15th Triennial World Congress, Barcelona, Spain

EMBEDDED CODE GENERATION FOR EFFICIENT REINITIALIZATION

Pieter J. Mosterman and John E. Ciolfi

Simulation and Real-Time Technologies, The MathWorks, Inc., Natick, MA 01760-2098,
[pieter_j_mosterman|ciolfi]@mathworks.com

Abstract: Embedded control system design involves continuous time, discrete event mode switching, and discontinuities in system behavior and requires support for: (i) continuous behavior, (ii) discrete event behavior, and (iii) re-initialization when discrete events occur. The graphical block diagram formalism that supports continuous behavior modeling is extended by logic components that seamlessly integrate. Re-initialization is supported by *state reset* ports and *state output* ports of the integrator component. In addition, the dynamic semantics of a formalism is specified by a *computational model*. For MATLAB-SIMULINK this consists of a number of interface methods that are called at pre-defined points in the model execution. Automatic generation of efficient embedded code allows the use of high-level modeling formalisms for design and analysis.

Keywords: Embedded systems, Hybrid systems, Induction motor design, Numerical simulation, Modeling

1. INTRODUCTION

Embedded computing power, often of a distributed nature, is increasingly becoming available in a wide variety of engineered systems ranging from consumer products to automobile control and aircraft data acquisition systems. To exploit the ultimately ubiquitous computing power, embedded software has to be produced that captures increasingly complex functionality. Advances in software design have shown that managing the complexity of large software is extremely difficult and requires systematic approaches. For embedded software the problem is exacerbated by the harsh requirements on performance and strict limitations on resources. In addition, requirements for embedded software are quite different from those of, e.g., interactive software (Lee, 2000). For example, for real-time systems, the result of a computation is only correct if it is available at a given point in time. Also, instead of guarantees for termination, the embedded software has to be guaranteed *not* to terminate or deadlock.

As a result, embedded software is mostly produced by domain engineers. Model integrated computing (Karsai *et al.*, 1998) addresses the need for domain engineers to design embedded systems by generating domain specific modeling formalisms that provide an intuitive and highly constrained modeling environment and model interpreters that facilitate automatic generation of executable code. In addition, this paradigm bridges the gap between physical system modeling and specification of the information system which facilitates holistic design approaches (e.g., mechatronics).

Now, the combination of continuous behavior exhibited by, e.g., physical systems and PID controllers, with discrete event behavior because of, e.g., discrete event and sampled data control, leads to systems that comprise both continuous as well as discrete phenomena. Such systems are referred to as *hybrid dynamic systems* and have been the subject of much research recently (Mosterman, 1997). Because it combines continuous behavior with discrete mode switching, embedded control is characteristic for hybrid dynamic systems.

To illustrate, consider the system in Fig. 1 that represents the control of the head of a CD player. The physical process, or plant, is modeled as a second order system. It is controlled by one of two control laws: When the head is far away from the desired track, a coarse control law quickly moves the head to the proximity of the desired position. High precision control then takes over to fine tune the position. A logic control structure selects the appropriate control law depending on the system variables, i.e., in this case position error.

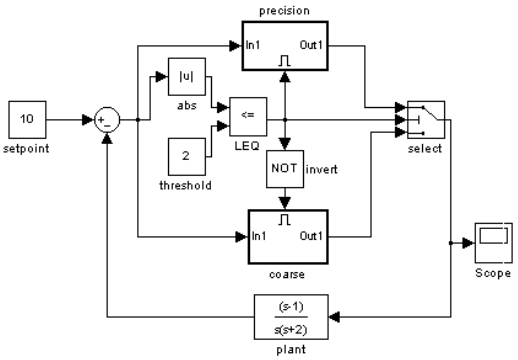


Fig. 1. Multiple-mode control.

The generation of efficient embedded software for hybrid dynamic systems has to be explicitly concerned with the nature of these systems. Efficiency can be achieved by separating the system into three distinct parts.

The continuous behavior is best described by differential equations, often in an explicit ordinary differntial equation (ODE) form. Numerical integration routines solves these ODEs to generate trajectories of continuous behavior and achieve real-time behavior because explicit implementations can be applied.

The discrete event behavior is most efficiently handled by event driven simulators. Sampled data behavior can be included in this part, in which case, however, interaction with the continuous behavior exists that may complicate simulation (e.g., VHDL-AMS simulator).

Upon startup, a set of values has to be available to initialize the system state. Furthermore, whenever events occur, the continuous model may change and new initial values may have to be computed. This (re-)initialization may be arbitrarily complex, not only in terms of the computations but also in their part of the execution order, and involve extensive logic conditions. In general, the model initialization part can be separated from the model part that specifies the dynamics of continuous behavior.

This paper evaluates the model (re-)initialization part of MATLAB-SIMULINK (SIMULINK, 1997) and how it combines this with the continuous and discrete model parts both in terms of a graphical formalisms as well as the *computational model* (Girault *et al.*, 1999). The modeling, simulation, and code generation of the electrical circuitry of an induction motor is described. This circuitry consists of a number of switches, commuting diodes, and inductors that may be switched in series, and, therefore, be directly coupled. Consequently, this constitutes a hybrid dynamic system with a variable set of state variables and run-time invocation of algebraic constraints between state variables, which corresponds to a higher index system of differential and algebraic equations (DAE) (Gear, 1988).

2. PROGRAM SYNTHESIS

Much of the design and analysis of control laws for engineered systems relies on computer simulation of the controlled plant system. In these phases, issues such as stability and performance are assessed. This requires a control law specification, or model, and typically this is available in the form of block diagrams. In addition, a model of the physical plant is required as well.

Once this analysis has resulted in a desirable implementation, the control is translated into embedded software. This software is executed on an embedded processor and again tested substituting the plant model for the actual physical process, i.e., hardware-in-the-loop simulation. This may result in changes to the control law and a new round of design changes is initiated first in simulation and then again the controller hardware implementation combined with a plant model. The repeated analysis and translation of the control law is costly, error-prone, and unwieldy.

Recently, automatic code generation facilities have become available that allow for automatic translation from a block diagram that captures a control law specification or plant model to embedded real-time code (Harel, 2001; Halbwachs *et al.*, 1991). This code generation step has to satisfy stringent constraints imposed on embedded code to fit the harsh requirements on resources in this domain (Lee, 2000). Moreover, the code has to be readable and transparant to provide a first pass of confidence.

A straightforward code generation approach uses a library of source and compiled code for the primitives of the modeling formalism. Analysis of the model topology generates the connection constraints and allows sorting the instances of library components used in the model to obtain an execution order. This then forms the basis for the main procedure that executes model behavior.

The functionality embodied by each of the model components is typically captured by a number of interface methods that have to be implemented. For example, the S-Function interface of MATLAB-SIMULINK requires the functions in Table 1 to be implemented. Similar functionality is present in Ptolemy (Davis, II *et al.*, 1999) and DSblock software. [1]

[1] See http://www.modelica.org/dsblock/dsblock4.0a.shmtl.

Table 1. S-Function Routines.

S-Function Routine	Simulation Stage
mdlInitializeSizes	Initialization
mdlGetTimeOfNextVarHit	Calculation of next sample hit (optional)
mdlOutputs	Calculation of outputs
mdlUpdate	Update discrete states
mdlDerivatives	Calculation of derivatives
mdlTerminate	End of simulation task

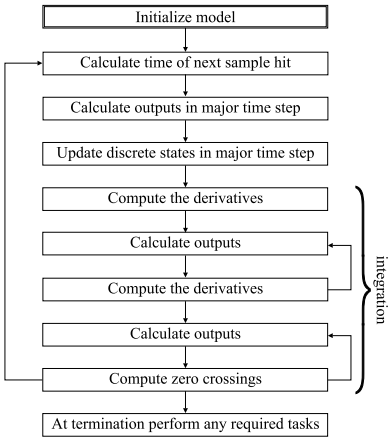


Fig. 2. Execution order.

The generated code calls the particular S-function routines at the correct point in the execution order. This execution order is depicted in Figure 1. When simulation starts, the initialization procedures are first executed. After this, sampled data systems allow computation of the next point in time at which an event occurs. Continuous simulation is then invoked up till this time point. To this end, first the entire system is evaluated so all variables are known, and the discrete variables, i.e., those that only change at events, are set. Continuous simulation then repeatedly evaluates the derivatives and outputs of the system to determine the continuous evolution. A monitoring module halts continuous integration when an event occurs before the pre-computed event time point by computing the zero crossing of system variables. Otherwise, integration is performed till this time point is reached. Next the discrete event model part is evaluated, which may include iteration, and the same procedure is repeated.

In order to generate efficient code, the models are separated in (i) the (re)initialization part that pertains to the computations required before continuous simulation can start, (ii) the discrete model part that entails the variables that are constant during continuous simulation, i.e., variables that only change at event times, and (iii) the continuous part that embodies the differential equations.

3. INITIALIZATION

A characteristic of hybrid dynamic systems is that during simulation events occur. At such an event, and when simulation starts, initial conditions have to be computed. For example, in case of a bouncing ball, when the ball hits the floor, the model does not change, but is reinitialized with new velocity $v = -\epsilon v^0$, with v^0 the velocity immediately before the collision and ϵ the coefficient of restitution.

A graphical modeling formalism has to support modeling the re-initialization of state variables which can be extremely involved, require access to different model variables and apply a manifold of initialization conditions. In the general case of a system of differential and algebraic equations, the generalized state space (Verghese *et al.*, 1981) is of a higher dimension than the space of valid or consistent initial values. This issue is not discussed any further in this paper.

4. GRAPHICAL FORMALISM

Graphical formalisms tailored to the needs and requirements of a particular domain are critical to prevent unnecessary errors because of, e.g., required work-arounds, non-intuitive semantics, missing and easily overlooked constraints. Control law design is deeply rooted in the block-diagram formalism, and, therefore, the MATLAB-SIMULINK graphical model development environment is well-suited for such tasks. In addition, physical system modeling approaches based on deriving a mathematical formulation (in contrast to energy based modeling approaches, e.g., bond graphs (Karnopp *et al.*, 1990)) also fit the block-diagram modeling paradigm very well.

Block diagrams facilitate a set of basic mathematical operations such as addition and multiplication, and more involved functions such as trigonometrics. An important place is taken by the time-integration operator. The differential equations because of this operator require the use of numerical solvers to generate behaviors. Combining it with other mathematical operations allows for complex nonlinear sets of ODEs.

The need for discrete event behavior in control law switching or to model physical switching phenomena is addressed by facilitating a set of logic operators. These satisfy the same graphical syntax and their execution is specified in a form that is compatible with the other block diagram elements, which allows a seamless combination of the different types of operators. For example, embedding a particular control law in a *sub-system* allows the enabling and disabling of these computations depending on the mode of the controller at a higher hierarchical level. This is graphically depicted by connecting a decision structure to an enable port of the sub-system as illustrated for the coarse and fine control in Fig. 1. The exact semantics of the activation can now be selected, i.e., on a rising or falling edge, both or for a particular level. In Fig. 1 the different control laws are activated based on the level of a logical signal.

When mixing continuous and discrete behavior, the re-initialization takes a prominent place. In its most basic form, this is present in the integrator block as an additional input, the *state reset* port, that connects to a computational structure solely present for this purpose. Again, this state reset port has to be given the desired semantics as to when to update the internal state of an integrator. To illustrate, consider the bouncing ball example in Fig. 3. When the logic decision structure in the collision detection block detects the rising edge of a zero-crossing, i.e., the ball hits the floor, the integrator value is re-initialized to reverse the velocity, while taking a coefficient of restitution into account. Note that the velocity change is computed from the velocity immediately before the bounce. As updating the value stored by the integrator would affect the computation, the *state output* port is used instead of the normal integrator output. This additional port is specifically included in the MATLAB-SIMULINK block diagram modeling formalism to facilitate such computations.

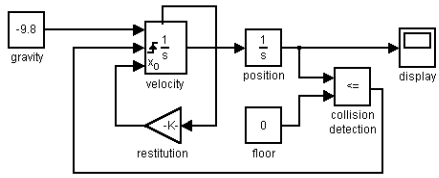


Fig. 3. Bouncing ball MATLAB-SIMULINK model.

Re-initialization takes place when an event occurs and the `mdlOutputs` method is executed.

```
void MdlOutputs(int_T tid)
{ if (ssIsContinuousTask(rtS, tid)) { /* Sample: [0.0, 0.0] */
    /* Integrator: '<Root>/position' */
    rtB.position = rtX.position_CSTATE;
    /* RelationalOperator: '<Root>/collision detection'
    * incorporates:
    * Constant: '<Root>/floor' */
    rtB.collision_detection = (rtB.position <= rtP.floor_Value);
    /* Gain: '<Root>/restitution'
    *
    * Regarding '<Root>/restitution':
    * Gain value: rtP.restitution_Gain */
    rtB.restitution = rtX.velocity_CSTATE * rtP.restitution_Gain;
    /* Integrator: '<Root>/velocity' */
    if (ssIsMajorTimeStep(rtS)) {
      ZCEventType zcEvent;
      /* evaluate zero-crossings */
      zcEvent = rt_ZCFcn(RISING_ZERO_CROSSING,
        &rtPrevZCSigState.velocity_ZCE, rtB.collision_detection);
      if (zcEvent|| rtDWork.velocity_IWORK.IcNeedsLoading) {
        rtX.velocity_CSTATE = rtB.restitution;
      }
      rtDWork.velocity_IWORK.IcNeedsLoading = 0;
    }
    rtB.velocity = rtX.velocity_CSTATE;
  }
}
```

This implies that the re-initialization equations (for the bouncing ball, this is the `rtB.restitution` assignment) are mixed with continuous behavior, and, therefore, evaluated during continuous integration as well. This incurs an efficiency penalty that can become significant when extensive re-initialization computations are required.

Note that the re-initialization becomes more complex as more complex model structures are to be facilitated. For example, it may be desirable to conditionally re-initialize the entire state vector of a sub-system. This could be the case in the multiple-mode control example in Fig. 1 when sophisticated high-order model based control laws are applied.

5. THE AC INDUCTION MOTOR

The electrical circuitry of an induction motor contains a number of cascaded inductances, I_i, each with parasitic resistance, R_i,, as shown in Fig. 4. To control the flux in each of the inductors, they are connected in series separated by a *bridge* consisting of a switch to ground and a switch to the source voltage, V_{cc}. Each switch is equipped with a commuting diode to protect the electrical circuit from voltage spikes.

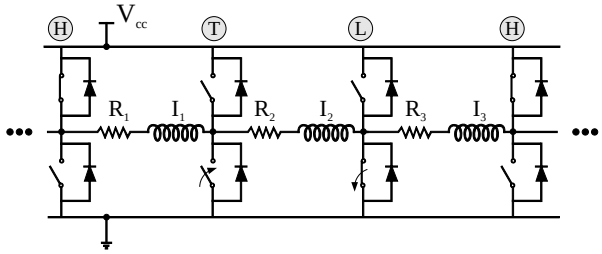


Fig. 4. Electrical circuit of an induction motor.

The motor is driven by changing the flux in the inductors, depending on the angle of the rotor. To achieve the desired flux values, each inductor can be connected to the source voltage and ground in two different directions, causing a voltage drop V_{cc} or $-V_{cc}$. A complex scheme closes and opens each of the switches. When a switch opens, the corresponding commuting diode may become active until the two connected inductors draw the same current and they can be coupled without inducing a spike. If a bridge closes a current path to V_{cc} it operates in its *high* (H) state, if it closes a current path to ground, it operates in its *low* (L) state, and if neither current path is closed, it is in its *tri* (T) state, see Fig. 4.

In case a bridge is in its tri state, the two connected inductors are coupled and their fluxes, p_i, are algebraically related. This means the state space reduces by one dimension and a redistribution, i.e., re-initialization, is required based on the fluxes immediately before switching to the tri state, p_i^0. This re-initialization is governed by conservation of flux principle and can be computed using the inductance values, L_i, of the inductors involved. Because, in general, multiple bridges may be in their tri state several inductors may be directly connected. To compute the new flux value in each inductor, the general formula

$$p_i = L_i \frac{\sum_j p_j^0}{\sum_j L_j}, \ (\forall j)(p_j \in C) \qquad (1)$$

can be applied (Mosterman *et al.*, 2000). Here C is the set of all states p_j that are coupled, i.e., algebraically

related, with p_i. Based on this computation, the re-initialization for each component only requires numerical knowledge of the total value of the combined states, p_j, that are collapsed into one, p_i, and the parameters that determine the weighting, L_j. This information is additionally supplied to each model component. No algebraic knowledge of a model component's internal structure and algebraic manipulations are required to execute the re-initialization.

The system is modeled in MATLAB-SIMULINK by a ring of inductor/resistor components. The block diagram of the constituent equations of these components are shown in Fig. 5. The crucial element in the model is the integrator (in the right-hand path). The state reset port of this integrator is connected to the block diagram structure in the left-hand part of the model that computes the new flux values based on the inductances and flux values of the connected inductor/resistor components. Note that this includes the flux value, i.e., value of the integrator block, of the inductor/resistor itself. If no provisions were taken, this would lead to circular dependency. Therefore, the *state output port* of the integrator block is used instead. This port supplies the value of the internal state updated at another point in the computational order, thereby breaking the dependencies.

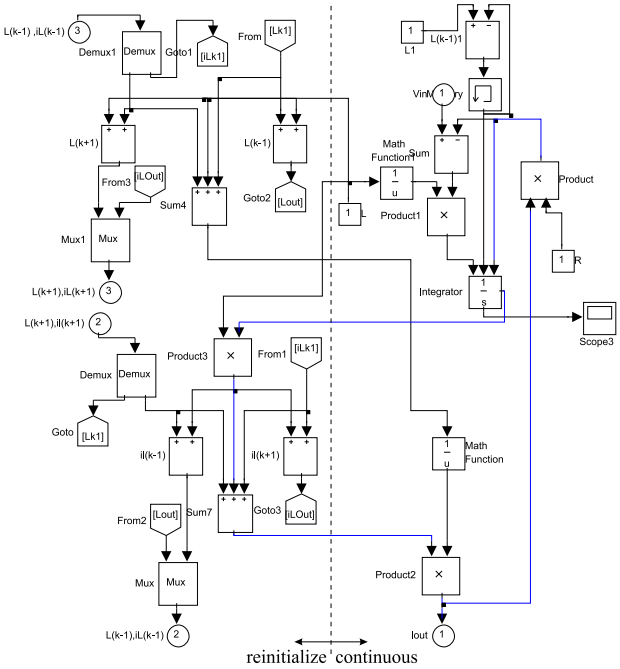
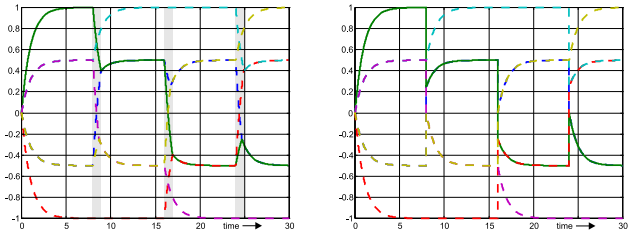


Fig. 5. Model of resistor/inductor component.

Control logic switching is modeled by a state transition table and comparators model the internal event diode switching. A simulation run of six diodes connected as a ring, with three bridge state changes is shown in Fig. 6. The solid curve shows how the current from one inductor changes over time to achieve desired flux values. The dashed curves represent neighboring currents that may be coupled with the current of the solid or dashed curves. In Fig. 6(a), the gray intervals show periods of time when the diodes become active, the *commuting* phase, resulting in C^0 hybrid behavior, i.e., trajectories are continuous. In Fig. 6(b) the continuous transients because of the commuting diodes are abstracted away, i.e., the diodes are removed from the model, to obtain faster simulation. As a result, the system includes discontinuities in state variables that are handled based on conservation of flux.



(a) (b)

Fig. 6. Induction motor simulation with (a) and without (b) commuting diodes.

The automatically generated embedded code contains two methods that pertain to the re-initialization. The method

```
void MdlOutputs(int_T tid)
{
   /* local block i/o variables */
   real_T rtb_Product3;
   real_T rtb_temp15;
   real_T rtb_temp16;
   if (ssIsSampleHit(rtS, 1, tid)) { /* Sample: [0.2, 0.0] */
      /* Memory: '<S1>/Memory' */
      rtB.Memory = rtDWork.Memory_PreviousInput;
   }
   if (ssIsContinuousTask(rtS, tid)) { /* Sample: [0.0, 0.0] */
      /* Product: '<S1>/Product3' incorporates:
      * Constant: '<S1>/L' */
      rtb_Product3 = rtP.L_Value * rtX.Integrator_CSTATE;
   }
   if (ssIsSampleHit(rtS, 1, tid)) { /* Sample: [0.2, 0.0] */
      /* Math: '<S1>/Math Function' incorporates:
      * Sum: '<S1>/Sum4'
      * Constant: '<S1>/L' */
      rtB.Math_Function = 1.0/((0.0 + rtP.L_Value + 0.0));
   }
   if (ssIsContinuousTask(rtS, tid)) { /* Sample: [0.0, 0.0] */
      /* Product: '<S1>/Product' incorporates:
      * Product: '<S1>/Product2'
      * Sum: '<S1>/Sum7'
      * Constant: '<S1>/R' */
      rtB.Product = ((0.0 + rtb_Product3 + 0.0) * rtB.Math_Function) * rtP.R_Valu
      /* Integrator: '<S1>/Integrator' */
      if (ssIsMajorTimeStep(rtS)) {
         ZCEventType zcEvent;
         /* evaluate zero-crossings */
         zcEvent = rt_ZCFcn(RISING_ZERO_CROSSING,
         &rtPrevZCSigState.Integrator_ZCE, rtB.Memory);
         if (zcEvent|| rtDWork.Integrator_IWORK.IcNeedsLoading) {
            rtX.Integrator_CSTATE = rtB.Product;
         }
         rtDWork.Integrator_IWORK.IcNeedsLoading = 0;
      }
      rtB.Integrator = rtX.Integrator_CSTATE;
   }
   if (ssIsSampleHit(rtS, 1, tid)) { /* Sample: [0.2, 0.0] */
      /* Sum: '<S1>/L(k+1)' incorporates:
      * Constant: '<S1>/L' */
      rtb_temp15 = rtP.L_Value + 0.0;
      /* Sum: '<S1>/L(k-1)' incorporates:
      * Constant: '<S1>/L' */
      rtb_temp15 = 0.0 + rtP.L_Value;
      /* Sum: '<S1>/L(k-1)1' incorporates:
      * Constant: '<S1>/L1' */
      rtB.L_k_1_1 = rtP.L1_Value - rtB.Memory;
      /* Math: '<S1>/Math Function1' incorporates:
      * Constant: '<S1>/L' */
      rtB.Math_Function1 = 1.0/(rtP.L_Value);
   }
   if (ssIsContinuousTask(rtS, tid)) { /* Sample: [0.0, 0.0] */
      /* Product: '<S1>/Product1' incorporates:
      * Sum: '<S1>/Sum' */
      rtB.Product1 = rtB.Math_Function1 * (0.0 - rtB.Product);
      /* Sum: '<S1>/il(k+1)' */
      rtb_temp16 = rtb_Product3 + 0.0;
      /* Sum: '<S1>/il(k-1)' */
      rtb_temp16 = 0.0 + rtb_Product3;
```

```
    }
}
```

is called at each system evaluation for computing the continuous behavior. However, most of the code pertains to the re-initialization of the integrator element `Integrator`. How to adjust the code generation facilities of MATLAB-SIMULINK to optimize the generated code for handling external blocks, producing compact and readable code is subject of current research.

6. CONCLUSIONS

Control law design requires support for: (i) continuous behavior, (ii) discrete event behavior, and (iii) re-initialization when discrete events occur. In the graphical block diagram formalism that supports modeling of continuous behavior logic components for discrete behavior are seamlessly integrated. The re-initialization is supported by *state reset* ports and *state output* ports of the integrator component.

The dynamic semantics of a graphical formalism is specified by a *computational model*. For MATLAB-SIMULINK this consists of interface methods that are called at pre-defined points in the model execution. Efficient real-time code generation eliminates the need for manual software design and allows the use of high-level modeling formalisms in the design and analysis stages to improve understanding of system behavior. Moreover, it closely fits modeling of physical systems and more quickly or even automatically experimenting with different implementations.

7. REFERENCES

Davis, II, J., *et al*. (1999). Ptolemy II – heterogeneous concurrent modeling and design in java. Dept. of EECS, UC Berkeley.

Gear, C. W. (1988). Differential-algebraic equation index transformations. *SIAM J. on Scientific and Statistical Computing* **9**(1), 39–47.

Girault, A., B. Lee and E. A. Lee (1999). Hierarchical Finite State Machines with Multiple Concurrency Models. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **18**(6), 742–760.

Halbwachs, N., P. Raymond and C. Ratel (1991). Generating efficient code from data-flow programs. In: *Third Intl. Symp. on Programming Language Implementation and Logic Programming*. Passau, Germany.

Harel, D. (2001). From play-in scenarios to code. *Computer* **34**(1), 53–60.

Karnopp, D.C., D.L. Margolis and R.C. Rosenberg (1990). *Systems Dynamics: A Unified Approach*. 2 ed.. John Wiley. New York.

Karsai, G., J. Sztipanovits and H. Franke (1998). Towards Specification of Program Synthesis in Model-Integrated Computing. In: *Proc. of the ECBS-98*. Jerusalem, Israel. pp. 226–233.

Lee, E. A. (2000). What's Ahead for Embedded Software. *Computer* **33**(9), 18–26.

Mosterman, P. J. (1997). Hybrid Dynamic Systems: A hybrid bond graph modeling paradigm and its application in diagnosis. PhD dissertation. Vanderbilt University.

Mosterman, P. J., P. Neumann and C. Preusche (2000). Modeling Systems With Variable Algebraic Contraints for Explicit Integration Methods. In: *ADPM*. pp. 251–256.

SIMULINK (1997). *Dynamic System Simulation for Matlab*. The MathWorks.

Verghese, G. C., B. C. Lévy and T. Kailath (1981). A generalized state-space for singular systems. *IEEE Trans. on Automatic Control* **26**(4), 811–831.