## 6. Conclusions and future work

The quality of the overall system architecture of the VirtMould test environment has been proved by a high user acceptance. About 10 PLC programmers in their daily work are currently using the environment; this number should be increased to 50 or more users, testing more than 1000 systems per year. `Modelica` has proved to be the ideal object oriented modelling language for building reusable libraries of simulation components, which is essential for automatic model generation.

The next step will be to add a test automation framework, such that *regression tests* can be performed automatically. An SVG (Scalable Vector Graphic) based visualization client will provide enhanced 2D animation of the simulation process. A 3D mechanical component visualization will also provide the possibility for collision checking.

In addition to software testing, the environment will also be used for computer-based training.

## References

[1] *Modelica™ – A Unified Object-Oriented Language for Physical Systems Modeling*, Tutorial, Version 1.4, by **Modelica Association**, Dec. 2000, downloadable from http://www.modelica.org.

[2] *Modelica™ – A Unified Object-Oriented Language for Physical Systems Modeling*, Language Specification, Version 1.4, by **Modelica Association**, Dec. 2000, downloadable from http://www.modelica.org.

[3] F. Iwanitz, J. Lange - *OLE for Process Control*, **Huethig GmbH,** Heidelberg, 2001.

[4] U. M. Ascher, L. R. Petzold – *Computer methods for ordinary differential equations and differential-algebraic equations*, **SIAM**, 1998.

[5] K. E. Brennan, S. L. Campbell, L. R. Petzold – *Numerical solution or initial-value problems in differential-algebraic equations*, **SIAM** 1996.

[6] C. Kaner, J. Falk, H. Q. Nguyen - *Testing Computer Software*, **John Wiley & Sons**, 1999.

[7] M. Fewster, D. Graham - *Software Test Automation*, **Addison-Wesley**, 1999.