From *Proc. IEEE Conference on Decision and Control*, San Antonio TX, December 1993. ©IEEE

Toward a Modeling Language Standard for Hybrid Dynamical Systems

James H. Taylor
Odyssey Research Associates (ORA)
301 Dates Drive, Ithaca, NY 14850
jim@oracorp.com

Abstract[1]

A rigorous means for modeling and evaluating hybrid systems is needed for the efficient and cost-effective development of embedded real-time software for a wide variety of applications (weapons, manufacturing, intelligent vehicles, process control, ...). The development of a standard hybrid systems modeling language (SHSML) and corresponding rigorous simulation environments represent important contributions to fulfilling this requirement.

1. Introduction

A standard hybrid systems modeling language called *SHSML* is under development, to serve five purposes:

- to define formally what is meant by the term "hybrid system",
- to provide a modeling language that matches recent advances in mathematical formalisms for hybrid systems [1], and thus permits their rigorous evaluation,
- to define an architectural description language for hybrid systems, in support of domain-specific / reference-architecture-based approaches to hybrid systems development [2],
- to provide the basis for a language-based "front end" for hybrid system simulation environments, and
- to serve as a definitive interface to software engineering tools for real-time code generation and composition of the overall architecture.

The specific target in regard to the fourth point is Hybrid DsTool [3], a package that very rigorously simulates hybrid systems but lacks such a front end.

[1]Support for the research described herein has been provided by the Advanced Research Project Agency (ARPA) through the U.S. Army AMCCOM at Picatinny Arsenal, NJ by Contract Number DAAA21-92-C-0013.

2. The Hybrid Systems Domain

SHSML is being designed to support a broad definition of a hybrid system, which we may express informally as being an arbitrary interconnection of components that are arbitrary instances of continuous-time, discrete-time and logic-based subsystems. As such, SHSML may be considered to be a general architectural description language (ADL) for hybrid systems, with a scope that encompasses not only the **digital software components** but also the **physical subsystems** (e.g., tank and propulsion unit, machine tool and drives) and **humans-in-the-loop** (by standard modeling approaches including perception, decision-making, neuro-muscular delay, etc.). This breadth is a prerequisite for the meaningful evaluation of embedded real-time software modules, as their dynamic behavior cannot be decoupled from that of the "real-world" portions of the system. Within this framework, SHSML will permit each component to be modeled with high fidelity (to whatever level the modeler deems appropriate) and provide features that will permit the hybrid system to be simulated with utmost accuracy.

SHSML is based on the conceptual definition of a hybrid system that underlies Hybrid DsTool [3] and on the modeling environment provided by Simnon [5]. Simnon was used as the starting point because it provides a solid *language-based* environment for building hierarchies of interconnected components of various types, and because it has a degree of rigor and encapsulation not found in most other modeling methods. In addition, features of the earlier standard CSSL [6], the ACSL modeling approach [7], and the MEAD definition of component interconnection [8] have been considered. Note that *graphical* model-building standards are not considered – we believe that the definition of a language standard must precede the definition of a graphical one.

2.1. Hierarchical Model Building:

Specific requirements addressed by this modeling language research include:

- Components at the lowest level are "pure" continuous-time, discrete-time or logic-based (symbolic) in type (see below).
- Pure components may be arbitrarily assembled into composite components.
- Each component is completely encapsulated. The only access to its variables is via **first-class input/output ports** (which may be connected to other component first-class i/o ports) or **second-class input/output ports** (which provide the means to change parameters or display (e.g. plot) variables; the latter cannot be connected to first-class i/o ports).
- Components (pure or composite) may be arbitrarily configured into hierarchical hybrid systems architectures.

These high-level features are designed to promote *model reuse* and *automatic component generation.*

2.2. Continuous-time Components (CTCs):

A continuous-time component may be described by an arbitrary ordinary differential equation set:

$$\begin{aligned} \dot{x_c} &= f_c(x_c, u_c, u_k, m_j, b_i, t) && (1) \\ y_c &= g_c(x_c, u_c, u_k, m_j, b_i, t) && (2) \end{aligned}$$

where x_c is the state vector, y_c is the output vector, u_c and u_k are numeric input signals (continuous- and discrete-time, respectively), b_i is a vector of discrete-time boolean variables, m_j is comprised of symbolic input variables, and t is the time; in general u_c, u_k, b_i and m_j are vectors. There are implicit "zero-order holds" operating on the elements of u_k, b_i and m_j, i.e., these inputs remain constant between those times when they change instantaneously.

- **Note:** The above notation is not all-inclusive – classes of differential-algebraic and implicit ordinary differential equations may also be supported.
- Rigorous handling of unpredictable state events (e.g., mechanical parts engaging and disengaging) will be supported [9]. In such cases the dynamics are unsmooth; transitions from one smooth model to another must be done carefully to avoid numerical integration errors.
- Support for models that undergo structural changes (e.g., changes in the definition or number of state variables) will be provided. In the case of mechanical parts engaging, the number of states decreases, producing a "higher-index" model that can be reduced using the Pantelides algorithm [10] to automatically reduce it to state–space form.

Example: a CTC may represent the continuous-time dynamics of an aircraft or land vehicle; an input u_c might represent wind-gust forces, u_k could be a controller actuation command (with implicit or explicit digital-to-analog conversion), b_i might signal whether or not an actuator has failed, and m_j might define a higher-level symbolic condition such as '`engine-has-stalled`'.

2.3. Discrete-time Components (DTCs):

A discrete-time component may be represented by an arbitrary up-date or difference equation set:

$$\begin{aligned} x_{k+1} &= f_k(x_k, u_c, u_k, m_j, b_i, k) && (3) \\ y_{k+1} &= g_k(x_{k+1}, u_c, u_k, m_j, b_i, k) && (4) \end{aligned}$$

where x_k is the discrete state vector, k is the index corresponding to the discrete time point t_k, y_{k+1} is the output vector, and u_c, u_k, b_i, m_j are as above. Note that there are implicit "sampling" operators on u_c, i.e., the input value $u_c(t_k)$ is used in updating x_k. The times t_k are usually – but not necessarily – uniformly spaced ($t_k = k{*}T_s$ where T_s is the "sampling time"); in any case we assume that the update times can be anticipated.

- Note that there may be computational delays (e.g., y_{k+1} may be output at time $t_k + \Delta$) – the language will allow offsets wherever required. This component type represents a particular digital module class that is reserved for pure numerical computations. The advantages of this particular taxonomy are that (i) the detailed structure of Eqns. $(3, 4)$ can be fully supported, and (ii) such components can meaningfully be linearized and analyzed while, in general, logic-based components (below) cannot.
- Digital modules are easier to emulate in a digital simulation environment; therefore, we do not anticipate that special features for state-event handling will be required.
- Support for models that undergo structural changes (e.g., changes in the definition or number of discrete state variables) will be provided.

Example: a DTC may represent the discrete-time numerical algorithm of a Kalman filter or LQR controller; an input b_i might govern whether or not the algorithm has to accommodate a sensor failure, and a symbolic input m_j may provide information for modifying the algorithm ('target-is-accelerating' might necessitate switching to a 9-state Kalman filter).

2.4. Logic-based Components (LBCs):

Each logic-based component may have numeric and/or symbolic inputs, symbolic outputs, and symbolic internal variables called "modes". At this point, it is not clear that these components have a "generic form" in mathematical terms as above except in terms of the categorization of input and output variables. Thus we *formally* write

$$m_{j+1} = \Phi_j(m_j, u_c, u_k, b_i, j) \qquad (5)$$

where m_j is the mode vector, j is the index corresponding to the discrete event triggering the LBC action, Φ_j is an undefined relationship (logic), and u_c, u_k, b_i are as above. The output of each LBC is the mode m_j which changes instantaneously at a discrete event (e.g., triggered by an event in a CTC such as a sensor failure); in contrast to the case in DTCs, we assume that mode changes usually can*not* be anticipated.

- There may be a computational delay between the trigger event and the mode change; this may be modeled with varying degrees of realism, from a fixed delay time to an actual emulation of the computational burden required in handling the event.

- LBCs will also exhibit unpredictable state- or discrete-event behavior – provision will provided for this as in the continuous-time case.

- Lack of a unifying paradigm for logic-based components precludes providing more than a "shell" definition for this class of component.

Example: an LBC may represent a discrete-event system or AI-based module that implements a failure detection, isolation and accommodation scheme, or serve as a means of managing the complicated sequence of continuous-time state events involved in reconfiguring a flight-control system when an aircraft engine stalls.

2.5. Composite Components (CCs):

As mentioned before, pure components may be arbitrarily assembled into composite components (CCs), and CCs in turn can be used to build arbitrary hierarchical hybrid systems. Combining pure components to create a CC is illustrated in Section 3.3.

3. SHSML Overview

The semantics of SHSML have been developed in detail and documented [11]. Its syntax is still undefined; this task will be completed in conjunction with the development of a SHSML interpreter. Page limitations do not permit a full description of the semantics of SHSML; therefore, we simply conclude by providing a few comments to establish the appropriate context for the developments in [11] and two small illustrative examples: a low-order CTC and a composite component.

3.1 Comments:

SHSML will naturally lack many of the features of a modern general-purpose high-order language:

- to avoid excessive complexity and unnecessary detail, and

- to allow model-specific support (see below).

Modeling support will include:

- syntactic checks (e.g., is there a state differential equation corresponding to each declared state variable, do all input/output connections involve consistent variable types), and

- semantic checks (e.g., detection of algebraic loops, checks for the correct use of structures for state-event handling, model structural changes, etc.)

Note that some of the features of the modeling language outlined here impose important requirements on the corresponding algorithms for numerical integration. Those considerations are also discussed in [11].

3.2 Illustration 1 - a Simple CTC:

The following SHSML model represents a nonlinear electro-mechanical component. There is one elementary state event, modeling Coulomb friction:

```
CTC turret is
%
% an electric-drive 'turret' example
%
interface         %  interface definition:
  input(volts,signal,(-vl,vl));  % applied voltage
  input(load_dist,signal);  % load disturbance
  input(Ksat,real);         % variable gain
  output(theta,signal);     % turret azimuth angle
  output(volt_lim,signal);  % aux. output variable
  knob(del_theta);          % supply any offset
  view(curr);               % displayed variable
end interface;
body              %  body definition:
  declarations      %  internal variables:
    state(theta,theta_dot); % state and deriv names
    state(theta_dot,moment);
    local(moment,signal,(-m_max,m_max));
    flag(funct01);          % state-event flag
  end declarations;
  initial           %  now handle initialization:
    % initial turret azimuth angle offset:
    theta = del_theta;
    % initial friction sign:
    sgn = if theta_dot < 0.0 then -1.0
          else if theta_dot > 0.0 then 1.0
          else sign(moment);
  end initial;
  event(funct01)    %  now handle the state event:
    funct01 = theta_dot    % zero-crossing => event
    negative-to-positive
      sgn = 1.0;
    end negative-to-positive;
    positive-to-negative
```

```
        sgn = -1.0;
      end positive-to-negative;
    end event;
    dynamics          %  now handle system dynamics:
      %  input is softly saturated
      volt_lim = volts/(1.0 + Ksat*abs(volts));
      curr = (volt_lim - Km*theta_dot)/R;  %  current
      frict = B * sgn;            %  Coulomb friction
      %  combine electrical, friction & load torques
      moment = (Km*curr - frict - load_dist)/J;
    end dynamics;
    assignments       %  now assign parameter values:
      vl: 15.0;          %  [volts]
      del_theta: 0.1;    %  [rad] - default value
      Ksat: 2.0;         %  [v**-1]
      Km: 6.2E-3;        %  [Nm/A]
      R: 5.3;            %  [Ohm]
      B: 1.0E-2;         %  [Nm]
      J: 7.5E-7;         %  [kg-m**2]
      m_max:  25.0;      %  [Nm]
    end assignments;
  end body;
  end turret;
```

The above example illustrates many general semantic features of the proposed SHSML language:

- Note that the notation `%` denotes the beginning of a comment; here we have used this feature to provide a road-map of the component model.
- Superfluous and / or redundant programming has been minimized, and the language has been kept simple. Simplicity is important because it allows the incorporation of model checking in the compiler or simulator that can perform equation sorting, detection of algebraic loops, checking for topological consistency, etc.
- The tentative syntax above is moderately terse rather than verbose. A more readable or novice-friendly notation might use key words (as in Ada); for example

  ```
  output(name=>theta,type=>signal);
  state(name=>theta,derivative=>theta_dot);
  ```

  eliminates the need to remember that the second element plays a very different role in these two statement types. On the other hand, the language is not cryptic, either – for example, the event subsection delimiter `positive-to-negative` could be `p2n`, and so on. Construction of more succinct or verbose variants of SHSML would be a straightforward task.

`interface` Section: Permitted variable categories are `input, output, knob` and `view`. As mentioned before, the first two types are primary input/output (i/o) ports, to which connections may be made, while `knob` and `view` statements designate secondary variables that can be accessed from outside the component – `knob`s are parameters (constants that can be varied during a set of simulation experiments), and `view`s are variables that the modeler might want to display after simulation. CTC `input` variables may be of type `signal, real, symbolic` or `boolean`, corresponding to the input list in Eqns. (1, 2); `output` variables are of type `signal`, by definition. The distinction between first-class inputs and outputs (which can be connected to other first-class i/o variables) and second-class ones (knob and view elements) is important when analyzing the topology of the system and performing operations such as linearization where unwanted inputs and outputs may be bothersome. Note that the modeler may declare a component `input` variable to be a `knob` at a higher level, and similarly for `output` and `view` variables (e.g., output `volt_lim` in `turret` is made a `view` variable in Illustration 2).

`declarations` Section: Key internal variables are categorized as `state, local` and `flag`. The `state` variables in a CTC are those dynamic states which will be evolved by numerical integration of the state equations (Eqns. 1, 2) or differential-algebraic equations; the states and their derivatives are named in the `state` statement, and the derivatives (and perhaps constraint equations) are evaluated in the `dynamics` section (and perhaps `constraint` section, see below). The `local` variables are arbitrary "other" signals which might need to be constrained (see example). The `flag` variables are state-event signatures, i.e., functions whose zero-crossings correspond to state events such as two mechanical parts engaging and disengaging.

Ranges: Variable ranges may be specified for `input` and `local` variables, as shown – e.g., `volts` must be in the range (`-vl`, `vl`); this feature is needed to prevent a component model from being driven outside its known domain of validity. The simulator would have to support checking such restrictions and either warning the user or stopping the simulation if they are violated.

Precision: Precision specification *could* be supported for representing the continuous-time signals in a CTC – however, many simulators do not provide this capability, so it is not clear there would be sufficient benefit to be worth the extra detail. Single-precision arithmetic is usually adequate, as automatic step-size-control algorithms generally reduce or eliminate the need for higher-precision arithmetic.

Units: Support for specifying units of key physical variables *might* be provided. This would prevent errors like connecting an output torque in foot-pounds to an input torque in newton-meters or worse yet to a voltage. However, checking that units are correct and consistent is a difficult task (is it appropriate to connect 0.3048*l(feet) times 4.4482*F(pounds) to a torque in Nm?) and a poorly implemented facility would be very frustrating to use.

`initial` Section: Initial condition calculations and other "set-up" evaluations are performed here. In this example we have specified that the turret angle is initially offset from zero by amount `del_theta`, and that the variable `sgn` takes the sign of `theta_dot` if that state is not zero, otherwise it takes the sign of `moment` ($\ddot{\theta}$).

`event` Sections: These provide for state-event handling (e.g., change of sign in the Coulomb friction term when `theta_dot` passes through zero); the occurrence of a state event is indicated by a zero-crossing in the variable contained in the `flag` statement. Separate subsections are provided to account for the negative-to-positive and positive-to-negative transitions. There would be a separate `event` section for each state event if there is more than one. A more detailed discussion of this feature is provided below.

`dynamics` Section: Differential equations are simply and naturally rendered. Here we have a "chain" of integrators, i.e., the derivative of state 1 is state 2; then the derivative of state 2 is `moment` – in mathematical terms, $\ddot{\theta} = M$ where $M =$ `moment` denotes the total moment acting on the turret. Therefore, `moment` is the only derivative evaluated in the `dynamics` section of this component.

`constraint` Section: We believe that support for an important class of differential-algebraic equations (DAEs), i.e., systems modeled by $\dot{x} = f(x, u, t)$ subject to $0 = h(x, u, t)$, could be incorporated by adding a `constraint . . . end constraint;` section to the formal definition of a CTC. The code in a `constraint` section could be similar to that in the `dynamics` section except each constraint would have to be stylized (e.g., `0.0 = <arbitrary expression>`), to distinguish it from other regular function evaluations. It is not clear whether other limitations or mechanisms would be required for this approach to handle all problems in this class – this extension is still tentative. Of course, the simulator would have to support nonlinear equation solving in coordination with numerical integration; DASSL [12] is the best-reputed solver for this class of DAEs. For a more general and rigorous discussion of DAE solvers, see [13].

Encapsulation: Each component is **rigorously encapsulated**, i.e., its internal variables and parameters cannot be confused with or influenced by those of other components. This is critical if components are to be interchangeable, reusable and arbitrarily interconnectable without having to worry about "side effects". We observe, however, that there are circumstances when it is tempting to permit circumscribed abuses to complete encapsulation. For example, if there were another CTC `tank_body` with azimuth angle `theta` one might wish to initialize `turret`'s state `theta` in terms of an offset from `tank_body`'s azimuth :

```
    theta = tank_body.theta + del_theta;
```

where a "global" designation `<component>.<element>` is used for the external variable. This would eliminate the need to create an artificial input variable to "connect" this external signal. Other cases are also considered in [11]; at this point this question is still open.

State Events: This CTC model contains a simple 'state event', i.e., the change in sign of the friction term when the angular velocity `theta_dot` passes through zero. Substantial errors in simulation may occur if such an event is not "captured" correctly [1, 3]; the purpose of the `event` section of the model above is to handle the state event rigorously [1]. State events can, of course, be more complicated than this example. In general, a state event can be formulated as an arbitrary zero-crossing condition:

$$S(x_c, u_c, u_k, m_j, b_i, t) = 0 \qquad (6)$$

where S is a general expression involving the state and perhaps input variables and time. The result of the state event in model `turret` is also extremely simple: change the sign of the friction term in the moment equation. This too can be more general in the dynamics section – for example, the switching variable `sgn` may be used in arbitrarily complicated expressions of the form `if sgn > 0.0 then do ... else do ... endif;`. Finally, the above example assumes that the dynamics are such that a trajectory can always cross the boundary – in conflict situations this may not be so, and additional machinery is needed to model motion along a boundary submanifold in such cases, as described in [11]. This example, however, is complete enough to convey the ideas and issues without getting involved in a lot of unnecessary complexity.

3.3 Illustration 2 - a Simple CC:

The component `turret` may be connected to a digital controller `dctrl` and a digital signal processor `az_filter` to form a composite component (CC), as follows:

```
CC turret_azimuth_control is
%
% composite component connection
%
interface    %  declare CC interface variables:
  input(theta_com,real); % controller command input
  input(disturbance,signal); % external disturbance
  input(noise,signal);       % sensor noise source
  knob(k_sat);               % two parameters to
  knob(k_noise);             % ... experiment with
  output(azimuth) = output(turret.theta);
  output(az_meas) = input(az_filter.theta);
  output(az_filt) = output(az_filter.theta_trk);
  % make turret.volt_lim a 'view' variable:
  view(volt_lim) = output(turret.volt_lim);
end interface;
body
  connections  %  define the component connections:
    input(dctrl.ref) = theta_com;
    input(dctrl.fdbk) = output(az_filter.theta_trk);
    input(turret.volts) = output(dctrl.command);
    input(turret.load_dist) = disturbance;
    % make turret.Ksat a 'knob' variable:
    input(turret.Ksat) = k_sat;
    input(az_filter.theta) = output(turret.theta)
                             + Knoise*noise;
  end connections;
  assignments
    k_noise: 1.0     %  default noise gain
  end assignments;
end body;
end turret_azimuth_control;
```

A graphic representation corresponding to this textual specification is also presented in [11]. Briefly, we note that:

- CC inputs and outputs are defined as in the pure component case. Each CC output variable inherits the type of the associated component variable; each input variable in the CC description must be typed in accordance with the typing in the component where it is consumed.
- A global variable "dot notation" is used to create unique identifiers for system variables. For example,

any number of components may have a variable `theta`; `turret.theta` is unique to component `turret`.

CC inputs: are *named* in the `interface` section; these names must be used to specify component inputs within the system. In this example, `disturbance` is a disturbance torque from an external module that acts on `turret`, `noise` corrupts the signal processed by the azimuth filter, etc. Inputs may be instantiated by connecting this CC to a "driver" (signal generator) or by using this module as a composite component of a larger system. Either way, such a connection defines the specific nature of the inputs, e.g. how `disturbance` varies with time, etc. Consistency between an input source and component constraints would be checked as part of validating the model when it is assembled.

CC outputs: are named in the `interface` section and must be (in the same statement) connected to an appropriate component input or output. In this example, `azimuth` is the azimuth angle of the current target being tracked, `az_meas` is the input to `az_filter`, etc. Note that we allow a component input to be a CC output only because we permit operations in the connection definition (see following point) – this would not be necessary or desirable if operations were not permitted in that context.

This example shows two simple operations beyond connection, i.e., multiplication by a gain factor `Knoise` and then addition to inject the signal into the track filter input `az_filter.theta`. It is an open question whether to allow more generality – e.g. to permit multiplication or division of variables or other nonlinear operations – or conversely to forbid all operations (in which case one must move the gain and addition into the `az_filter` component).

The validity of each component connection is checked and enforced by the interpreter / compiler before a system is deemed acceptable for use (simulation or analysis).

4. Conclusion

The lack of precise tools for modeling and evaluating hybrid control systems is a major impediment to the efficient and cost-effective development of embedded real-time software for a wide variety of applications. The development of a standard hybrid systems modeling language (SHSML) and corresponding rigorous simulation environments (e.g., Hybrid DsTool) represent important contributions to eliminating this barrier. In addition, SHSML has been devised to provide a direct interface with software engineering tools for implementing such systems. The result should be a substantial reduction in the design-cycle time and life-cycle cost of hybrid control systems.

Acknowledgements: The author would like to thank colleagues at ORA (Jin Lu, Matt Stillerman) and in the DSSA [2] Community (Bob Balzer, USC/Information Sciences Institute; Frank Belz, TRW; Christine Braun, GTE Contel Federal Systems; Lee Erman, Teknowledge Federal Systems; Marc H. Graham, CMU Software Engineering Institute; David Luckham, Stanford University; Erik Mettala, ARPA SISTO (retired); Will Tracz, IBM Federal Systems Company; Steve Vestal, Honeywell System Research Center; and Dave Wile, USC/Information Sciences Institute) for many helpful discussions and suggestions.

5. References

1. Kohn, W. and Nerode, A., "Foundations of Hybrid Systems", to appear in *Logical Methods: A Symposium in Honor of Anil Nerode's 60th Birthday*, Birkhauser, 1993.
2. Mettala, E. and Graham, M. H., Eds., *The Domain-Specific Software Architecture Program*, Special Report CMU/SEI-92-SR-9, Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA 15213, June 1992.
3. Guckenheimer, J. and Nerode, A., "Simulation for Hybrid Systems and Nonlinear Control", *Proc. IEEE Conference on Decision and Control*, Tucson, AZ, December 16-18, 1992.
4. Cellier, F. E., Elmqvist, H., Otter, M., and Taylor, J. H., "Guidelines for Modeling and Simulation of Hybrid Systems", *Proc. 12th IFAC World Congress*, Sydney, Australia, July 18-23, 1993.
5. Elmqvist, H., "SIMNON - An Interactive Simulation Program for Non-Linear Systems", in *Proc. of Simulation '77*, Montreux, France, 1977.
6. Augustin, D. C., Strauss, J. C., Fineberg, M. S., Johnson, B. B., Linebarger, R. N., and Sansom, F. J., "The SCi Continuous System Simulation Language (CSSL)", *Simulation*, Vol. 9, No. 6, December 1967.
7. *Advanced Continuous Simulation Language (ACSL) Reference Manual.* Mitchell & Gauthier Associates, Concord MA 01742.
8. Taylor, J. H., Frederick, D. K. Rimvall, C. M. and Sutherland, H. A., "The GE MEAD Computer-Aided Control Engineering Environment", *Proc. IEEE Symposium on CACSD,* Tampa, FL, December 16, 1989.
9. Cellier, F. E., "Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools", PhD Thesis ETH 6438, Swiss Federal Institute of Technology, Zurich, Switzerland, 1979.
10. Pantelides, C. C., "The Consistent Initialization of Differential-Algebraic Systems," *SIAM J. on Scientific Computing*, Vol. 9, No. 2, pp. 213–231, 1988.
11. Taylor, J. H., *A Proposed Modeling Language Standard for Hybrid Dynamical Systems*, Odyssey Research Associates Technical Report, April 1993.
12. Brenan, K. E., Campbell, S. L. and Petzold, L. R., *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*, North Holland, 1989.
13. Mattsson, S. E. and Söderlind, G., "A New Technique for Solving High-Index Differential-Algebraic Equations Using Dummy Derivatives", *Proc. CACSD'92, IEEE Computer-Aided Control Systems Design Conference,* Napa, Calif., pp. 218–224, March 17–19, 1992.