SNE SIMULATION NEWS EUROPE

Special Issue:

Object-oriented and Structural-dynamic Modeling and Simulation II

Volume 18 Number 2 August 2008, ISSN 0929-2268

Journal on Developments and Trends in Modelling and Simulation

Special Issue

ARGESIM

Editorial SNE Special Issue

Object-oriented and Structural-dynamic Modelling and Simulation II

The SNE special issues on *Object-oriented and Structural-dynamic Modelling and Simulation* emphasize recent developments in languages and tools for object-oriented modelling of complex systems and on approaches, languages, and tools for structural-dynamic systems.

Computer aided modelling and simulation of complex systems, using components from multiple application domains, have in recent years witnessed a significant growth of interest. In the last decade, novel equation-based object-oriented (EOO) modelling languages, (e.g. Modelica, gPROMS, and VHDL-AMS) based on acausal modelling using equations have appeared. These languages allow modelling of complex systems covering multiple application domains at a high level of abstraction with reusable model components.

This need and interest in EOO languages additionally raised the question for modelling approaches and language concepts for structural dynamic systems. Appropriate control structures like state charts in EOO languages also allow composition of model components 'in serial' – an interesting new strategy for modelling structural- dynamic systems.

There exist several different communities dealing with both subjects, growing out of different application areas. Efforts for bringing together these disparate communities resulted in a new workshop series, EOOLT workshop series, and established special sessions on structural-dynamic modelling and simulation (SDMS) within simulation conferences. In August 2007, the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools – EOOLT 2007 – took place in Berlin, followed by the 2nd workshop EOOLT 2008 in Cyprus, July 2008, and a special session at EUROSIM 2007 Congress (September 2007, Ljubljana) focused on structural dynamic modelling (EUROSIM 2007- SDMS Special Session), to be continued at MATHMOD 2009 in Vienna.

SNE 17/2, the SNE special issue on *Object-oriented and Structural-dynamic Modelling and Simulation – I* presented selected contributions from EOOLT 2007 and from EUROSIM 2007 – SDSM. This SNE special issue *Object-oriented and Structural-dynamic Modelling and Simulation – II* - SNE 18/2 - continues with overview, state-of-the-art, and development of object-oriented and structural-dynamic modelling and simulation with further four contributions from EUROSIM 2007 – SDSM and with three contributions from EOOLT 2008.

The first two contributions investigate and describe structural-dynamic changes by means of hybrid automata and UML state charts, resp., trying to combine continuous and discrete world views ('Discrete Hybrid Automata Approach to Structural and Dynamic Modelling and Simulation', G. Mušic and B. Zupancic; 'Modeling of Structural-dynamic Systems by UML Statecharts in AnyLogic', N. Popper et al).

The paper 'Classical and Statechart-based Modeling of State Events and of Structural Changes in the Modelica Simulator Mosilab' by G. Zauner et al. compares classical IF- and WHEN – clause constructs with state charts for modelling state events, with examples in a Java-based simulator.

The fourth and the fifth paper, 'Numerical Solution of Continuous Systems with Structural Dynamics' by O. Enge-Rosenblatt, and 'Selection of Variables in Initialization of Modelica Models' by M. Najafi' discuss algorithmic and numerical aspects in handling structural changes and variable initialising, resp.

The sixth contribution 'Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling' by V. Sanz underlines that the Modelica approach is also suited for discrete-event modelling. The issue concludes with the contribution 'Multi-Aspect Modeling in Equation-Based Languages' by D. Zimmer, addressing general topics and further developments.

It is intended to publish related contributions from EOOLT 2008 and from MATHMOD 2009 SDMD special session in coming regular issues of SNE.

The editors would like to thank all authors for their co-operation and for their efforts, e.g. for sending revised versions of their contributions for SNE, and hope, that the selected papers present a good overview and state-of-the-art in object-oriented and structural-dynamic modelling and simulation.

Peter Fritzson, Linköping University, Sweden
François Cellier, ETH Zurich, Switzerland
Christoph Nytsch-Geusen, University of Fine Arts, Berlin, Germany
Peter Schwarz, Fraunhofer EAS – Dresden, Germany
Felix Breitenecker, Vienna Univ. of Technology, Austria
Borut Zupancic, Univ. Ljubljana, Slovenia

Proceedings EUROSIM 2007 - 6th EUROSIM Congress on Modeling and Simulation, B. Zupancic, R. Karba, S. Blazic (Eds.); ARGESIM / ASIM, Vienna (2007), ISBN: 978-3-901608-32-2;

Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools – EOOLT 2008, P. Fritzson, F. Cellier, Ch. Nytsch-Geusen (eds), Linköping University Electronic Press 2008, ISSN (online): 1650-3740; www.ep.liu.se/ecp/024/

Contents

Dear Readers,

Due to the big interest in object-oriented and structural-dynamic modelling we decided to continue in 2008 with this subject, publishing this year a further SNE Special Issue 'Object-oriented and Structural-dynamic Modelling and Simulation II' – SNE 18/2. SNE 17/2, 'Object-oriented and Structural-dynamic Modelling and Simulation I', contained revised and/or extended versions from contributions to EOOLT 2007 workshop and from EUROSIM 2007 special session on structural-dynamic systems . This issues continues with further contributions from EUROSIM 2007 special session on structural-dynamic systems, and from EOOLT 2008 workshop- all fulfilling the editorial policy of SNE Special Issues. Further contributions, which were suggested as candidates (e.g. from Modelica Conference 2008) will be published in regular SNE issues (SNE 18/3-4, SNE 19/1) – so that the subject 'Object-oriented and Structural-dynamic Modelling and Simulation' has become an emphasis for SNE in the years 2007, 2008, and 2009.

The already announced SNE Special Issue on 'Verification and Validation' is postponed to 2009, to appear with new title "Quality Aspects in Modeling and Simulation' (SNE 19/2).

I would like to thank all authors and people who helped in managing this SNE Special Issue, especially the Guest Editors, Peter Schwarz (Fraunhofer EAS, Dresden, Germany), Borut Zupancic (Univ. Ljubljana, Slovenia), Peter Fritzson (Linköping University, Sweden), François Cellier (ETH Zurich, Switzerland), and David Broman, (Linköping University, Sweden).

Felix Breitenecker, Editor-in-Chief SNE; Felix.Breitenecker@tuwien.ac.at

Call for Contributions

SNE Special Issue 2009 "Quality Aspects in Modeling and Simulation"

Simulation is an important method which helps to take right decisions in system planning and operation. Building high-quality simulation models and using the right input data are preconditions for achieving significant and usable simulation results. For this purpose, a simulation model has to be well-defined, consistent, accurate, comprehensive and applicable.

The ASIM-Working Group *Simulation in Production and Logistics* accommodates the increased significance of quality aspects in simulation studies and will publish the forthcoming special issue in the Simulation News Europe (SNE). Papers on one or more of the following topics will be welcome:

- Quality Aspects in Simulation Studies
- Procedure Models and Methods for Information and Data Acquisition
- Procedure Models for Verification and Validation
- Verification and Validation Techniques
- Certification and Accreditation
- Model Management and Documentation Aspects
- Statistical Significance of Simulation Results
- Case Studies and Practical Experiences

The guest editors of this SNE Special Issue, Sigrid Wenzel (University Kassel), Markus Rabe (Fraunhofer Institute IPK, Berlin), and Sven Spieckermann (SimPlan AG, Maintal) invite for submitting a contribution.

Contributions should not exceed 8 pages (template provided at ASIM Webpage, ww.asim-gi.org, Menu *International*) and should be mailed directly to the editors not later than April 15, 2009. Contributions will be peer reviewed.

Sigrid Wenzel
Dept. of Mechanical Engineering, University of Kassel
Kurt-Wolters-Strasse 3, D-34125 Kassel, Germany
sigrid.wenzel@uni-kassel.de

TN

Discrete Hybrid Automata Approach to Structural Dynamic Modelling and Simulation

Gašper Mušič, Borut Zupančič, University of Ljubljana, Slovenia, *gasper.music@fe.uni-lj.si*

The paper presents the discrete hybrid automata (DHA) modelling formalism and related HYSDEL modelling language. The applicability of the framework in the context of modelling of structural-dynamic systems is discussed. High level and partially modular modelling capabilities of HYSDEL are presented and the possibility of modelling structural-dynamic systems is shown and illustrated by a simple example. To model structural dynamics, standard HYSDEL list structures are employed, and additional dynamic modes are introduced when state re-initializations are necessary at mode switching. For the derived DHA models an efficient simulation algorithm is presented. The main features of the framework are compared to characteristics of other modelling and simulation tools capable of capturing structural dynamics. Although DHA modelling framework only permits the simulation of a corresponding maximal state space model, and the simulation precision is limited, it offers other advantages, e.g. straightforward translation of the model to various optimization problems that can be solved by standard linear or quadratic programming solvers.

SNE 18/2, August 2008

Introduction

Hybrid systems were recognized as an emerging research area within the control community in the past decade. With improvements to the control equipment the complexity of modern computer-control systems increases. Various aspects of discrete-event operation, such as controller switching, changing operating modes, communication delays, and interactions between different control levels within the computercontrol systems are becoming increasingly important. Hybrid systems, defined as systems with interacting continuous and discrete-event dynamics, are the most appropriate theoretical framework to address these issues.

Mathematical models represent the basis of any system analysis and design such as simulation, control, verification, etc. The model should not be too complicated in order to efficiently define system behaviour and not too simple, otherwise it does not correspond to the real process and the behaviour of the model is inaccurate. Many modelling formalisms for hybrid systems were proposed in the engineering literature [1, 2, 3] and each class of models is usually appropriate only for solving a certain problem.

A common approach to analyse the behaviour of the developed model is to apply simulation and observe the response in the time domain. When hybrid models are dealt with, a number of problems must be resolved, such as detection of state-events, generated when a predefined boundary in the state-space is reached by the state trajectory, or a proper treatment of discontinuities, such as re-initialization of the state at the so-called state jumps, etc. A number of related simulation techniques and tools has been developed that deal successfully with these problems. One of the most challenging issues from the simulation viewpoint is a proper treatment of state dependent changes in the model structure during the simulation run. This means that in dependency of events, which are triggered from the state of the model or its environment, the number and types of equations can change during the simulation. These changes are often designated by a term model structural dynamics.

In the paper an approach is presented, where the system is modelled as a *discrete hybrid automaton* (DHA) using a HYSDEL (HYbrid System DEscription Language) modelling language [4, 5]. Using an appropriate compiler, a DHA model described by the HYSDEL modelling language can be translated to different modelling frameworks, such as *mixed logical dynamical* (MLD), *piecewise affine* (PWA), *linear complementarity* (LC), *extended linear complementarity* (ELC) or *max-min-plus-scaling* (MMPS) systems [6]. The system described as an MLD system [7] can be effectively simulated using an additional information from the HYSDEL compiler. The approach was applied to a simple example of a structural-dynamic system, which illustrates the applicability of the framework.

1 Discrete hybrid automata

According to [4] a *Discrete hybrid automaton* (DHA) is the interconnection of a *finite state machine* (FSM), which provides the discrete part of the hybrid system,

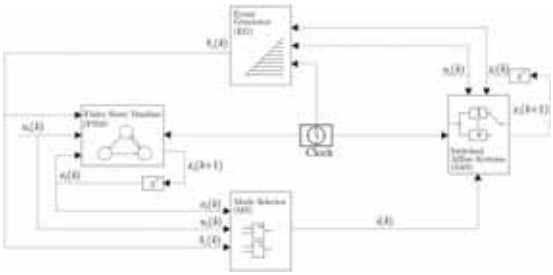

Figure 1. A discrete hybrid automation (DHA)

with a *switched affine system* (SAS) providing the continuous part of the hybrid dynamics. The interaction between the two is based on two connecting elements: the *event generator* (EG), which extracts logic signals from the continuous part, and *mode selector*, which defines the mode (continuous dynamics) of the SAS based on logic variables (states, inputs and events). The DHA system is shown on figure 1.

A switched affine system (SAS) represents a sampled continuous system that is described by the following set of linear affine equations:

$$x_r(k+1) = A_{i(k)}x_r(k) + B_{i(k)}u_r(k) + f_{i(k)} \quad (1a)$$

$$y_r(k) = C_{i(k)}x_r(k) + D_{i(k)}u_r(k) + g_{i(k)} \quad (1b)$$

where $k \in \mathbb{Z}_{\geq 0}$ represents the independent variable (time step) ($\mathbb{Z}_{\geq 0} \triangleq \{0,1,...\}$ is a set of nonnegative integers) $x_r \in \mathcal{X}_r \subseteq \mathbb{R}^{n_r}$ is the continuous state vector, $u_r \in \mathcal{U}_r \subseteq \mathbb{R}^{m_r}$ is the continuous input vector, $y_r \in \mathcal{Y}_r \subseteq \mathbb{R}^{p_r}$ is the continuous output vector, $\{A_i, B_i, f_i, C_i, D_i, g_i\}_{i\in\mathcal{I}}$ is a set of matrices of suitable dimensions, and $i(k) \in \mathcal{I}$ is a variable that selects the linear state update dynamics. A SAS of the form (1) changes the state update equation when the switch occurs, i.e. $i(k) \in \mathcal{I}$ changes. An SAS can be also rewritten as the combination of linear terms and *if-then-else* rules. The state-update function (1a) can also be written as:

$$z_1(k) = \begin{cases} A_1 x_r(k) + B_1 u_r(k) + f_1 & i(k) = 1 \\ 0 & \text{otherwise} \end{cases} \quad (2a)$$

…

$$z_s(k) = \begin{cases} A_s x_r(k) + B_s u_r(k) + f_s & i(k) = s \\ 0 & \text{otherwise} \end{cases} \quad (2b)$$

$$x_r(k+1) = \sum_{i=1}^{s} z_i(k) \quad (2c)$$

An event generator (EG) generates a logic signal according to the satisfaction of linear affine constraints:

$$\delta_e(k) = f_H(x_r(k), u_r(k), k) \quad (3)$$

where $f_H : \mathbb{R}^{n_r} \times \mathbb{R}^{m_r} \times \mathbb{Z}_{\geq 0} \to \mathcal{D} \subseteq \{0,1\}^{n_e}$ is a vector of descriptive functions of a linear hyperplane. The relation f_H for time events is modeled as $[\delta_e^{\,i}(k) = 1] \leftrightarrow [kT_s \geq t_i]$, where T_s is the sampling time, while for threshold events is modeled as $[\delta_e^{\,i}(k) = 1] \leftrightarrow [a_i^{\,T} x_r(k) + b_i^{\,T} u_r(k) \leq c_i]$, where a_i, b_i, c_i represent the parameters of a linear hyperplane. $\delta_e^{\,i}$ denotes the i-th component of a vector $\delta_e(k)$.

A finite state machine (FSM) is a discrete dynamic process that evolves according to a logic state update function:

$$x_b(k+1) = f_B(x_b(k), u_b(k), \delta_e(k)) \quad (4)$$

where $x_b \in \mathcal{X}_b \subseteq \{0,1\}^{n_b}$ is the Boolean state, $u_b \in \mathcal{U}_b \subseteq \{0,1\}^{m_b}$ is the Boolean input, $\delta_e(k)$ is the input coming from the EG, and $f_B : \mathcal{X}_b \times \mathcal{U}_b \times \mathcal{D} \to \mathcal{X}_b$ is a deterministic logic function. An FSM may have also associated Boolean output:

$$y_b(k) = g_B(x_b(k), u_b(k), \delta_e(k)) \quad (5)$$

where $y_b \in \mathcal{Y}_b \subseteq \{0,1\}^{p_b}$.

A mode selector (MS) selects the dynamic mode $i(k)$ of the SAS according to the Boolean state $x_b(k)$, the Boolean inputs $u_b(k)$ and the events $\delta_e(k)$ using the Boolean function $f_M : \mathcal{X}_b \times \mathcal{U}_b \times \mathcal{D} \to \mathcal{I}$. The output of this function

$$i(k) = f_M(x_b(k), u_b(k), \delta_e(k)) \quad (6)$$

is called the *active mode*.

2 HYSDEL modelling language

DHA models can be built by using the HYSDEL modelling language [4], which was designed particularly for this class of systems. The HYSDEL modelling language allows the description of hybrid dynamics in textual form. The HYSDEL description of hybrid systems represents an abstract modelling step. Once the system is modelled as DHA, i.e. described by HYSDEL language, the model can be translated into an MLD model using an associated HYSDEL compiler. At this point, we will give just a brief introduction into the structure of a HYSDEL list.

A HYSDEL list is composed of two parts: the `INTERFACE`, where all the variables and parameters are declared, and the `IMPLEMENTATION`, which consists of specialised sections, where the relations between the variables are defined.

The AD section allows the definition of Boolean variables and is based on the semantics of the *event generator* (EG), i.e. in the AD section the δ_e variables are defined. The LOGIC section allows the specification of arbitrary functions of Boolean variables. Since the *mode selector* is defined as a Boolean function, it can be defined in this section. The DA section defines the switching of the continuous variables according to *if-then-else* rules depending on Boolean variables, i.e. part of *switched affine system* (SAS), namely z_i variables (see Equation (2)) are defined. The CONTINUOUS section defines the linear dynamics expressed as difference equations, i.e. defines the remaining Equation (2c) of the SAS. The LINEAR section defines continuous variables as an affine function of continuous variables, which in combination with the DA and the CONTINUOUS section enables more flexibility when modelling SAS. The AUTOMATA section specifies the state transition equations of the *finite state machine* (FMS) as a Boolean function (4), i.e. defines Boolean variables x_b. The MUST section specifies constraints on continuous and Boolean variables, i.e. defines the sets $\mathcal{X}_r$, $\mathcal{X}_b$, $\mathcal{U}_r$ and $\mathcal{U}_b$.

For more detailed description on the functionality of the modelling language HYSDEL and the associated compiler (tool HYSDEL), the reader is referred to [4, 5].

3 Structural-dynamic systems and HYSDEL

In general, discontinuities are modelled in HYSDEL by the use of auxiliary variables. Two types of such variables exist: Boolean or discrete (δ) and continuous (z).

3.1 Modelling of discontinuities

Discrete auxiliary δ variables may be defined based on continuous variables in the AD section of the HYSDEL list, which has the following syntax:

AD{ *ad-item*$_+$ }

and each *ad-item* is one of the following:

var = *affine-expr* <= *real-number* ;
var = *affine-expr* >= *real-number* ;

The affine expression is a linear affine combination of real variables

$$a_0 + a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \qquad (7)$$

where a_i is a function of parameters, and x_i are real (state, input, output, and auxiliary) variables [5]. The δ variables defined in such a way represent outputs of the event generator (EG) in Fig. 1.

Continuous auxiliary z variables are defined in the DA section of the HYSDEL list, which has the following syntax:

DA{ *da-item*$_+$ }

and each *da-item* is one of the following:

var = { IF *Boolean-expr* THEN *affine-expr* };
var = { IF *Boolean-expr* THEN *affine-expr*
ELSE *affine-expr* };

if the ELSE part is omitted, it is assumed to be 0.

The z variables defined this way can be used to implement switching dynamic part (SAS) of the HDA in figure 1. Using this approach, also the changes in the model structure can be easily implemented.

The actual continuous dynamic of the system is implemented in discretized form in the CONTINUOUS section, which has the following syntax:

CONTINUOUS{ *cont-item*$_+$ }

and each *cont-item* is one of the following:

var = *affine-expr* ;

Typically, a list of such *cont-item*s looks like:

```
x1 = z11 + z12 + ... + z1m ;
x2 = z21 + z22 + ... + z2m ;
...
xn = zn1 + zn2 + ... + znm ;
```

where n is the number of states and m the number of dynamical nodes. Auxiliary variables z_{11} to z_{nm} are defined within the DA section.

When the mode is not active the z_{ij} variables can be zero or may be held at any other value, depending on the problem.

The conditions related to reset of the state at switching or other similar conditions can be easily taken into account if a new mode is defined, which is active only at a single sampling instant.

With regard to structural changes, it is obvious that the states can not be created or deleted during the simulation run but can only be held 'inactive' when they are not required. Therefore the simulation runs by employing a corresponding maximal state space model.

3.2 Example

To illustrate the HYSDEL modelling of structural-dynamic systems a simple example will be shown. A system under consideration has two dynamic modes, the first one being active when the system output is below 0.5 and the second one when the output is above 0.5.

In the first mode the system dynamics is described by

$$\dot{y} + 0.5y = 0.5u \tag{8}$$

where u is the input to the system and y is the system output.

The second mode is described by

$$\ddot{y} + 2\dot{y} + y = u \tag{9}$$

The system is written in the state space form by assigning the state variables $x_1 = y$ and $x_2 = \dot{y}$, and discretized at the sampling time $T_s = 0.1\text{s}$. Then the first mode is described by:

$$\begin{aligned} x_1(k+1) &= a_{11}^1 x_1(k) + b_{11}^1 u_1(k) \\ y(k) &= x_1(k) \end{aligned} \tag{10}$$

and the second mode by

$$\begin{aligned} \begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} &= \begin{bmatrix} a_{11}^2 & a_{12}^2 \\ a_{21}^2 & a_{22}^2 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} b_{11}^2 \\ b_{21}^2 \end{bmatrix} u(k) \\ y(k) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} \end{aligned} \tag{11}$$

Equations (10) and (11) are coded in the `IMPLEMENTATION` part of HYSDEL list as follows:

```
1  IMPLEMENTATION {
2  AUX {
3    BOOL d, df;
4    REAL z1, z21, z22; }
5  AD {
6    d = x1 <= limit;
7    df = a11_1*x1 + b11_1*u >= limit; }
8  DA {
9    z1 = {IF d THEN
             a11_1*x1 + b11_1*u
10         ELSE
             a11_2*x1 + a12_2*x2 + b11_2*u};
11   z21 = {IF ~d THEN
             a21_2*x1 + a22_2*x2 + b21_2*u};
12   z22 = {IF d & df THEN
             (a11_1 - a11_2)/a12_2*x1 +
             (b11_1 - b11_2)/a12_2*u}; }
13 CONTINUOUS {
14       x1 = z1;
15       x2 = z21 + z22; }
16 OUTPUT {
17       y = x1; }
18 }
```

The *limit* parameter is set to 0.5, while other parameters are obtained by discretization procedure. It can be observed that an additional dynamic mode is introduced, which is active when x_1 is below the *limit* but the value of x_1 in the next time step exceeds the *limit*. This way the time of mode switching is predicted and x_2 is set to the value which causes a smooth transition to the new mode (both y and $\dot{y}$ are continuous). Both z_{21} and z_{22} are forced to zero when corresponding modes are inactive.

4 Simulation

Once a DHA system is modelled by the HYSDEL modelling language, the companion HYSDEL compiler generates the equivalent MLD model [4]. In [7] the authors proposed discrete-time hybrid systems denoted as *mixed-logical dynamical* (MLD) systems:

$$x(k+1) = Ax(k) + B_1 u(k) + B_2 \delta(k) + B_3 z(k) \tag{12a}$$

$$y(k) = Cx(k) + D_1 u(k) + D_2 \delta(k) + D_3 z(k) \tag{12b}$$

$$E_2 \delta(k) + E_3 z(k) \le E_1 u(k) + E_4 x(k) + E_5 \tag{12c}$$

where $x = [x_r, x_b]' \in \mathbb{R}^{n_r} \times \{0,1\}^{n_b}$ is a vector of continuous and logic states, $u = [u_r, u_b]' \in \mathbb{R}^{m_r} \times \{0,1\}^{m_b}$ are continuous and logic inputs, $y = [y_r, y_b]' \in \mathbb{R}^{p_r} \times \{0,1\}^{p_b}$ are continuous and logic outputs, $\delta \in \{0,1\}^{r_b}$, $z \in \mathbb{R}^{r_r}$ auxiliary logic and continuous variables, respectively, and A, B_1, B_2, B_3, C, D_1, D_2, D_3, E_1, …, E_5 are matrices of suitable dimensions. Inequalities (12c) can also contain additional constraints on the variables (states, inputs and auxiliary variables). This permits the inclusion of additional constraints and the incorporation of heuristic rules into the model.

4.1 The structure of an MLD form

Hybrid systems consist of continuous and logic variables. Relations between latter can be described by propositional calculus [7]. Propositional calculus enable statements to be combined in compound statements by means of connectives: "$\wedge$" (and), "$\vee$" (or), "$\neg$" (not), etc. Each compound statement can be translated into a conjunctive normal form (CNF) or product of sums (POS) of the following form

$$\wedge_{j=1}^{m} (\vee_{i \in P_j} X \ \vee_{i \in N_j} \neg X) \tag{13}$$

where P_j and N_j are sets of indices of literals X_i and inverted literals $\neg X_i$. By associating logical (binary) variables $\delta_i \in \{0,1\}$ with each propositional variable X_i then the compound statement (13) can be equiva-

lently translated into a following set of integer linear inequalities:

$$\begin{aligned} 1 &\le \sum_{i\in P_1}\delta_i + \sum_{i\in N_1}(1-\delta_i) \\ &\vdots \\ 1 &\le \sum_{i\in P_m}\delta_i + \sum_{i\in N_m}(1-\delta_i) \end{aligned} \tag{14}$$

This translation technique can be adopted to model logic parts of processes, logic constraints of the plant and heuristic knowledge about plant operation, as integer linear inequalities.

A/D interface: Propositional variable X, defined by statement $X \triangleq [f(x_r) \le 0]$, i.e. $[f(x_r \le 0)] \leftrightarrow [\delta = 1]$, can be can be translated into the following set of mixed-integer inequalities

$$\begin{aligned} f(x_r) &\le M(1-\delta) \\ f(x_r) &\ge \varepsilon + (m-\varepsilon)\delta \end{aligned} \tag{15}$$

where ε is a small positive real number and M and m are constants defined by $M \triangleq \max f(x_r)$ and $M \triangleq \min f(x_r)$.

D/A interface: In this case the results of logical events define values of continuous variables. The most common D/A interface is the `IF-THEN-ELSE` construct, `IF` X `THEN` $z = f_1(x_r)$ `ELSE` $z = f_2(x_r)$, which can be translated into the following set of mixed-integer inequalities:

$$\begin{aligned} (m_2 - M_1)\delta + z &\le f_2(x_r) \\ (m_1 - M_2)\delta - z &\le -f_2(x_r) \\ (m_1 - M_2)(1-\delta) + z &\le f_1(x_r) \\ (m_2 - M_1)(1-\delta) - z &\le -f_1(x_r) \end{aligned} \tag{16}$$

where z is an auxiliary continuous variable defined by auxiliary logical variable δ associated to literal X. M_i and m_i are defined as in Equation (15).

Linear part enables to define linear relations as a system of inequalities and is defined as

$$\begin{aligned} z &\le f(x_r) \\ -z &\le -f(x_r) \end{aligned} \tag{17}$$

Continuous dynamical part is described by linear difference equations (discrete time domain) as follows

$$\begin{aligned} x_r(k+1) &= A_r x_r(k) + B_r u_r(k) \\ y_r(k) &= C_r x_r(k) + D_r u_r(k) \end{aligned} \tag{18}$$

By considering Equations (14,15,16,17,18) the mixed logical dynamical (MLD) system is derived and is presented by Equation (12). For more detailed description of the MLD structure the reader is referred to [4, 7].

4.2 Simulation of an MLD system

Using the current state $x(k)$ and input $u(k)$, the time evolution of (12) is determined by solving $\delta(k)$ and $z(k)$ from inequalities (12c), and then updating $x(k+1)$ and $y(k+1)$ from equalities (12a) and (12b), respectively. The MLD system (12) is assumed to be completely well-posed, i.e. for a given state $x(k)$ and input $u(k)$ the inequalities (12c) have a unique solution for $\delta(k)$ and $z(k)$. Obtaining the values of the auxiliary variables $\delta(k)$ and $z(k)$ presents a bottleneck in a simulation of a hybrid system modelled as an MLD system.

The variables $\delta(k)$ and $z(k)$ are defined by the system of inequalities (12c) and can be computed by defining and solving a mixed integer problem. It has to be pointed out that in this case the optimization is only used to find a feasible solution. Because the system is *well posed* the solution is unique and only one solution to the system of inequalities exists, which does not depend on the cost function. The disadvantage of this approach is the usage of the mixed integer optimization algorithms, which can be time consuming or even not able to find a feasible solution because of numerical sensitivity.

One of the reasons why the optimization approach is time consuming lies in the branch and bound nature of the underlying algorithm and in the fact that, once that the delta variables have been fixed, the inequalities (16) are actually equalities, i.e. $z = f_1(x_r)$ or $z = f_2(x_r)$.

To overcome the problem, which appears when using optimization approach, a special algorithm was developed. It is based on the knowledge of transformation procedure from DHA into MLD form and is able to compute values of $\delta(k)$ and $z(k)$ "explicitly", i.e. without iterations. Such approach is of course much faster. The algorithm involves a direct E_1, …, E_5 matrix manipulation.

The algorithm abstracts the inequalities (12c) into sets based on an origin of a certain row. The result are six sets: `AD` set containing the inequalities from AD part of a system, `LOGIC` set containing the inequalities of logical relations, `LINEAR` set containing the linear relations, `DA` set containing inequalities from DA part of a system, `LOGIC MUST` set containing all logical

constraints and `CONTINUOUS MUST` set containing all continuous constraints. The following algorithm exploits the definition of the variables $\delta(k)$ and $z(k)$ to define them:

1. Given $x(k)$ and $u(k)$.
2. **Repeat**
   a. Define $\delta_{AD}(k)$ variables for which all right hand side variables are defined.
   b. Define $\delta_{LO}(k)$ variables for which all right hand side variables are defined.
   c. Define $z_{LI}(k)$ variables for which all right hand side variables are defined.
   d. Define $z_{DA}(k)$ variables for which all right hand side variables are defined.
3. **Until** all $\delta(k) = \left[\delta_{AD}(k) \quad \delta_{LO}(k)\right]^T$ and $z(k) = \left[z_{LI}(k) \quad z_{DA}(k)\right]^T$ are defined.
4. Check logical constraints
5. Check continuous constraints
6. **If** all containts are fulfilled define $\delta(k)$, $z(k)$, new state $x(k+1)$ and output $y(k)$ **else** return error.

All the computation is based on direct E_1, …, E_5 matrix manipulation and does not rely on any mixed integer solver but relies on additional information provided by the HYSDEL tool, such as row origin information (AD, DA...). A similar algorithm is implemented in the HYSDEL tool, version 2.0.5 [4, 5].

4.3 Example

The described simulation algorithm is applied to the example model introduced in section 3.2. A periodic pulse signal with the amplitude 1 is defined as an input to the system and simulation results are shown in Fig. 2 and 3.

It can be observed that dynamics is changed when the system output crosses the boundary at 0.5. The change in the dynamics can be seen if the shapes of the rising and falling responses are compared. It can also be observed that x_2 state is initialized to the appropriate value whenever the second dynamic mode is entered. This value guarantees a smooth transition to the new mode. On the contrary, x_2 is switched to zero when the second mode is exited, because it is not needed anymore.

5 Comparison to other tools

A number of simulation techniques and tools has been developed in recent years that deal successfully with hybrid phenomena. Structural-dynamics as an important part of hybrid dynamics can be seen in one of the two distinct ways. In one way, state events can be seen as a mechanism that switches on and off algebraic conditions, which freeze certain states for certain periods. In another way, local model with fixed state spaces are controlled by a global model. Following this, two different approaches for simulating structural-dynamic systems are developed: the maximal state space approach and the hybrid decomposition approach [8].

Most currently available simulation tools follow the maximal state space model approach, e.g. Modelica, VHLD-AMS, Dymola. Matlab incorporates a simulation tool Simulink, which also works with a maximal state space. Simulink supports triggered sub-models, which can be executed only at event occurrences. Recent versions also include support for Statechart-

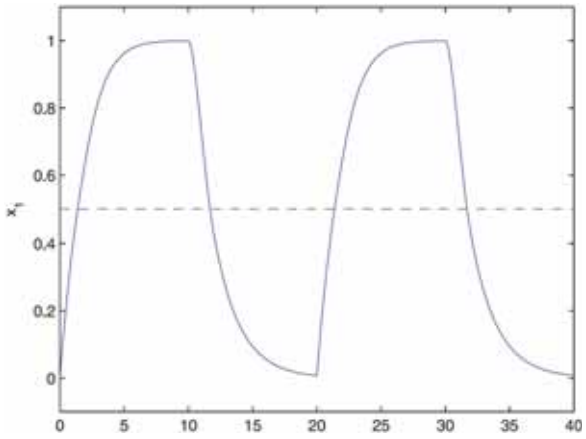

Figure 2. Simulated response ($x_1 = y$)

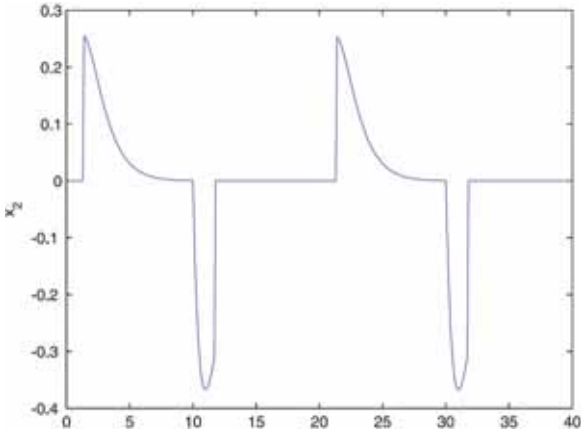

Figure 3. Simulated response (x_2)

based description of state-machines and discrete-event simulation by SimEvents Blockset, based on the entity concept. On the other hand, Simulink can not deal with Differential-Algebraic Equations (DAEs).

At the moment, the developments of hybrid decomposition approach are mainly focused on various extensions of Modelica. One of such extensions, which closely follows all basic principles of the hybrid decomposition approach, is a modelling description language Mosila (Modelling and Simulation Language). In Mosila, dynamical object structures are introduced to represent variable models. Objects represent state attributes and behaviour in a form of equations, and the equation system may be changed when a structural change is triggered. A corresponding simulator MOSILAB has been successfully applied to simulation of a number of case studies [9].

The DHA modelling and simulation approach presented in this paper belongs to the group of maximal state space model approaches. Since it operates in discrete-time, it is less elaborated from the simulation viewpoint. State and time events may be detected with a limited precision that is mainly influenced by the chosen sampling-time. On the other hand, the description has a sound theoretical framework and models can be converted to other formal descriptions of hybrid systems. This enables analytical exploration of important system properties, e.g. stability. Furthermore, the models converted to the MLD form can be used for defining various optimization problems that can be solved by standard linear or quadratic programming solvers.

6 Conclusions

The discrete hybrid automata (DHA) modelling formalism and related HYSDEL modelling language can be applied also to modelling and simulation of structuraldynamic systems. The modelling is simple and requires only the description of the switching boundaries in the state space and a discretization of the corresponding dynamics. The coding into HYSDEL list is straightforward and could also be automated based on a higher level description of the model. The simulation is fast and relatively simple. Compared to other tools the accuracy of simulation is limited, but on the other hand, the underlying DHA description can be easily transformed to other descriptions of hybrid systems and also used as a basis for analysis and optimization.

References

[1] P. J. Antsaklis. *A brief introduction to the theory and applications of hybrid systems*. Proceedings of the IEEE, 88(7):879–887, 2000.

[2] S. Engell, G. Frehse, E. Schnieder. *Modelling, Analysis and Design of Hybrid Systems*. Lecture Notes in Control and Information Sciences.

[3] G. Labinaz, M.M. Bayoumi, K. Rudie. *A survey of modeling and control of hybrid systems*. Annual Reviews in Control, 21:79–92, 1997.

[4] F. D. Torrisi, A. Bemporad. *Hysdel - a tool for generating computational hybrid models for analysis and synthesis problems.* IEEE Transactions on Control Systems Technology, 12:235–249, 2004.

[5] F. D. Torrisi, A. Bemporad, G. Bertini, P. Hertach, D. Jost, D. Mignone. *HYSDEL 2.0.5 - User Manual*. Automatic Control Laboratory, ETH, Zürich, Switzerland, 2002.

[6] W. P. M. H. Heemels, B. De Schutter, A. Bemporad. *Equivalence of hybrid dynamical models*. Automatica, 37(7):1085–1091, 2001.

[7] Bemporad, M. Morari. *Control of systems integrating logic, dynamic, and constraints*. Automatica, 35(3):407–427, 1999.

[8] F. Breitenecker, N. Popper. *Structure of simulation systems for structural-dynamic systems*. In Proceedings of the First Asia International Conference on Modelling & Simulation (AMS'07), pages 574–579, 2007.

[9] Nytsch-Geusen, T. Ernst, A. Nordwig, P. Schneider, P. Schwarz, M. Vetter, C. Wittwer, A. Holm, T. Nouidui, J. Leopold, G. Schmidt, U Doll, A Mattes. Mosilab: *Development of a modelica based generic simulation tool supporting model structural dynamic*. In Proceedings of the 4th International Modelica Conference, pages 527–535, 2005. Proc. EUROSIM 2007 (B. Zupančič, R. Karba, S. Blažič) 9-13 Sept. 2007, Ljubljana, Slovenia ISBN

Corresponding author: Gašper Mušič
University of Ljubljana
Faculty of Electrical Engineering
1000 Ljubljana, Trzaška 25, Slovenia
gasper.music@fe.uni-lj.si

Accepted EUROSIM 2007, June 2007
Received: September 20, 2007
Revised: June 12, 2007
Accepted: July 10, 2007

Modeling of Structural-dynamic Systems by UML Statecharts in AnyLogic

Daniel Leitner, Johannes Kropf, Günther Zauner, TU Vienna, Austria, *dleitner@osiris.tuwien.ac.at*
Yuri Karpov, Yuri Senichenkov, Yuri Kolesov, XJ Technologies St. Petersburg, Russia

With the progress in modeling dynamic systems new extensions in model coupling are needed. The models in classical engineering are described by differential equations. Depending on the general conditions of the system the description of the model and thereby the state space is altered. This change of system behavior can be implemented in different ways. In this work we focus on AnyLogic and its ability to switch between different sets of equations using UML statecharts. Different possibilities of the coupling of the state spaces are compared. This can be done either using a parallel model setup, a serial model setup, or a combined model setup. The analogies and discrepancies can be figured out on the basis of three classical examples. The first is the constrained pendulum as defined in ARGESIM comparison C7, where the dimension of the state space is unaltered. Second is the free pendulum on a string, where the dimension of the state space changes. The third example is a thermal storage model at which between different accuracies of the discretization is switched.

SNE 18/2, August 2008

Introduction

In this work three different structural dynamic systems are under investigation. The models are implemented in AnyLogic using its UML statecharts representation for discrete event based changes of their governing differential equations and their state spaces. In the first part AnyLogic will be described, and then the different models will be presented. After that different methods of solutions are introduced with a special focus on how UML statecharts can be used to control the model structure. With these methods implementations for AnyLogic of the structural dynamic systems are given. As a conclusion the benefits and drawbacks of the hybrid simulator are examined.

AnyLogic is a multiparadigm simulator supporting Agent Based modeling as well as Discrete Event modeling, which is flowchart-based, and System Dynamics, which is a stock-and-flow kind of description. Due to its very high flexibility AnyLogic is capable of capturing arbitrary complex logic, intelligent behaviour, spatial awareness and dynamically changing structures. It is possible to combine different modeling approaches which make AnyLogic a hybrid simulator. AnyLogic is highly object oriented and based on the Java programming language. To a certain degree this ensures a compatibility and reusability of the resulting models.

The development of AnyLogic in the last years has been towards business simulation. In version 6 of AnyLogic it is possible to calculate typical problems from engineering, but there are certain restrictions. For example the integration method cannot be chosen freely and there is no state event finder.

When a model starts, the equations are assembled into the main differential equation system (DES). During the simulation, this DES is solved by one of the numerical methods built in AnyLogic. AnyLogic provides a set of numerical methods for solving ordinal differential equations (ODE), algebraic-differential equations (DAE), or algebraic equations (NAE).

AnyLogic chooses the numerical solver automatically at runtime in accordance to the behaviour of the system. When solving ordinal differential equations, it starts integration with forth-order Runge-Kutta method with fixed step. Otherwise, AnyLogic plugs in another solver, the Newton method. This method changes the integration step to achieve the given accuracy.

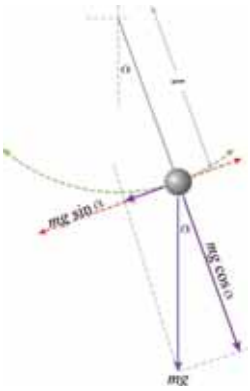

Figure 1. Force diagram of a simple gravity pendulum

1 Modeling

In this section three different models will be explained. They have in common that discrete events change the

model structure. In the first model the state space is not altered, thus the process can be described as parameter event that occur at a discrete time point. In the second and third model this is not possible because the dimension of the state space changes. This change of model structure and its implementation using UML statecharts in AnyLogic shall be investigated.

1.1 Constrained pendulum

A classical and simple nonlinear model in simulation techniques is the so called constrained pendulum. This model has been presented in the definition of ARGESIM comparison C7, full solutions in AnyLogic can be found in [1] or [2]. There is no exact analytical solution to this problem Therefore the results must be obtained by numerical methods. In this section a description of the model shall be given.

The motion of the pendulum is given by

$$\begin{aligned} ml\dot{\omega} &= -mg\sin(\alpha) - dl\omega \\ \dot{\alpha} &= \omega \end{aligned} \tag{1}$$

where α denotes the angle measured in counterclockwise direction from the vertical position and ω is the angular velocity. The parameter m is the mass and l is the length of the pendulum. The damping is realized with the constant d.

In the case of a constrained pendulum a pin is fixed at a certain position given by the angle α_p and the length l_p. If the pendulum is swinging it may hit the pin. In this case the pendulum swings on with the position of the pin as the point of rotation and the shortened length

$$l_s = l - l_p. \tag{2}$$

In ARGESIM comparison C7, the initial values of two experiments are predetermined:

1. The first example is given by
$$\alpha = \pi/6, \quad \omega = 0, \quad d = 0.2, \quad \alpha_p = -\pi/12 \tag{3}$$
2. The second example is given by
$$\alpha = -\pi/6, \quad \omega = 0, \quad d = 0.1, \quad \alpha_p = -\pi/12 \tag{4}$$

Both examples have the general parameters:

$$m = 1.02, \quad l = 1, \quad l_p = 0.7 \ (l_s = 0.3), \quad g = 9.81 \tag{5}$$

1.2 Free pendulum on a string

The second example is a slightly more complicated pendulum. The massive bob of the pendulum is fixed on a string. In case of a rollover of the pendulum it can start to fall freely until the constraints of the string apply again. This can happen if the pendulum swings higher than $\pm\frac{\pi}{2}$ and the centrifugal force is smaller than the gravitational force, see figure 1. Thus the model has two different states: the normal pendulum movement *normal* and the free fall *fall*. The pendulum movement is given in equation 1. The equations of free fall are given by

$$\begin{aligned} \dot{v}_y &= -g \\ \dot{v}_x &= 0 \end{aligned} \tag{6}$$

1.3 Solar system heating

The third example is motivated by the work of Nytsch-Geusen, who describes a complex energy system [3]. In this work only a small subsystem will be investigated, which can demonstrate the abilities of AnyLogic to deal with structural dynamics.

A one dimensional thermal storage model shall be calculated with different accuracies that are dependent of the gradient of the temperature in the storage system. This happens for example when hot water enters the storage system. The effect is realized by using to different systems with different thermal layering as presented in figure 2.

The example demonstrates not only the dynamics but although AnyLogic's ability to connect with external solvers which happens in this example because specialized finite element method (FEM) solvers are needed for the calculations.

2 Solution approaches

New advantages in computer numerics and the fast increase of computer capacity lead to necessity of new modeling and simulation techniques. In many cases of modern simulation problems state events have to be handled. There exist different categories of

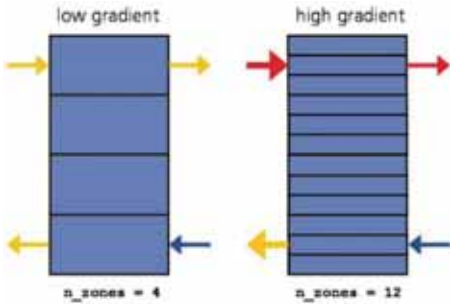

Figure 2. Structural variable storage model, which uses a different number of zones in dependency of the current thermal layering.

structural dynamic systems which should be focused on and solved. The first class of hybrid systems are the one, where the state space dimension does not change during the whole simulation time and also the system equations stay the same. Only so called parameter events occur at discrete time points. These are the more or less simplest form of state events. AnyLogic does not differ between different types of events. The implementation of parameter events and state space changes does not differ, making AnyLogic a truly hybrid simulation environment.

AnyLogic supports the usage of UML statecharts [4]. This is a very intuitive and convenient way to describe a system which contains multiple discrete states. In the combination with dynamical equations this approach enables a simple implementation of structural dynamics. The dynamic equations or parameters are dependent of the discrete state of the model. On the other hand the transitions of the states are influenced by the dynamic variables, see [5].

The different kinds of description will be explained by means of the constrained pendulum. In this case the states are normally swinging (state *long*) or swinging with shortened length around the pin (state *short*). The discrete state of the model depends on the angle α and the pins angle α_p. The state alters the model parameters or the models set of equations, see figure 3.

2.1 Switching states

When the state of a system changes, often the state space of the model stays unchanged, thus the same set of differential equation can be used for different states. In this situation only certain parameters must be changed when a state is entered.

In case of the constrained pendulum the differential equation for movement stays the same for both states *long* and *short*. If the state changes the parameter length and angular velocity are updated before the calculation can continue, see figure 4.

2.2 Switching models

Often the previous approach is not possible. Sometimes situation occur where the state space of the model changes, thus a simple change of parameters is not possible. Normally the whole set of differential equations, thus the complete model, must be switched. In many simulation environments this approach can lead to complications.

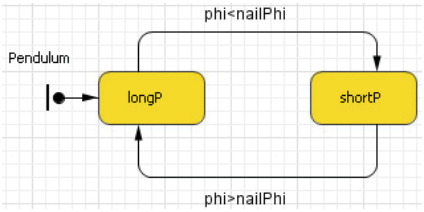

Figure 3. UML statechart controlling the pendulum.

Figure 4. The parameters of the model are changed by an UML statechart

In case of the constrained pendulum two differential equations are set up describing the movement of the pendulum. One describes the normal pendulum the other one the shortened pendulum. Which equation is set to be active is determined by the state diagram. When the states are switched the initial values must be passed on. The current equation must be activated and the other one must be frozen, see figure 5.

3 Constrained pendulum

The implementation of the constrained pendulum has been done in two different ways. In the first approach only the parameter states have been switched corresponding to section 3.1, in the second approach the whole differential equation is switched corresponding to section 3.2. Both examples from chapter 2.1 have been simulated using both approaches. The results in AnyLogic are identical in both methods because the times of the state transitions are the same.

In the first approach the model consists of two ordinary differential equations describing the movement

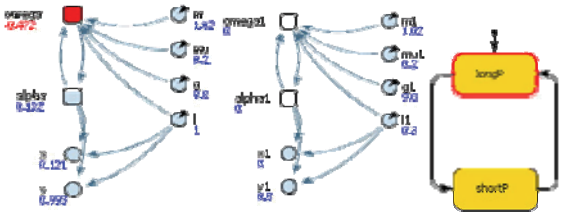

Figure 5. The differential equations of the system are switched in dependence of the UML state diagram

of the pendulum. The differential equations are realized in AnyLogic using two stock variables, the angle α and the angular velocity ω. In these equations four parameters are used: length l, mass m, damping d and gravity g. Further a state diagram with states *long* and *short* and two transitions are used to update the equations. When the state changes, the length l and angular velocity ω are updated. The results calculated by AnyLogic 6 are plotted in figures 6 and 7.

The second approach uses two separate models. The implemented model consists of two times two ordinary differential equations, thus four stock variables (α_1, α_2, ω_1, ω_2). Both equations have four parameters separately: length l, mass m, damping d and gravity g. A state diagram is implemented analog to the first approach. If the state changes the right differential equations are activated and their initial values are set, while the other differential equation is frozen.

4 Pendulum on a string

When implementing the pendulum on a string in AnyLogic, to totally different submodels must be considered:

1. The pendulum is described by the formula given by equation 1. The equation is realized in AnyLogic with the use of two stock variables, the angle α and the angular velocity ω. Further the four parameters describe the pendulum, length l, mass m, the damping factor μ and gravity g.
2. The equations of free fall uses a completely different state space. The stock variables x and y describe the position of the massive bob. In vertical direction another stock variable v_x is needed because in this direction exists acceleration due to gravity. The vertical acceleration is described with a parameter a_y. In horizontal direction a parameter v_y is sufficient.

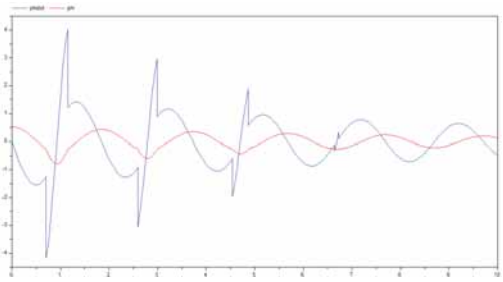

Figure 6. The angle (red, inner graph) and the angular velocity (blue) of the constrained pendulum (example 1)

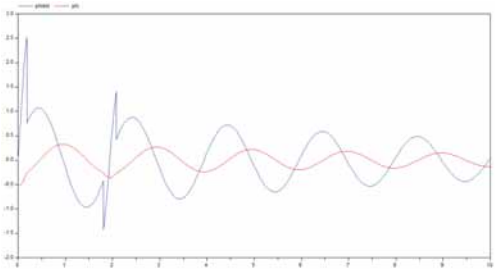

Figure 7. The angle (red, inner graph) and the angular velocity (blue) of the constrained pendulum (example 2)

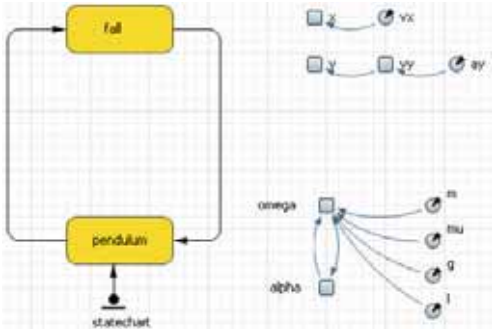

Figure 8. UML statechart controlling the pendulum on a string.

Which model is active is controlled by an UML statechart, see figure 8. Therefore two different states are needed, the state *pendulum* for the first submodel and the state *fall* for the second submodel. Two transitions control the state of the system. The condition of the transition from state *pendulum* to state *fall* is given by

$$\omega^2 r < g \tag{7}$$

expressing that the pendulum begins to fall when the gravity force is larger than the centrifugal force. The condition of the transition from state *fall* to state *pendulum* is naturally given by the constraint of the pendulum length, thus

$$x^2 + y^2 \geq l \tag{8}$$

When a state is entered all initial values for the system must be calculated from the previous submodel. When switching from *pendulum* to *fall* following initial values are preset:

$$\begin{aligned} a_y &= g \\ v_x &= \cos(\alpha)\,\omega l \\ v_y &= -\sin(\alpha)\,\omega l \\ x &= \sin(\alpha)\,l \\ y &= \cos(\alpha)\,l \end{aligned} \tag{9}$$

In the case of the transition fall to pendulum following initial values are chosen:

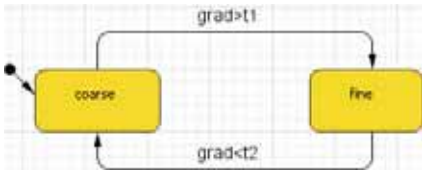

Figure 9. UML statechart controlling the accuracy of the thermal layers.

$$\alpha = \arcsin\left(\frac{x}{l}\right)$$
$$\omega = \cos(\alpha)\, v_x - \sin(\alpha)\, v_y \qquad (10)$$

The example show the possibilities of UML statecharts controlling the model structure. The main benefit of this kind of representation is the that model is clearly arranged and dependencies are shown in an intuitive way.

5 Solar heating system

The UML statechart in figure 9 controls the two different components of the problem, which solve the thermal model with a different number of nodes.

In the original solution the model is linked to external simulation software for numerical FEM calculations. In AnyLogic this link to external code can be established easily. The reason for that is that AnyLogic can be extended by arbitrary Java code. This make it possible to either start applications or to communicate with external code via the Java Native Interface (JNI).

JNI is a programming framework that allows Java code to call and be called by native applications and libraries written in other languages such as C or C++. With JNI it is possible to export and import parameters using a predetermined interface. In the case of C++ this interface look like

```
1 // C++ code
  JNIEXPORT void JNICALL Java_ClsName_MethodName
     (JNIEnv *env, jobject obj, jstring javaStr)
2 {
     // For example a string is imported from AnyLogic
     const char *nativeString =
        env->GetStringUTFChars(javaString, 0);
3
     // Do something with the nativeString
4
     // Don't forget to release the string
5    env->ReleaseStringUTFChars
6       (javaString, nativeString);
7 }
```

6 Conclusion

AnyLogic is a hybrid simulator which supports a multitude of different modeling approaches, particularly UML statecharts, System Dynamics, Agent Based simulation and Dynamic Systems. In this work it has been focused on UML statecharts in combination with Dynamic Systems for the description of structural dynamic systems. In theory all different approaches can be freely combined.

AnyLogic is feasible tool to create UML statecharts and can handle structural dynamic systems in an very intuitive way. AnyLogic works strictly object oriented and translates the models to Java code and further Java code is used within the models. This ability of AnyLogic makes it easy to extend and thus applicable to a huge range of applications.

The main scope of AnyLogic version 6 is business simulation. For engineering application it is a drawback that the integration method cannot be chosen freely. Further there is no state event finder which can lead to a significant reduction of the step size in the temporal domain.

References

[1] A. Filippov and A. Kornev. *C7 constrained pendulum anylogic: Hybrid modeling approach model level*. Simulation News Europe, 32, 2001.

[2] F. Judex. *C7 constrained pendulum anylogic: Hybrid approach*. Simulation News Europe, 35, 2002.

[3] Ch. Nytsch-Geusen et al. *Advanced modeling and simulation techniques in Mosilab—a system development case study*. Proceedings of the 5th international modelica conference, 2006.

[4] A V. Borshchev, Y. B. Kolesov, and Y. B. Senichenkov. *Java engine for UML based hybrid state machines*. Winter Simulation Conference, 2000.

[5] A. Borshchev. *Anylogic 4.0: Simulating hybrid systems with extended uml-rt*. Simulation News Europe, 31:15–16, 2001.

Corresponding author: Daniel Leitner
Vienna University of Technology
Department of Analysis and Scientific Computing
Wiedner Hauptstraße 8-10, 1040 Wien, Austria
dleitner@osiris.tuwien.ac.at

Accepted EUROSIM 2007, June 2007
Received: September 21, 2007
Revised: May 15, 2008
Accepted: June 10, 2008

SNE 18/2, August 2008

Classical and Statechart-based Modeling of State Events and of Structural Changes in the Modelica Simulator Mosilab

Günther Zauner, dieDrahtwarenhandlung Simulation Services Vienna, Austria
Florian Judex, Vienna University of Technology, Austria
Peter Schwarz, Fraunhofer Institute for Integrated Circuits Dresden, Germany

Mosilab (MOdelling and SImulation LABoratory) a new simulation system developed by Fraunhofer understands Modelica, offers different modeling approaches, and supports structural dynamic systems. This will be discussed on the basis of a main example, the classical constrained pendulum. We show how the solution can be done using only standard Modelica components, where the benefits are and which kind of switching the states can be done. As we will see there is no possibility to define separate submodels with different state space dimensions and switch between these systems during one simulation run.

The next point of view lies on an extension of the Modelica framework. The most important new feature of this model description language is the definition of a statechart framework. With this construction the next three solutions of the constrained pendulum are done. The first approach is mathematically similar to the Modelica solution and defines poor parameter events within the statechart construct. This approach cannot handle events of higher order. The second approach for the model is done with two different submodels, one for the case that the rope of the pendulum is short and one for the case it is long. In the statechart the two models are then connected and disconnected to the main program and thereby switched between active and off. A third approaches with only one submodel but two instances of the system will conclude our model inspection.

We focus on how the numerical approaches are done in general and where are the benefits comparing to the other solutions. A final step is to look at the numerical quality of the output of the different approaches. This is done by validation with another example for which an analytical solution exists.

General

In the last decade a broad amount of knowledge in model description theory and modeling and simulation techniques, which could not be solved with the older systems, have their renaissance. Increasing power of computers and better algorithms lead to advanced modeling environments. One benefit are the customer friendly interfaces.

Nevertheless, these advanced modeling environments ask for well educated experts in the field of simulation. In nowadays definition of a project it is very often important to model a part of a system in detail, but when the system switches to another state the description is done imprecisely. Another often needed approach is that a state event makes restrictions to the actual model which leads to a change in the degree of freedom. Both here explained cases result in a change in the state space dimension or even a parameter change for the given system.

The new generation of simulation systems handles this challenge with different methods. One approach is to define a discrete class, where state event handling is done (e.g. ACSL), others restrict their system. They allow only parameter changing state events (e.g. Dymola/Modelica) or to blow up the whole system. A third class, which we will focus on in a selected example, is simulators with an implemented state machine. This group of simulators can handle state events of both sorts: the classical ones where the dimension of the state space remains the same and the hybrid switch between separate models.

1 The simulation environment

MOSILAB (MOdeling and SImulation LABoratory) [1] is a simulator developed by Fraunhofer-Instituts FIRST, IIS/EAS, ISE, IBP, IWU and IPK within the research project GENSIM [2]. It is a generic simulation tool for modeling and simulation of complex multidisciplinary technical systems. The simulation environment supports the procedures modeling, simulation and post processing. The model description in MOSILAB is done in the Modelica [3] standard. Additional features are implemented to assure high

flexibility during modeling the concept of structural dynamics. This is done by extending the Modelica standard with state charts to control dynamic models.

The resulting model description language is called MOSILA [4]. The textual editor in which the model setup is done is expanded by the component diagram as in other Modelica simulators. But there exits another graphical layer which supports state chart definition by using UML diagrams (Unified Markup Language). This is one main benefit compared to some other tools, because the event handling can be done intuitively and the thereby defined program code can be modified and extended in the textual layer as well. Moreover, simulator coupling with standard tools (e.g. MATLAB/Simulink, FEMLAB) is realized. Features for coupling a new simulator with MATLAB are in general used for optimization. The included MATLAB algorithms can be used and so, the system runs in co-simulation, whereby the model is defined intuitively in Modelica standard with additional states and the optimization routine is started in MATLAB/Simulink.

Mosilab offers a list of explicit and implicit integration methods for solving the defined system of DAEs (Differential Algebraic Equations). The default method is the IDA Dassl routine. This method is capable to handle stiff systems. The other implemented methods are Explicit Euler, Implicit Euler, Implicit Trapeze and Explicit Trapeze.

2 Modeling

In this section three different models will be explained in detail. The first one, the constrained pendulum, is used to show the high flexibility of Mosilab and to represent the different ways of implementing a state event. The second is a linear model [6] for which an analytical solution exists and which is used to show the mathematical correctness of the implemented solution algorithms.

The third one gives an overview about advanced modeling and simulation with Mosilab/Modelica, it is a model of the free pendulum. Out of the given model definition we will see that a pure Modelica solution is not possible any more, because the dimension of the state space changes. This happens when a statechart is inevitable in Mosilab.

2.1 Constrained pendulum

The constrained pendulum is a classical nonlinear model in simulation techniques. To make the problem easier than it is in real life, we assume the mass m is large enough so that, as an approximation, we state that all the mass is contained at the bob of the pendulum (that is the mass of the rigid shaft of the pendulum is assumed negligible). This model has been presented in the definition of ARGESIM comparison C7 [5]. There is no exact analytical solution to this problem. Therefore, the results have to be obtained by numerical methods. In this section a description of the model will be given.

$$ml\ddot{\varphi} = -mg\sin(\varphi) - dl\dot{\varphi} \tag{1}$$

Hereby φ denotes the angle in radiant measured in counter clockwise direction from the vertical position. The parameters in the model are the mass m and the length of the rope l. The damping is realized with the constant d. In Mosilab it is an important difference, if the modeler is using *constant* or *parameter*!

As it is a constrained pendulum a pin is fixed at a certain position. This position is given by the angle angle φ_p and the length l_p. Every time when the rope of the pendulum hits the pin the length of the pendulum has to be shortened. In this case the pendulum swings on with the position of the pin as the point of rotation and the shortened length

$$l_s = l - l_p \tag{2}$$

We will focus on the first example defined in the ARGESIM comparison C7, where the following parameters, constants, and initial values are defined:

$$\begin{aligned} &m = 1.02, \quad l = 1, \quad l_p = 0.7, \quad g = 9.81 \\ &\varphi_{\text{start}} = \pi/6, \quad \dot{\varphi}_{\text{start}} = 0, \quad d = 0.2, \quad \varphi_p = -\pi/12 \end{aligned} \tag{3}$$

2.2 Two state model

The here defined model is based on the definition of the ARGESIM comparison C5 [6]. This is a system with two coupled differential equations with a classical parameter state event. The reason why we chose this more or less simple example is, that in contrast to the system defined in 2.1 this system can be solved analytically and therefore we can compare the solution generated in Mosilab with the original analytical solution. Furthermore the different model approaches can be compared pertaining to the solution quality.

This example tests the ability of the simulator to handle discontinuities of the aforementioned type in a satisfactory way. The problem is as follows:

$$\begin{aligned} \dot{y}_1 &= c_1(y_2 + c_2 - y_1) \\ \dot{y}_2 &= c_3(c_4 - y_2) \end{aligned} \tag{4}$$

This ordinary differential equation (ODE) system is essentially a simple linear stiff problem with exponential decays as analytical solution. One of these is a very rapid transient, and the stationary solution of the slow decay varies from the two states of the model. This actually 'drives' the model (and the discontinuity).

The parameter c_1 and c_3 stay unchanged during simulation. The parameter c_2 is 0.4 and c_4 is 5.5 when the model is in state 1 (also the initial state). The initial values are $y_1(0) = 4.2$ and $y_2(0) = 0.3$. The model remains in state 1 as long as $y_1 < 5.8$. The choice of c_2 and c_4 ensures that y_1 will grow past 5.8.

When the model switches to state 2, parameters c_2 and c_4 change to $c_2 = -0.3$ and $c_4 = 2.73$. The model remains in state 2 as long as $y_1 > 2.5$. When passing this instance the model switches back to state 1; the choice of c_2 and c_4 ensures that this will happen.

Analytical solution values can be found. We are focusing on a simulation period starting at time point 0 and ending at time point 5. For comparison we state that the last discontinuity occurs at time 4.999999646 and the $y_1(5.0)$ value should be approximately 5.369.

2.3 Free pendulum on a string

Until now the definitions of systems of interest have been looking on models where the state space dimension does not change during simulation. The state events can all be interpreted as simple parameter events. Now a system is given where the state space dimension has to be changed for real.

This example is a little bit more complicated. Let us again consider a pendulum. The massive bob of the pendulum is fixed on a string. The general structure of the system is depicted in Fig. 1 [5].

In case of a rollover of the pendulum it can start to fall freely until the constraints of the string apply again. This can happen if the pendulum swings higher than $\pm\pi/2$ and the centrifugal force is smaller than the gravitational force.

Accordingly, this model has two different states:

- The normal pendulum movement, and
- the free fall case.

The movement of the pendulum is given in equation (1). We have to define the equations for the free fall case. They are given by

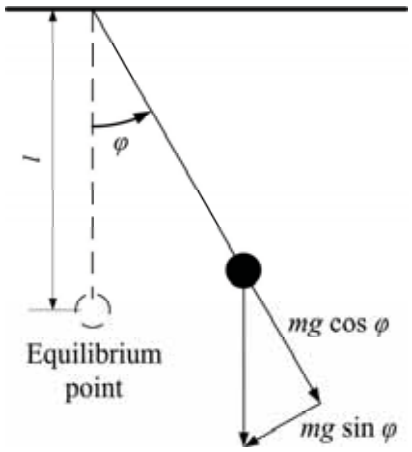

Figure 1. Force diagram of the model.

$$\dot{v}_y = -g$$
$$\dot{v}_x = 0 \qquad (5)$$

For our model we have an additional constraint, which is based on the fixed length l of the pendulum:

$$x^2 + y^2 \le l \qquad (6)$$

This model cannot be solved using simple parameter state events and is defined to give an example that problems in simulation of technical systems as well as in biology, genetics, etc. occur not only in very sophisticated systems. As seen here the need for state space switching in nowadays modeling and simulation techniques is quite common.

After the definition of the main tasks and the extra example we will have a closer look on the implementation approaches of the constrained pendulum and test the simulator by solving different solution of the two state model and comparing them with the analytical solution.

3 Solutions of the constrained pendulum

In this chapter the most important different solution approaches in Modelica of the classical constrained pendulum are discussed. Benefits and restrictions of the different implementations are listed. In the implementations of the constrained pendulum the tangential velocity is used instead of angular velocity. This has the benefit that only the length of the pendulum has a discrete change in case of hitting or leaving the pin.

3.1 Standard Modelica approach

In this approach only standard Modelica code is used. It is defined in the Mosilab equation layer, which is

part of the model editor. The model can be formulated as implicit law, which means that it is not necessary to transform the equations to an explicit form:

```
1 equation
2   v = l1*der(phi);  vdot = der(v);
3   mass*vdot/l1 + mass*g*sin(phi) + damping*v=0
```

The state event, which appears every time when the rope of the pendulum hits the pin or looses the connection to it, is modeled in an `algorithm` section with `if` (or `when`) – conditions:

```
4 algorithm
5   if (phi<=phipin) then length := ls; end if;
6   if (phi>phipin)  then length := l1; end if;
```

This section defines the length of the rope depending on the actual state of the constrained pendulum. Mosilab handles the `if`-command by means of a state event finder. This is important to find the time point of the state event in a given time slice. The solution of the so defined system is depicted in Fig. 2.

In compare with the solutions done in another Modelica simulator (Dymola, in SNE [6]) and the reference solution, this outcome seems reasonable.

3.2 Mosilab state chart approaches

These approaches make use of an additional feature of Mosilab, namely modeling of discrete elements by state charts.

Parameter event solution

The state chart is used instead of the algorithm section and therefore instead of the `if`- or `when`-construct. This has the benefit of much higher flexibility and readability in case of complex conditions. Boolean variables define the status of the system and are managed by the state chart. This can be solved as follows:

```
1 event Boolean lengthen(start=false),
                shorten(start=false);
2 equation
3    lengthen = (phi>phipin);
4    shorten  = (phi<=phipin);
5 statechart
6 state LengthSwitch extends State;
7    State Short, Long, Initial(isInitial=true);
8    transition Initial->Long end transition;
9    transition Long->Short event shorten
        action length := ls;
10   end transition;
11   transition Short->Long event lengthen
        action length := l1;
12   end transition;
13 end LengthSwitch
```

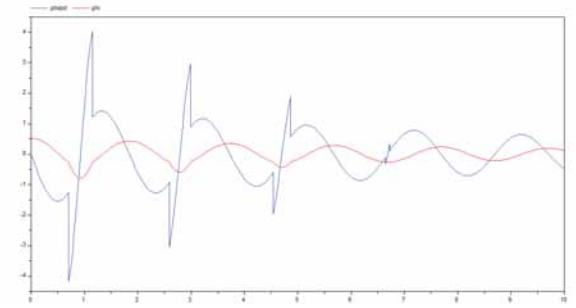

Figure 2. Solution of the task defined in section 3.1, the red (inner) curve represents the angle, the blue curve depicts the angular velocity.

From the modeling point of view, this is equivalent to the description with `if`-clauses. The Mosilab translator generates an implementation with different internal equations. Mosilab performs a simulation by handling the state event within the integration over the simulation period.

Mosilab switching solution

As already explained Mosilab's state chart engine is not only an alternative to the Modelica `if`- or `when`-construct, it is much more powerful.

This system allows any kind of hybrid model composition with models of different state spaces and also of different types. For the constrained pendulum we decompose the system into two different models:

- SHORT, for the case that the rope has contact to the pin, and
- LONG, for the standard damped pendulum.

These two models are than controlled by a state chart, defined in a similar way as shown in the UML-diagram in Fig. 3.

As seen, the new model description comprehends now three parts, the main program which also consists of the state chart and two submodels. These two submodels can be defined separately, or because of the special structure, can be instances of one defined class.

The following source code is using the first method for implementation and first defines the two separate models and afterwards the main program.

```
1 model ConstrainedPendulum
2 model Long
3   equation
4      mass*vdot/l1 + mass*g*sin(phi) +
          damping*v = 0;
5 end Long;
```

```
model Short
  equation
     mass*vdot/ls + mass*g*sin(phi) +
        damping*v = 0;
end Short;
event discrete Boolean lengthen(start=true),
                       shorten(start=false);
  equation
     lengthen = (phi>phipin);
     shorten  = (phi<=phinpin);
statechart
state ChangePendulum extends State;
  State Short,Long,startState(isInitial=true);
  transition startState -> Long
    action
       L := new Long(); K := new Short();
       add(L);
  end transition;
  transition Long -> Short  event shorten
    action
       disconnect ...; remove(L);
       add(K); connect ...;
  end transition;
  transition Short -> Long  event lengthen
    action
       disconnect ...; remove(K);
       add(L); connect ...;
  end transition;
end ChangePendulum;
end ConstrainedPendulum;   // end of model
```

The transitions organize the switching between the pendulums (`remove, add`). The `connect` statements are used for mapping local states to global state variables.

Summing up the results

In center of interest is also the difference in time behavior of the different solution methods. As this is a nonlinear model we can only calculate the numerical solutions and compare, for example, the time points where the last state event appears. This is the moment when the rope of the pendulum looses the connection to the pin the last time. In the model under investigation, this happens after the fourth time shortening the pendulum, which means after eight state events all together.

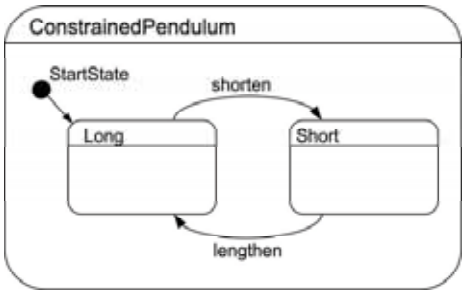

Figure 3. UML-diagram of the statechart solution of the constrained pendulum. The main model controls the two submodels. In this example, the LONG mode is the initial state.

Simulation method	Time of last event	Solution method
Pure Modelica	6.7199	Impl. Trapez Min. step 1E-6 Max. step 1E-4
Switch models	6.7204	IDA Dassl Min step 1E-6 Max step 0.08

Table 1. Comparison of the results.

The solutions are calculated with the default simulation method, if possible. With this approach we try to test the simulation environments from the user's point of view. Many programmers and modelers do not care a lot about the implemented integration methods. For this reason the standard method has to produce reliable results in an appropriate calculation time.

As depicted in Tab. 1, the solutions are quite close but not identically the same. An explanation therefore is that the standard Modelica solution cannot be done with the standard integration method. This could be examined making further tests with different choice of the minimal and maximal step size in each solution method.

4 Two state model

As this is another model where only parameters are changed in the case of the arrival of a state event, this model can be solved in four different ways as explained for the constrained pendulum in chapter 4.

The main differences in compare to the pendulum model are:

- An analytical solution exists for the model.
- Only the first derivative has to be calculated.
- The system is stiff.

The first solution is done again in standard Modelica notation. The most interesting part of the source code is implemented as follows:

```
algorithm
   when (y1 >= 5.8) then
      c2 := -0.3; c4 := 2.73;
   end when;
   when (y1 <= 2.5) then
      c2 := 0.4; c4 := 5.5;
   end when;
```

```
 8 equation
 9    der(y1) =c1*(y2+c2-y1);
10    der(y2) =c3*(c4-y2);
```

The second way of solving this stiff system is to define a state chart in which only the parameters c2 and c4 are changed when an event occurs. We have two cases for this parameter state event. The one when the value of the variable y1 gets higher than 5.8 and the second when this value falls below 2.5. For this model approach we only need one model in which the parameter rules are defined on a higher level.

```
11 event Boolean e(start = false),
                 f(start = false);
12 equation
13    ...
14 statechart
15 state SCSol extends State;
16   State ax, bx, ix(isInitial=true);
17   transition ix -> ax end transition;
18   transition ax -> bx event e
19      action c2:=-0.3; //-1.25;
               c4:=2.73; //4.33;
20   end transition;
21   transition bx -> ax event f
        action c2:=0.4; c4:=5.5;
22   end transition;
23 end SCSol;
```

The source code above shows the easy way of implementation of this task.

The third way of implementation we will focus on concerning this example is to define two separate models which will then toggle between the states. This is the safest way for general implementation of systems with different states. On the one hand this cannot be done with the main part of simulators, on the other hand Mosilab is able to use this structure in different ways: the first approach is to define a submodel and switch between different instances of one and the same class (this would be enough in our case). The second solution is the general switching of active submodels to solve the system. This is done in the implemented solution.

```
 1 model Zustand1 when (y1>=5.8) then
 2   constant Real c1 = 2.7*10^6;
 3   constant Real c3 = 3.5651205;
 4   constant Real c2 = 0.4;
 5   constant Real c4 = 5.5;
 6   Real y1; Real y2;
 7   equation
 8      der(y1) = c1*(y2 + c2 - y1);
 9      der(y2) = c3*(c4 - y2);
10 end Zustand1;
11 model Zustand2 ... end Zustand2
12 event Boolean e(start = false),
                 f(start = false);
13 Real y1(start = 4.2);
14 Real y2(start = 0.3);
15 dynamic Zustand1 Z1;
16 dynamic Zustand2 Z2;
17 equation
18    e = (y1 > 5.8) or (y1 == 5.8);
19    f = (y1 < 2.5) or (y1 == 2.5);
20 statechart
21 state Zustandswechsel extends ANDState;
22   State ax, bx, ix(isInitial=true);
23   transition ix -> ax action
24      Z1:= new Zustand1();
25      Z2:= new Zustand2();
26      add (Z1); Z1.y1:=y1;
27      Z1.y2:=y2;
28      connect(Z1.y1,y1);
29      connect(Z1.y2,y2);
30   end transition;
31   transition ax -> bx event e action
32      disconnect(Z1.y1,y1);
33      disconnect(Z1.y2,y2);
34      remove(Z1);
35      Z2.y1:=y1;
36      Z2.y2:=y2; add(Z2);
37      connect(Z2.y1,y1);
38      connect(Z2.y2,y2);
39   end transition;
40   transition bx -> ax event f action
41      disconnect(Z2.y1,y1);
42      disconnect(Z2.y2,y2);
43      remove(Z2);
44      Z1.y1:=y1;
45      Z1.y2:=y2; add(Z1);
46      connect(Z1.y1,y1);
47      connect(Z1.y2,y2);
48   end transition;
49 end Zustandswechsel;
```

The compendium of the code above shows the basic structure of the problem solution. The next part represents the output part of the system.

4.1 The three solutions compared

After defining the source code, the main interest of the user will focus on the quality of different implementations. From mathematical point of view the implemented solutions are equivalent.

The solutions are all calculated with standard solution method *IDADassl*. Two different step sizes are defined for the experiment. The first with

maxStep = 1e-6, minStep = 0.08,

the second experiment with

maxStep = 1e-12, minStep = 0.0008.

The other settings are all chosen by their default values. No changes are made. The results are all read out of the graphical interface. For more detailed output information the user can take a look at the generated

data files to get more digits in the representation.

The results are depicted in table 2. As we can see, the values are exactly the same in all three approaches. This is very good for model reliability and necessary for further development. In compare with the exact solution of this problem (last event at time point 4.999999646 and the value $y_1(5.0)$ should be approximately 5.369.), we see that our simulation method works in an acceptable quality range. The imprecision of the output occurs also because the user gets only four digits after semicolon for the calculated value. Of course this is normally enough for standard technical system solution, but in our case, namely, for comparing with the exact analytical solution, it is not good enough. The value of the function at the time point 5 is in the allowed range.

5 Summary and outlook

As pointed out in chapters 4 and 5 Mosilab is capable to handle as well nonlinear as linear stiff systems. The Modelica extension for state event handling is a strong tool for advanced modeling concepts. Nevertheless it is important to develop more features and work on the compatibility with the Modelica syntax, so that model exchange can be carried out.

The state chart extension of the Modelica notation is a very useful feature for modeling complex hybrid systems. Because of the state space switching ability it can be used to minimize the simulation time. Furthermore, the models get simpler and the number of equations, that have to be solved are the minimal number during computation.

The possibility to couple the simulation environment with Matlab/Simulink is another important feature of Modelica. As Matlab is very wide spread, a combination of both tools, especially in combination with modern simulation system development and optimization can be done efficiently.

	Settings 1	**Settings 2**
Event	5.0169	5.0000
Value at time 5.0	5.7935	5.8000 - 5.0998

Table 2. Calculated values

6 References

[1] http://mosilab.de/

[2] http://mosilab.de/forschungsprojekt-gensim

[3] http://www.modelica.org/

[4] Thilo Ernst, André Nordwig, Christoph Nytsch-Geusen, Christoph Claus, André Schneider: *MOSILA Modellbeschreibungssprache, Spezifikation, Version 2.0*, from webpage: www.mosilab.de/downloads/dokumentation

[5] http://www.sparknotes.com/physics/oscillations/applicationsofharmonicmotion/section1.html

[6] http://www.argesim.org/comparisons/index.html

[7] F. Breitenecker, H .Ecker and I. Bausch-Gall. *Simulation mit ACSL: eine Einführung in die Modellbildung, numerischen Methoden und Simulation*. Braunschweig: Vieweg, 1993. - XI, 399 S

Corresponding author: Günther Zauner
"die Drahtwarenhandlung" Simulation Services,
Neustiftgasse 57-59, 1070 Wien
guenther.zauner@drahtwarenhandlung.at

Accepted EUROSIM 2007, June 2007
Received: September 15, 2007
Revised: May 5, 2008
Revised: July 10, 2008
Accepted: July 30, 2008

Numerical Simulation of Continuous Systems with Structural Dynamics

Olaf Enge-Rosenblatt, Jens Bastian, Christoph Clauß, Peter Schwarz
Fraunhofer Institute for Integrated Circuits, Germany, *olaf.enge@eas.iis.fraunhofer.de*

In this paper, "continuous systems with structural dynamics" shall be understood as dynamical systems consisting of components with continuous and/or discrete behaviour. (This notation should not be confused with the term "structural dynamics" in the context of Finite Element simulation). Continuous systems with structural dynamics—or so-called "hybrid systems"—can often be investigated only by a so-called "hybrid simulation" which means a simultaneous simulation of continuous-time dynamics (modelled by differential equations or differential-algebraic equations (DAE)) and discrete-event dynamics (modelled e.g. by Boolean equations, finite state machines, or statecharts). To this end, an algorithm for numerical simulation of hybrid systems must be able to both solve a DAE system within a "continuous" time progression as well as to deal with event-driven phenomena.

In the paper, the point of view is emphasized that the structure of a continuous system is closely combined to the structure of the DAE system which describes the continuous system's dynamical behaviour. In this context, discrete-time events are considered as phenomena which may cause a change of the DAE system's structure. Furthermore, a distinction between systems with variable structure and models with variable structure is explained. The main part of the paper deals in detail with a simulation algorithm suitable for hybrid systems. This algorithm consists of a "continuous phase" (for numerical integration of the DAE system) and a "discrete phase" (for interpreting the event, establishing the new valid DAE system, calculating the new initial values). Some simulation results dealing with selected models and using the multi-physics language Modelica will complete the paper.

SNE 18/2, August 2008

1 System structure—what is it?

This paper deals with changes of the "structure" of a dynamic system during a simulation process. But what is the structure of a dynamic system? Many properties could be considered to possibly belong to the structural description of such a system. In mechanical domain, the number of interacting bodies and the number of joints between them belong to the structural information as well as the fact which two bodies are connected by which kind of joint. A body's geometrical shape is of no importance in this context. In electrical domain, the number and types of electrical components and their galvanic connections among each other belong to the structural information. It does not care whether, e.g., a voltage source has a constant value or a sinusoidal time behavior. Similar descriptions can be found for other physical domains (hydraulic, pneumatic, thermodynamic, etc.).

To sum up all these different properties, we assume in this paper that the structure of a system can be interpreted as the structure of its mathematical model, i.e. the number, types and structure of differential and/or algebraic equations belonging to the model. Finally, this structure manifests itself within the fill-in structure of the equation system's Jacobian.

In this paper, a mathematical model which possesses the possibility to change its structure because of some kind of "events" will be denoted to as a model with structural dynamics.

2 Why structural dynamics?

Many physical or technical systems change their properties during operation. Variation of model parameters is a common situation in simulating dynamic systems. But very often, changes of properties occur depending on events which may appear at certain points in time (time-discrete phenomena). In these cases, the complete system shows both time-continuous and time-discrete behavior. Such systems are often called *hybrid systems*. They arise in many fields including robotics, embedded systems, transportation systems, process control, biological and chemical systems, mixed signal (analogue-digital) integrated circuits, etc. Events occurring in hybrid systems can be distinguished into

- events depending only on time (i.e. they can be collected within a time queue) and
- events depending on other physical quantities of the system (i.e. they happen if a variable crosses the zero border value).

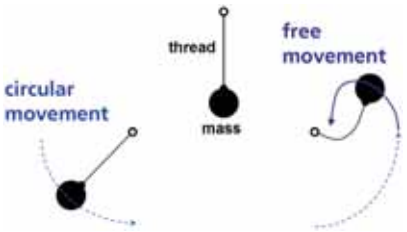

Figure 1. String pendulum

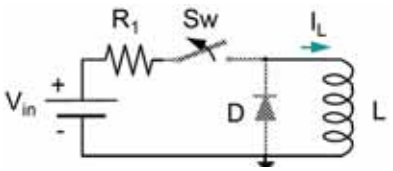

Figure 2. Switched diode circuit

Investigation of hybrid systems has a long-lasting history (see e.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]). Their dynamic simulation is supported by some simulators (see [8]), e.g. Matlab/Simulink, Aspen, gPROMS, Dymola, Saber, Mosilab, and various VHDL-AMS simulators. But most of them only support very simple model variations. A fundamental change of model structure—such as adding or removing differential and/or algebraic equations (which we call "structural dynamics")—is not possible in most simulators. Such behaviour leads to complicated mathematical problems. Mainly, it has to be guaranteed after a structural change that, first, the correct differential-algebraic equations are chosen and, second, a set of consistent initial values of the state variables can be calculated.

From the application point of view, it is important to distinguish between *systems* with a varying structure and *models* with varying structure (but the system itself is not varying). A *system* having a varying structure is characterized either by existence of so-called unilateral constraints (see e.g. [16, 17, 18, 19]) or by appearance of switches for activating or deactivating parts of the system. Such a system does really change its structure or at least its structural information in the behavioural equations during operation. Examples may be found in different application areas:

- *mechanics*: clutches, collision of masses, Coulomb friction, "maximum distance" phenomena (see Fig. 1),
- *electronics*: parts of the system are suspended for a certain time period (e.g. for saving electrical power in mobile communication devices),
- *power electronics*: switches and relays as well as diodes and thyristors (if they are considered as ideal switches, see Fig. 2),
- *adaptive manufacturing machines and roboters*: they have to handle different objects and have to adjust themselves to changing situations.

Other reasons may lead to *varying models* of the same system because system's behaviour shall be investigated under different circumstances. Examples of such reasons may be:

- accuracy shall be adjustable to a more or less detailed model during simulation (to be able to "simulate as accurate as necessary", see Fig. 3),
- usage of different model designs for "dynamic mode" (transient investigation) and "steady-state mode" with the intention to switch between them during simulation (see e.g. [20]).

From our point of view, investigation of hybrid models is much more than a simultaneous simulation of continuous-time dynamics (modelled by differential equations or differential-algebraic equations (DAE)) and discrete-event dynamics (modelled e.g. by Boolean equations, finite state machines, or state charts). A hybrid model should rather be considered as a model which, beyond its continuous time and discrete-event properties, possesses the possibility to change the structure of behavioural equations at cer-

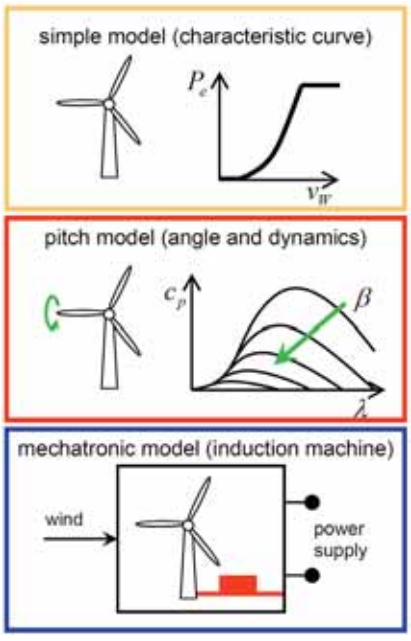

Figure 3. Three levels of wind generator modeling

tain points in time (also called events) because of various reasons. Hence, a simulator for hybrid models has not only to be able to handle continuous and discrete parts with appropriate numerical solvers but, furthermore, should provide concepts and statements for the definition of models with variable structure. This includes a practical solution of the expected numerical issues coming up with a change from one set of differential-algebraic equations to another. The next section gives an overview what types of structural changes may occur with dynamic systems.

3 Types of structural changes

Investigating practical simulation problems, structural changes may arise in different ways. A summary is shown in Fig. 4. The most important issue is "change model behavior" in the first row. This issue includes

- simple substitution of one differential equation by another one,
- exchange of a system of differential equations for another one but with the same order,
- replacement of behavioral or structural description of a component by a totally other one (e.g. a drastic variation of model order, change between continuous and discrete behavior, substitution of a model description by coupling with another simulator).

But also the interconnections between components and, therefore, the structure of the system may change (see rows two to five of Fig. 4). Adding and deleting of certain blocks to/from the complete model (issue "Additional blocks", second row) requires a correct handling of these connectors which are sometimes "opened" (i.e. not connected). In the "open"-case, an additional equation has to be added automatically to the model that enforces the vanishing of the flow variable of the concerning connector. The issue "Change connections" (third row) yields a simple change of parts of some algebraic equations. "Additional blocks and connections" (row four) combines the issues above. The "Change number of ports"-issue may be a consequence of changing the block content from a simple model to a very detailed one or vice versa.

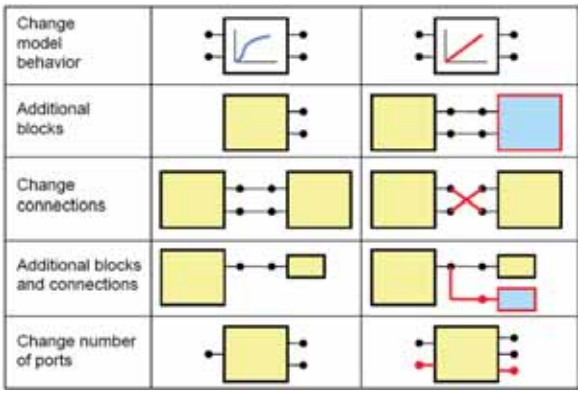

Figure 4. Structural dynamics

4 A hybrid simulation algorithm

4.1 Algorithm principle

The simulation of continuous-discrete systems is supported by many powerful tools. But in handling varying model structures, most simulators have strong restrictions. The Modelica simulator Dymola, e.g., allows that equations may change in an `if-then-else` clause, but the *number of equations* in both branches must be the *same*. Similar restrictions exist in many other simulators which allow the usage of hybrid models.

In this section, an approach for simulating hybrid models is proposed which is able to deal with structural variability. This approach was implemented within the experimental simulator *Mosilab* (see [21, 22, 23]). This simulator was developed within the German applied research project GENSIM by some Fraunhofer Institutes. Within Mosilab, an extension of the language Modelica by a concept for dealing with structural dynamics has been intended. For this purpose, a description of state charts in graphical and textual way was implemented.

In the following, the *structural variability* of a model is characterized using *state charts*. Roughly speaking, every state stands for a certain set of differentialalgebraic equations and every transition realizes a change between different model structures.

The basic algorithm is shown in Fig. 5. It consists of two phases, a *discrete phase* and a *continuous phase*. The main issue in the discrete phase is to update all state machines of the hybrid model (one state machine is described by one state chart) and to establish the new set of differential-algebraic equations if necessary. Hence, all structural changes of a hybrid model are carried out within the discrete phase. In this context, it is assumed that structural changes occur at discrete points in time, i.e. they shall not be

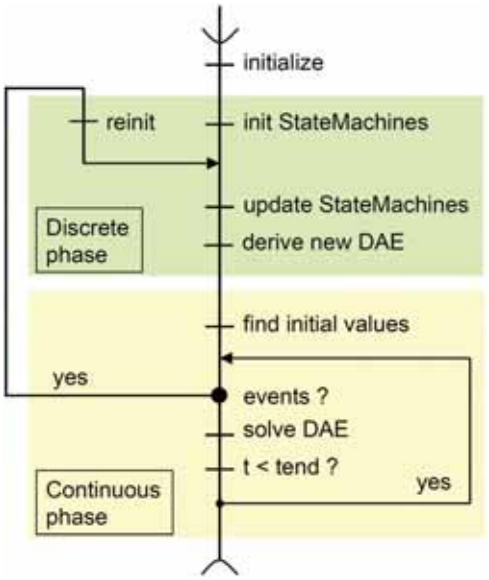

Figure 5. A hybrid simulation algorithm

executed within a certain time interval (zero time assumption). Between two successive discrete points in time, only analogue simulation is performed. This analogue simulation is carried out by a numerical DAE solver and must continue for a minimum time interval which is greater than zero. The length of such a simulation interval may, a priori, either be known or be unknown. If the next discrete-time event happens at a determined (time-fixed) moment then the interval's length is known. Otherwise, e.g. if the next event being expected is triggered by a zero crossing of a variable, the length of the simulation interval is unknown. In the latter case, the relevant quantity has to be monitored in an appropriate way.

4.2 Discrete phase

Fig. 6 shows a more detailed outline (compared to Fig. 5) of the discrete phase of the hybrid simulation algorithm. Please note that *simulation time* keeps *constant* during the complete discrete phase. The algorithms of the discrete phase influence only the discrete parts of the hybrid model. Hence, states and transitions of the state chart diagram are under special focus. But the *model structure* of the continuous submodel (including the DAE set belonging to) and the discrete variables *may be affected*, too.

At start of numerical simulation, all state machines must be initialized by evaluating the initial states and their associated transitions. The main loop of the discrete phase consists of one or more updating processes of the state machines and, after every updating process, the question for new events which may be raised within the last update. During every updating process, two sets of events are to be distinguished: the set of *active events* and that of *waiting events*. One updating process handles all active events and fires the associated transitions successively. New events, which may be raised by fired transitions or by exit actions or entry actions of the associated states, are collected in the set of waiting states. If no more active events are available, one updating process is finished. If now the set of waiting events is empty then the main loop can be closed. Otherwise, another updating process is necessary. For this purpose, all waiting events are transferred into the set of active events and the next update is started.

After leaving the main loop, it has to be proved whether the model structure has changed. This can be done in a very simple way assuming that different activation configurations of the state machines before and after the current discrete phase refer to a change of the structure of the continuous submodel. If the structure is unchanged, the discrete phase can be finished and the following continuous phase is ready to go. In case of structural changes, the set of behavioral equations has to be changed, too. The new set of differential-algebraic equations has to be chosen according to the currently active states in all state machines. At start of the following continuous phase, consistent initial conditions have to be found. To simplify this task—or even perhaps to enable a solution—it may be necessary for the user to define a mathematical algorithm how some initial values of

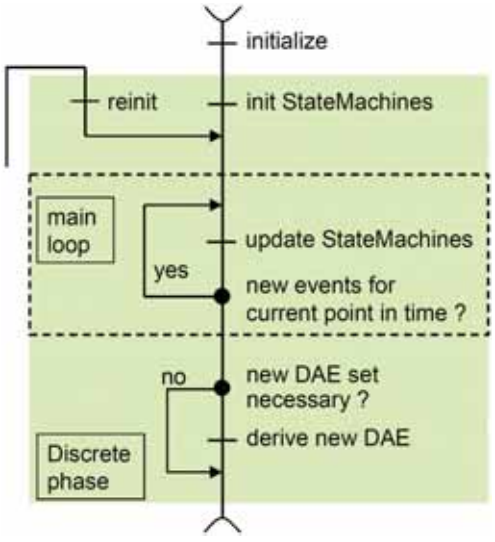

Figure 6. Discrete phase of hybrid simulation algorithm.

the new model structure are to be calculated from the values of the old one. Such an algorithm would have to be specified within the transitions which are responsible for the appropriate structural change.

4.3 Continuous phase

A more detailed outline (compared to Fig. 5) of the continuous phase of the hybrid simulation algorithm is shown in Fig. 7. Please note that within this phase, the *structure* of the complete hybrid model keeps *unchanged*. The algorithms of the continuous phase only affect the continuous parts of the hybrid model. Hence, finding consistent initial values (see e.g. [24, 25, 26]) as well as solving the present set of differential-algebraic equations is the main issue of this phase. But the recognition of possibly occurring events is also important.

The continuous phase begins with an initialization process. In case of carrying out this step for the first time ($t = 0$), the user-given initial values for physical quantities of the continuous model are accepted. Otherwise, the values of the physical quantities calculated within the last discrete phase are used as initial values. Using these values as a start configuration, consistent initial values—i.e. values, which fulfill the constraints of the DAE—have to be found in the next step.

The main task of the continuous phase is to solve numerically an initial value problem of the form

$$F(t, y, \dot{y}) = 0 \qquad y(t_0) = y_0, \quad \dot{y}(t_0) = \dot{y}_0 \tag{1}$$

where y_0 and $\dot{y}_0$ are consistent initial values, i.e. they fulfil the residuum $F(t, y, \dot{y}) = 0$. The vector y consists of both differential variables (the relevant $\dot{y}$-element appears in the DAE) and algebraic variables (no relevant $\dot{y}$-element appears in the DAE). An appropriate numeric solver can be used to solve the problem (1) with advancing time. (In Mosilab, the numeric solver IDA [27] is used.) The numerical integration process may possibly be continued until the end time of simulation t_{end} is reached. However, there are some reasons for stopping the numerical integration at an earlier point in time.

The first reason is the possible appearance of a structural change. In such a case, the numeric solver would have to be stopped at a point in time which lies as near as possible to the moment of event. In order to recognize structural changes, so-called "event variables" are defined. These variables are differential or algebraic variables which may cause an event in the sense of structural dynamics. The event variables are monitored during the numerical integration process.

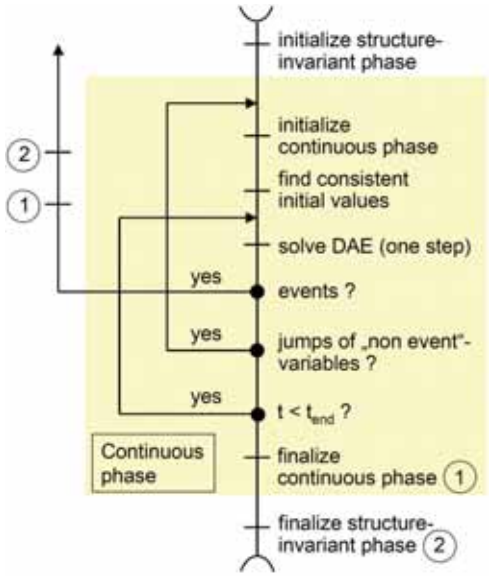

Figure 7. Continuous phase of hybrid simulation algorithm

After each time step of the solver, all event variables are compared to their values before the last integration step. If a change of an event variable is detected then the first moment of changing this variable within the current integration interval must be determined. This can be done e.g. by a root finding algorithm. In this context, it is important to use only numerical solutions at points in time before the event occurs. Otherwise, the accuracy of the calculated moment of event may be affected negatively. After determination of event point in time, the continuous phase is finished and the next discrete phase is started.

The second reason for stopping the numerical integration before reaching t_{end} is the possible jump of a so-called "non event"-variable. Such variables are differential or algebraic variables which are not associated to any event of structural changes. In case of jumping of such a variable, the IDA's integration interval becomes smaller and smaller. If the integration interval drops below a certain border value (denoted by $\Delta t_{\min}$), the solver is reinitialized at the point in time $t_{i+1} = t_i + \Delta t_{\min}$ and new consistent initial values are computed. After that, a new numerical integration process is started.

4.4 Special aspects

The necessary calculation of consistent initial values at each beginning of a continuous phase or after a

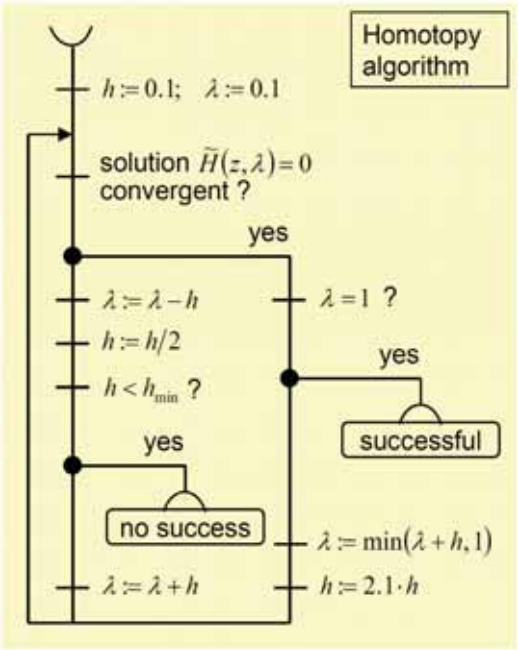

Figure 8. Homotopy method algorithm

jump of a non event variable is sometimes a crucial task. Therefore, the finding procedure may fail. Some helping methods were implemented into hybrid simulation algorithm of Mosilab to overcome this problem. One of them, the homotopy method, shall be mentioned here.

The homotopy method is a procedure to calculate the solution of the problem

$$H(z) = 0 \tag{2}$$

starting from a known solution z_0. For this purpose, the original problem (2) is substituted by the following problem

$$\tilde{H}(z,\lambda) = \lambda H(z) - (1-\lambda)H(z_0) = 0 \tag{3}$$

If $\lambda = 0$, this equation is trivial. By increasing λ stepwise, new problems of the form (3) are established. Generally, the solution z_{k-1} of the preceding problem (3) is used to find a solution z_k of the current problem (3). In case of convergence, this solution is used in the next step (with furthermore increased λ). If no solution z_k can be found then λ is decreased and a new trial is started using z_{k-1}. The complete algorithm as used in Mosilab is shown in Fig. 8.

5 Simulation experiment

In order to show the function of the presented algorithm, some simulation results of a simple 2D string pendulum are given. A detailed sketch of the example is shown in Fig. 9.

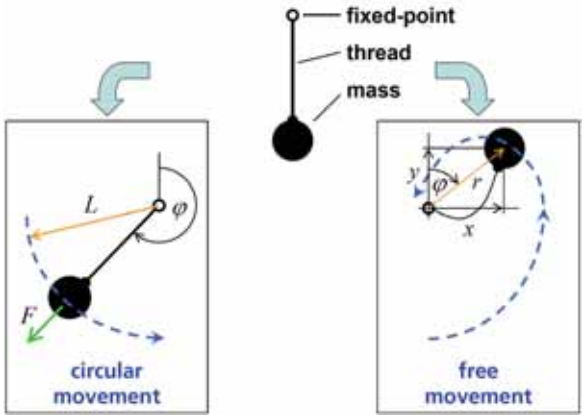

Figure 9. Sketch of string pendulum

A point mass (having the mass parameter m) is attached to a fixed point by a non-elastic thread. The maximum length of the thread shall be denoted to as L. The mass can perform two kinds of movements: a circular movement in case of a fully stretched thread (Fig. 9, left hand side) and a free movement in case of a non-stretched thread (Fig. 9, right hand side). An appropriate state chart is depicted in Fig. 10. The model has two states called `bound` and `free`. Within the circular movement, one differential equation of second order is valid (having the state quantities $\dot{\varphi}$ and φ, where g means the gravity constant and k denotes a damping coefficient). Within the free movement, however, two differential equations of second order are needed (having the positions in x- and y-direction and their time derivatives as state quantities, where k means a damping coefficient and r denotes the current distance between point mass and fixed point).

The system remains in the `bound` state as long as the centrifugal force of the mass holds the thread at its

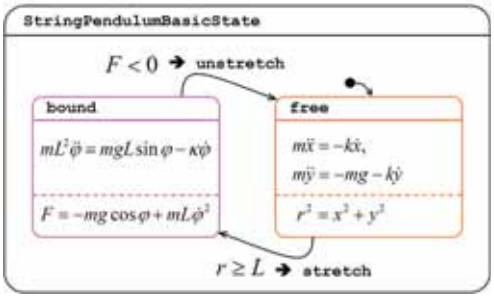

Figure 10. State chart of string pendulum

full length. If the sum of forces acting on the mass drops below zero ($F < 0$), this state will be leaved and the `free` state will become active. The relevant transition is called `unstretch`. Within this transition, the current position and velocities have to be calculated from the last valid values of the physical quantities of the `bound` state. On the other hand, the `free` state is valid as long as the distance between point mass and fixed point is less than the full length of the thread. If the full length is reached or exceeded ($r \geq L$), the system will change from the `free` state into the `bound` state. The relevant transition is called `stretch`. Within this transition, the current angle and angular velocity have to be calculated from the last valid values of the physical quantities of the `free` state. Please note that the energy conservation law may not be fulfilled during this structural change.

Fig. 11 shows an x - y -plot of the string pendulum under the assumption that the mass is located near its rest position at start of simulation and an initial velocity in positive x -direction is given. The pendulum performs two cycles followed by a decreasing oscillation. In the first cycle, two structural changes occur (denoted by no. 1 and 2). The first one switches from circular to free movement, the second one changes contrarily. The same appears within the second cycle (structural changes no. 3 and 4). After the fourth switch, the thread remains stretched to its full length during the decreasing oscillations.

The following figures show time histories of some interesting physical quantities. Fig. 12 depicts the time association to the x - y -plot in Fig. 11. In case of free movement, x and y are differential variables of the DAE, while in case of circular movement, both

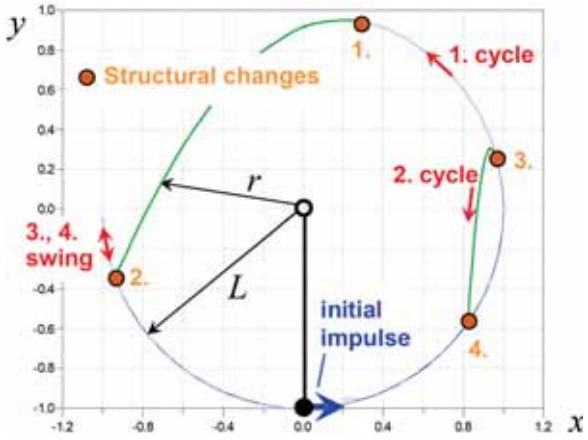

Figure 11. Simulation result: x - y -plot

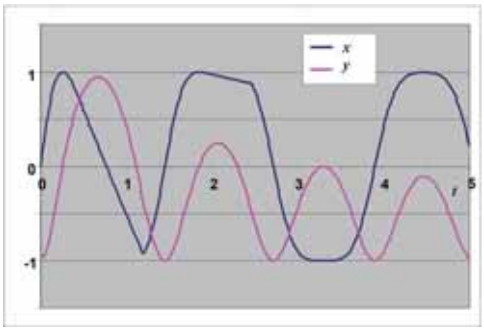

Figure 12. Simulation result: x - and y -coordinates

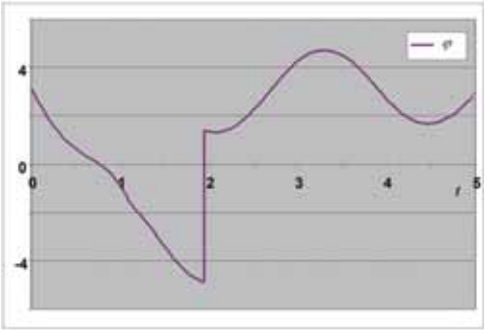

Figure 13. Simulation result: angle

variables have to be computed from the current angle. Contrary to this, Fig. 13 shows the angle φ which is known during the circular movement and has to be calculated within the free movement. The structural changes can be determined best in the curves of the two monitoring variables: the force within the thread (see Fig. 14) and the distance between the mass and the fixed point (see Fig. 15).

6 Conclusion

The numerical simulation of continuous systems with structural dynamics requires simultaneous handling of continuous-time dynamics and discrete-event dynamics. Hence, a tool suitable for simulating such systems must offer facilities to describe both phenomena. In particular, the interactions between both worlds, i.e. triggering events by the continuous model as well as changing the continuous model's structure by events, have to be taken into account.

In the paper, different types of structural changes are listed. The full variety of these cases is hardly supported by well-known simulation tools. Hence, the paper presents a hybrid simulation algorithm consisting of a discrete phase and a continuous phase. The simulator switches between these two phases at certain points in time in an appropriate way. The discrete phase influences only the discrete parts of the hybrid

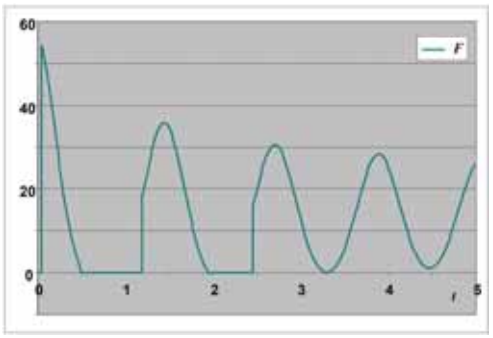

Figure 14. Simulation result: force

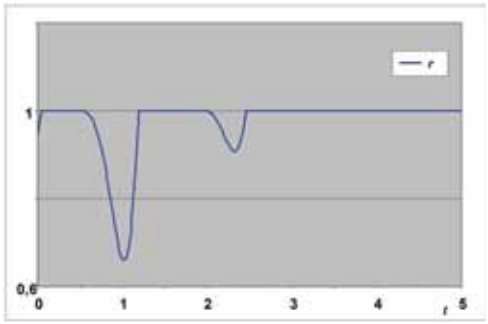

Figure 15. Simulation result: distance

model while the simulation time keeps constant. Execution of events and their consequences on changes of the model structure are under focus. The continuous phase affects only the continuous parts of the hybrid model while the model structure keeps unchanged. The main issue is to find consistent initial values and to carry out numerical integration of the DAE while monitoring relevant variables for recognition of possibly occurring events.

Beyond that, a homotopy method for supporting the overcome of the consistent initial values finding problem is presented. Finally, some simulation results of a string pendulum are given.

References

[1] P. Antsaklis, X. Koutsoukos, J. Zaytoon. *On hybrid control of complex systems: a survey*. J. Européen des Systèmes Automatisés, 32:1023-1045, 1998.

[2] P.I. Barton, C.K. Lee. *Modeling, simulation, sensitivity analysis, and optimization of hybrid systems*. ACM Trans. Mod. Comp. Sim., 12:256-289, 2002.

[3] D.A. van Beek, J.E. Rooda. *Languages and applications in hybrid modelling and simulation: Positioning of Chi*. Control Engineering Practice, 8:81-91, 2000.

[4] F. Breitenecker, I. Troch. *Simulation software – development and trends*. In: H. Unbehauen, editor, Control Systems, Robotics and Automation, Theme in Encyclopedia of Life Support Systems, UNESCO / EOLSS Publishers, Oxford/UK 2004, Article No. 6.43.7.7 [http://www.eolss.net].

[5] F.E. Cellier. *Continuous System Modeling*. Springer. 1991.

[6] F.E. Cellier, H. Elmqvist, M. Otter, J.H. Taylor. *Guidelines for modeling and simulation of hybrid systems*. Proc. IFAC World Congress, Sydney, Australia, 1993, vol.8, 391-397.

[7] H. Gueguen, M.-A. Lefebvre. *A comparison of mixed specification formalisms*. J. Européen des Systemès Automatisés, 35:381-394, 2001.

[8] `http://www.laas.fr/cacsd/hds`

[9] *Hybrid Systems: Computation and Control*. Springer Lecture Notes in Computer Science (LNCS), Proceedings of the HSCC workshops.

[10] KONDISK: German research project on continuous-discrete systems; see `http://www.ifra.ing.tu-bs.de/kondisk/`

[11] E.A. Lee, H. Zheng. *Operational semantics of hybrid systems*. Proc. HSCC 2005, Zurich, Switzerland, Springer LNCS 3414, 25-53.

[12] P. Mosterman. *An overview of hybrid simulation phenomena and their support by simulation packages*. Proc. HSCC 1999, Berg en Dal, The Netherlands, Springer LNCS 1569, 165-177.

[13] M. Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter*. Dissertation, Fortschrittberichte VDI, Reihe 20, Nr. 147, VDI-Verlag, 1995.

[14] M. Otter, M. Remelhe, S. Engell, P. Mostermann. *Hybrid models of physical systems and digital controllers*. J. Automatisierungstechnik, 48:426-437, 2000.

[15] R. Saleh, S.J. Jou, A.R. Newton. *Mixed-Mode Simulation and Analog Multilevel Simulation*. Kluwer, 1994.

[16] O. Enge. *Analyse und Synthese elektromechanischer Systeme*. Dissertation, Shaker, Aachen, 2005.

[17] O. Enge, P. Maißer. *Modelling electromechanical systems with electrical switching components using the linear complementarity problem*. Journal Multibody System Dynamics, 13:421-445, 2005.

[18] C. Glocker. *Dynamik von Starrkörpersystemen mit Reibung und Stößen*. Dissertation, Fortschrittberichte VDI, Reihe 18, Nr. 182, VDI-Verlag, 1995.

[19] F. Pfeifer, C. Glocker. *Multibody dynamics with unilateral contacts*. John Wiley & Sons, 1996.

[20] O. Enge et al. *Quasi-stationary AC analysis using phasor description with Modelica*. Proc. 5th Modelica Conf., Vienna, Austria, 2006, 579-588.

[21] `http://www.mosilab.de`

[22] C. Nytsch-Geusen et al. *Mosilab: Development of a Modelica based generic simulation tool supporting model structural dynamics*. Proc. 4th Modelica Conf., Hamburg, Germany, 2005, 527-535.

[23] C. Nytsch-Geusen et al. *Advanced modeling and simulation techniques in Mosilab: A system development case study*. Proc. 5th Modelica Conf., Vienna, Austria, 2006, 63-71.

[24] P.N. Brown, A.C. Hindmarsh, L.R. Petzold. *Consistent initial condition calculation for differential-algebraic systems*. SIAM J. Sci. Comp., 19:1495-1512, 1998.

[25] C. Pantelides. *The consistent initialization of differential-algebraic systems*. SIAM J. Sci. Stat. Comput., 9:213-231, 1998.

[26] J. Unger, A. Kröner, W. Marquardt. *Structural analysis of differential-algebraic equation systems theory and applications*. Computers Chem. Engng., 19:867-882, 1995.

[27] A.C. Hindmarsh, R. Serban, A. Collier. *User Documentation for IDA v2.5.0*, UCRL-SM- 208112, www.llnl.gov/casc/sundials, 2006.

Corresponding author: Olaf Enge-Rosenblatt
Fraunhofer Institute for Integrated Circuits,
Design Automation Division,
Zeunerstraße 38, 01069 Dresden, Germany
olaf.enge@eas.iis.fraunhofer.de

Accepted EUROSIM 2007, June 2007
Received: June 5, 2008
Revised: July 10, 2008
Accepted: July 20, 2008

SNE - Editors /ARGESIM
c/o Inst. f. Analysis and Scientific Computation
Vienna University of Technology
Wiedner Hauptstrasse 8-10, 1040 Vienna, AUSTRIA
Tel + 43 - 1- 58801-10115 or 11455, Fax – 42098
sne@argesim.org; www.argesim.org

Editorial Info – Impressum

SNE Simulation News Europe ISSN 1015-8685 (0929-2268).

Scope: Technical Notes and Short Notes on developments in modelling and simulation in various areas /application and theory) and on bechmarks for modelling and simulation, membership information for EUROSIM and Simulation Societies.

Editor-in-Chief: Felix Breitenecker, Inst. f. Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria; Felix.Breitenecker@tuwien.ac.at

Layout: Markus Wallerberger, ARGESIM TU Vienna; markus.wallerberger@gmx.at

Printed by: Grafisches Zentrum, TU Vienna, Wiedner Hauptstrasse 8-10, 1040, Vienna, Austria

Publisher: ARGESIM/ASIM; ARGESIM, c/o Inst. for Scientific Computation, TU Vienna, Wiedner Hauptstrasse 8-10, 1040 Vienna, Austria, and ASIM (German Simulation Society), c/o Wohlfartstr. 21b, 80939 Munich

Selection of Variables in Initialization of Modelica Models

Mosoud Najafi, INRIA-Rocquencourt, Domaine de Voluceau, *masoud.najafi@inria.fr*

In Scicos, a graphical user interface (GUI) has been developed for the initialization of Modelica models. The GUI allows the user to fix/relax variables and parameters of the model as well as change their initial/guess values. The output of the initialization GUI is a pure algebraic system of equations which is solved by a numerical solver. Once the algebraic equations solved, the initial values of the variables are used for the simulation of the Modelica model. When the number of variables of the model is relatively small, the user can identify the variables that can be fixed and can provide the guess values of the variables. But, this task is not straightforward as the number of variables increases. In this paper, we present the way the incidence matrix associated with the equations of the system can be exploited to help the user to select variables to be fixed and to set guess values of the variables during the initialization phase.

SNE 18/2, August 2008

Introduction

Scicos (`www.scicos.org`) is a free and open source simulation software used for modeling and simulation of hybrid dynamical systems [3, 4]. Scicos is a toolbox of SciLab (`www.scilab.org`) which is also free and open-source and used for scientific computing. For many applications, the SciLab/Scicos environment provides an open-source alternative to MatLab/Simulink. Scicos includes a graphical editor for constructing models by interconnecting blocks, representing predefined or user defined functions, a compiler, a simulator, and code generation facilities. A Scicos diagram is composed of blocks and connection links. A block corresponds to an operation and by interconnecting blocks through links, we can construct a model, or an algorithm. The Scicos blocks represent elementary systems that can be used as building blocks. They can have several inputs and outputs, continuous-time states, discrete-time states, zero-crossing functions, etc. New custom blocks can be constructed by the user in C and Scilab languages. In order to get an idea of what a simple Scicos hybrid models looks like, a model of a control system has been implemented in Scicos and shown in Figure 1.

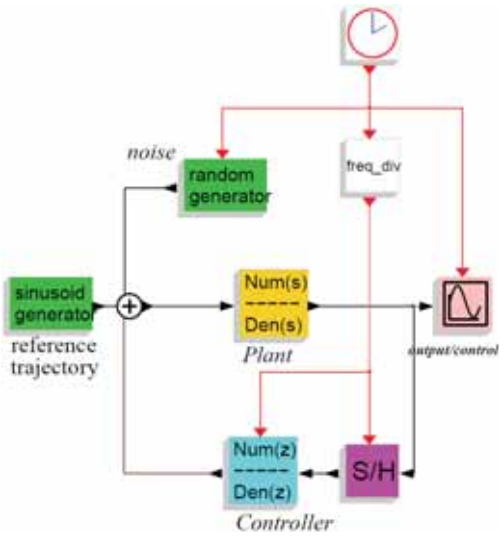

Figure 1. Model of a control system in Scicos

Besides causal or standard blocks, Scicos supports a subset of the Modelica (`www.modelica.org`) language [7]. The diagram in Figure 2 shows the way a simple DC-DC Buck converter has been modeled in Scicos. The electrical components are modeled with Modelica while the blocks that are used to control the On/Off switch are modeled in standard Scicos. The Modelica compiler used in Scicos has been developed in the SIMPA (Simulation pour le Procédé et l'Automatique) project. Recently the ANR/RNTL SIMPA2 project has been launched to develop a more complete Modelica compiler. The main objectives of this project are to extend the Modelica compiler resulted from the SIMPA project to fully support inheritance and hybrid systems, to give the possibility to solve inverse problems by model inversion for static and dynamic systems, and to improve initialization of Modelica models.

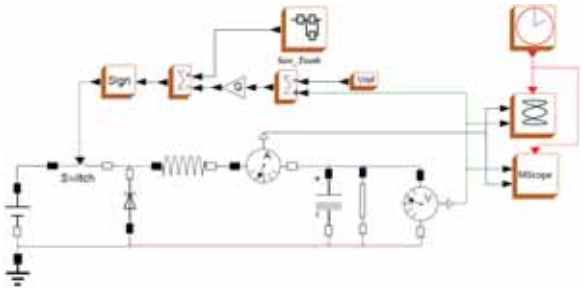

Figure 2. Model of a DC-DC Buck converter in Scicos using Modelica components.

SNE 18/2, August 2008

An important difficulty when simulating a large Modelica model is the initialization of the model. In fact, a model can be simulated only if it is initialized correctly. The reason lies in the fact that a DAE (Differential-Algebraic Equation) resulting from a Modelica program can be simulated only if the initial values of all the variables as well as their derivatives are known and are consistent.

A DAE associated with a Modelica model can be expressed as

$$0 = F(x', x, y, p) \tag{1}$$

where x', x, y, p are the vector of differential variables of size N_d, derivative of differential variables of size N_d, algebraic variables of size N_a, and model parameters of size N_p, respectively. $F(\cdot)$ is a nonlinear vector function of size $(N_d + N_a)$. Since, the Modelica compiler of Scicos supports index-1 DAEs [1, 2], in this paper we limit ourselves to this class of DAEs.

In Scicos, in order to facilitate the model initialization, the initialization phase and the simulation phase have been separated and two different codes are generated for each phase: The initialization code (an algebraic equation) and the simulation code (a DAE). In the Initialization phase, x' is considered as an algebraic variable (i.e., dx) and then an algebraic equation is solved. Modelica parameters p are considered as constants unless they are relaxed by the user. There are $(N_d + N_a)$ equations and $(2N_d + N_a + N_p)$ variables and parameters that can be considered as unknowns. In order to have a square problemsolvable by the numerical solver, $(N_p + N_d)$ variables/parameters must be fixed. The values of x and p are often fixed and given by the user and the values of dx and y are computed. But the user is free to fix or relax any of variables and parameters. For example, in order to initialize a model at the equilibrium state, dx is fixed and set to zero whereas x is relaxed to be computed. Another example is parameter sizing where the value of a parameter is computed as a function of a fixed variable.

In this case, the parameter p is relaxed and the variable x is fixed. In the simulation phase, the values obtained for x, dx, y, p are used for starting the simulation. During the simulation, the value of p (model parameters) does not change.

In Modelica, the `start` keyword can be used to set the start values of the variables. The start values of derivatives of the variables can be given within the `initial equation` section. For small programs, this method can easily be used but as the program size grows, it becomes difficult to set start values and change the `fixed` attribute of variables or parameters directly in the Modelica program; initialization via modifying the Modelica model is specially difficult for models with multiple levels of inheritance; the visualization and fixing and relaxing of the variables and the parameters are not easy. Furthermore, the user often needs to have a model with several initialization scenarios. For each scenario a copy of the model should be saved.

In Scicos, a GUI has been developed to help the user to initialize the Modelica models. In this GUI, the user can easily change the attributes of the variables and the parameters such as `initial/guess value`, `max`, `min`, `nominal`, etc. Furthermore, it is possible to indicate whether a variable, the derivative of a variable, and a parameter must be fixed or relaxed in the initialization phase.

In the following sections, the initialization methodology for Modelica models and the initialization GUI features will be presented.

1 Initialization and simulation of Modelica models

The flowchart in Figure 3 shows how initialization and simulation of Modelica models are carried out in Scicos. The first step in both tasks is removing inheritances. This provides access to all variables and generates a flat model. The flat model is used to generate the initialization and the simulation codes. Note that the initialization data used for starting the simulation is passed to the simulation part by means of an XML file containing all initial values.

In Scicos, three external applications are used in initialization and simulation: `Translator`, `XML2Modelica`, and `ModelicaC`.

`Translator` is used for three purposes:

- Modelica Front-end compiler for the simulation: when called with appropriate options, `Translator` generates a flat Modelica program. For that, `Translator` verifies the syntax and semantics of the Modelica program, applies inheritance rules, generates equations for `connect` expressions, expands `for` loops, handles predefined functions and operators, performs the implicit type conver-

sion, etc. The generated flat model contains all the variables, the derivatives of differential variables, and the parameters defined with attribute `fixed=false`. Constants and parameters with the attribute `fixed=true` are replaced by their numerical values.

- Modelica Front-end for initialization: when called with appropriate options, `Translator` generates a flat Modelica program containing the variables and the parameters defined with attribute `fixed=false`. The derivatives of the variables are replaced by algebraic variables. Furthermore, the flat code contains the equations defined in the `initial equation` section in the Modelica programs. Constants and parameters with the attribute `fixed=true` are replaced by their numerical values.
- XML generator: when called with `-xml` option, Translator generates an XML file from a flat Modelica model. The generated XML file contains all the information in the flat model.

Once the XML file generated, the user can change variable and parameter attributes in the XML file with the help of the GUI. The modified XML file have to be reconverted into a Modelica program to be compiled and initialized. This is done by `XML2Modelica`.

`ModelicaC`, which is a compiler for the subset of the Modelica language, compiles a flat Modelica model and generates a C program for the Scicos target. The main features of the compiler are the simplification of the Modelica models and the generation of the C program ready for simulation. It supports zero-crossing and discontinuity handling and provides the analytical Jacobian of the model. It does not support DAEs with index greater than one. Another important feature of the Modelica compiler is the possibility of setting the maximum number of simplification carried out during the code generation phase. Thus, the compiler can be called to generate a C code with no simplification or a C code with as much simplification as possible. This is an important feature for the debugging of the model.

A new feature of `ModelicaC` is generating the incidence matrix. When a C code is generated, the corresponding incidence matrix is generated in an XML file. The incidence matrix is used by the initialization GUI to help the user.

As shown in Figure 3, once the user requests the initialization of the Modelica model, the Modelica

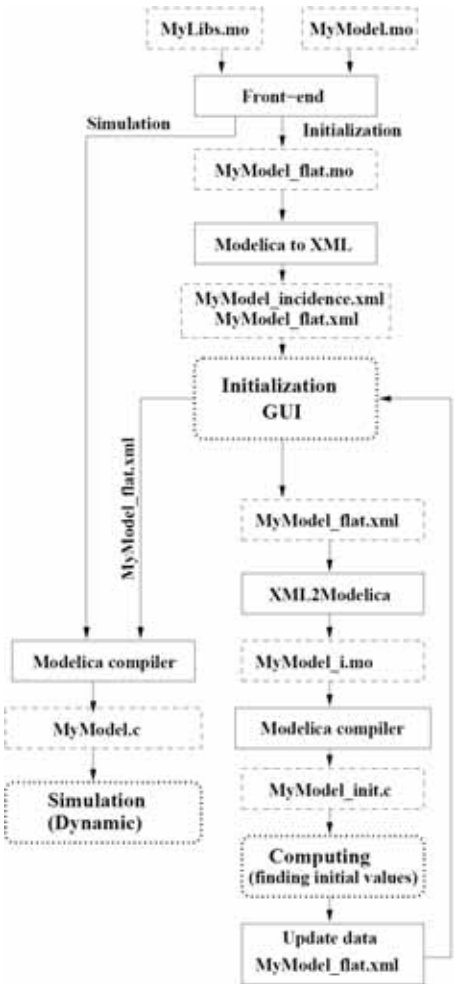

Figure 3. Initialization flowchart in Scicos

front-end generates a flat Modelica model as well as its corresponding XML file. The XML file is then used in the initialization GUI. In the GUI, the user can change the variable and parameter attributes defined in the XML file. The modified XML file is then translated back to a Modelica program. The Modelica program is compiled with the Modelica compiler and a C program is generated. The C program is used by the Scicos simulator to compute the value of unknowns. Once the initialization finished, whether succeeded or failed, the XML file is updated with the most recent results. The GUI automatically reloads and displays the results. The user can then decide whether the simulation can be started or not.

In order to simulate the Modelica model, similar to the model initialization, a flat model is generated. Then, the Modelica compiler simplifies the model and generates the simulation code. The generated code is simulated by a numerical solver. The initial

values, needed to start the simulation, are read directly from the XML file. The end result of the simulation can also be saved in another XML file to be used as a starting point for another simulation.

2 Initialization GUI

In Scicos, a GUI can be used for the initialization of the Modelica models. Figure 4 illustrates a screen shot of the GUI corresponding to the Modelica parts of the Scicos diagram of Figure 2. In this GUI, the Modelica model is displayed in the hierarchical from, as shown in Figure 4. Main branches of the tree represent components in the Modelica model. Sub-branches are connectors, partial models, etc. If the user clicks on a branch, the variables and parameters defined in that branch are displayed and the user can modify their attributes. In the following subsections, some main features of the GUI will be presented.

2.1 Variable/parameter attributes

Any variable/parameter has several attributes which are either imported directly from the Modelica model such as `name`, `type`, `fixed` etc. or defined and used by the GUI *i.e.,* `id` and `selection`.

- `name` is the name of the variable/parameter used in the Modelica program. The user cannot change this attribute in the GUI.
- `id` is an identification of the variable/parameter in the flat Modelica program. The user cannot change this attribute in the GUI.
- `type` indicates whether the original type has been `parameter` or `variable` in the Modelica program. The user cannot change this attribute in the GUI.
- `fixed` represents the value of the `'fixed'` attribute of the variable/parameter in the Modelica program. The user cannot change this attribute in the GUI.
- `weight` is the confidence factor. In the current version of Scicos, it takes either values 0 or 1. `weight=0` corresponds to the `fixed=false` in Modelica whereas `weight=1` corresponds to `fixed=true`. The default value of `weight` for the parameters and differential variables is one, whereas for the algebraic variables and the derivatives of differential variables (converted to variables) is zero.
- `value` is the value of the variable/parameter. If the `weight=1`, the given `value` is considered as the final value and it does not change in the initialization. If `weight=0`, the given `value` is considered as a guess value. If the user does not provide any value, it is automatically set to zero. The user can modify this value in the GUI.

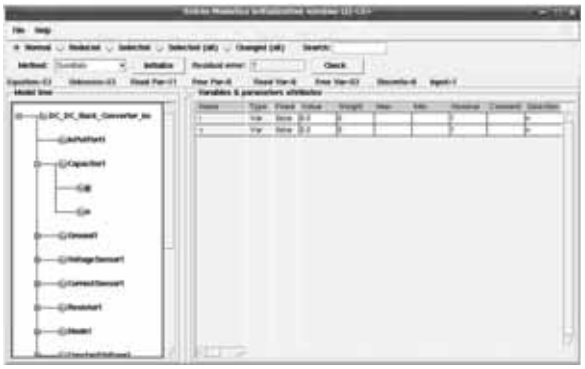

Figure 4. Screenshot of the initialization GUI in Scicos for the electrical circuit of figure 2.

`selection` is used to mark the variables and parameters. This information will be used by the GUI for selective display of variables/parameters and to influence the Modelica compiler in the model simplification phase.

Note that if the user sets the `weight` attribute of a variable to one, it will be considered as a constant and in the initialization phase it will be replaced by its numerical value. On the other hand, if the user sets the `weight` attribute of a parameter to zero, the parameter will be considered as an unknown and its value will be computed in the initialization phase. This is in particular useful when the user tries to find a parameter value as a function of a variable in the Modelica model.

2.2 Display modes

Accessing to variables and parameters of the model becomes easier, if different display modes of the GUI are used:

- **Normal** mode is the default display mode. Clicking on each branch of the model tree, the user can visualize/ modify the variables/parameters defined in that part of the Modelica model.
- **Reduced** mode is used to display the variables of the simplified model. When the user pushes the initialization button, the flat Modelica model is compiled and a simplified model is generated. In this display mode, only the remaining variables are displayed. This display mode is in particular useful when the numerical solver cannot con-

verge and the user should help the solver either by influencing the compiler to eliminate the undesirable variables or by giving more accurate guess values.

- **Selected** mode is used to display only the marked variables and parameters of the active branch. A variable or parameter can be marked by putting `'y'` in its selection field in the GUI. By default, all parameters, all differential variables and all algebraic variables whose start values are given are marked. Marking is useful in particular when a branch has many variables/parameters whereas the user is interested in a few ones. In this display mode, unmarked variables/parameters are not shown.
- **Selected (all)** mode is used to display all marked variables and parameters of the Modelica model.
- **Changed** mode is used to display the variables and the parameters whose `weight` attributes have been changed, such as the relaxed parameters.

2.3 Initialization methods

Once the user modified the attributes of the variables and the parameters, the initialization process can be started by clicking on the "Initialize" button. The initialization consists of calling a numerical solver to solve the final algebraic equation. There are several algebraic solvers available in Scicos such as `Sundials` and `Fsolve` [8, 9, 10].

Once the solver finished the initialization, the obtained results, either successful or not, are put back into the XML file and new values are displayed in the GUI. If the result is not satisfactory, the user can either select another initialization method or help the solver by giving initial values more accurately. This try and error can be continued until satisfactory initialization results are obtained. Then, the simulation can be started.

3 Problems in variable fixing and variable selection

The initialization of DAE (1) can be formulated as the following algebraic problem

$$0 = F(dx_0, x_0, y_0, p_0) \tag{2}$$

where x_0, dx_0, and y_0 are solutions or the initial values of differential variables, derivative of differential variables, algebraic variables, and parameter values, respectively. The degree of freedom of the equation (2) is $N_d + N_p$, therefore the user should fix $N_d + N_p$ variables or parameters and let the solver find the values of the remaining $N_d + N_a$ unknowns.

Fixing the variables/parameters and giving the start values of the relaxed variables/parameters are essential in the initialization of models. But they are not easy and straightforward for large models. In the next subsections the way these problems are handled in Scicos will be explained.

3.1 Fixing the variables

Consider the following equation set, composed of two equations and three unknowns.

$$F: \begin{cases} 0 = f(x) \\ 0 = g(x, y, z) \end{cases} \tag{3}$$

Since the degree of freedom is one, the user should provide and fix the value of a variable. But, it is clear that x cannot be fixed, because its value is imposed by the first equation. In this case, the GUI should prevent the user from fixing x.

Consider the next set of equations composed of three equations and five unknowns.

$$F: \begin{cases} 0 = f(x,u) \\ 0 = g(x,z) \\ 0 = h(x,y,z,v) \end{cases} \tag{4}$$

Although the degree of freedom is two, the user cannot fix (u,z), (x,z), or (x,u) at the same time. In general, it is not easy to identify the set of variables that can be fixed. This is in particular important when the number of equations increases. In this case, if the user tries to fix an inadmissible variable, the GUI should raise an error message and prevent the user from fixing the variable.

This problem can be solved using the incidence matrix of the Modelica model. For example, this is the incidence matrix of (3):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Fixing u and z means removing u and z from the equations which results in the following equation set and the incidence matrix.

$$F: \begin{cases} 0 = f(x,u_0) \\ 0 = g(x,z_0) \\ 0 = h(x,y,z_0,v) \end{cases} \quad \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{5}$$

Although, there are three unknowns and three equations, the incidence matrix is not structurally full rank. This means that u and z cannot be fixed at the same time. Computing the structurally rank of the incidence matrix is a straightforward way to determine if the user is allowed to fix variables or parameters of the model. Since the incidence matrix is very often large and sparse in practical models, we should use special methods for sparse matrices. In the GUI, a *maximum matching* method (also called a maximum transversal method) is used to compute the structural rank of the incidence matrix. The maximum matching method is a permutation of the matrix so that its k^{th} diagonal is zero-free and $|k|$ is uniquely minimized. With this method, the structural rank of the matrix is the number of non-zero elements of the matrix diagonal [6]. When the user tries to fix a variable or a parameter, the initialization GUI computes the new structural rank of the incidence matrix. If the fixing operation lowers the rank, an error message will raised and the modification will be inhibited.

3.2 Selection of variables to be eliminated

Another recurrent problem in solving algebraic equations is the convergence failure of the solver. Newton methods are convergent if the initial guess values of unknowns are not too far from the solution. So, the user should provide reasonable initial guess values. If the problem size is small and the user knows the nominal values of the unknowns, the user can provide the guess values. But in large models, it is nearly impossible to give all guess values. In medium size Modelica models, we usually end up with models with many variables whose start values are not specified by the user. In this case, their initial guess values are automatically set to zero which is not often a good choice. Furthermore, many variables of a model are redundant and the user does not know for which ones the initial guess should be given. This often happens with variables linked by the `connect` operator in Modelica. Suppose that two Modelica components are connected via a connector, *e.g.,*

```
connect(Block1.x, Block2.y);
```

During the model simplification, the compiler may eliminate either `Block1.x` or `Block1.y`. Even if the user knows the guess values of both, it is not reasonable to ask the user to provide them. Since the user has no influence on the compiler's variable selection, this may cause a problem in solving the initialization equation. Consider, *e.g.,* the following situation.

$$F:\begin{cases} 0=\dfrac{x-3}{(x-3)^2+1}-0.1 \\ 0=x-y \end{cases} \tag{6}$$

Here, if the user sets the initial guess of y to 10 and leaves the guess value of x unspecified *i.e.,* $x=0$, although $y=10$ is close to the solution, the Newton's method will likely fail. The reason is that the solver ignores the initial value of y and uses that of x. In fact, there is no way to tell the solver the guess value which is "more" correct than the others.

The solution is to formally simplify the equations by eliminating the variables whose guess-values are not given, by replacing them with the variables having given guess values. For that, in the initialization GUI, variables with known guess-values are marked and the Modelica compiler is told to eliminate the unmarked variables. The user, of course, can modify the list of these marked variables.

The compiler tries to eliminate the variables as much as possible, but a problem may arise when the compiler fails to eliminate all of unmarked variables. Since, the simulator sets their guess-value to zero, the original problem still persists. In this case, the user should be asked to provide the guess-value of the remaining variables. But, usually the user has no idea about the nominal values of the remaining variables or even does not know the physical interpretation of them. As an example, consider the following set of equations for which no guess-values are given.

$$F:\begin{cases} 0=f(x) \\ 0=x-y \end{cases} \tag{7}$$

Suppose that the compiler eliminates y, but the user does not know the start value of x while y has a physical interpretation and its nominal value can be given. In this case, the initialization GUI should propose the user all variables that can replace x, i.e., y.

Proposing alternative variables for formal simplification is done in the initialization GUI. In the next sections, it will be shown the way these problems can be handled by the use of the incidence matrix of the model. This is done using the maximum flow algorithms.

4 Maximum flow problem

The maximum flow problem is to find the maximum feasible flow through a single-source, single-sink flow network [5]. The maximum flow problem can be

seen as a special case of more complex network flow problems. A directed graph or digraph G is an ordered pair $G := (V, A)$ with

- V is the set of vertices or nodes,
- A is the set of ordered pairs of vertices, called directed edges or arcs.

An edge $e = (u, v)$ is considered to be directed from u to v; v is called the head and u is called the tail of the edge; v is said to be a direct successor of u, and u is said to be a direct predecessor of v. The edge (v,u) is called the inverted edge of (u,v).

Given a directed graph $G(V,E)$, where each edge u, v has a capacity $c(u,v)$, the maximal flow f from the source s to the sink t should be found. There are many ways of solving this problem, such as linear programming, Ford-Fulkerson algorithm, Dinitz blocking flow algorithm, etc [12, 11].

4.1 Ford-Fulkerson algorithm

The *Ford-Fulkerson algorithm* computes the maximum flow in a flow network. The name "Ford-Fulkerson" is often also used for the Edmonds-Karp algorithm, which is a specialization of Ford-Fulkerson. The idea behind the algorithm is very simple: as long as there is a path from the source to the sink, with available capacity on all edges in the path, we send flow along one of these paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

Algorithm: Consider a graph $G(E,V)$, with capacity $c(u,v)$ and flow $f(u,v) = 0$ for the edge from u to v. We want to find the maximum flow from the source s to the sink t. After every step in the algorithm the following is maintained:

- $f(u,v) \le c(u,v)$. The flow from u to v does not exceed the capacity.
- $f(u,v) = -f(v,u)$. Maintain the net flow between u and v. If in reality a units are going from u to v, and b units from v to u, maintain $f(u,v) = a - b$ and $f(v,u) = b - a$.
- $\sum_v f(u,v) = 0 \Leftrightarrow f_{in}(u) = f_{out}(u)$ for all nodes u, except s and t. The amount of flow into a node equals the flow out of the node.

This means that the flow through the network is a legal flow after each round of the algorithm. We define the residual network $G_f(V,E_f)$ to be the network with capacity $c_f(u,v) = c(u,v) - f(u,v)$ and no flow. Notice that it is not certain that $E = E_f$, as sending flow on u,v might close u,v (it is saturated), but open a new edge v,u in the residual network.

1. $f(u,v) \leftarrow 0$ for all edges (u,v)
2. While there is a path p from s to t in G_f, such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$:
   a. Find $c_f(p) = \min_{(u,v)\in p} c_f(u,v)$
   b. For each edge $(u,v) \in p$
      i. $f(u,v) \leftarrow f(u,v) + c_f(p)$
      ii. $f(v,u) \leftarrow f(v,u) - c_f(p)$

The path p can be found with, e.g., a *breadth-first* search or a *depth-first* search in $G_f(V,E_f)$. The former which is called the Edmonds-Karp algorithm has been implemented in Scicos.

By adding the flow augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. When the capacities are integers, the runtime of Ford-Fulkerson is bounded by $O(E * f_{\max})$, where E is the number of edges in the graph and $f_{\max}$ is the maximum flow in the graph. This is because each augmenting path can be found in $O(E)$ time and increases the flow by an integer amount which is at least 1. The Edmonds-Karp algorithm that has a guaranteed termination and a runtime independent of the maximum flow value runs in $O(VE^2)$ time.

4.2 Problem of proposition of alternative variables

In order to handle this problem, we build the bipartite graph shown in Figure 5. The left-hand side vertices indicate unknowns, and each vertex at the right-hand side indicates an equation. The edges are bidirectional and their capacity is infinite.

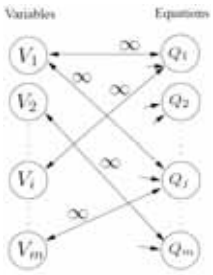

Figure 5. Bipartite graph of variables and equations.

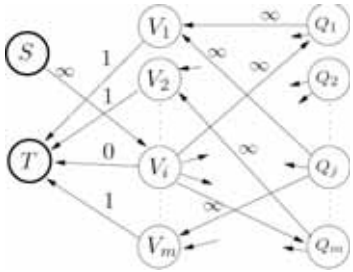

Figure 6. Directed graph for the problem of proposing all alternative variables for V_i.

Note that, at this stage of initialization, the number of unknowns and the number of equations are identical and the incidence matrix is full rank.

For the problem of proposing alternative variables that can be initialized instead of a variable V_i, based on the bipartite graph in Figure 5, we build another directed graph as shown in Figure 6. In this graph, a source vertex and a target (sink) vertex have been added to the graph. The edge connecting the source vertex to V_i has infinite capacity. All m edges connecting the target vertex to the variable vertices have the capacity 1 (except the edge connected to the vertex V_i). The edges are mono-directional.

Now, the problem of finding all alternative variables for V_i is transformed into that of finding of all feasible paths from the source to the target. All predecessors of the target are possible alternative variables that can be used instead of V_i. In the initialization GUI, when the user double-clicks on a variable, its alternative variables are displayed. This is a useful help during the initialization.

5 Initialization iterations

The role of the GUI and the marking in the initialization loop (see the flowchart in the Figure 3) can be summarized in the following algorithm.

1. The GUI automatically marks the model parameters, the differential variables and the algebraic variables whose guess value are given.
2. In the GUI, the user can
   a. visualize/modify the fixed attribute of the variables and the parameters.
   b. change the guess values of variables and parameters (final values if they are fixed).
   c. modify whether a variable or a parameter is marked or not.
3. Initialization is invoked.
   a. If necessary, the model is compiled. The Modelica compiler tries to reduce the number of unknowns by performing several stages of substituting and elimination. In this phase the marked variables are more likely to be eliminated by the compiler.
   b. A numerical solver is used to find the solution of the reduced model.
   c. The obtained solution values are send back to the GUI to be displayed.
4. If the obtained results are satisfactory, goto step 7.
5. The user can readjust the guess values of the remaining unknowns. If there are still unmarked unknowns in the reduced model, either the user can provide more accurate guess values for them or can click on the variables to see their alternatives variables. The alternative variables should be marked to be remained in the reduced model.
6. Goto step 2
7. Start the simulation

6 Example

The model of a thermo-hydraulic system is shown in Figure 7. In this model, there are a pressure source, two pressure sinks, three pipes (pressure losses), a constant volume chamber, and two flow-meter sensors linked to a Scicos scope.

As shown in Figure 8, the initial non-simplified model is composed of 132 equations, 131 relaxed variables and 1 relaxed parameter (*i.e.*, 132 unknowns). The number of fixed parameters and variables are 36 and 1, respectively.

When the model is simplified, the model size is reduced to only 11 unknowns. In Figure 9, where the display mode is `Reduced`, the remaining variables as well as their solution values are shown.

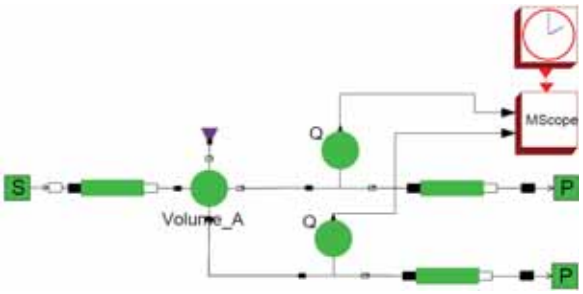

Figure 7. A thermo-hydraulic system.

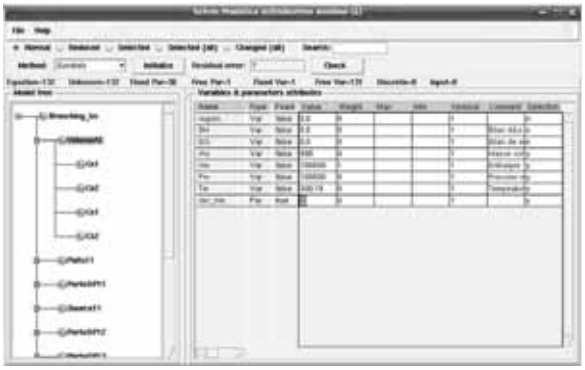

Figure 8. The initialization GUI for the model in Figure 6 (the display mod is `normal` and the variables and the parameters of the block `Volume` are shown)

Figure 9. The remaining variables as well as their initial values after the model simplification. The display mode is reduced.

7 Conclusion

In the Modelica models, initialization is an important stage of the simulation. At the initialization, variables and parameters can be fixed or relaxed and their start values can be changed by the user. In this paper, we presented a special GUI to facilitate the task of selecting fixed and relaxed variables.

Acknowledgements

The author would like to thank Sébastien Furic (LMS. Imagine Co.) for a number of helpful comments. This work is supported by the ANR/SIMPA2-C6E2 project.

References

[1] K.E. Brenan, S.L. Campbell, L.R. Petzold. *Numerical solution of initial-value problems in differential-algebraic equations*. SIAM pubs., Philadelphia, 1996.

[2] P.N. Brown, A.C. Hindmarsh, L.R. Petzold. *Consistent initial condition calculation for differential-algebraic systems*. SIAM Journal on Scientific Computing, 19(5):1495– 1512, 1998.

[3] S.L. Campbell, J-Ph. Chancelier, R.Nikoukhah. *Modeling and simulation Scilab/Scicos*. Springer Verlag, 2005.

[4] J.P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, S. Steer. *An introduction to Scilab*. Springer Verlag, Le Chesnay, France, 2002.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[6] T.A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[7] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[8] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, C.S.Woodward. *Sundials: Suite of nonlinear and differential/algebraic equation solvers*. ACM Transactions on Mathematical Softwar 31(3), pages 363–396, 2005.

[9] A.C. Hindmarsh. *The pvode and ida algorithms*. LLNL technical report UCRL-ID-141558, 2000.

[10] M. Najafi, R. Nikoukhah. *Initialization of modelica models in scicos*. Conference Modelica 2008, Bielefeld, Germany., 2008.

[11] R.L. Rivest, C.E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1990.

[12] D.D. Sleator, R.E. Tarjan. *A data structure for dynamic trees*. In STOC '81: Proc. 13th annual ACM symposium on Theory of computing, pages 114–122, New York, NY, USA, 1981. ACM. 118

Corresponding author: Masoud Najafi
INRIA-Rocquencourt, Domaine de Voluceau,
BP 105, 78153, Le Chesnay, France
masoud.najafi@inria.fr

Accepted EOOLT 2008, June 2008
Received: July 30, 2008
Revised: August 10, 2008
Accepted: August 15, 2008

Introducing Messages in Modelica for Facilitating Discrete-Event System Modeling

Victorino Sanz, Alfonso Urquia, Sebastian Dormido, ETSII Informática, UNED, Spain
{vsanz,aurquia,sdormido}@dia.uned.es

The work performed by the authors to provide to Modelica more discrete-event system modeling functionalities is presented. These functionalities include the replication of the modeling capacities found in the Arena environment, the SIMAN language and the DEVS formalism. The implementation of these new functionalities is included in three free Modelica libraries called ARENALib, SIMANLib and DEVSLib. These libraries also include capacities for random number and variates generation, and dynamic memory management. They are freely available for download at `http://www.euclides.dia.uned.es/`. As observed in the work performed, discrete-event system modeling with Modelica using the process-oriented approach is difficult and complex. The convenience to include a new concept in the Modelica language has been observed and is discussed in this contribution. This new concept corresponds to the model communication mechanism using messages. Messages help to describe the communication between components in a discrete-event system. They do not substitute the current discrete-event modeling capabilities of Modelica, but extend them. The proposed messages mechanism in Modelica is discussed in the manuscript. An implementation of the messages mechanism is also proposed.

SNE 18/2, August 2008

Introduction

Several Modelica libraries have been developed by the authors in order to provide to Modelica more discrete-event system modeling capabilities. The work performed is specially based in modeling systems using the process oriented approach, reproducing the modeling functionalities of the Arena simulation environment [10] in a Modelica library called ARENALib. The functionalities of the SIMAN modeling language [18], used to describe components in Arena, have also been reproduced in a Modelica library called SIMANLib. One objective of the development of this library is to take advantage of the Modelica object-oriented capabilities to modularize as much as possible the development of discrete-event system models. Also, the use of a formal specification to describe SIMANLib components helped to understand, develop and maintain them. SIMANLib blocks can be described using DEVS specification formalism [21]. Event communication in DEVS and block communication in SIMANLib match perfectly. An implementation of the Parallel DEVS formalism [23] has been developed in a Modelica library called DEVSLib, and used to describe the components in SIMANLib. All the performed work with Modelica has been developed using the Dymola modeling environment [1]. The problems encountered during the development of the ARENALib, SIMANLib and DEVSLib Modelica libraries, and the solutions applied to those problems are discussed.

The Modelica language includes several functionalities for discrete-event management, such as `if` expressions to define changes in the structure of the model, or `when` expressions to define event conditions and the actions associated with the defined events [16].

Other authors have contributed to the discrete-event system modeling with Modelica. Depending on the formalism used to define the discrete-event system, contributions can be found using finite state machines [7, 14, 17], Petri nets [15] or the DEVS formalism [2, 3, 4, 8]. On the other hand, other authors have developed tools to simulate discrete event systems in conjunction with Modelica. For example, translating models developed using a subset of the Modelica language to the DEVS formalism. The translated models are then simulated using the CD++ DEVS simulator [5]. Also, other authors describe the discrete-event system with an external tool that translates a block diagram to Modelica code [19].

All these contributions use the event-scheduling approach for describing the discrete-event systems [12]. Events are scheduled to occur in a future time instant. The simulation evolves executing the actions associated with the occurrence of the events.

Due to the difficulties and problems encountered during the development of the mentioned Modelica libraries, the convenience of introducing a new concept in Modelica has been identified. This new con-

cept will facilitate the development of discrete-event systems, extending the current Modelica capacities. This new concept is the model communication using the messages mechanism. The main characteristics and functionalities of this mechanism are also discussed in this manuscript.

1 Process-oriented modeling in Modelica

A discrete-event system modeled using the process-oriented approach is described from the point of view of entities [10]. These entities flow through the components of the system, and some processes are applied to them using the available resources of the system. Some of the information associated with the entities are the serial number, the type, the statistical indicators, the attributes, the creation time, and the processing time among others. An example of this kind of system can be a beverage manufacturing system. The entities of this system are the bottles. A tank fills bottles with the beverage. Once filled, the bottles are labeled and quality controlled before they are accepted for distribution (first and second class bottles). Bottles without the required quality are cleaned and re-labeled. The components of this kind of systems are usually stochastic. For example, the labeling and cleaning processes are modeled using the Triangular probability distribution. The quality controls are represented by two-way decisions whose percentage is based on the values of uniform random variates.

The process-oriented approach is supported by the Arena simulation environment to model discrete-event systems. Arena has *data modules*, that represent the entities, the resources, and some other static elements of the system, and *flowchart modules*, that represent the processes performed on the entities across the system. The implementation of the beverage manufacturing system using Arena is shown in Figure 1a. It is modeled as a hybrid system, because the tank is represented by a continuous time model.

Arena allows some simple hybrid modeling by describing level variables that change continuously over time, and rate variables, that represent how fast the level variable changes its value. Each pair of level/rate variables represents a differential equation that is simulated using Euler, RKF or any user-implemented integration method.

1.1 ARENALib

ARENALib reproduces the Arena data and flowchart modules that have to be combined and connected to

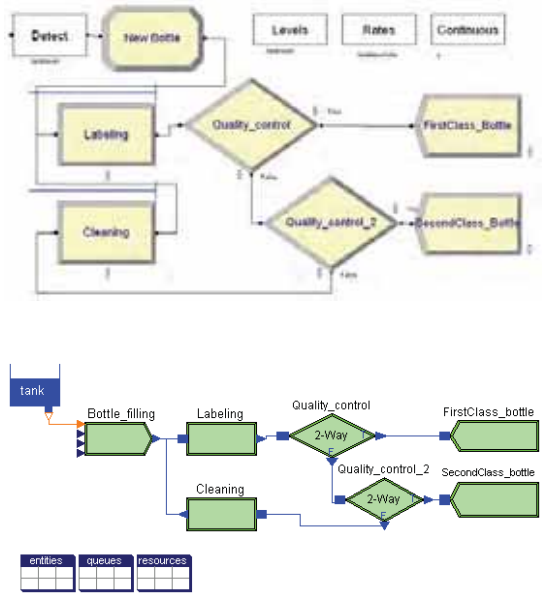

Figure 1. Beverage manufacturing system. An example of hybrid discrete-event system developed using: a) Arena; and b) ARENALib.

model the system. This library is freely available for download at [6]. At the moment, the Create, Process, Dispose and Decide flowchart modules and the Entity, Queue, Resource and Variable data modules, of the Arena Basic Process panel, have been implemented.

The library also allows hybrid system modeling, combining the current Modelica continuous-time system modeling functionalitieswith the components of ARENALib.A detailed description of the library can be found in [20]. The model of the beverage manufacturing system composed using ARENALib is shown in Figure 1b. In this figure, the *Bottle_filling* module corresponds to a Create module, *Labeling* and *Cleaning* correspond to Process modules, *Quality_control* and *Quality_control_2* are Decide modules and the *FirstClass_bottle* and *SecondClass_bottle* are Dispose modules. *Entities*, *queues* and *resources* contain the data modules required for this system.

The main tasks accomplished during the development of the ARENALib library were: a) the model communication mechanism; b) the entity management; c) the management of the statistical information and; d) the generation of stochastic data. These tasks and the solutions proposed and implemented to the problems encountered during the development of the ARENALib library are discussed below.

1.2 Model communication mechanism

Entities are generated in the system during the simulation, flow across the components of the system and, if necessary, are disposed. Generally, the number of entities in the system changes during the simulation run, depending on the behavior of the system.

Usually an entity arrives to a module, is processed and sent to the following module. Entity communication is an important part of the simulation process.

Model interaction in Modelica can be performed using connectors. A connector is an special class and contains some variables that are linked with the ones in another connector using a connect equation. The connect equation relates variables either equaling them, or summing them and equaling the sum to zero.

Several approaches have been studied, implemented and evaluated during the development of ARENALib in order to perform the entity transmission between modules. The approach used to perform the entity transmission is completely transparent for the end user. At the user level, the communication is just defined by connecting the output ports of some modules to the input ports of other modules. The mentioned approaches are discussed next.

Direct transmission

It consists of specifying all the variables that define a type of entity inside the connector. The values assigned to the variables of one connector represent an entity. These values are assigned, because of the connect equation, to the connector of the next model. In this way, an entity is directly transmitted from one model to another. Different types of entities require different connectors, one for each type. This is the simplest way for communicating models, but presents a problem: the simultaneous reception of several entities at one model. There are three possible situations for this problem:

- One-to-one connection: one model sends several entities to another model at the same time.
- Many-to-one connection: several models simultaneously send one entity to another model.
- A combination of the previous cases: several models simultaneously send one, or more, entities to another model.

The two following solutions have been applied to this problem:

1. Synchronizing the entity transmission between models using semaphores. The synchronization allows the sender and receiver to manage the flow of entities between both models, using a send/ACK mechanism like in the TCP/IP communication. Thus, the sender model will send an entity to the receiver and wait for an ACK. On the other hand, the receiver model will receive entities when it is ready to process them, and only send the ACK back if still ready to continue processing more entities. A model of the semaphore synchronizationmechanism, based on a previous work by Lundvall and Fritzson [9], has been implemented and is freely available for download at [6]. A disadvantage of this solution is the performance degradation due to the event iteration that takes place during the synchronization phase of the entity transmission.
2. Including in the connector a `flow` variable that represents the number of entities sent from a model. So, the model receiving the entities will know the number of entities received, even with many senders. However, the information that describes several entities can not be transmitted simultaneously using the direct transmission approach. The variables of the connector that describe the entity can not be assigned with different values, that represent the different transmitted entities, at the same time. Anyway, the text file storage and dynamic memory storage approaches, discussed below, allow to solve this problem using the flow variable.

Text file storage

The idea is to define an intermediate storage for the transmitted entities. This storage behaves as a communication buffer between two or more modules.

The storage is implemented in a text file that stores in each line of text the information related to each transmitted entity. The connector contains a reference to the text file, its file-name, and the flow variable indicating the number of entities received. This reference is shared between the models connected to that connector, allowing them to access the file. Each module is able to receive entities, creates an storage text file and sets the reference to that file in the connector. Functions to read/write entities from/to the file have been developed. A model writes one or several entities to the file using the write function. Another function is used by the receiver to check the number

of entities in the file. When there is any entity to be read, the receiver reads the entities and processes them. Thus, this approach allows the simultaneous reception of several entities.

A disadvantage associated with this approach is the poor performance due to the high usage of I/O operations to access the files. Also, the structure of the information stored in the files is not very flexible if any additional information has to be included. If new types of entities need to be used, or the attributes of an entity have to be changed, the file management functions (i.e. read and write) have to be re-implemented to correctly parse the text file to support these new changes.

Dynamic memory storage

In order to improve the performance of the text file approach, the intermediate storage was moved from the file system to the main memory. Using the Modelica external functions interface, a library in C was created to manage the intermediate storage using dynamic memory allocation. An entity is represented in Modelica using a `record` class, and in C using its equivalent `struct` data structure. Entities are stored using linked-lists structures during their transmission from one model to another. This library is freely distributed together with the ARENALib Modelica library.

Instead of a reference to the file, the connector contains a reference to the memory space that stores the entities, together with the flow variable that indicates the number of entities received. That reference is the memory address pointing to the beginning of the linked-list. It is stored in an integer variable in the connector. Similarly to the text file approach, each model able to receive entities initializes the linked-list and sets the reference to it in the connector. Entities can be transferred to the queue using the write function, and can be extracted using the read function. Another function is used to check the availability of received entities, in order to process them.

This approach also allows the simultaneous reception of several entities. The performance is highly increased compared to the text file approach. And, the structure of the information only depends on the data structures managed by the functions. To modify any attribute or entity type, it is only necessary to change a data structure and not all the functions used to manage that structure.

1.3 Entity management

Regarding the entity management, it has to be mentioned that an additional problem appears when implementing processes that delay the entity. Arena process module can include a delay time that represents the time spent processing the entity. This delay time is usually randomly selected from a probability distribution. It has to be noticed that since the delay time is usually random, the order of the arrived entities need not correspond to the order of the entities leaving the process. These processes have to include a temporal storage for the entities that are being delayed. This problem can be solved using the text file storage or the dynamic memory storage as an additional storage for delayed entities. Due to performance reasons, the dynamic memory approach was used to manage entity storage during delays in ARENALib and SIMANLib.

Together with the initialization of the linked-queue for entity communication, a process module initializes a temporary storage, represented by a linked-list in memory, for delayed entities. The reference to that list is also stored in an integer variable. Every time the process module has to delay an entity, it stores the entity in the list using a write function. Entities are inserted in the list in increasing order, according to the time they must leave the process. The insertion of an entity in the list returns the leaving-time for the first entity in the queue. When the simulation time reaches the next leaving-time, the entity or entities leaving the process are extracted from the list and sent to the next module.

1.4 Stochastic data generation

Discrete-event models usually contain some kind of stochastic information. Random processing times, delays or inter-arrival times help to construct a more realistic model of a given system.

The Modelica language specification does not include any functionality for random number generation. Dymola, the modeling environment used to develop and test the mentioned Modelica libraries, includes two functions for generating random uniform and random normal variates [1]. The generation of random variates following other probability distributions is not covered by these random number generation functions. Also, the application of variance reduction techniques is not supported by these functions.

A random number generator (RNG) was developed by the authors. The RNG algorithm selected for its

implementation in Modelica is the same that is used in the Arena environment. This allows the validation of the ARENALib models using the Arena environment, because both use the same source of random numbers. This RNG algorithm was proposed by Pierre L'Ecuyer and is called Combined Multiple Recursive Generator. A detailed description of the RNG is given in [13].

Additionally to the implementation of the RNG, some functions for generating random variates were also developed by the authors of this manuscript. The new RNG and the random variates generation functions are packaged in a Modelica library called RandomLib, which is freely available for download at [6].

1.5 Statistical information management

Simulation results are usually reported using statistical indicators, due to the stochastic nature of discrete-event systems. Some of these statistical indicators have to be calculated during the simulation and some others at the end. The amount of data that has to be stored to calculate some of these indicators changes depending on the length of the simulation.

Modelica does not allow the declaration of variables with an undefined length or size, which are required to store the statistical data. A mechanism to declare variables of undefined length in Modelica needs to be defined, giving the possibility to increase or decrease the size of the variable during the simulation run.

This problem is very similar to the previously mentioned one about intermediate entity storage for transmission or delay management. So, the mentioned dynamic memory storage has been used in ARENALib to record the information regarding the statistical indicators of the simulation. The indicators calculated in each ARENALib module are shown in Tab. 1. Statistical indicators calculated include the number of entities arrived, the number of entities departed, processing times, the number of entities in queue, and the number of entities in the ystem, among others. The information calculated for each indicator is the mean, the maximum value, the minimum value, the final value and the number of observations. These values are updated during the simulation. On the other hand, all the intermediate values have to be recorded and used to calculate the confidence interval at the end of the simulation. A variable in Modelica stores a reference to the memory space that contains the stored data for each indicator. That space is managed using external functions written in C.

Module	Indicator	Values
Create	System.NumberIn	Obs
Process	NumberIn	Obs
	NumberOut	Obs
	VATime Per Entity	Avg, Min, Max, Final, Obs
	NVATime Per Entity	Avg, Min, Max, Final, Obs
	TotalTime Per Entity	Avg, Min, Max, Final, Obs
	Queue.NQ	Avg, Min, Max, Final
	Queue.WaitTime	Avg, Min, Max, Final, Obs
Dispose	System.NumberOut	Obs
EntityType	NumberIn	Obs
	NumberOut	Obs
	VATime	Avg, Min, Max, Final, Obs
	NVATime	Avg, Min, Max, Final, Obs
	TranTime	Avg, Min, Max, Final, Obs
	WaitTime	Avg, Min, Max, Final, Obs
	OtherTime	Avg, Min, Max, Final, Obs
	Work In Progress	Avg, Min, Max, Final

Table 1. Statistical indicators and values calculated in the ARENALib modules

1.6 SIMANLib

The first approach for the development of ARENALib was to write all its components, except the mentioned external functions and data types which are written in C, in plain Modelica code. This generated large and complex models that were difficult to understand, maintain and extend.

The idea then was to divide the actions performed by each module into simpler actions that combined will offer the same functionality than the original module.

The same structure can be observed in the Arena environment, where the modules are based and constructed using a lower level simulation language called SIMAN [18].

SIMANLib contains low-level components for discrete event system modeling and simulation. These are low-level components compared to the modules in ARENALib, which represent the high-level modules for system modeling. Flowchart modules of both libraries are shown in Figure 2. ARENALib modules can be described using a combination of SIMANLib components. For example, the process module of ARENALib is composed by the Queue, Seize, Delay and Release blocks of SIMANLib, as shown in Fig. 3.

Components in SIMANLib are divided, as well as in the SIMAN language, in two groups: blocks and elements. The blocks represent the dynamic part of the system, and are used to describe its structure and define the flow of entities from their creation to their disposal. The elements represent the static part of the system, and are used to model different components such as entities, resources, queues, etc.

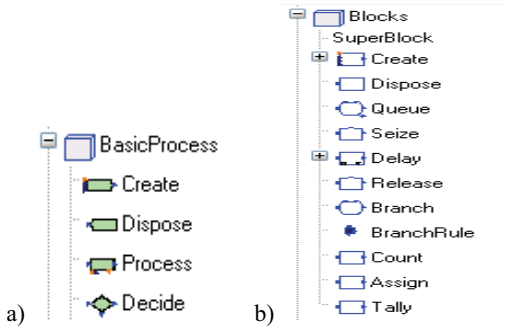

Figure 2. Flowchart modules: (a) ARENALib; and (b) SIMANLib

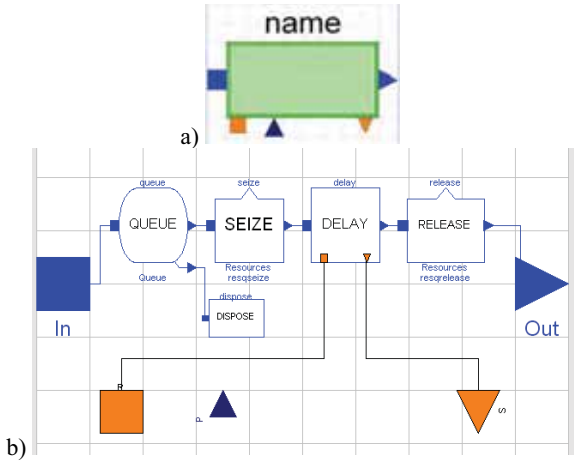

Figure 3. ARENALib process module: a) icon; b) internal structure composed using SIMANLib components.

An example of a model developed using SIMANLib is shown in Figure 4. This system is very similar to the beverage manufacturing system mentioned above. The entities are pieces to be machined. The pieces arrive to the system and are processed by a machine, one at a time. After processed, the pieces are inspected by a supervisor and classified as Good, Reject and Repair. Repaired pieces are sent back for reprocessing.

2 Parallel DEVS in Modelica

The main objective of the implementation of the DEVSLib library has been to closely follow the definition of the Parallel DEVS formalism and implement all its features without restrictions. The functionalities of DEVSLib are similar to the ones offered by other DEVS environments such as DEVSJAVA [24] or CD++ [22]. These similarities include the new atomic and coupled models construction based on predefined classes, the redefinition of the internal, external, output and time advance functions in each atomic model as required by the user and the management of model

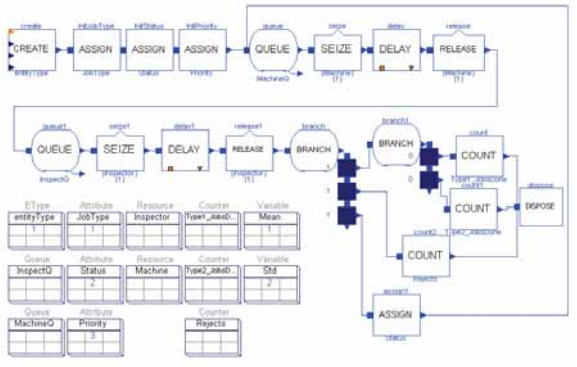

Figure 4. Manufacturing system model composed using SIMANLib components

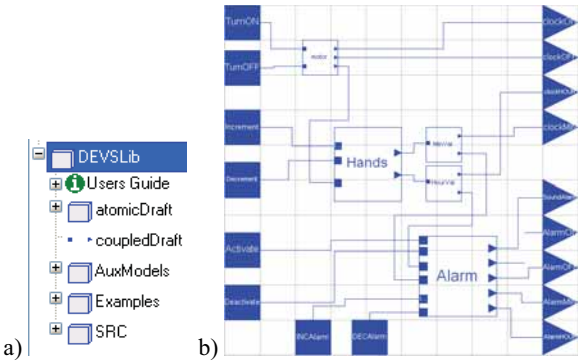

Figure 5. The DEVSLib Modelica library: a) architecture; b) case of use (model of a pendulum clock).

input and output ports as needed. However, due to the capacities of the Modelica language, DEVSLib still presents some restrictions that will be discussed below.

2.1 DEVSLib architecture

The architecture of the library is rather simple. It is shown in Figure 5a. It contains two main models, atomicDraft and coupledDraft, that represent the basic structures for building any new atomic or coupled DEVS models. Together with the main models there are several auxiliary models and functions for managing event transmission. Additionally, some examples of atomic and coupled systems have been included. One of the included examples is the hybrid model of a pendulum clock [11], which is shown in Figure 5b. In this system a continuous-time model of a pendulum generates tics, acting as the motor of the clock. The rest of the clock receives the tics, calculates the current time (in hours and minutes) and manages the alarm of the clock.

2.2 Model development with DEVSLib

When building a new atomic model, the user has to specify the actions to be performed by the external

```
model processor
  extends AtomicDEVS(redeclare record State = st);
    redeclare function Fcon = con;
    redeclare function Fint = int;
    redeclare function Fext = ext;
    redeclare function Fta = ta;
    redeclare function initState =
      initst(dt=processTime);
    parameter Real processTime = 1;
    Interfaces.outPortManager outPortManager1(
       redeclare record State = st,
       redeclare function Fout = out, n=1);
    Interfaces.outPort outPort1;      // output port
    Interfaces.inPort inPort1;        // input port
    equation
       iEvent[1] = inPort1.event;
       iQueue[1] = inPort1.queue;
       connect(outPortManager1.port, outPort1);
end processor;
```

Listing 1. Modelica code of a processor system modeled using DEVSLib

transition, internal transition, output and time advance functions. This can be performed by re-declaring the functions `Fext`, `Fint`, `Fout` and `Fta`, initially declared in the atomicDraft model. The user can specify any desired behavior for these functions, while maintaining the defined function declaration. Any new atomic model has to extend the AtomicDEVS model and to re-declare the mentioned functions. The Modelica code of a processor system [23] developed using DEVSLib is shown in Listings 1, 2 and 3.

The desired number of input and output ports can also be included in the new model and managed with the mentioned functions. The user can drag and drop new input and output ports into the model. The prototypes of the external transition and the output function allow the user to check the port where an incoming event has been received, or to specify the output port to send the event. All these ports could be connected later to other models.

A coupled DEVS model, like the one shown in Figure 5b, can be easily build using previously defined atomic or coupled models, and connecting them as required. The input and output ports have to be included and connected to any of the model components

2.3 DEVSLib modeling restrictions

One restriction in DEVSLib is the impossibility to perform one-to-many connections. These kinds of connections are not considered in ARENALib or SIMANLib because neither Arena nor SIMAN permits them. However, the Parallel DEVS formalism allows this kind of connection so they have been taken into account.

```
function con "Confluent Transtition Function"
   input st s, Real e, Integer q, Integer port;
   output st sout, soutput;
algorithm
   soutput := s;
   sout := ext(int(s),e,q,port);
end con;

function int "Internal Transition Function"
   input st s;
   output st sout;
algorithm
   sout := s;
   sout.phase := 1; sout.job := 0;
   sout.delta := Modelica.Constants.inf;
end int;

function ext "External Transition Function"
   input st s, Real e, Integer q, Integer port;
   output st sout;
protected
   Integer numreceived;
   stdEvent x;
algorithm
   sout := s;
   numreceived := numEvents(q);
   if s.phase == 1 then
      for i in 1 : numreceived loop
         x := getEvent(q);
         if i == 1 then
            sout.job := x.Value;
            Modelica.Utilities.Streams
              .print("* Event to process");
         else
            Modelica.Utilities.Streams
              .print("* Event balked");
         end if;
         sout.received := sout.received + 1;
      end for;
      sout.phase := 2;       // active
      sout.delta := s.dt;    // processing_time
   else
      sout.delta := s.delta -e;
   end if;
end ext;

function out "Output Function"
   input st s, Integer port, Integer queue;
   output Boolean send;
protected
   stdEvent y;
algorithm
   if s.phase == 2 then
      send := true;
      y.Type := 1;
      y.Value := s.job;
      sendEvent(queue,y);
   else
      send := false;
   end if;
end out;

function ta "Time Advance Function"
   input st s;
   output Real delta;
algorithm
   delta := s.delta;
end ta;
```

Listing 2. Modelica code of the functions redeclared in the processor system.

```
record st "State of the model"
  Integer phase;       // 1 = passive, 2 = active
  Real delta;          // internal transitions interval
  Real job;            // current processing job
  Real dt;             // default processing time
  Integer received;    // num of jobs received
end st;

function initst "State Initialization Function"
  input Real dt;
  output st out;
algorithm
  out.phase := 1;     // passive
  out.delta := Modelica.Constants.inf;
  out.job := 0;
  out.dt := dt;
  out.received := 0;
end initst;
```

Listing 3. Modelica code of the state and state initialization of the processor system.

This restriction appears because the way the port and the event communication mechanism is managed, using dynamic memory storage. As mentioned before, each receiver initializes its linked-queue to receive entities. A one-to-many connection cannot be performed because the sender can not store in just one integer variable the references to all the linked-queues created by the receivers. A solution has been implemented in the DEVSLib library. This solution consists in an intermediate model that can be used to duplicate the events and send them to the receivers. Examples of this intermediate model are the *MinValue* and the *HourValue* models shown in Figure 5b.

By default, the information transmitted between models in DEVSLib, at event instants, is composed by two values: the type of the event and a real value. The information communication mechanism using dynamic memory is relatively complex. It will not be easy for a user to change the structure of the information, type and value, transmitted in events. Anyway, it can be performed modifying the Modelica and C data structures that support the communication mechanism. In order to improve the mechanism for managing the information transmitted in events, additional information structures will be included to the DEVSLib library, e. g., giving the possibility to transmit arrays or matrices instead of only real values.

3 Introducing messages in Modelica

A conclusion of the performed work is that discrete event system modeling with Modelica, using the process-oriented approach, is not an easy task. The components required for modeling these kind of systems and the solutions proposed for the problems are relatively complex. The developed libraries provide some functionalities for discrete-event system modeling with Modelica, using the process-oriented approach. Still, there are some problems without a solution, like the one-to-many connections in DEVSLib and the polymorphism of the information transmitted at event instants.

In this section the model communication using messages in Modelica is presented. The authors also propose a possible implementation of this mechanism that will be discussed in Section 4.

3.1 Motivation

The main difficulty observed in the presented work is the model communication mechanism. This is the way models are connected and communicate.

The connection of models in Modelica is represented by the `connect` equation. In a connection equation the value of the variables at the ends of the connection are either equaled, or summed and equaled to zero. A connection between discrete-event models does not establish any relation between variables of both models, but is used to communicate some information that has been generated in one model and is transmitted to another. Both connection concepts mean different things.

Event management is also different between Modelica and DEVS discrete-event systems. An event in Modelica involves a change in the value of a boolean condition that either makes the structure of the model to change, or performs a change in the discrete time variables or the state variables of themodel. Events in DEVS discrete-event systems represent a change in the state of the system or its discrete time variables, and usually also involves the exchange of information between models. This is an instantaneous transmission/reception of an impulse of information between models at the time of an event. Event management in discrete-event systems involve additional things than in Modelica, because of this information communication.

In order to make the development of discrete-event systems more simple and easy, a new concept is proposed and introduced in Modelica. This concept is the messages communication mechanism. The messages mechanism provides the capacity for communicating impulses of information between models at event instants.

3.2 Messages and mailboxes

The model communication mechanism using messages involves two parts: the message itself and the mailbox. The message represents the information either traveling from one model to another, or inside a model itself. The mailbox receives the incoming messages and stores them until they are read. The mailbox also represents the concept of a bag of events in the Parallel DEVS formalism.

The characteristics of the model communication using messages are the following:

- A message can be sent to any available mailbox. Available mailboxes are the ones that can be referenced from the model that sends the message, either accessing directly or using a connection.
- The mailbox warns the model when new incoming messages are received.
- Once received, the message can be read from the mailbox.
- The transmission of messages between models has to be performed instantly. Any message sent from one model will be immediately received by another model.
- Messages can be received simultaneously, either in the same or different mailboxes.
- The information transported by a message, the content, is independent from the message communication mechanism. It is a task of the user to define the structure of that information using the existing components of the Modelica language, so it can be managed by the models.
- Messages can be of different types. A mailbox can store any message independently of its type. The type of the message has also to be independent from the content of the message.
- Received messages have to be stored temporarily in the mailbox, until they are read.
- Message communication has to be performed in two stages: sending and reception. The sending involves the transmission of any message in the system at a given point in time, so all the messages sent are stored in the mailbox at the end. After the sending, all the messages are available for reception in each mailbox and can be read and managed as required. If a model sends several messages to the same mailbox, all the sent messages have to be stored in the mailbox before the first message can be read by the receiver.

3.3 Message sending, transmission, detection and treatment

A message can be sent from one model to any other model that contains a mailbox, even if no connection between models is available.

Mailboxes can also be shared between models. Sharing a mailbox represent that several models can access to the message storage that it represents. Each model sharing the mailbox can access the messages stored, reading or extracting them from it. Read messages are kept in the mailbox until they are extracted, or fetched, from it.

A special case of mailboxes are the ones defined inside connectors. Two mailboxes, inside connectors, connected using a connect equation represent a bidirectional message communication pipe. They will act as input/output mailboxes instead of only receiving messages. A message sent to one end of the pipe will be transported to the opposite end, and vice-versa. If more than two models are connected to the same pipe, a copy of the message will be transported to each receiver connected to the pipe. This provides a message broadcast functionality that also emulates the event transmission in DEVS, however in DEVS the communication is not bidirectional. The connect equation functionalities in Modelica have to be extended in order to support this mailbox behavior. An example of this behavior is shown in Figure 6.

The detection of a message is implicit in the action of sending it, since they are transferred instantly. Every time a model sends a message to a mailbox, the simulator knows that the message will be received by another model and will have to be treated properly.

The treatment of each message has to be defined by the user. The mailbox warns when a new message has arrived. The mailbox activates a listener function that can be used as a condition to detect any incoming message, used with statements like `when` or `if` in Modelica. This does not mean that the new message condition has to be effectively checked at each simulation step, because it is notified by the send message operation. Once a new message arrives to a mailbox, the arrived message or messages have to be read and treated.

4 Proposal of implementation

This section contains a proposal of implementation in Modelica of the previously described message communication mechanism. This implementation is based

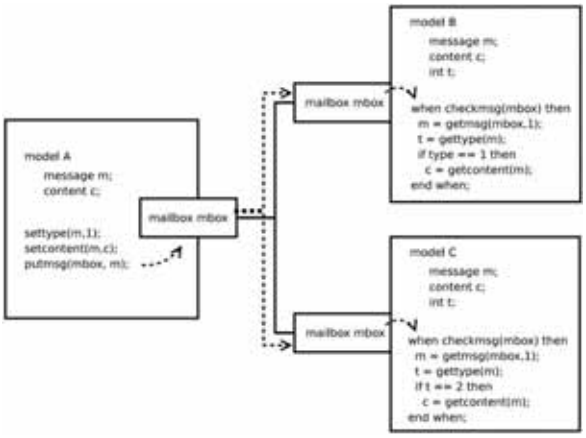

Figure 6. Model communication with messages using connectors.

on the definition of data structures that support the message and mailbox concepts, and the definition of the operations that can be performed with both data structures. Messages and mailboxes have to be defined as new predefined classes that have to be treated in a singular way, allowing objects of type message or mailbox. Due to the current Modelica language specification, the proposed implementation differs from the mechanism described above. The Modelica language will need to be extended in order to support the messages mechanism.

4.1 Data structures

There are two data structures needed to manage the messages mechanism. These are the definition of the message itself and the structure to support the mailbox that receives the defined messages.

The message structure contains two components: the type and the content. The type of a message can be represented with an integer value. It is used to separate the messages of the system in different classes. The content represents the information transported by the message. The content of a message is defined by the user and has to be independent from the message management mechanism. Thus, any mailbox can receive messages with any content and of any type. It is a task of the user to distinguish between the types of the messages and their contents. The content of the message is represented by a reference to an external data structure in C defined by the user. The user has to provide this data structure and the functions required to manage it using the reference in Modelica. Because of this definition, a message will be composed by two integer values: the type and the reference to the content.

The second structure required in the messages mechanism is the mailbox. A mailbox is a temporary storage for messages. If a message is sent to a mailbox, it is stored in the mailbox until the receiver reads it. The number of stored messages in a mailbox is not limited, so this structure has to be able to change its dimension depending on the number of stored messages. The implementation of a mailbox is very similar to the currently implemented linked-lists for storing delayed entities during processes.

4.2 Operations

The operations that can be performed with the previously described structures are defined below. Each operation is defined with its parameters and a short description of its behavior.

Mailbox operations

- `newmailbox(mailbox)`. Initializes the mailbox.
- `checkmsg(mailbox)`. Warns about the arrival of a new message. It changes its value from false to true and immediately back to false at each message arrival event.
- `newmsg()`. Detects the arrival of a message to any of the mailboxes declared in the model. This helps to manage the simultaneous arrival of messages in different mailboxes.
- `nummsg(mailbox)`. Returns the number of waiting messages stored in the mailbox.
- `readmsg(mailbox,select)`. Reads a message from the mailbox. The select parameter represents a user-defined function used to select the desired message to be read from the mailbox.
- `getmsg(mailbox,select)`. Fetches a message from the mailbox, deleting it. The select parameter is used in the same way as in the `readmsg` function.
- `putmsg(mailbox,message)`. Sends the message to the mailbox.

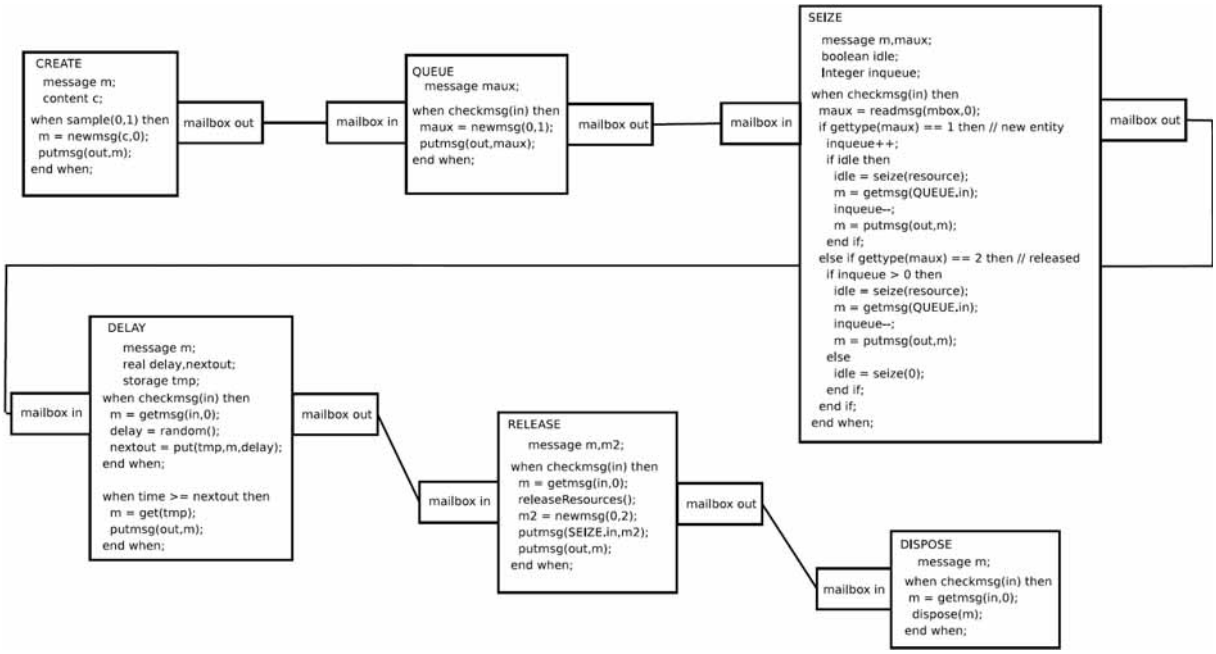

Figure 7. Example of a SIMAN single-queue system modeled using messages.

Message operations

- `newmsg(content,type)`. Creates a new message with the defined type and content.
- `gettype(message)`. Returns the type of the message.
- `settype(message,newtype)`. Updates the type of the message to the value of newtype.
- `getcontent(message)`. Reads the content of the message.
- `setcontent(message,newcontent)`. Inserts the newcontent into the message.

An example of a SIMAN single-queue system, with the Create, Queue, Seize, Delay, Release and Dispose blocks, modeled using the described messages mechanism is shown in Figure 7. Each block of the figure contains the pseudo-code that implements the basic actions for the entity management and communication. The select function, in the `readmsg` and `getmsg` functions, has been simplified and only represents the type of message to be read or extracted.

5 Conclusions

It has been observed that process-oriented modeling of discrete-event systems in Modelica is a difficult task. Several Modelica libraries have been developed to provide more discrete-event system modeling functionalities to Modelica, especially for modeling systems using the process-oriented approach. The implementation of these libraries present some problems and restrictions, and the solutions proposed and implemented are complex, hard to understand and difficult to maintain. In order to facilitate the development of discrete-event system models in Modelica, the message communication mechanism has been introduced and described. A possible implementation of this mechanism in Modelica has also been proposed.

Acknowledgments

This work has been supported by the Spanish CICYT, under DPI 2007-61068 grant, and by the IV PRICIT (Plan Regional de Ciencia y Tecnología de la Comunidad de Madrid 2005-2008), under S-0505/DPI/0391 grant.

References

[1] Dynasym AB. *Dymola Dynamic Modeling Laboratory User's Manual*. http://www.dymola. com/ , 2006.

[2] T. Beltrame. *Design and Development of a Dymola/ Modelica Library for Discrete Event-Oriented Systems Using DEVS Methodology*. Master's thesis, ETH Zürich, March 2006.

[3] T. Beltrame, F.E. Cellier. *Quantised State System Simulation in Dymola/Modelica using the DEVS Formalism*. In Proceedings of the 5th International Modelica Conference, pages 73–82, 2006.

[4] F.E. Cellier, E. Kofman. *Continuous System Simulation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] M.C. D'Abreu, G.A. Wainer. *M/CD++: Modeling Continuous Systems Using Modelica and DEVS*. In Proc. 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pages 229–236, 2005.

[6] http://www.euclides.dia.uned.es/.

[7] J.A. Ferreira and J.P. Estima de Oliveira. *Modelling Hybrid Systems using Statecharts and Modelica*. In Proc. 7th IEEE International Conference on Emerging Technologies and Factory Automation, pages 1063–1069, 1999.

[8] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2003.

[9] H. Lundvall, P. Fritzson. *Modelling concurrent activities and resource sharing in Modelica*. In Proceedings of the SIMS 2003 - 44th Conference on Simulation and Modeling, 2003.

[10] W.D. Kelton, R.P. Sadowski, D.T. Sturrock. *Simulation with Arena (4th ed.)*. McGraw-Hill, Inc., New York, NY, USA, 2007.

[11] J. Kriger. Trabajo práctico 1: Antiguo reloj des pertador. http://www.sce.carleton.ca/faculty /wainer/wbgraf/samplesmain_1.htm.

[12] A.M. Law. *Simulation Modelling and Analysis (4th ed.)*. McGraw-Hill, 1221 Avenue of the Americas, New York, NY, 2007.

[13] P. L'Ecuyer. *Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators*. Oper. Res., 47(1):159–164, 1999.

[14] S.E. Mattsson, M. Otter, H. Elmqvist. *Modelica Hybrid Modeling and Efficient Simulation*. In Proc. 38th IEEE Conference on Decision and Control, pages 3502–3507, 1999.

[15] P.J. Mosterman, M. Otter, H. Elmqvist. *Modelling Petri Nets as Local Constraint Equations for Hybrid Systems using Modelica*. In Proceedings of the Summer Computer Simulation Conference, pages 314–319, 1998.

[16] M. Otter, H. Elmqvist, S.E. Mattsson. *Hybrid Modeling in Modelica Based on the Synchronous Data Flow Principle*. In CACSD'99, pages 151–157, 1999.

[17] M. Otter, K.-E. Årzén, I. Dressler. *StateGraph - A Modelica Library for Hierarchical State Machines*. In Proc. 4th International Modelica Conference, pages 569–578, 2005.

[18] C.D. Pegden, R.P. Sadowski, R.E. Shannon. *Introduction to Simulation Using SIMAN*. McGraw-Hill, Inc., New York, NY, USA, 1995.

[19] M.A. Pereira Remelhe. *Combining Discrete Event Models and Modelica - General Thoughts and a Special Modeling Environment*. In Proc. 2nd International Modelica Conference, pages 203–207, 2002.

[20] V. Sanz, A. Urquia, S. Dormido. *ARENALib: A Modelica Library for Discrete-Event System Simulation*. In Proc. 5th International Modelica Conference, pages 539–548, 2006.

[21] V. Sanz, A. Urquia, S. Dormido. *DEVS Specification and Implementation of SIMAN Blocks Using Modelica Language*. In Proc. Winter Simulation Conference 2007, pages 2374–2374, 2007.

[22] G. Wainer. *CD++: A Toolkit to Develop DEVS Models*. Softw. Pract. Exper., 32(13):1261–1306, 2002.

[23] B.P. Zeigler, Tag Gon Kim, H. Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2000.

[24] B.P. Zeigler, H.S. Sarjoughian. *Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models*. http://www.acims.arizona.edu/PUBLICATIONS/.93

Corresponding author: Victorino Sanz
Dpto. Informática y Automática,
ETSII Informática, UNED
Juan del Rosal 16, 28040 Madrid, Spain
vsanz@dia.uned.es

Accepted EOOLT 2008, June 2008
Received: July 30, 2008
Revised: August 15, 2008
Accepted: August 20, 2008

Multi-Aspect Modeling in Equation-Based Languages

Dirk Zimmer, Inst. of Computational Science, ETH Zürich, Switzerland, *dzimmer@inf.ethz.ch*

Current equation-based modeling languages are often confronted with tasks that partly diverge from the original intended application area. This results out of an increasing diversity of modeling aspects. This paper briefly describes the needs and the current handling of multi-aspect modeling in different modeling languages with a strong emphasis on Modelica. Furthermore a small number of language constructs is suggested that enable a better integration of multiple aspects into the main-language. An exemplary implementation of these improvements is provided within the framework of Sol, a derivative language of Modelica.

Motivation

Contemporary equation-based modeling languages are mostly embedded in graphical modeling environments and simulators that feature various types of datarepresentation. Let that be for instance a 3D-visualization or a sound module. Consequently the corresponding models are accompanied by a lot of information that describes abundantly more than the actual physical model. This information belongs to other aspects, such as the modeling of the iconographic representation in the schematic editor or the preference of certain numerical simulation techniques. Hence, a contemporary modeler has to cope with many multiple aspects.

In many modeling languages such kind of information is stored outside the actual modeling files, often in proprietary form that is not part of any standard. But in Modelica [6], one of the most important and powerful EOO-languages, the situation has developed in a different way. Although the language has been designed primarily on the basis of equations, the model-files may also contain information that is not directly related to the algebraic part. Within the framework of Modelica, the most important aspects could be categorized as follows:

- *Physical modeling:* The modeling of the physical processes that are based on differential-algebraic equations (DAEs). This modeling-aspect is also denoted as the primary aspect.
- *System hints:* The supply of hints or information for the simulation-system. This concerns for example hints for the selection of state-variables or start values for the initialization problem.
- *3D Visualization:* Description of corresponding 3Dentities that enable a visualization of the models
- *GUI-Representation:* Description of an iconographic representation for the graphical user-interface (GUI) of the modeling environment.
- *Documentation:* Additional documentation that addresses to potential users or developers.

We will use this classification for further analysis, since it covers most of the typical applications fairly well. Nevertheless, this classification of modeling aspects is of course arbitrary, like any other would be.

Let us analyze the distribution of these aspects with respect to the amount of code that is needed for them. Figure 1 presents the corresponding pie-charts of three exemplary models of the Modelica standard library. These are the “FixedTranslation” component for the MultiBodylibrary, the PMOS model of the electrical package and the “Pump and Valve” model in the Thermal library. The first two of them represent single components; the latter one is a closed example system.

In the first step of data-retrieval, all unnecessary formatting has been removed from the textual model-files. For each of these models, the remaining content has then been manually categorized according to the classification presented above. The ratio of each aspect is determined by counting the number of characters that have been used to model the corresponding aspect.

The results reveal that the weight of the primary aspect cannot be stated to be generally predominant. The distribution varies drastically from model to model. It varies from only 14% to 53% for these examples.

Yet one shall be careful by doing an interpretation of the pie-charts in figure 1. The weight of an aspect just expresses the amount of modeling code with respect to the complete model. This does not necessarily

correlate with the invested effort of the modeler and even less it does correlate with the overall importance of an aspect. It needs to be considered that code for the GUIrepresentation is mostly computer-generated code that naturally tends to be lengthy. On the other hand side, the code that belongs to the primary aspect of equation-based modeling is often surprisingly short. This is due to the fact that this represents the primary strength of Modelica. The language is optimized to those concerns and enables convenient and precise formulations. Unfortunately, this can hardly be said about the other aspects in our classification.

The discussion about the Modelica and other EOOlanguage is often constrained to its primary aspect of physical modeling. But in typical models of the Modelica standard-library this primary aspect often covers less than 25% of the complete modeling code. Any meaningful interpretation of figure 1 reveals that the disregard on other modeling aspects is most likely inappropriate especially when we are concerned with language design. For any modeling language that owns the ambition to offer a comprehensive modeling-tool, the ability to cope with multiple aspects has become a definite prerequisite.

It is the aim of this paper to improve modeling languages with respect to these concerns. To this end, we will suggest certain language constructs that we have implemented in our own modeling language: Sol. The application of these constructs will be demonstrated by a small set of examples. But first of all, let us take a look at the current language constructs in Modelica and other modeling languages.

1 Current handling of multiple aspects

1.1 Situation in VHDL-AMS, Spice, gPROMS, Chi

The need for multiple aspects originates primarily from industrial applications. Hence this topic is often not concerned for languages that have a strong academic appeal. One example for such a language is Chi [3]. For the sake of simplicity and clarity, this language is very formal and maintains its focus on the primary modeling aspect.

In contrast, languages like SPICE3 [9] or VHDL-AMS [1,10] and Verilog-AMS[12] are widely used in industry. Unlike Modelica, these languages do typically not integrate graphical information into their models. The associated information that describes the schematic diagram and the model icons is often separately stored, often in a proprietary format. For instance, the commercial product Simplorer [11] generates its own proprietary files for the model-icons. The corresponding VHDL-code does not relate to these files.

However, different solutions are possible: both AMSlanguages contain a syntax-definition for attributes. These can be used to store arbitrary information that relate to certain model-items. Since there is only a small-number of predefined attributes (as unit descriptors, for instance), most of the attributes will have to be specified by the corresponding processing tools.

Furthermore these two languages and SPICE3 own an extensive set of predefined keywords. This way it is possible to define output variables or to configure simulation parameters. The situation is similar in ABACUSS II [5], which is the predecessor to gPROMS [2]. This language offers a set of predefined sections that address certain aspects of typical simulation run like initialization or output.

1.2 Multiple aspects in Modelica

The Modelica language definition contains also a number of keywords that enable the modeler to describe certain aspects of his model. For instance, the attributes `stateSelect` or `fixed` represent system-hints for the simulator. In contrast to other modeling languages, Modelica introduced the concept of annotations. These items are placed within the definitions of models or the declarations of members and contain content that directly relates on them. Annotations are widely used within the framework of Modelica. The example below presents an annotation that describes the position, size and orientation of the capacitor icon in a graphic diagram window.

```
1 Capacitor C1(C=c1) "Main capacitor"
2    annotation (extent  =[50, -30; 70, -10],
3                rotation=270);
```

Listing 1. Use of an annotation in Modelica

Since annotations are placed alongside the main modeling code, they inflate the textual description and tend to spoil the overall clarity and beauty. A lot of annotations contain also computer-generated code that hardly will be interesting for a human reader. Thus, typical Modelica editors mostly hide annotations and make them only visible at specific demand of the user. However, this selection of code-visibility comes with a price. First it reduces the convenience of textual editing, since cut, copy and paste opera-

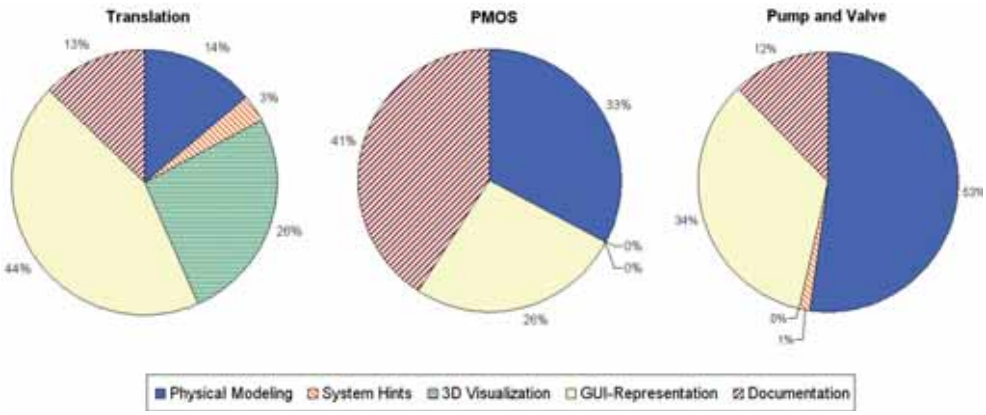

Figure 1. Code distribution of aspects in Modelica models.

tions may involve hidden annotations. Second, the selection of visibility happens on a syntactical level not on a semantic level.

Storing data for GUI-representation or other specific hints and information has been initially a minor topic in the design process of Modelica. Still, there was a compelling need for it. To meet these urgent requirements, the Modelica community decided to introduce the concept of annotations into the modeling language. Already the first language definition of Modelica contained the concept of annotations and also presented some applications for GUI-representation and documentation. The corresponding annotations have been used as a quasi-standard despite the fact that they only have been weakly documented. Annotations served also as an official back-door entrance to non-official, proprietary functionalities. Since it happens frequently in software engineering that certain things just grow unexpectedly, many further annotations have been introduced meanwhile. Nowadays, annotations contain a lot of crucial content that revealed to be almost indispensable for the generation of effective portable code. Therefore it is no surprise that just recently a large set of annotations had to be officially included in version 3 of the Modelica language definition [8]. This way, what started out as a small, local and semi-proprietary solution, became now a large part in the official Modelica standard.

To store the information that belongs to certain aspects, different approaches are used in Modelica and often more than one language-tool is involved. The following list provides a brief overview on the current mixture of data representation:

- The physics of a model is described by DAEs and is naturally placed in the main Modelica model.
- Hints or information for the simulation-system are mostly also part of the main Modelica language but some of them have to be included in special annotations.
- Information that is used by the GUI is mostly included in annotations. But the GUI uses also uses information from textual descriptions that are part of the main-language.
- The description of 3D-visualization is done by dummy-models within main-Modelica code.
- Documentation may be extracted from the textual descriptions that accompany declarations and definitions, but further documentation shall be provided by integrating HTML-code as a text-string into a special annotation. Other annotations store information about the author and the library version.

1.3 Downfalls of the current situation

Obviously, this fuzzy mixture of writings and language constructs reveals the lack of a clear, conceptual approach. As nice as the idea of annotations appears in the first moment, it also incorporates a number of problematic insufficiencies.

The major drawback is that only pre-thought functionalities are applicable. The modeler has no means to define annotation by its own or to adapt given constructs to his personal demands. Furthermore, syntax and semantics of each annotation needs to be defined in the language definition. Since there is always a demand for new functionalities, the number of annotations will continue to increase. This leads to a foreseeable inflation of the Modelica language definition.

1.4 Lack of expressiveness

These downfalls originate from a lack of expressiveness in the original Modelica language. Whenever one is concerned with language design [7], it is important to repetitively ask some fundamental questions. How can it be that a language so powerful to state highly complicated DAE-systems is unable to describe a rectangle belonging to an iconographic representation? Why do we need annotations at all?

These questions are clearly justified and point to the fact that the development scope of the Modelica language might have been too narrowly focused on the equation based part. Therefore, extension that would have been of great help in other domains, have been left out:

- There is no suitable language construct that enables the declaration of an interface to an environment that corresponds to a certain aspect.
- Instances of objects cannot be declared anonymously within a model.
- The language provides no tool for the user that enables him or her to group statements into semantic entities.
- The language offers no means to refer on other (named) objects, neither statically nor dynamically.

By removing these four lacks, we will demonstrate that the use of annotations can be completely avoided and that the declarative modeling of multiple aspects can be handled in a conceptually clear and concise manner. The following section will discuss this in more detail and provide corresponding examples.

2 Multi-aspect modeling in Sol

Sol is a language primarily conceived for research purposes. It owns a relatively simple grammar (see appendix) that is similar to Modelica. Its major aim is to enable the future handling of variable-structure systems. To this end, a number of fundamental concepts had to be revised and new tools had to be introduced into the language. The methods that finally have become available suit also a better modeling of multiple aspects. These methods and their application shall now be presented.

2.1 Starting from an example

In prior publications on Sol [13,14] the "Machine" model has been introduced as standard example. It contains a simple structural change and consists of an engine that drives a flywheel. In the middle there is a simple gear box. Two versions of an engine are available: The first model `Engine1` applies a constant torque. In the second model `Engine2`, the torque is dependent on the positional state, roughly emulating a piston-engine. Our intention is to use the latter, more detailed model at the machine's start and to switch to the simpler, former model as soon as the wheel's inertia starts to flatten out the fluctuation of the torque. This exchange of the engine model represents a simple structural change on run-time.

```
1  model Machine
2  implementation:
3     static Mechanics.FlyWheel F{inertia<<1};
4     static Mechanics.Gear G{ratio<<1.8};
5     dynamic Mechanics.Engine2 E {meanT<<10};
6
7     connection c1(a << G.f2, b << F.f);
8     connection c2(a << E.f, b << G.f1);
9     when F.w > 40 then
10      E <- Mechanics.Engine1{meanT << 10};
11    end;
12 end Machine;
```

Listing 2. Simple machine model in Sol.

The first three lines of the implementation declare the three components of the machine: fly-wheel, gear-box and the engine. The code for the corresponding connections immediately follows. The third component that represents the engine is declared dynamically. This means that the binding of the corresponding identifier to its instance is not fixed and a new instance can be assigned at an event. This is exactly what happens in the following declaration of the when-clause. A new engine of compatible type is declared and transmitted to the identifier E. The old engine-model is thereby implicitly removed and the corresponding equations are automatically updated.

This model contains the physics part only. We now want to add other aspects to the model. We would like to add a small documentation and to specify the simulation parameters. Furthermore we want to add information about model's graphical representation in a potential, graphical user-interface. The following subsections will present the necessary means and their step by step application.

2.2 Environment packages and models

Many modeling aspects refer to an external environment that is supposed to process the exposed information. This environment may be the GUI of the modeling environment or a simulator program. Therefore it needs to be specified how a model can address a

potential environment. To this end, Sol features environment packages and models that enable to define an appropriate interface. Let's take a look at an example:

```
1  environment package Documentation
2     model Author
3       interface:
4          parameter string name;
5     end Author;
6     model Version
7       interface:
8          parameter string v;
9     end Version;
10    model ExternalDoc
11      interface:
12         parameter string fname;
13    end ExternalDoc;
14 end Documentation
```

Listing 3. Environment package.

This example consists in a package that contains models which can be used to store relevant information for the documentation of arbitrary models. The keyword `environment` does specify that the models of the corresponding package address the environment and are therefore not self-contained. They merely offer an interface instead. The actual implementation and semantics of the package remains to be specified by the environment itself.

It is important to see that stipulating the semantics would be a misleading and even futile approach. Different environments will inevitable have to feature different interpretations of the data. For instance, a pure simulator will complete ignore the "Documention" models whereas a modeling editor may choose to generate an HTML-code out of it. Nevertheless it is very meaningful to specify a uniform interface within the language. This provides the modeler with an overview of the available functionalities. Furthermore the modeler may choose to customize the interface for its personal demands using the available object-oriented means of the Sol-language.

2.3 Anonymous declaration

The language Sol enables the modeler to anonymously declare models anywhere in the implementation. The parameters can be accessed by curly brackets whereas certain variable members of the model's interface are accessible by round brackets. This way, the modeler can address its environment in a convenient way just by declaring anonymous models of the corresponding package. An application of this methodology is presented below in listing 4 for the Machine model.

Anonymous declarations are an important element of Sol, since they enable the modeler to create new instances on the fly, for example at the execution of an event. This is very helpful for variable-structure systems. However, within the context of multi-aspect modeling, anonymous declarations serve primarily convenience. It is of course possible to assign names to each of the documentation items. They can be declared with an identifier like any other model, but this is mostly superfluous and would lead to bulky formulations.

```
1  model Machine
2    implementation:
3       […]
4       when F.w > 40 then
5          E <- Mechanics.Engine1{meanT << 10 };
6       end;
7       Documentation.Author{name<<"DirkZimmer"};
8       Documentation.Version{v << "1.0");
9       Documentation.ExternalDoc
                     {fname<<"MachineDoc.html"};
10 end Machine;
```

Listing 4. Use of anonymous declarations.

2.4 Model sections

Sol has been extended by the option for the modeler to define sections using an arbitrary package name. Sections incorporate three advantages: One, code can be structured into semantic entities. Two, sections add convenience, since the sub-models of the corresponding package can now be directly accessed. Three, section enable an intuitive control of visibility. A modern text editor may now hide uninteresting sections. The user may then be enabled to toggle the visibility according to its current interests. This way, the visibility is controlled by semantic criteria and not by syntactical or technical terms.

```
1  model Machine
2    implementation:
3       […]
4       when F.w > 40 then
5          E <- Mechanics.Engine1{meanT << 10 };
6       end;
7       section Documentation:
8          Author{name << "Dirk Zimmer"};
9          Version{v << "1.0"};
10         ExternalDoc{fname<<"MachineDoc.html"};
11      end;
12      section Simulator:
13         IntegrationTime{t << 10.0};
14         IntegrationMethod{method<<"euler",
15         step << "fixed", value << 0.01};
16      end;
17 end Machine;
```

Listing 5. Sections

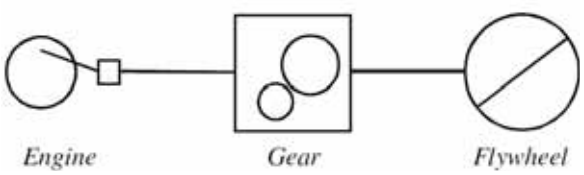

Figure 2. Diagram representation

The documentation part of the machine model has now been wrapped within a section. A second section addresses another environment called "Simulator" and shows an exemplary specification of some simulation parameters. Both sections could be hidden by an editor if the user has no interest in their content.

2.5 Referencing of model instances

The provided methods so far, are fully sufficient for simple application cases. The proper implementation of a GUI-representation is yet a more complex task that demands a more elaborate solution. In the classic GUI-framework for object-oriented modeling, each model owns an icon and has a diagram window that depicts its composition from sub-models. Figure 2 displays the aspired diagram of the exemplary machine-model that contains the icons of its three sub-models. The connections are represented by single lines. The following paragraphs outline one possible solution in Sol.

The problem is that many models will own GUI information but only the information of certain model instances shall be acquired. This originates in the need for language constructs that enable hierarchical or even mutual referencing between model-instances. Sol meets these requirements by giving model-instances a first-class status [4]. This means that model-instances cannot only be declared anonymously but also these instances can be transmitted to other members or even to parameters.

This capability had already been applied in listing 2 to model the structural change of the engine. The statement

```
E <- Mechanics.Engine1(meanT << 10)
```

declares anonymously an instance of the model "Engine1" and then transmits this instance to the dynamic member E. Hence the binding of the identifier to its instance gets re-determined which causes a structural change.

A similar pattern will occur in our solution for the GUI-design. Let us take a look at the corresponding environment-package.

- environment package Graphics
  - model Line
  - model Rectangle
  - model Ellipse
  - model Canvas
    - model Line
    - model Rectangle
    - model Ellipse
  - model GraphicModel

Figure 3. Structure of the Graphics package.

Figure 3 gives a structural overview of the environment package `Graphics`. This package provides rudimentary tools for the design of model-icons and diagrams. These are represented by models for rectangles, ellipses and lines. The package contains also a `Canvas` model that enables drawings on a local canvas. Furthermore the package contains a partial model `GraphicModel` that serves as template for all models that support a graphical GUI-representation. It defines two sub-models: one for the icon-representation and one for the diagram representation. Models that own a graphical representation are then supposed to inherit this template model. Please note that the icon has a canvas model as parameter.

```
1 model GraphicModel
2 interface:
3    model Icon
4    interface:
5       parameter Canvas c;
6    end Icon;
7    model Diagram
8    end Diagram;
9 end GraphicModel;
```

Listing 6. A template for graphical models.

A graphical modeling environment may now elect to instantiate one of these sub-models. This will cause further instantiations of models belonging to the "Graphics"-package that provide the graphical environment with the necessary information. Below we present an exemplary icon model for our engine that corresponds to the icon in Figure 2.

```
10 model Engine2 extends Interfaces.OneFlange;
11                          // that extends GraphicalModel
12 interface:
13   parameter Real meanT;
14   redefine model Icon
15   implementation:
16      c.Ellipse(sx<<0.0, sy<<0.2,
                  dx<<0.6, dy<<0.8);
17      c.Rectangle(sx<<0.9, sy<<0.45,
                    dx<<1.0, dy<<0.55);
18      c.Line(sx<<0.3, sy<<0.3,
               dx<<0.9, dy<<0.5);
```

```
19     end Icon;
20  implementation:
21     […]
22  end Engine2;
```

Listing 7. An implementation of an icon

The icon of listing 7 "paints" on a local canvas that is specified by the corresponding parameter c. The transmission of this parameter is demonstrated in Listing 8 that represents the whole diagram of figure 2. This model declares the icons of its sub-models and creates a local canvas for each of them by an anonymous declaration. The two connections c1 and c2 also own a Line-model for their graphical representation.

```
1  model Machine extends Graphics.GraphicalModel;
2  interface:
3     redefine model Diagram
4     implementation:
5        section Graphics:
6           F.Icon{c<<Canvas{x<<10, y<<10,
                            w<<10, h<<10}};
7           G.Icon{c<<Canvas{x<<30, y<<10,
                            w<<10, h<<10}};
8           E.Icon{c<<Canvas{x<<50, y<<10,
                            w<<10, h<<10}};
9           c1.Line(sx<<20, sy<<15,
                    dx<<30, dy<<15);
10          c2.Line(sx<<40, sy<<15,
                    dx<<50, dy<<15);
11          c.Rectangle(0,0,70,30);
12       end;
13    end Diagram;
14 implementation:
15    […]
16    section Documentation:
17    […]
18    section Simulator:
19    […]
20 end Machine;
```

Listing 8. An implementation of a diagram

The "GraphicalModel" involves another key-concept of Sol. The language enables the modeler to define models also as member-models in the interface section. When instantiated, these models belong to their corresponding instance and are therefore not independent. This means that the Diagram or Icon model always refer to their corresponding super-instance. Consequently, they also have access to all the relevant parameters and can adapt.

Please note that the resulting GUI-models are potentially much more powerful than their annotation-based counterparts in Modelica. All the modeling power of Sol is now also available for the graphical models. For instance, only a minimal effort is needed to make the look of an icon adapt to the values of a model-parameter. No further language construct would be required. A model could even feature "active" icons that display the current system-state and hence enable a partial animation of the system within the diagram-window. Even the structural change of the machine-model could be made visible in the diagram during the simulation. Such extensions (if desired or not) become now feasible and demonstrate the flexibility of this approach.

However, the provided examples are merely a suggestion and represent just one possible and convenient solution within the framework of Sol. There are also many other language constructs that would lead to feasible or even more general solutions. Many of them could easily be integrated into equation-based languages. Some of them are featured in Sol. With respect to Modelica, this is unfortunately not the case yet.

3 Conclusion

Handling complexity in a convenient manner and organizing modeling knowledge in a proper form have always been primary motivations for the design of modeling languages. The introduction of object-oriented mechanism has yield to a remarkable success and drastically simplified the modeling of complex systems. Object-orientation essentially enabled the modeler to break models into different levels of abstraction. Hence, the knowledge could be organized with respect to depth.

However, certain models combine many different aspects that have to be linked together at a top level. Here the knowledge needs to be organized with respect to breadth. For those tasks, current mechanisms in EOOlanguages are underdeveloped.

This paper focuses on *four conceptual language constructs* for EOO-languages that in combination drastically increase the ability to deal with multiple aspects. These are:

1. *Environment-packages* that enable the aspectspecific declaration of interfaces.
2. *Anonymous declarations* of model instances.
3. *Sections* can be used to form semantic entities and control visibility.
4. *Referencing mechanisms* between model-instances. (In Sol, these mechanisms are pro-

vided by giving model-instances a first class status and enabling socalled member-models.)

The proposed constructs have been implemented in our experimental language Sol and their application is demonstrated by a set of corresponding examples. The resulting advantages of this approach are manifold:

- The methods how to address a potential environment are made available within the language. The modeler may browse through the provided functionalities like she or he is used to do it for standard libraries.
- The existing object-oriented mechanisms can be applied on these environment-models. Hence the modeler can customize the interface for its personal demands and is not constrained to a predefined solution.
- Anonymous declarations enable a convenient usage of these models, anywhere in the implementation. The resulting statements are naturally readable and integrate nicely into the primary, equation-based part.
- User-defined sections help to organize the model and offer an excellent way to filter for certain modeling aspects. Uninteresting information may consequently be hidden without hindering the editing of the code. The filtering criteria are not based on syntax anymore, there are based on semantic entities that have been formed by the modelers themselves. Furthermore sections enable a clear separation of computer generated modeling code.
- The embedment into an existing object-oriented framework enables a uniform approach for a wider range of modeling aspects. Furthermore, it increases the interoperability between these aspects.

However, the most important conclusion is that the ability of the language to help and to extend itself by its own means has been drastically improved with respect to other languages like Modelica. Further development is now possible within the language does not require a constant update and growth of the language definition.

4 Appendix

The following listing of rules in extended Backus-Naur form (EBNF) presents an updated version of the core grammar for the Sol modeling language. The rules are ordered in a top-down manner listing the high-level constructs first and breaking them down into simpler ones. Non-terminal symbols start with a capital letter and are written in bold. Terminal symbols are written in small letters. Special terminal operator signs are marked by quotation-marks. Rules may wrap over several lines.

The inserted modifications concern solely the modeling of multiple aspects. With respect to a prior version of the grammar [13], the changes are minor and concern only 3 rules: `ModelSpec`, `Statement` and `Section`.

```
Model       = ModelSpec Id Header
              [Interface] [Implemen] end Id ";"
ModelSpec   = [redefine | partial | environment]
              (model | package | connector | record)

Header      = {Extension} {Define} {Model}
Extension   = extends Designator ";"
Define      = define (Const | Designator) as Id ";"
Interface   = interface ":" {(IDecl | ParDecl) ";"} {Model}
ParDecl     = parameter Decl
IDecl       = [redelcare] LinkSpec [IOSpec] [CSpec] Decl
ConSpec     = potential | flow
IOSpec      = in | out

Implemen    = implementation ":" StmtList
StmtList    = [Statement {";" Statement }]
Statement   = [ Section | Condition | Event |
              Declaration | Relation ]

Section     = section Designator ":" StmtList end [section]
Condition   = if Expression then StmtList ElseCond
ElseCond    = (else Condition)|([else then StmtList]
              end [if])
Event       = when Expression then StmtList ElseEvent
ElseEvent   = (else Event)|([else then StmtList]
              end [when]
Declaration = [redeclare] LinkSpec Decl
LinkSpec    = static | dynamic
Decl        = Designator Id [ParList]

Relation    = Expression Rhs
Rhs         = ("=" | "<<" | "<-") Expression

ParList     = "{" [Designator Rhs {"," Designator Rhs }] "}"
InList      = "(" [Designator Rhs {"," Designator Rhs }] ")"

Expression  = Comparis {(and|or) Comparis }
Comparis    = Term [("<"|"<="|"=="|"<>"|">="|">")Term]
Term        = Product {( "+" | "-" ) Product }
Product     = Power { ("*" | "/") Power }
Power       = SElement {"^" SElement }
SElement    = [ "+" | "-" | not ] Element
Element     = Const | Designator [InList] [ParList]
              | "(" Expression ")"
Designator  = Id {"." Id }
Id          = Letter {Digit | Letter}
Const       = Number | Text | true | false
Number      = ["+"|"-"] Digit { Digit }
              ["." {Digit }] [e ["+"|"-"] Digit { Digit }]
Text        = "\"" {any character} "\""
Letter      = "a" | ... | "z" | "A" | ... | "Z" | "_"
Digit       = "0" | ... | "9"
```

Listing 9. EBNF grammar of Sol

Acknowledgements

I would like to thank Prof. Dr. François E. Cellier for his helpful advice and support. This research project is sponsored by the Swiss National Science Foundation (SNF Project No. 200021-117619/1).

References

[1] P.J. Ashenden, G.D. Peterson, D.A. Teegarden. *The System Designer's Guide to VHDL-AMS* Morgan Kaufmann Publishers. 2002.

[2] P.I. Barton and C.C. Pantelides. *Modeling of Combined Discrete/Continuous Processes*. American Institute of Chemical Engineers Journal. 40, pp.966-979, 1994.

[3] D. A. van Beek, J.E. Rooda. *Languages and Applications in Hybrid Modelling and Simulation: Positioning of Chi*. Control Engineering Practice, 8(1), pp.81-91, 2000

[4] R. Burstall. *Christopher Strachey – Understanding Programming Languages*. Higher-Order and Symbolic Computation 13:52, 2000.

[5] J.A. Clabaugh, *ABACUSS II Syntax Manual, Technical Report*. Massachusetts Institute of Technology. http://yoric.mit.edu/abacuss2/syntax.html. 2001.

[6] P. Fritzson. *Principles of Object-oriented Modeling and Simulation with Modelica 2.1*, John Wiley & Sons, 897p. 2004.

[7] C.A.R. Hoare. *Hints on Programming Language Design and Implementation*. Stanford Artificial Intelligence Memo, Stanford, California, AIM-224, 1973.

[8] *Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling* Language Specification Version 3.0. Available at www.modelica.org .

[9] T.L. Quarles. *Analysis of Performance and Convergence Issues for Circuit Simulation*. PhDDissertation. EECS Department University of California, Berkeley Technical Report No. UCB/ERL M89/42, 1989.

[10] P. Schwarz, C. Clauß, J. Haase, A. Schneider. *VHDL-AMS und Modelica - ein Vergleich zweier Modellierungssprachen*. Symposium Simulationstechnik ASIM2001, Paderborn 85-94, 2001.

[11] Ansoft Corporation: *Simplorer* Avaiable at: http://www.simplorer.com .

[12] *Verilog-AMS Language Reference Manual Version 2.2* Available at http://www.designers-guide.org/VerilogAMS/.

[13] D. Zimmer. *Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems*. Proc. 6th International Modelica Conference, Bielefeld, Germany, Vol.1 47-56, 2008.

[14] D. Zimmer. *Enhancing Modelica towards variable structure systems*. Proc. 1st International Workshop on Equation-Based Object-Oriented Languages and Tools, Berlin, Germany, 61-70, 2007.

Corresponding author: Dirk Zimmer
Institute of Computational Science
ETH Zürich, Switzerland,
dzimmer@inf.ethz.ch

Accepted EOOLT 2008, June 2008
Received: July 30, 2008
Accepted: August 10, 2007

COMSOL MULTIPHYSICS™

Proceedings CD der Konferenz zur Multiphysik-Simulation

ANWENDUNGSBEREICHE:

- Akustik und Fluid-Struktur-Interaktion
- Brennstoffzellen
- Chemietechnologie und Biotechnologie
- COMSOL Multiphysics™ in der Lehre
- Elektromagnetische Wellen
- Geowissenschaften
- Grundlegende Analysen, Optimierung, numerische Methoden
- Halbleiter
- Mikrosystemtechnik
- Statische und quasi-statische Elektromagnetik
- Strömungssimulation
- Strukturmechanik
- Wärmetransport

Bestellen Sie hier Ihre kostenlose Proceedings CD mit Vorträgen, Präsentationen und Beispielmodellen zur Multiphysik-Simulation:

TITEL, NACHNAME

VORNAME

FIRMA / UNIVERSITÄT

ABTEILUNG

ADRESSE

PLZ, ORT

TELEFON, FAX

EMAIL

Fax: +49(0)551 / 99 721- 29

www.comsol.de

www.comsol.de/conference2005/cd/