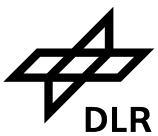
Deutsches Zentrum
für Luft- und Raumfahrt e.V.

Bericht DLR-IB-515-00/2

A Simulation Environment for Hybrid Dynamic Systems

Pieter J. Mosterman

Institute of Robotics and
Mechatronics
Oberpfaffenhofen



A Simulation Environment for Hybrid Dynamic Systems

Pieter J. Mosterman

Institute of Robotics and
Mechatronics
Oberpfaffenhofen

36 Seiten
10 Bilder
6 Tabellen
45 Literaturstellen

A Simulation Environment for Hybrid Dynamic Systems

Deutsches Zentrum
für Luft- und Raumfahrt e.V.

Abstract

This report describes the architecture and implementation of a simulator environment for models of physical systems with mixed continuous/discrete, *hybrid*, behavior in the DSblock 4.0 model specification format.

Contents

List of Figures

List of Tables

1 Overview

This report describes the implementation of a simulator environment for models with mixed continuous/discrete, *hybrid*, behavior. [1]

1.1 The Simulation Supervisor

The simulation of a hybrid dynamic system model specification requires the coordination between a number of tasks. First the user supplied initial conditions are copied into the model. In case of an implicit model formulation, based on these values, consistent values have to be found for the generalized state of the model. In case of index 1 systems, this may require changes in the user supplied values as well.

After consistent initial values are computed, based on these values a mode change may occur. This change is executed by sending the appropriate command to the DSblock model specification. This new mode again requires the computation of consistent initial conditions and this process repeats until no further mode changes occur.

Once a consistent mode and initial conditions are found, the next point in time is determined when data needs to be communicated from the numerical solver to the graphing tool. Also, the next occurrence of a predictable event is retrieved from the DSblock model specification. Because only dependent on the independent variable time, such events are referred to as *time events*. Next, simulation is initiated until the first of the two time points is reached. If an event that was not predicted occurs before the next time event, a so-called *state event*, the numerical solver simulates up till the corresponding point in time and returns with an event flag set (also set when a time event occurs).

When an event (either time or state event) occurs, exactly like on initialization, a new consistent mode and initial values have to be determined. The same iterative procedure is invoked and simulation continues once it has converged (simulation is aborted when the number of iterations exceeds a pre-defined maximum).

When the numerical solver returns the model state at either the predetermined communication time point or at an event time, the complete model state is stored in a file. This allows the computation of variables that are selected by the user to be displayed without re-executing the simulation run.

[1] The research reported here was performed in the context of the focused research program "Continuous Discrete Dynamic Systems" (KONDISK) and sponsored by the *Deutsche Forschungsgemeinschaft* under grant OT174/1-2. This support is gratefully acknowledged.

1.2 The Hybrid Model Specification

The DSblock model specification required by the simulator architecture is automatically generated from a hybrid bond graph by the prototype simulator HYBRSIM, developed in the first phase of the project. To this end, it can export a DSblock model specification in Java and C/C++ in either an explicit or implicit formulation.

A critical issue of this code generation is the treatment of discrete state changes that lead to a combinatorial explosion of modes of operation. Because the DSblock model specification requires an executable system of equations, such a system of equations has to be exported in its sorted and solved form for each mode. This complexity is circumvented by using a local specification of mode changes so only one equation changes its form. To handle the change in computational causality that this would invoke, an implicit formulation is used. Combined with the use of an implicit numerical solver and consistent initial condition computation this has shown to be a viable approach. In spirit the approach is similar to the one used for handling mode changes in systems that do not cause index changes. The possibility to handle variable index systems presents a significant improvement.

1.3 The Simulator Architecture

The overall simulation algorithm coordinates the described tasks, embodied by the following software modules:

- A graphical user interface (GUI) written in Java using Swing library components to enter simulation conditions (end time, output grid, tolerances), modify model properties (start values), and to display model variables from stored data. It also contains the simulator supervisor module to initiate and coordinate communication between each of the components of the entire simulator environment architecture as described in Section 1.1.

- A graphing tool written in Java to store and display simulation results. The data of a simulation run is stored in the form of the model *state* at each point used to communicate data between the numerical solver and the simulator supervisor. The model state is the minimal data required to unequivocally compute all variable values in the model. In case of an implicit model formulation, this includes the state vector, x, its time derivative $\dot{x}$, the input u, the independent variable time, t, the values of discrete boolean and real state variables (i.e., those that are operated on by the *pre* operator.

- A model specification in the DSblock (Dynamic Systems block) 4.0 format, either in Java or in C/C++. This contains the processed model equations (i.e., executable) and further information (e.g., minimum and maximum values of a variable). The internal structure supports hierarchical, object-oriented, system specification, which is close to models specified by the Modelica language for modeling physical systems. From a model specification, it derives an efficient interface to the numerical algorithms for behavior generation. Furthermore, it

includes some event handling features (e.g., automatic generation of indicator functions) and supports the efficient handling of events with an *a priori* known time of occurrence (*time events*).

- A suite of numerical solvers in Java and C to generate model behaviors. This includes
  - explicit solvers with a
    - fixed step size (forward Euler, Runge-Kutta 4^{th} order) and
    - variable step size (Runge-Kutta $4^{th}/5^{th}$ order)

    and root-finding by means of a bisectional search, and
  - an implicit solver with a variable step size and root-finding (DASSL).
- Numerical algorithms to compute consistent initial conditions of implicit model formulations (all implemented in C):
  - A least-mean square solver based on the Levenberg-Marquardt algorithm (MINPACK-LMDIF) to find consistent values of implicit algebraic variables and behavior gradients of implicit derivatives. This only applies to index 0 model formulations.
  - Software to compute consistent projections for model formulations up to index 2, made available by René Lamour at the Humboldt Universität in Berlin. These projections are, however, not yet used to compute consistent initial values.
  - A method based on decomposing the general model formulation into a pseudo Kronecker Canoncial Form (KCF) developed at the DLR Oberpfaffenhofen with the help of Andras Varga. This method applies to index 1 and certain types of index 2 systems and is currently used by the simulation environment.

The software architecture modules are implemented in Java and C/C++. Java is used for modules that require graphical interaction with the user because of its superior qualities in this respect, mainly because of the Swing library of GUI components. C/C++ is used for computationally complex operations (e.g., numerical calculation) and efficient memory allocation. Interaction between the two relies on the Java Native Interface (JNI) specification.

1.4 Evaluation

The present architecture allows:

- continuous simulation of a class of index 2 systems,
- implicit, explicit, and mixed formulations,
- time events and state events,

- run time equation changes that may cause index changes, and
- the use of *a priori* values around a discontinuous change.

At present, the models are automatically generated from a hybrid bond graph formulation by HYBRSIM. This lacks flexibility and, because HYBRSIM does not allow hierarchy, support for modeling of large systems. This restriction is eliminated as soon as the DSblock 4.0 model specification is supported by a Modelica translator (e.g., Dymola) or an object-oriented equation based modeling tool in general.

1.5 Publications from September 1999

Dieter Moormann, Pieter J. Mosterman, and Gert-Jan Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, (3):115–126, 1999.

Pieter J. Mosterman. Implicit Modeling and Simulation of Discontinuities in Physical System Models. In *ADPM*, 2000. Invited paper.

Pieter J. Mosterman and Gautam Biswas. A modeling and simulation methodology for hybrid dynamic physical systems. *SCS Transactions*. to be published.

Pieter J. Mosterman and Gautam Biswas. Building Hybrid Automata of Physical Systems for Real-time Application. In *Conference on Decision and Control*, December 1999. Invited paper.

Pieter J. Mosterman and Gautam Biswas. A comprehensive methodology for building hybrid models of physical systems. *AI Journal*, 2000. in press.

Pieter J. Mosterman and Gautam Biswas. Towards Procedures for Systematically Deriving Hybrid Models of Complex Systems. In Nancy Lynch and Bruce Krogh, editors, *Hybrid Systems: Computation and Control*, pages 324–337, 2000. Lecture Notes in Computer Science.

Pieter J. Mosterman, Peter Neumann, and Carsten Preusche. Modeling Systems With Variable Algebraic Contraints for Explicit Integration Methods. In *ADPM*, 2000.

Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling in Control System Design. In *IEEE International Symposium on Computer-Aided Control Systems Design*, 2000.

Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. Sliding mode model semantics and simulation for hybrid systems. In Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems V*, pages 218–237. Springer-Verlag, 1999. Lecture Notes in Computer Science.

Martin Otter, Manuel A. Pareira Remelhe, Sebastian Engell, and Pieter J. Mosterman. Hybrid models of physical systems and discrete controllers. *at - Automatisierungstechnik*, 2000.

1.6 Special Sessions

To investigate the use of meta-modeling and share knowledge of multi-paradigm modeling gained by designing multiple formalisms in DOME, a special session on Computer Automated Multi-Paradigm Modeling was organized at the IEEE International Symposium on Computer-Aided Control Systems Design, Anchorage, Alaska. The session comprised a part on heterogeneous modeling and one on behavior analysis, especially mixed continuous/discrete behaviors.

1.6.1 Heterogeneous Modeling

Chairs: Hans Vangheluwe and Pieter J. Mosterman

Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling in Control System Design. Institute of Robotics and Mechatronics, *DLR Oberpfaffenhofen*, Germany and School of Computer Science, *McGill University*, Montreal, Canada

Johannes Ernst and Scott Washburn. Zero-latency engineering^{tm} for control design. *Aviatis Corp.*, Campbell, CA

John R. James. Thoughts on Information Operation Detection as a Nonlinear, Mixed-Signal Identification Problem: A Control Systems View Department of Electrical Engineering and Computer Science, *United States Military Academy West Point*, NY

Eric Engstrom and Jonathan Krueger. A meta-modeler's job is never done: Building and evolving domain-specific tools with DOME. Honeywell Technology Center, *Honeywell*, Minneapolis, MN

Gabor Karsai, Greg Nordstrom, Akos Ledeczi, and Janos Sztipanovits. Specifying Graphical Modeling Systems Using Contraint-based Metamodels. Institute for Software Integrated Systems, *Vanderbilt University*, Nashville, TN

Jie Liu and Edward A. Lee. Component-based hierarchical modeling of systems with continuous and discrete dynamics. Department of Electrical Engineering and Computer Science, *University of California*, Berkeley, CA

1.6.2 Behavior Analysis

Chairs: Pieter J. Mosterman and Hans Vangheluwe

Paul I. Barton. Modeling, simulation, and sensitivity analysis of hybrid systems: Mathematical foundations, numerical solutions, and sofware implementations. Department of Chemical Engineering, *Massachusetts Institute of Technology*, Cambridge, MA

Karl Henrik Johansson, John Lygeros, Shankar Sastry, and Jun Zhang. Hybrid automata: A formal paradigm for heterogeneous modeling. Department of Electrical Engineering and Computer Science, *University of California*, Berkeley, CA

Hans Vangheluwe. DEVS as a common denominator for multi-formalism hybrid system modeling. School of Computer Science, *McGill University*, Montreal, Canada

Taeshin Park. Verification of large-scale hybrid systems using implicit model representation. *Mitsubishi Cars Research Innovation Center*, Inc., Cambridge, MA

Simin Nadjm-Tehrani. Formal methods for analysis of heterogeneous models of embedded systems. Department of Computer and Information Science, *Linköping University*, Linköping, Sweden

2 Transition Behavior

Hybrid dynamic systems may contain complex state space transition behavior that approximates detailed continuous transients. Consider the hydraulic cylinder in Fig. 2.1. The valve at the top of the cylinder controls oil flow into and out of the cylinder, and the flow rate is a function of the control pressure p_{in}. The flow of oil determines the position of the piston in the cylinder, and this in turn determines the position of the load, e.g., the elevator control surface of an airplane. To prevent damage to the actuator system, a relief valve on the left side of the cylinder opens when the pressure in the cylinder exceeds a certain value.

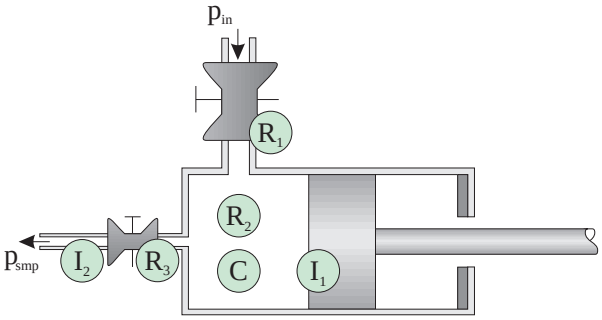
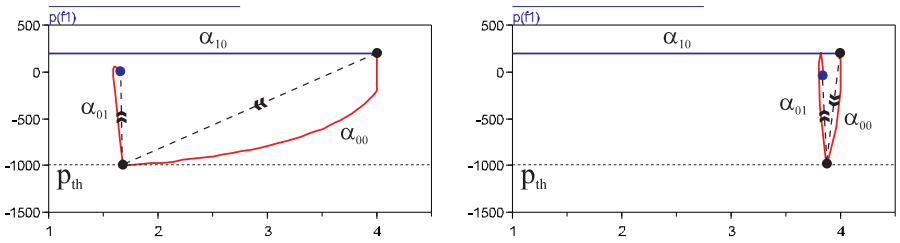


Figure 2.1: Hydraulic cylinder with relief valve.

If the valve behaviors are approximated and simplified to be discrete, the actuator can be modeled as a hybrid automata with four states: α_{00}, both valves closed, α_{01}, relief valve open and control valve closed, α_{10}, control valve open and relief valve closed, and α_{11}, both valves open. The dynamic behavior in each of these modes can be derived from the actuator parameters, that include R_1, the resistance of the open control valve, R_2, the internal dissipation parameter for the oil, R_3, the resistance of the open relief valve, C, the oil elasticity, I_1, the piston inertia, and I_2, the relief valve fluid inertia.

2.1 Pinnacles

Figure 2.2 shows the detailed phase space transition behavior for two values of C in a hybrid model with relatively small and large parameters. ($R_{1,l} = R_{3,l} = 0.01$, $R_{1,h} = R_{3,h} = 1 \cdot 10^7$, $R_2 = 100$, $I_1 = 1$, $I_2 = 0.01$, and $p_{th} = 1000$), i.e., it is of C^0 hybrid type. Velocity f_1 is plotted on the x-axis and pressure p is plotted on the y-axis. The



(a) $C = 1 \cdot 10^{-5}$ (b) $C = 1 \cdot 10^{-6}$

Figure 2.2: Dominant C phase space switching behavior.

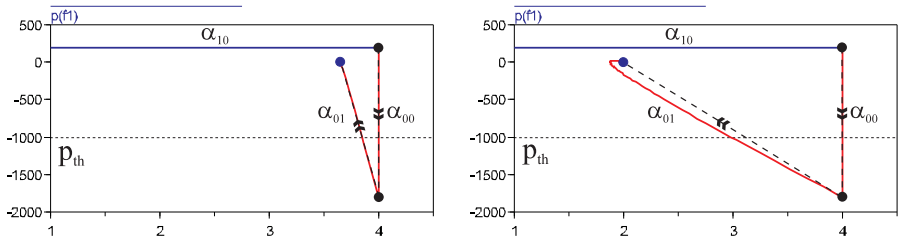
discontinuous approximations are superimposed by dotted lines.[1] When the control valve closes, f_1 has value 4, and the pressure in the cylinder starts to rise quickly This behavior consists of an immediate change in p caused by the term $f_1 R_2$, and a quick continuous change because of the pressure build up.

If the absolute pressure exceeds 1000, the mode switch to α_{01} occurs. In this mode, there is another quick change in f_1, this time governed by the dependency between I_1 and I_2. Because I_1 is several orders of magnitude larger than I_2, only a small change in f_1 occurs. The interaction between the three generalized state variables in the C^0 hybrid model, f_1, f_2, and p_1, causes oscillatory behavior in α_{01}. This is clearly seen in Fig. 2.2(b). The discontinuous jump does not include this behavior, but immediately reaches the final value. The phase space behaviors present examples of two consecutive discontinuous state variable value changes that are of different types. The intermediate value of f_1 is achieved in a so-called *pinnacle* mode [34, 35], and the final value is governed by a subspace projection. Note that the pinnacle is crucial in computing the correct final value of the variable in α_{01}, when continuous behavior resumes.

2.2 Mythical Modes

If $R_2 > 250$, $f_1 R_2$ becomes the dominant factor in the phase space transition behavior, as shown in Fig. 2.3 for $C = 1 \cdot 10^{-5}$ and $R_2 = 500$. The consecutive switch to α_{01} follows immediately after the switch to α_{00}. As a consequence, α_{00} does not affect the value of f_1, therefore, this is a so-called *mythical mode* [13, 38]. Mode α_{00} is not intrinsically a mythical mode because the state variable values when the mode is entered determine whether it is exited immediately. Only in such situations mythical behavior occurs. The projection in α_{01} that follows is shown more clearly in Fig. 2.3(b) for a larger value of I_2. For these parameters, the value $f_1^+ = 2$ ($f_1 = 4$, $f_2 = 0$) is computed based on the decomposition in Section 3, which also shows that larger values of I_2 have a greater effect on the magnitude change of f_1. Again, the fast oscillatory behavior of the subspace projection is abstracted away in the discontinuous approximation.

[1] These approximations are not simulation results.



(a) $I_2 = 0.1$ (b) $I_2 = 1$

Figure 2.3: Dominant R_2 phase space switching behavior.

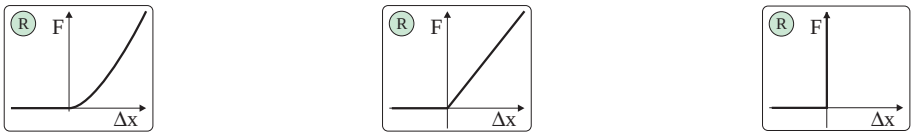
2.3 Sliding Modes

For specific sets of parameter values, the model may generate transition behavior interspersed with infinitely short intervals of continuous behavior. During numerical simulation this results in the simulation time advancing very slowly, and, effectively, simulation comes to a halt. One mechanism to avoid this problem is by not performing event location computations and using a fixed step size numerical integration. However, this reduces accuracy and may preclude models with varying time scales from simulation.

The alternative is the use of a dedicated so-called *sliding mode* algorithm when chattering is detected [33, 36]. This algorithm relies on the computation of equivalent dynamics along the chattering surface and allows efficient and accurate simulation.

2.4 Transition Sequences

In general, hybrid models of physical systems arise when certain abstractions are applied [39]. These can be classified as (i) parameter and (ii) time scale abstractions [18, 25]. This is illustrated by a model of the valve behavior in the hydraulic cylinder in Fig. 2.1. In Fig. 2.4(a), behavior is modeled by a smooth, nonlinear, characteristic. The nonlinearity can be approximated by piecewise linear behavior with a transition point where behavior is continuous but the first time-derivative is not, i.e., behavior is C^0. If the steep gradient is simplified even further, a causal change occurs at the switching point and discontinuous changes may occur.



(a) Nonlinear model. (b) C^0 hybrid model. (c) Hybrid model.

Figure 2.4: Levels of abstraction of a cylinder end-stop.

Detailed analysis of the transition behavior that results from these approximations [26]

shows that they can be characterized in terms of a phase space transition ontology. In other work [34, 35], three principal transition functions were analyzed in phase space: (i) transition to a mythical mode, (ii) transition to a pinnacle, and (iii) transition to a continuous mode which can be (a) into the interior of the mode, (b) onto its boundary, and (c) to a sliding mode. For the hydraulic cylinder, when switching to α_{00}, the two possible scenarios are

1. $p < p_{th}$ in which case a projection of f_1 onto $f_1 = 0$ occurs, and the system remains in α_{00}. This represents a transition to a *continuous* mode.

2. $p \geq p_{th}$ in which case

   - there may be a distinct drop in f_1 before switching to α_{01}. This is a transition to a *pinnacle*, or
   - the switch to α_{01} has occurred before any significant change in f_1 occurs. This represents a transition to a *mythical mode*.

The switch to α_{01} may also include a discontinuous state change because of the subspace projection that immediately follows the pinnacle or mythical mode. Note that such sequences of transitions are difficult to model systematically in a compositional framework [29].

An apparent drawback of hybrid systems that contain pinnacle behavior is that it hampers composability [21, 23]. These issues become apparent when the complex transition behavior is made explicit in the form of hybrid automata [24]. Note, however, that although hybrid automata provide a powerful and formal framework for system analysis, they still allow paradoxal behavior [30].

3 Computing Discontinuous Changes

In case of a model formulation by differential and algebraic equations, a so-called DAE system, only a subspace of the generalized state space contains admissible simulation trajectories [8, 17, 43, 44]. In case of index 1 systems, the initial values have to be in this subspace for numerical solvers (e.g., DASSL [42]) to be able to generate behaviors within this space. Initial conditions that are supplied by the user or generated from system behavior before a configuration change may not be in this subspace. This requires a projection from the given point into the subspace, which satisfies the instantaneous dynamics of the active DAE. Previous work [14] has shown that, in case of physical systems, the instantaneous dynamics result in a physically consistent projection, i.e., conservation laws (e.g., conservation of mass and momentum) are satisfied. Furthermore, DASSL requires a consistent set of values for both X and XD, and, therefore, the initial direction of behavior evolution, XD, needs to be computed as well.

To compute the projected values and initial direction of behavior, the given system can be transformed into the Kronecker canonical form (KCF). In this form, the subspace of admissible behaviors is decoupled from the projection space and the computation of the projection and direction of behavior evolution is trivial. However, because of the numerical difficulty to compute the KCF, a pseudo canonical form is used.

3.1 Computing the Pseudo KCF

First, the matrices of the system

$$E\dot{x} + Ax + Bu = 0 \tag{3.1}$$

are derived from a DSblock model using the standard interface routines. Note that since the DSblock model may be nonlinear, this may include linearization. The system in Eq. (3.1) is then transformed by the decomposition

$$QEZZ^{-1}\dot{x} + QAZZ^{-1}x + QBu = 0 \tag{3.2}$$

with Q and Z orthogonal matrices, to arrive at the form

$$\begin{bmatrix} \tilde{E}_{11} & \tilde{E}_{12} \\ 0 & \tilde{E}_{22} \end{bmatrix} \begin{bmatrix} \dot{\tilde{x}}_1 \\ \dot{\tilde{x}}_2 \end{bmatrix} + \begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ 0 & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix} + \begin{bmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{bmatrix} \begin{bmatrix} u \end{bmatrix} = 0 \tag{3.3}$$

where the top and bottom rows correspond to the finite and infinite eigenvalues, respectively. In case of an index 1 system, the matrix $\tilde{E}_{22} = 0$ [44], and, therefore, $\tilde{x}_2$

can be computed explicitly to be $\tilde{x}_2 = -\tilde{A}_{22}^{-1}\tilde{B}_2 u$. However, substitution in the finite part to compute $\tilde{x}_1$ requires the time derivative of u because of the $\tilde{E}_{12}$ cross coupling. Therefore, a further coordinate transformation $\bar{x} = P^{-1}\tilde{x}$ is performed with

$$P = \begin{bmatrix} I & P_{12} \\ 0 & I \end{bmatrix} \tag{3.4}$$

to arrive at the form

$$\begin{bmatrix} \bar{E}_{11} & 0 \\ 0 & \bar{E}_{22} \end{bmatrix} \begin{bmatrix} \dot{\bar{x}}_1 \\ \dot{\bar{x}}_2 \end{bmatrix} + \begin{bmatrix} \bar{A}_{11} & \bar{A}_{12} \\ 0 & \bar{A}_{22} \end{bmatrix} \begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \begin{bmatrix} u \end{bmatrix} = 0 \tag{3.5}$$

For this transformation, P_{12} is computed from the requirement that $\tilde{E}_{11}P_{12} + \tilde{E}_{12} = 0$. Straightforward computation shows that $\bar{E}_{11} = \tilde{E}_{11}$, $\bar{E}_{22} = \tilde{E}_{22}$, $\bar{A}_{11} = \tilde{A}_{11}$, $\bar{A}_{22} = \tilde{A}_{22}$, and $\bar{A}_{12} = \tilde{A}_{11}P_{12} + \tilde{A}_{12}$.

3.2 Computing X for Index 1 Systems

To compute the initial values, the Laplace transform ($\mathcal{L}(\dot{x}) = sX - x^-$) is applied. Because only the initial conditions in the finite part can be chosen freely, this yields

$$(s\bar{E}_{11} + \bar{A}_{11})\bar{X}_1 + \bar{A}_{12}\bar{X}_2 + \tilde{B}_1 U = \bar{E}_{11}\bar{x}_1^- \tag{3.6}$$

The Laplace tranform of the system in original coordinates is

$$(sQE + QA)X + QBU = QEx^- \tag{3.7}$$

and using the inverse Laplace transform leads to

$$\bar{x}_1 = \bar{E}_{11}^{-1}Q_1 E x^- \tag{3.8}$$

where $\bar{E}_{11}$ ($= \tilde{E}_{11}$) is of full rank and Q_1 contains the rows corresponding to the finite part of the decomposition. For an index 1 system, $\bar{E}_{22} = 0$, and, therefore, $\bar{x}_2 = -A_{22}^{-1}B_2 u$. Combined with Eq. (3.8), the consistent initial values of the system in original coordinates can be computed to be

$$x = ZP\bar{x} = ZP \begin{bmatrix} \bar{E}_{11}^{-1}Q_1 E x^- \\ -A_{22}^{-1}B_2 u \end{bmatrix} \tag{3.9}$$

These computations are applied to a model of the actuator cylinder that is automatically generated from a hybrid bond graph model by HYBRSIM [20, 27]. Results are reported in [17].

3.3 Computing XD for Index 1 Systems

Before DASSL can start simulation, it also needs consistent values for $\dot{x}$. These are computed from the same decomposed system in Eq. (3.3). The bottom row in Eq. (3.3) combined with the transformation in Eq. (3.4) results in $\bar{x}_2 = \bar{A}_{22}^{-1}\bar{B}_2 u$ and from this

$$\dot{\bar{x}}_2 = \bar{A}_{22}^{-1}\bar{B}_2 \dot{u}$$

is computed. From the top row in Eq. (3.3) and the transformation in Eq. (3.4), it follows that

$$\dot{\bar{x}}_1 = -\bar{E}_{11}^{-1}(\bar{A}_{11}\bar{x}_1 - \bar{A}_{12}\bar{A}_{22}^{-1}\bar{B}_2 u + \bar{B}_1 u)$$

Transforming this back results in the solution in original coordinates

$$\dot{x} = ZP\dot{\bar{x}} = ZP \begin{bmatrix} -\bar{E}_{11}^{-1}(\bar{A}_{11}\bar{x}_1 - \bar{A}_{12}\bar{A}_{22}^{-1}\bar{B}_2 u + \bar{B}_1 u \\ -A_{22}^{-1}B_2\dot{u} \end{bmatrix} \tag{3.10}$$

from which $\dot{x}$ can be computed under the assumption that $\dot{u} = 0$.

3.4 Acknowledgement

Thanks to Andras Varga (DLR Oberpfaffenhofen) for his help in developing the calculations and implementing the numerics.

4 Simulator Components

The overall simulator structure consists of the following software modules: (i) the DSblock model specification, (ii) the numerical solvers (e.g., DASSL), (iii) the graphical user interface (GUI), (iv) the graphing facilities, and (v) initial value computations. The modules are implemented as dynamic link libraries (DLLs) and Java classes and are described in this section.

4.1 DSblock

The DSblock (Dynamic Systems block) software architecture embodies a neutral input-output description of a general hybrid differential-algebraic equation system. The internals of a DSblock can be used to describe explicit ordinary differential equations (ODEs)

$$\dot{x} = f(x, u, t)$$

as well as implicit differential-algebraic equations (DAE)

$$0 = f(\dot{x}, x, u, t)$$

where x are the generalized state variables, the dot notation means derivative with respect to time, u is the input, and t is time. Furthermore, it facilitates root finding functionality and contains some event handling mechanisms.

One DSblock model may contain a combination of ODEs and DAEs. This leads to the following general mathematical structure

$$\dot{x}_e = f_e(x_e, x_i, x_a, u, t) \tag{4.1}$$
$$res_i = f_i(\dot{x}_i, \dot{x}_e, x_i, x_e, x_a, u, t) \tag{4.2}$$
$$res_a = f_a(x_i, x_e, x_a, u, t) \tag{4.3}$$
$$y = f_y(x_e, x_i, x_a, u, t) \tag{4.4}$$
$$z = f_z(x_e, x_i, x_a, u, t) \tag{4.5}$$

Here, Eq. (4.1) contains the explicit ODE part, Eq. (4.2) the implicit differential equations and Eq. (4.3) the implicit algebraic equations. Variables x_e are the explicit state variables, x_i the implicit states, x_a the implicit algebraic variables, and z the indicator variables that specify assumptions. The variables res_i and res_a are the residues of the implicit derivative and algebraic variables, respectively. These are computed to be close to 0 by numerical methods.

function	return value
nx()	the number of generalized state variables
nx_e()	the number of explicit state variables
nx_i()	the number of implicit state variables
nw_i()	the number of implicit algebraic variables
nx_z()	the number of indicator functions

Table 4.1: DSblock interface dimension routines.

function	comment
setArg(XD, WI) → XD are the time derivative variables → WI are the algebraic variables with residuals	sets the time-derivative of the generalized implicit state variables and the implicit algebraic variables of a DAE system
setArg(X, U, TIME) → X are the generalized state variables → U are the input variables → TIME is the current time	sets the generalized state and input of a DAE system
setArgODE(X, U, TIME) → X are the generalized state variables → U are the input variables → TIME is the current time	sets the state and input of an ODE system
setArg(X, XD, U, TIME) → X are the generalized state variables → XD are the time-derivatives of the generalized state variables → U are the input variables → TIME is the current time	sets the generalized state, its time-derivatives, the input, and the current time of a DAE system
derivatives(DELTA, xDot_e) ← DELTA are the computed residuals of the implicit variables ← xDot_e are the values of the explicit time-derivatives of state variables	computes the dynamics of a DSblock DAE system
derivativesODE(XD) ← XD are the time-derivatives of the state variables	computes the dynamics of a DSblock ODE system
crossings(Z) ← Z are the values of the indicator functions	evaluates the indicator functions
outputs(Y) ← Y are the output variables	computes the output

Table 4.2: DSblock interface routines.

The DSblock model specification is object-oriented and structured as classes (either in C++ or Java). It consists of two parts: (i) the model part that contains the functional description of the model to be simulated and (ii) the system part that contains the mathematical structure and interface to the numerical solver. The model part is a close mapping of a Modelica model specification [7] and the component classes are organized hierarchically according to the Modelica model structure (see Appendix A).

Access to the DSblock internal structure is provided by a set of interface functions. Each model contains the standard functions listed in Table 4.1 and Table 4.1. The system part of DSblock contains the interface to the numerical operations performed on the model (a.o., the interface to the numerical solver), this includes inquiry functions such as `nx_i()` to obtain the number of implicit state variables.

Upon initialization, the model structure is created as a tree that maps onto the hierarchy of model classes. The leafs of this tree are variables of one of the standard pre-defined types (Real, Boolean, Integer). Each variable contains a set of optional properties such as its unit, maximum, minimum, and start value. Furthermore, when initialized, the model variables are classified based on their functionality in the system (see Table 4.1). The addresses of interface variables are accumulated and stored in contiguous memory to ensure efficient data transfer. To this end, the addresses of all model variables that are explicit states are stored in a vector x_e and their derivatives in

class	properties
- ModelicaClass	name, description, class (class, connector, record, package, model, block, function, crossing, variable, real, boolean), variability (timevarying, discrete, constant, parameter), access (protected, output, input), enable
- Variable	property (known, explicitAlgebraic, implicitAlgebraic, explicitDerivative, implicitDerivative, state, residue, pre, alias, inverseAlias), propertyRef
- Real	start, min, max, quantity, unit, displayUnit, flow
- Boolean	start, evaluate

Table 4.3: DSblock model component class structure (partial) and properties.

variable	ASCII	comment
x_i	x_i	implicit state variables
x_e	x_e	explicit state variables
$\dot{x}_i$	xDot_i	time derivative of implicit state variables
$\dot{x}_e$	xDot_e	time derivative of explicit state variables
w_i	w_i	implicit algebraic variables
res_i, res_a	residue	residuals for algebraic and derivative implicit variables
u	u	input variables
y	y	output variables
z	z	indicator variables

Table 4.4: DSblock internal variables.

a vector $\dot{x}_e$. The addresses of implicit state variables are stored in a vector x_i, their time derivatives in a vector $\dot{x}_i$, and those of their residues in a vector res_i. The addresses of implicit algebraic variables are stored in a vector x_a and those of their residues in a vector res_a. See Table 4.1 for a list of variables and the relation with those in Eq. (4.1) through Eq. (4.5).

4.2 Numerical Solvers

To generate continuous behavior trajectories, numerical simulation is used. At present, four algorithms are available (i) forward Euler, (ii) Runge-Kutta 4^{th} order, (iii) Runge-Kutta-Fehlberg $4^{th}/5^{th}$ order, and (iv) 1^{st} to 5^{th} order backward difference applied by DASSL. The two first ones are implemented in Java. The latter two are translated from FORTRAN code into C by the f2c compiler.

4.2.1 Class Structure

The numerical solvers are implemented according to the class hierarchy in Table 4.2.1. The fixed step solvers are implemented in Java based on a forward Euler or Runge-Kutta 4^{th} order algorithm. They both inherit a root-finding mechanism by means of a bisectional search from their abstract superclass JavaExplicitSolver.

In order to be able to generate code for general ODE and DAE solvers, the Euler, RK4, and RKF4-5 classes need to inherit a general abstract interface. Because multiple inheritance of classes is not possible in Java, this abstract interface is implemented in Java as a so-called *interface*. The three ODE solvers implement the ODESolver interface and DASSL implements the DAESolver interface. This allows the use of Java and C/C++ DSblock modules by the same simulator implementation.

class	comment
NumericalSolver	
- JavaExplicitSolver	contains bisectional root finding
- Euler	forward Euler
- RK4	Runge-Kutta 4^{th} order
- ImplicitNumericalSolver	
- DASSL	native interface to DASSL
- RKF4-5	native interface to Runge-Kutta-Fehlberg $4^{th}/5^{th}$ order

Table 4.5: Numerical solver class structure.

4.2.2 DASSL

DASSL [42] is a numerical solver for DAEs that may be of index 1 [44]. The standard call is of the form

$$\delta = f(\dot{x}, x, t)$$

where δ embodies the residues. Some versions of DASSL include a root-finding mechanism which requires the specification of an additional indicator function

$$0 = g(x, t)$$

The roots of this function are found ordered in time on the simulated interval and their time of occurrence is determined within a specified accuracy.

DASSL is called by the simulator software with input arguments the generalized state vector, x, the values of its derivatives, $\dot{x}$, the start time, t, and the next time instant, $tout$, at which data needs to be communicated. Furthermore, the functions f and g are provided to DASSL to evaluate the model and compute roots of the indicator functions. When the root of at least one of the indicator functions is found, the corresponding time of occurrence is determined, simulation is halted even if $tout$ is not yet reached, and the values of $\dot{x}$, x, and t at this time point are returned to the simulator.

4.3 Graph

The Graph class is used to coordinate and plot simulation results. The output data points are stored in an array for plotting and the state variables at each data point are written to a file. This file allows off-line plotting of variables not initially selected as output. Also, the graphical window has some rudimentary features to zoom in and out.

4.4 Initial Value Computation

Before simulation of implicit systems can start, consistent initial conditions of the state and its gradient have to be computed. To this end, a least-mean square solver based on the Levenberg-Marquardt algorithms (LMDIF) was first implemented. However, this only applies to index 0 model formulations. Therefore, a method was developed based on decomposing the general model formulation into a pseudo Kronecker Canoncial Form (KCF). The details are presented in Section 3 and the implementation of all

operations that are described relies on the LAPACK [2] and BLAS library. This method applies to index 1 and certain types of index 2 systems and is currently used by the simulation environment. Alternatively, software to compute consistent projections for model formulation up till index 2 made available by René Lamour at the Humboldt Universität in Berlin is being investigated.

4.5 The Simulator GUI

When the simulator is started, it displays the main control panel, shown in Fig. 4.1. This panel contains buttons to (i) start a simulation, (ii) change simulation options (see Fig. 4.2), (iii) set initial conditions, and (iv) examine and change model properties (parameter values, output variables, etc.).

Figure 4.1: Main simulator window.

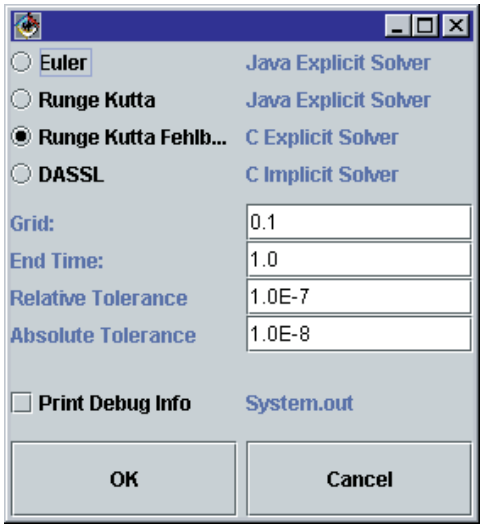


Figure 4.2: Simulator options.

The simulator is written as a Java class called CppSim. The options shown in Fig. 4.2 are directly connected to variables in the CppSim class. The GUI allows entry of simulation end time, the step size used for communicating points to be graphed (i.e., the *grid*), absolute and relative tolerances, and which numerical solver to use. The model properties that are changed through the user interface shown in Fig. 4.3 change (i) the start value of boolean and real model variables and (ii) the accessibility of a variable (e.g., output, protected, and input) and it (iii) allows selection of variables for off-line plotting (i.e., without executing the simulation). Note that changing the start values requires the model DLL to be loaded (the model class to be instantiated, in case of the Java implementation) before sending the change command to the DLL. Otherwise, the newly assigned start values would be overwritten by the ones in the DSblock code.

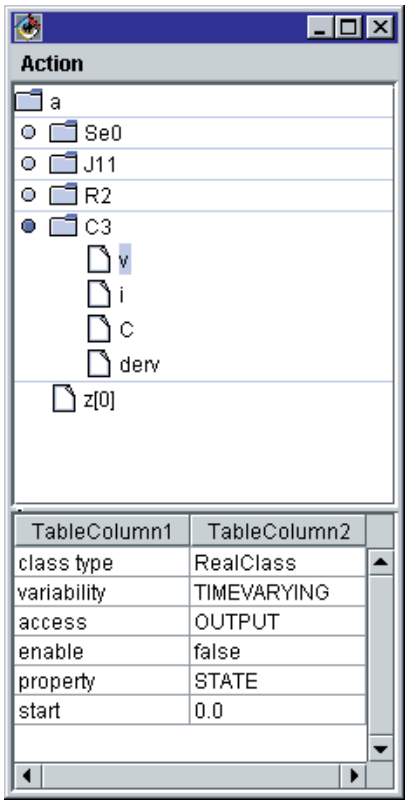


Figure 4.3: The model properties window.

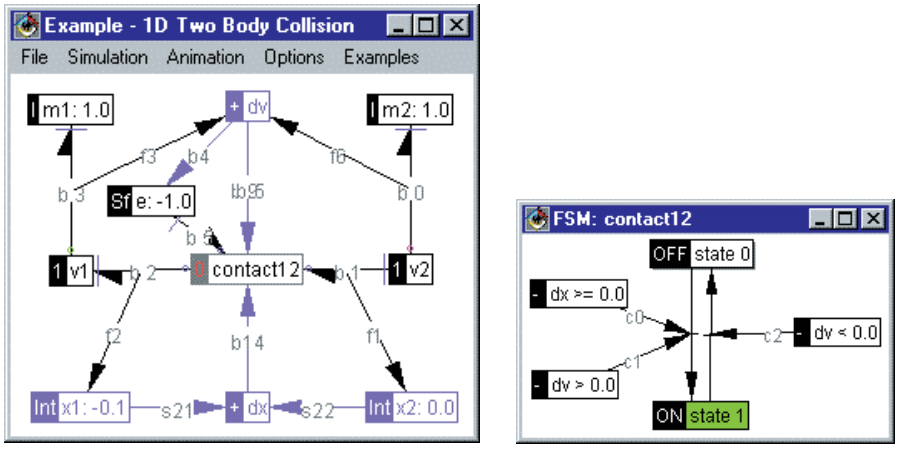
The simulator allows simulation of DSblock models specified in either Java or C/C++. In case a Java specification is used, only the Java based explicit numerical solvers can be used (forward Euler and Runge-Kutta 4^{th} order). If a C/C++ DSblock specification is used, the C/C++ solvers, i.e., DASSL and the Runge-Kutta-Fehlberg $4^{th}/5^{th}$ order, can be used also.

4.6 The Modeling Environment

The Java and C/C++ specifications can be automatically exported from the hybrid bond graph modeling and simulation tool HYBRSIM [20, 27]. In case of C/C++ it allows a choice whether the equations should be exported in explicit or implicit form. In case of the latter, a residue formulation is used that requires an implicit numerical solver such as DASSL.

An example of a hybrid bond graph model in HYBRSIM is given in Fig. 4.4(a). This is the model of a one-dimensional collision between two bodies, m_1 and m_2, modeled as inertias (indicated as being of type I) with velocities v_1 and v_2. Note that velocities are indicated by components of type 1 and velocity differences by components of type 0.[1] Upon collision, a power connection between the bodies is activated represented by *contact12*. This connection computes the difference between velocities and, by means of the e component of type Sf (a fixed velocity), it enforces the desired velocity differ-

[1]The type of each component is shown at the left in its bounding box.



(a) Bond graph. (b) Finite state machine.

Figure 4.4: HYBRSIM model of a one-dimensional collision.

ence after collision, Δv. This is based on the velocity difference before collision, Δv^-, according to Newton's collision rule

$$\Delta v = -\epsilon \Delta v^- \tag{4.6}$$

with ϵ the coefficient of restitution.

The control logic that switches the power connection *contact12* on and off is shown in Fig. 4.4(b). Initially, *contact12* is off. When the bodies touch, $dx \geq 0$, and they are moving towards each other, $dv > 0$, then *contact12* switches on. It switches off when the bodies are moving away from each other, $dv < 0$, which occurs as soon as the original momentum is redistributed according to Eq. (4.6).

Such a graphical specification is automatically transformed into a DSblock specification with a sorted and solved system of equations (see Appendix A).

5 Architecture and Interfaces

Each of the discussed components is connected according to the architecture in Fig. 5.1. The complete software structure consists of a part written in Java, the top part, and a part written in C/C++, the bottom part. Java is used for the graphical user interface (GUI) type parts of the simulator environment because of the powerful Swing library of GUI components. C/C++ is used for the computationally intensive parts such as the numerical algorithms.

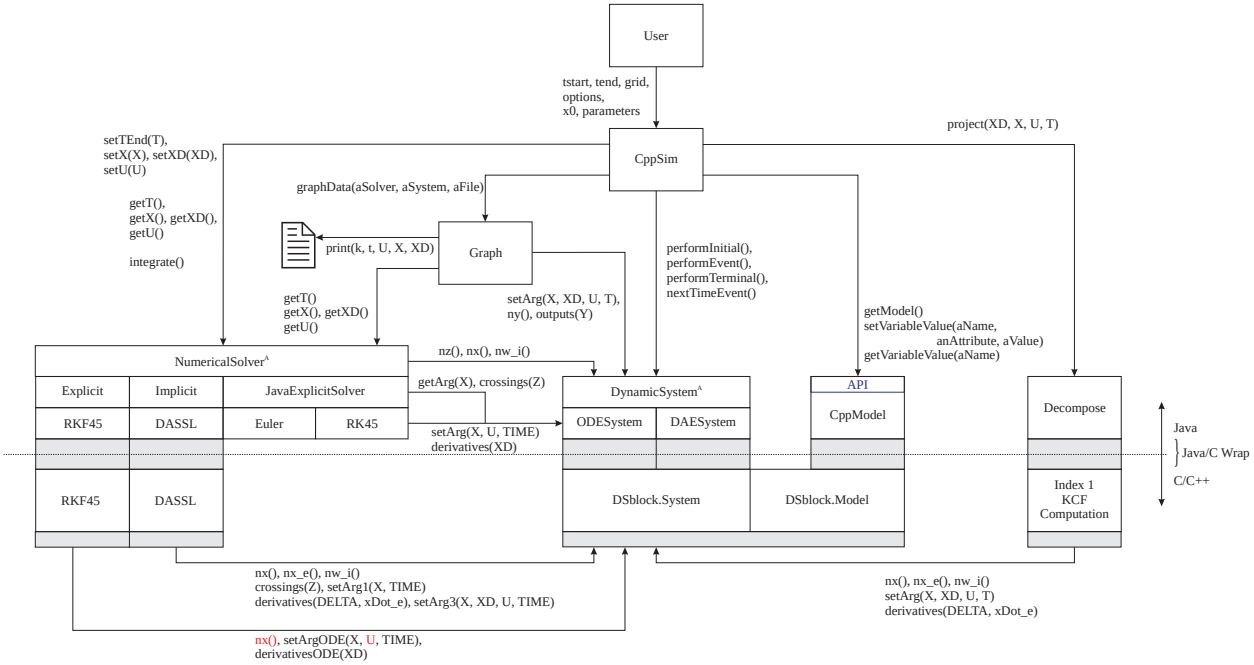


Figure 5.1: Simulator software architecture.

All the way at the top, the user input is transferred to the CppSim class that embodies the simulator interface. The class CppSim initiates and coordinates the flow of control between the simulator components. To this end, it exchanges information with (i) the numerical solvers, (ii) the graphing component, (iii) the DSblock model of the system dynamics, and (iii) a subroutine to compute consistent initial conditions.

The components that are implemented in C/C++ are compiled into separate Dynamic Link Libraries (DLLs) and communicate with Java by means of the Java Native Interface (JNI) specification. For each DLL this requires the definition of a corresponding class in Java that defines the accessible 'native' methods. These native methods correspond to exported subroutines in each DLL.

To interface DASSL with the DSblock code, the variables and subroutines have to be appropriately mapped. This is achieved by a wrapper subroutine around the DASSL internals that relies on standard DSblock calls only. Efficiency is enhanced by facilitating direct calls between DLLs at the C/C++ level. For example, DASSL can directly call the DSblock module to evaluate the dynamics for a set of generalized state values. This requires an additional layer of communication software. To illustrate, consider a call of DASSL to DSblock to compute the system dynamics (derivatives). To this end, the DASSL software requires a function heading of the form `F(T, X, XD, DELTA)` but this function is not explicitly available in the DSblock interface. Therefore, a wrapper routine is required that performs a sequence of tasks available through this interface that have the desired effect, i.e., first the function `setArg(X, XD, U, TIME)` is called to copy the state values and the input into the DSblock model where

$$X = \begin{bmatrix} x_i \\ x_a \\ x_e \end{bmatrix} \tag{5.1}$$

and

$$XD = \begin{bmatrix} \dot{x}_i \\ 0 \\ 0 \end{bmatrix} \tag{5.2}$$

Note the input, U, is constant during numerical integration. Next, the function evaluation can be requested by means of the `derivatives(DELTA, xDot_e)` function. Finally, the computed values have to be copied in the return vector according to

$$DELTA = \begin{bmatrix} res_i \\ XD_e - \dot{x}_e \end{bmatrix} \tag{5.3}$$

to arrive at the structure expected by DASSL. Here, XD_e is the part of XD that corresponds to the explicit state variables, x_e. Also, the residue, res_i, consists of a part for the implicit state variables, x_i, and the implicit algebraic variables, x_a.

Such wrapper routines are written for f, g, and Jac routines.

Finally, an Application Programming Interface (API) is being defined to allow direct access to DSblock model functionality and variables. This API is based on two routines: (i) `setVariableValue(aName, anAttribute, aValue)` to allow changing of a variable's variability (see Table 4.1) and its start value and (ii) `getVariableValue(aName)` to quickly graph behavior of variables from the state variable values (boolean and real) stored during simulation.

6 Simulation Algorithm

This section describes the simulation of a DSblock model that is of index 1, i.e., it requires an implicit solver (in this case, DASSL), and, therefore the DSblock model is specified in C/C++. When a simulation run is initiated, the sequence of actions shown in Algorithm 1 is performed.

Algorithm 1 Hybrid Simulation.

instantiate and initialize a DSblock system, a graph, a numerical solver, $t_{begin}, t_{final}, n_{grid}$
set input
compute consistent initial conditions
while new event **do**
  graph data
  compute consistent initial conditions
end while
t_{event} = next time event
repeat
  $t_{grid} = t_{begin} + n_{grid} * t_{step}$
  $t_{next} = (t_{event} \leq t_{grid})?t_{event} : t_{grid}$
  integrate till t_{next}
  if an event occurred **then**
    graph data
    re-initialize the numerical solver
    while new event **do**
      graph data
      compute consistent initial conditions
    end while
    t_{event} = next time event
    graph data
  else
    graph data
    $n_{grid} = n_{grid} + 1$
  end if
until $t_{next} \geq t_{final}$ **or** an error occurred
graph data

First, the DSblock DLL is loaded into memory. Next, an instance of the dynamic model in the DSblock DLL is created as a tree that represents the hierarchy of model components, followed by a system of the form specified in Section 4.1, which requires traversing the tree and evaluating variable properties. Then the DSblock model is initialized[1] by copying the start values into the variable fields. These start values pertain to boolean variables (e.g., whether a valve is open or closed) and real variables (e.g., the velocity of a mass). The boolean variables determine the active model configuration, which, in turn determines the system dimensions (i.e., the size of the generalized state vector, the number of indicator functions, etc.). The interface functions are used to query these dimensions to instantiate and initialize the numerical solver and graph.

[1]Instantiation occurs only once, when the simulator is started. Otherwise, no different start values can be assigned to its variables.

The numerical solver is initialized, a.o., by retrieving the start values from the DSblock specification. These values may not be in the admissible subspace. Therefore, the next step computes consistent initial conditions. For index 1 systems, this is performed by the computations in the KCF software module, described in Section 3.

To illustrate, consider the hydraulic cylinder with relief valve in Fig. 2.1. Because the start values of the real variables may not be in the admissible subspace of the generalized state space, consistent initial conditions are computed. In case of the hydraulic cylinder, if the control valve is open and the relief valve is closed, there is no flow through the relief valve, $f_2 = 0$, and the velocity of the piston, f_1, can be chosen freely. If the values of the variables in the admissible subspace cause no further configuration changes, continuous simulation starts. At each grid point, the numerical solver communicates the state of the system. When an indicator function crosses 0, an event occurs, the time of occurrence is located, and continuous integration is halted. For example, if at a given piston velocity, v_{th}, the control valve closes, the numerical solver simulates up till this point in time. Note that the time of occurrence of events that are generated by a function of the independent variable (time) can be a priori computed. These so-called *time events* (as opposed to *state events*) [4]) are efficiently treated by adjusting the numerical solver end time, t_{next}, before simulation of the next grid interval is initiated.

Now, the simulator substitutes the returned state variable values into the DSblock model and executes the corresponding event, i.e., the control valve closes. This changes the structure of the model, and, therefore, a projection into the changed admissible subspace may be required. Again, the newly computed variable values may trigger a further event that causes another configuration change. For example, when the control valve is closed, the pressure in the cylinder changes, which may cause the relief valve to open.

This sequence of events and the corresponding model configuration changes and projections are executed till no further events occur and continuous integration is resumed. This process continues and intervals of continuous behavior interspersed with event points are generated till the final time of the requested simulation interval is reached.

7 Conclusions

To evaluate the status of the present simulator architecture, its functionality is classified with respect to the following characteristic hybrid simulation phenomena identified in previous work [15]:

- **state events:** If events occur because of continuous system variables crossing threshold values [3, 4, 41],
  - the event needs to be **detected**, and
  - its time of occurrence needs to be **located**.
- **time events:** If time crosses a threshold value, the time of occurrence is known *a priori*, which can be treated efficiently [4].
- **simulation model:** The system of equations that describes model behavior may change.
  - Blocks of sorted and solved equation may simply appear or disappear (e.g., a vehicle entering and leaving a highway), and, therefore, can be dynamically **added/removed**.
  - In some cases equations can be replaced by others, changing computational causality, and the system of equations may have to be **sorted** again.
  - In other cases, algebraic constraints between state variables may become active and the system of equations needs to be **solved** again to reduce the index of the system [40] (e.g., the rod making contact with the floor).
- **reinitialization:** There may be a discontinuous change in state variable values.
  - This change may be **explicitly** specified by the user by a new initial state equation (e.g., Eq. (4.6) of the colliding bodies).
  - The system of equations may have to be **integrated** to derive physically consistent initial values for a new mode. This ensures that physical conservation principles hold [14].[1]
- **event iteration:** When an event occurs, new system variable values may immediately trigger a further event. Two types of event iteration exist [19],
  - the state vector is **invariant** across the entire iteration (e.g., dominant R_2 effect in Fig. 2.3), and

[1] There may be instantaneous dissipation of energy.

		HYBRSIM	NuSim
state events	detection	√	√
	location	√	√
time events			√
simulation model	add/remove	√	√
	re-sort	√	√
	re-solve	√	√
reinitialization	explicit	√	√
	integration	√	√
event iteration	invariant	√	
	update	√	√
Dirac pulses		√	
chattering			

Table 7.1: Present status for index 1 systems.

  - the state vector is **updated** after each iteration step (e.g., the dominant C effect in Fig. 2.2).

- **Dirac pulses:** Discontinuous changes in continuous variables may cause Dirac pulses to occur. If their magnitudes are numerically approximated, comparison may be affected by non-Dirac type variables (e.g., the sliding condition for the falling rod). To ensure numerically precise treatment, Dirac pulse values should be distinguished from non-Dirac pulse values and evaluation of Dirac pulses can be based on their areas [13].

- **chattering:** If the system chatters between modes, root finding to locate the exact time of occurrence of the event causes continuous integration to become excessively slow. An equivalence relation eliminates the fast chattering motion, but preserves the dynamics of the slow motion along the chattering surface [36].

Table 7 presents the results of the evaluation of the newly developed simulator and the prototype HYBRSIM simulator (that is still used as a modeling environment). Note that the results are limited to index 1 systems. Improvement with respect to the prototype is the handling of time events. Furthermore, it should be noted that the newly developed simulator relies on compiled execution (instead of interpreted), and, therefore, is more efficient. Also, the newly developed simulator allows the use of more accurate numerical solvers such as DASSL whereas HYBRSIM only has the forward Euler algorithm available.

Table 7 also shows some features not yet implemented in the newly developed simulator. Note that the equation sorting and solving relies on an implicit formulation of model configuration changes. This prevents the explicit generation of all possible modes of behavior, which, in the worst case is exponential in the number of switches. In related work [31] it was shown how systems with run-time equation and index changes can be modeled and simulated in MATLAB-SIMULINK by using a two stage numerical simulation approach: after each integration step the subspace projection is executed.

When studying the aircraft elevator control benchmark example [12], it was found that modeling the detailed hydraulic behavior of the elevator redundancy control leads to a model with widely different time scales that cannot be simulated with state of the art tools such as Dymola [6]. One approach to mitigate these problems is by pre-processing the model before simulation and performing automated model reduc-

tion [26]. This has several advantages and implications: (i) simulation speed-up, (ii) the model can be designed without regards for simulation efficiency (often it is more convenient to model much detail), (iii) one 'mother' model that is the most detailed can be used for many tasks by automatically deriving less detailed models (e.g., for control design a low order model is desired), which guarantees consistency across all model incarnations that are used (e.g., for CACSD [28], fault detection and isolation [22], and training [16]), (iv) a systematic model reduction stage generates a complete model where manual reduction may not [26].

Another issue is the plethora of modeling formalisms used in present day system design and analysis (e.g., Statecharts [9], Petri nets [37], Grafcet [5], data flow and control flow diagrams [10, 45], bond graphs [11], and hybrid automata [1]). To provide a tailored modeling environment for each of these, an efficient approach is the modeling of the modeling formalisms. Such a meta-modeling approach appears to be well suited for handling heterogeneous systems with their multi-paradigm modeling requirement [32].

A Example DSblock Model Specification

For the two colliding bodies in Fig. 4.4 in Section 4, the automatically generated specification looks like ($v_1 = m1.i.\_$, $v_2 = m2.i.\_$, $x_1 = x1.s.\_$, $x_2 = x2.s.\_$, $m_1 = m1.I.\_$, and $m_2 = m2.I.\_$)

```
v2.i._ = m2.i._;
v1.i._ = m1.i._;
contact12.L._ = (contact12.preState == True) ?  1.0 :  0.0;
m2.v._ = - contact12.v._;
v2.v._ = m2.v._;
m2.v._ = v2.v._;
m2.res = -m2.deri._ + m2.v._ / m2.I._;
m1.v._ = + contact12.v._;
v1.v._ = m1.v._;
m1.v._ = v1.v._;
m1.res = -m1.deri._ + m1.v._ / m1.I._;
e.i._ = e.Iin._;
contact12.res = (- v2.i._ + v1.i._ - e.i._) * contact12.L._ + (contact12.L._ - 1) * contact12.v._;
x1.ders._= + v1.i._;
dv.s._ = - m2.i._ + m1.i._;
e.newIin._ = -1.0 * dv.s._;
x2.ders._= + v2.i._;
dx.s._ = + x1.s._ - x2.s._;
```

Note that the contact status is specified locally by a residue equation for `contact12`. Therefore, no global permutations are required. Also note that in case of contact the index of the system changes. This is correctly handled by reinitializing the generalized state based on the method that uses the pseudo Kronecker canonical form described in Section 3.

Bibliography

[1] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Lecture Notes in Computer Science*, volume 736, pages 209–229. Springer-Verlag, 1993.

[2] E. Anderson, Z. Bai, J. Bishop, J. Demmel, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide, Second Edition*. SIAM, Philadelphia, 1995.

[3] M. Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD dissertation, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

[4] François E. Cellier. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. PhD dissertation, ETH, Zurich, Switzerland, 1979.

[5] René David and Hassane Alla. *Petri Nets & Grafcet*. Prentice Hall Inc., Englewood Cliffs, NJ, 1992. ISBN 0-13-327537-X.

[6] H. Elmqvist, D. Brück, and M. Otter. *Dymola — User's Manual*. Dynasim AB, Research Park Ideon, Lund, 1996.

[7] Hilding Elmqvist *et al*. Modelica^{tm}–a unified object-oriented langauge for physical systems modeling: Language specification, December 1999. version 1.3, http://www.modelica.org/.

[8] Eberhard Griepentrog and Roswitha März. *Differential-Algebraic Equations and Their Numerical Treatment*. BSB Teubner, Leipzig, 1986. ISBN 3-322-00343-4.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[10] Derek J. Hatley and Imtiaz Pirbhai. *Strategies for Real-Time Systems Specification*. Dorset House Publishing Co., New York, New York, 1988.

[11] D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach*. John Wiley and Sons, New York, 2 edition, 1990.

[12] Dieter Moormann, Pieter J. Mosterman, and Gert-Jan Looye. Object-oriented computational model building of aircraft flight dynamics and systems. *Aerospace Science and Technology*, (3):115–126, 1999.

[13] Pieter J. Mosterman. *Hybrid Dynamic Systems: A hybrid bond graph modeling paradigm and its application in diagnosis*. PhD dissertation, Vanderbilt University, 1997.

[14] Pieter J. Mosterman. State Space Projection onto Linear DAE Manifolds Using Conservation Principles. Technical Report #R262-98, Institute of Robotics and System Dynamics, DLR Oberpfaffenhofen, P.O. Box 1116, D-82230 Wessling, Germany, 1998.

[15] Pieter J. Mosterman. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569, pages 164–177. Lecture Notes in Computer Science; Springer-Verlag, March 1999.

[16] Pieter J. Mosterman. Towards Model Manipulation for Efficient and Effective Simulation and Instructional Methods. In *Distributed Modelling and Simulation of Complex Systems for Education, Training and Knowledge Capitalisation*, Eze, France, May 1999.

[17] Pieter J. Mosterman. Implicit Modeling and Simulation of Discontinuities in Physical System Models. In *ADPM*, 2000.

[18] Pieter J. Mosterman and Gautam Biswas. A modeling and simulation methodology for hybrid dynamic physical systems. *SCS Transactions*. to be published.

[19] Pieter J. Mosterman and Gautam Biswas. Principles for Modeling, Verification, and Simulation of Hybrid Dynamic Systems. In *Fifth International Conference on Hybrid Systems*, pages 21–27, Notre Dame, Indiana, September 1997.

[20] Pieter J. Mosterman and Gautam Biswas. A Java Implementation of an Environment for Hybrid Modeling and Simulation of Physical Systems. In *ICBGM99*, pages 157–162, San Francisco, January 1999.

[21] Pieter J. Mosterman and Gautam Biswas. Building Hybrid Automata of Physical Systems for Real-time Application. In *Conference on Decision and Control*, December 1999.

[22] Pieter J. Mosterman and Gautam Biswas. Building Hybrid Observers for Complex Dynamic Systems using Model Abstractions. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, pages 178–192, 1999. Lecture Notes in Computer Science; Vol. 1569.

[23] Pieter J. Mosterman and Gautam Biswas. Deriving Discontinuous State Changes for Reduced Order Systems and the Effect on Compositionality. In 13^{th} *International Workshop on Qualitative Reasoning*, pages 160–168, Lock Awe Hotel, Scotland, June 1999.

[24] Pieter J. Mosterman and Gautam Biswas. Hybrid automata for modeling discrete transitions in complex dynamic systems. In *AAAI Spring Symposium Working Notes: Hybrid Systems and AI*, pages 136–141, Stanford University, CA, March 1999.

[25] Pieter J. Mosterman and Gautam Biswas. A comprehensive methodology for building hybrid models of physical systems. *AI Journal*, 2000. in press.

[26] Pieter J. Mosterman and Gautam Biswas. Towards Procedures for Systematically Deriving Hybrid Models of Complex Systems. In Nancy Lynch and Bruce Krogh, editors, *Hybrid Systems: Computation and Control*, pages 324–337, 2000. Lecture Notes in Computer Science.

[27] Pieter J. Mosterman, Gautam Biswas, and Martin Otter. Simulation of Discontinuities in Physical System Models Based on Conservation Principles. In *SCS Summer Simulation Conference*, pages 320–325, Reno, Nevada, July 1998.

[28] Pieter J. Mosterman, Gautam Biswas, and Janos Sztipanovits. A hybrid modeling and verification paradigm for embedded control systems. *Control Engineering Practice*, (6):511–521, 1998.

[29] Pieter J. Mosterman and Peter Breedveld. Some Guidelines for Stiff Model Implementation with the Use of Discontinuities. In *ICBGM99*, pages 175–180, San Francisco, January 1999.

[30] Pieter J. Mosterman, Jan F. Broenink, and Gautam Biswas. Model Semantics and Simulation of Time Scale Abstractions in Collision Models. In *Eurosim 98*, pages 230–237, Helsinki, Finland, April 1998.

[31] Pieter J. Mosterman, Peter Neumann, and Carsten Preusche. Modeling Systems With Variable Algebraic Contraints for Explicit Integration Methods. In *ADPM*, 2000.

[32] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling in Control System Design. In *IEEE International Symposium on Computer-Aided Control Systems Design*, 2000.

[33] Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. Model semantics and simulation for hybrid systems operating in sliding regimes. In *AAAI Fall Symposium on Model Directed Autonomous Systems*, pages 48–55, Boston, MA, 1997.

[34] Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. A Study of Transitions in Dynamic Behavior of Physical Systems. In 12^{th} *International Workshop on Qualitative Reasoning*, Cape Cod, MA, May 1998.

[35] Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. An Ontology for Transitions in Physical Dynamic Systems. In *AAAI98*, pages 219–224, July 1998.

[36] Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. Sliding mode model semantics and simulation for hybrid systems. In Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems V*, pages 218–237. Springer-Verlag, 1999. Lecture Notes in Computer Science.

[37] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[38] T. Nishida and S. Doshita. Reasoning about discontinuous change. In *Proceedings AAAI-87*, pages 643–648, Seattle, Washington, 1987.

[39] Martin Otter, Manuel A. Pareira Remelhe, Sebastian Engell, and Pieter J. Mosterman. Hybrid models of physical systems and discrete controllers. *at - Automatisierungstechnik*, 2000.

[40] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal of Scientific and Statistical Computing*, 9(2):213–231, March 1988.

[41] Taeshin Park and Paul I. Barton. State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation*, 6(2):137–165, April 1996.

[42] Linda R. Petzold. A description of DASSL: A differential/algebraic system solver. Technical Report SAND82-8637, Sandia National Laboratories, Livermore, California, 1982.

[43] A. J. van der Schaft and J. M. Schumacher. The complementary-slackness of hybrid systems. *Math. Contr. Signals Syst.*, (9):266–301, 1996.

[44] George C. Verghese, Bernard C. Lévy, and Thomas Kailath. A generalized state-space for singular systems. *IEEE Transactions on Automatic Control*, 26(4):811–831, August 1981.

[45] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.