A RIGOROUS MODELING AND SIMULATION PACKAGE FOR HYBRID SYSTEMS

NATIONAL SCIENCE FOUNDATION
PHASE I FINAL REPORT

Prepared for:

Ritchie B. Coryell, Program Official
Division of Design, Manufacture and Industrial Innovation
Small Business Innovation Research Program

Award Date: 12/27/93

Effective Date: 01/01/94 End of Performance: 06/30/94

Prepared by:

Dr. James H. Taylor, PI
Formerly at: Odyssey Research Associates, Inc.
Ithaca, NY 14850-1326

Now at: Department of Electrical Engineering
University of New Brunswick
Fredericton, NB CANADA E3B 5A3
Tel: (506) 453-5101 FAX: (506) 453-3589

Award No. III-9361232

This report is based upon work supported by the National Science Foundation under award number III-9361232. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation. Note that this is not the official SBIR Final Report; it has been extended in both concept and length in substantial ways.

NATIONAL SCIENCE FOUNDATION
PHASE I FINAL REPORT

Program Official: Ritchie B. Coryell, DMII
Division of Design, Manufacture and Industrial Innovation

Program Name: SBIR/Division of Design, Manufacture and Industrial Innovation
Award Date: 12/27/93

Effective Date: 01/01/94 End of Performance: 06/30/94

Dr. James H. Taylor, PI
Odyssey Research Associates, Inc.
301 Harris B. Dates Drive
Ithaca, NY 14850-1326
Tel: (607) 277-2020 FAX: (607) 277-3206

Award No. III-9361232

A RIGOROUS MODELING AND SIMULATION PACKAGE FOR HYBRID SYSTEMS

a.) Technical Summary: Existing commercial modeling and simulation environments do not adequately predict the behavior of complex computer-controlled systems being developed and fielded today. These “hybrid” systems involve discontinuous phenomena such as mechanical parts engaging and disengaging (e.g., spacecraft docking), control units switching, and software components reconfiguring. In many circumstances this deficiency presents a major barrier to showing that a system will perform as required, or conversely that a newly designed system may have safety-critical defects.

In Phase I we investigated (i) an appropriate *class* of hybrid systems to support, (ii) *integration methods* that will simulate discontinuous nonlinear hybrid systems with complete accuracy, and (iii) a *modeling language* for representing such phenomena. The integration algorithms will rigorously catch and handle all discontinuous events. The language is based on modern standards, to facilitate and enforce correct modeling practices.

b.) Commercial Potential: Such a package for modeling and simulation will be of great value for design and validation of: Flexible manufacturing systems, robots, intelligent vehicles and highways, spacecraft maneuvers and rendezvous, flight dynamics and control, chemical and materials processing, and other hybrid systems.

Phase II Intentions: ORA does not intend to submit a Phase II Proposal.

This material is based upon work supported by the National Science Foundation under award number III-9361232. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Preface

The methods and approaches described in the following report are the product an intensive six-month study of requirements and the state of the art in hybrid systems modeling and simulation undertaken in 1994. Since then the author has returned to an academic environment, and shifted emphasis and direction in this area of research. Primarily, these changes have involved:

1. adopting MATLAB as the platform of choice for implementation of a hybrid systems modeling and simulation environment, and building on their most powerful algorithms for integrating ordinary differential equations and other needed functionality;

2. focussing on rigorous handling of discontinuous effects, and particularly those involving multivalued nonlinearities, such as hysteresis; and

3. defining a MATLAB modeling schema for the interconnection of continuous- and discrete-time components to create hybrid systems.

The next phase of development would be to develop a "MATLAB compiler" to accept models in a language similar to (but probably more modern than) the hybrid system modeling language construct defined herein.

The research and development outlined above is described, at a high level, in the following:

1. J. H. Taylor, "Rigorous Hybrid Systems Simulation with Continuous-time Discontinuities and Discrete-time Agents", *Proc. 3rd IMACS/IEEE International Multiconference on Circuits, Systems, Communications and Computers,* Athens, Greece, July 1999.
2. J. H. Taylor and D. Kebede, "Rigorous Hybrid Systems Simulation of an Electromechanical Pointing System with Discrete-time Control", *Proc. American Control Conference,* Albuquerque, NM, pp. 2786-2789, June 1997.
3. J. H. Taylor and D. Kebede, "Modeling and Simulation of Hybrid Systems in MATLAB", *Preprints of the IFAC World Congress, Vol. J*, San Francisco, CA, pp. 275-280, July 1996.
4. J. H. Taylor and D. Kebede, "Modeling and Simulation of Hybrid Systems", *Proc. IEEE Conference on Decision and Control,* New Orleans, LA, pp. 2685-2687, December 1995.
5. J. H. Taylor, "Rigorous Handling of State Events in MATLAB", *Proc. IEEE Conference on Control Applications,* Albany, NY, pp. 156-161, September 1995.
6. J. H. Taylor, "A Modeling Language for Hybrid Systems", *Proc. Joint Symposium of Computer-Aided Control System Design,* Tucson, AZ, pp. 337-344, March 1994.

The above publications represent the thread of R & D starting from the method and approach described in this report, to the most recent phase in 1998-99. Many of these articles, and software for rigorous handling of discontinuous / multivalued effects, are available on the author's web site, at `http://www.ee.unb.ca/jtaylor/`.

James H. Taylor — 16 May 2004
Department of Electrical & Computer Engineering
University of New Brunswick
PO Box 4400
Fredericton, NB CANADA E3B 5A3
e-mail: jtaylor@unb.ca

Contents

1 Executive Summary **1**

1.1 Phase I Research Objectives 1

1.2 Suggested Phase I Research Changes 2

1.3 Phase I Research Overview 2

1.3.1 The Domain 2

1.3.2 The Problem 3

1.4 Research Findings and Results 4

1.4.1 Defining the Class of Hybrid Systems 4

1.4.2 Hybrid SystemModeling Language Specification 4

1.4.3 Methods and Algorithms for Hybrid System Modeling and Simulation . 4

1.4.4 Preliminary Software Development Plan 4

1.5 Potential Applications of the Research 4

2 Hybrid Systems: The HSML Modeling Domain **6**

2.1 Domain Overview 6

2.2 Pure Components – Mathematical Formulation 7

2.2.1 Pure Continuous-Time Components (CTCs) 7

2.2.2 Pure Discrete-Time Components (DTCs) 9

2.2.3 Pure Logic-Based Components (LBCs) 10

2.3 Hybrid System Building 11

2.3.1 Building a Composite Component 11

2.3.2 Building Hybrid Systems 12

3 Simulation of Hybrid Systems **14**

3.1 Protocol for Simulating Hybrid Systems 14

3.2 Continuous-Time State Evolution - Overview 15

3.3 Evolution of Continuous Class $\mathcal{D}^0$ Models 16

3.3.1 Starting the Evolution of Class $\mathcal{D}^0$ Models 16

3.3.2 Equation Sorting for Class $\mathcal{D}^0$ Models 16

3.4 Evolution of Continuous Class $\mathcal{D}^1$ Models 18

3.5 Discontinuity (State-Event) Handling 18

3.5.1 Handling Predictable Discontinuities 18

3.5.2 Handling Switch-Type Nonlinearities 19
3.5.3 Handling State-Changing Nonlinearities 21
3.5.4 Handling Structure-Changing Nonlinearities 21

4 Preliminary HSML Language Definition 25
4.1 Describing Components in General 25
4.2 Describing a Continuous-Time Component (CTC) 26
4.3 Describing a Discrete-Time Component (DTC) 29
4.4 Describing a Logic-Based Component (LBC) 31
4.5 Describing a Composite Component 32
4.6 Describing a Hybrid System 33
4.7 Component Encapsulation 34

5 HSML Modeling Example 36
5.1 A Composite Component Connection Definition 36
5.2 Pure Component Model Definitions 38
5.2.1 `turret` – A Continuous-Time Component 39
5.2.2 `pid_control` – A Discrete-Time Component 42
5.2.3 `engagement_mgr` – A Logic-Based Component 44
5.3 Completing a Hybrid System Model 45

6 Conclusions 48

7 References 49

A Rigorous Modeling and Simulation Package for Hybrid Systems

James H. Taylor
Department of Electrical Engineering
University of New Brunswick
PO Box 4400
Fredericton, NB CANADA E3B 5A3

1 Executive Summary

The background for the Phase I effort is provided by restating the objectives in the Phase I Proposal and outlining certain changes in research agenda suggested by the Proposal reviewers. Next an overview of the Phase I Research is presented, comprising an outline of the domain and the problem; then the key finding and results are summarized. This Executive Summary is concluded by an outline of potential applications.

1.1 Phase I Research Objectives

The following is a verbatim restatement of the Phase I Proposal Research Objectives:

> Preliminary work has established a set of requirements for a quality hybrid system simulation environment (QHSSE) and shown that there is a strong need for one. An outline for such a software package has also been generated and presented here. At this stage of development, we need to continue research in the following directions:
>
> 1. Complete the definition of the precise class of hybrid systems for which it is feasible to create a QHSSE (see Section 2); in particular, decide on what classes of differential-algebraic equations and LBCs can be supported.
> 2. Complete the definition of a semantics and syntax of the most general possible hybrid system modeling language (see Section 4).
> 3. Determine which existing routines are best suited for use as the numerical integration algorithms of a QHSSE,and which logic is best suited for state-event handling (see Section 3.5).
> 4. Create a preliminary Software Development Plan for producing a QHSSE.
> 5. Determine the technical and commercial viability of (i) joining with an existing commercial simulation package vendor to upgrade their environment, or (ii) create a QHSSE from public-domain subroutine packages.
> 6. Develop a plan for Phase II research and development.
>
> There are many technical details to be investigated and resolved. Nonetheless, we believe that the technical feasibility of producing a software plan for a QHSSE for a broad class of hybrid systems is established, by virtue of the detailed plan

for handling at least the class outlined in Section 2. Highly precise methods and algorithms exist for the numerical solution of that class and broader ones, e.g., certain types of differential-algebraic equations (Section 2), so there is no doubt that existing commercial simulation packages could be extended or a new hybrid system simulation environment could be created that exploit existing resources to achieve the objectives of accuracy and rigor.

There are many applications and potential users of commercial simulation packages (Section 1.3.1). Demand for a QHSSE will be driven by the increasing complexity of hybrid systems and growing awareness that rigorous modeling and simulation approaches are needed to provide definitive answers to questions like the existence of oscillations and chaos (Section 1.3.2). Thus, the commercial viability of this project is also assured. However, many questions remain to be answered regarding the nature of such a commercial venture; we plan to resolve them in Phase I.

1.2 Suggested Phase I Research Changes

Reviewers of our Phase I Proposal indicated that issues related to numerical algorithms for simulation of hybrid systems were not adequately addressed. We agree that, although sound codes exist for integration of continuous-time dynamics including state-event handling and interplay with discrete-time components, higher-level methods and algorithms for modeling and simulation of hybrid systems along the lines envisioned in our Proposal do not exist. Emphasis on this aspect was substantially increased from our original plan.

The stronger focus on numerical methods and algorithms resulted in a decrease in effort in the area of business planning (objective 5 above). We believe this change in emphasis has been highly beneficial in developing a firm foundation for hybrid system modeling and simulation.

1.3 Phase I Research Overview

1.3.1 The Domain

Modeling and simulation of nonlinear, computer-controlled dynamical systems is a well established technical discipline. The corresponding activity is widespread throughout the scientific and engineering fields. As an illustration of the ubiquitous nature of this endeavor, computer-controlled dynamical systems are all about us – in everyday terms, they include modern automobiles, appliances, automatic teller machines etc.; in higher technology areas they encompass flexible manufacturing systems, robots, intelligent vehicles and highway systems, spacecraft, flight dynamics and control, chemical and materials process control, automated drug administration and health monitoring systems, to name only a few more recent applications. Along another dimension, modeling and simulation of a dynamical system may be used to predict behavior of a device that is in the conception and early design stages of development, or to refine the design of a system that has been prototyped or is targeted for redesign (upgrade), or to perform experiments that cannot be executed in the real world (e.g., study failure modes in a flight control system or nuclear reactor control logic).

The last decade has seen immense growth in the complexity and sophistication of computer-controlled dynamical systems being developed and put into service. The physical components (e.g., airframe, robot, chemical process) may not be much more complicated, but the discrete-time components and computer control software have taken on more complex and higher-level functionality, as in the area of automonous vehicles, for example. These developments have resulted in a large surge of interest in "hybrid systems", as the modern class of computer-controlled dynamical systems has come to be called.

The mathematical understanding of the properties and behaviors of computer-controlled dynamical systems has also made major strides in recent years. First *bifurcation analysis* came to prominence [26]; this involves searching for critical parameter values that give rise to qualitative behavioral changes in the system. For example, a parameter α in a model may have a critical value $\alpha^\star$ such that for $\alpha < \alpha^\star$ the system is stable but for $\alpha > \alpha^\star$ limit cycles (nonlinear oscillations) exist. The existence of a bifurcation point near the design value of a parameter may have important implications regarding the safe operating envelope of a computer-controlled dynamical system. Then the existence of *chaotic behavior* was established [37]; again, for example, a nonlinear system may have a critical range of a parameter α for which the motion of the system is so sensitive to initial conditions that it appears to be random (unpredictable) – this is usually highly undesirable. The existence of these phenomena underscores the need for careful modeling and simulation of hybrid systems before they are built and fielded.

1.3.2 The Problem

Much of the work in modeling and simulation is being done today with outmoded and possibly dangerous tools. The modeling languages and numerical integration algorithms that are incorporated in most commercial simulation packages are the outgrowth of theoretical developments that were undertaken decades ago to handle continuous-time and continuous-state systems, i.e, systems that obey ordinary differential equations where the derivatives of the variables are continuous functions of time and state. To introduce a specific notation, such systems are represented by:

$$\begin{aligned} \dot{x} &= f(x,u,t)\,,\ f \in \mathcal{C}^m \\ y &= g(x,u,t) \end{aligned}$$

where x is the continuous-time *state vector*, y is the *output vector*, u denotes a continuous-time *input signal*, and $\mathcal{C}^m$ denotes the class of functions with up to m continuous derivatives. As an example of the limitations of commonplace integration algorithms, all predictor-corrector methods for numerically integrating such system models assume $m = 2$ or more in their derivation and error analysis. Use of methods and software designed for this class of dynamical systems is clearly inappropriate for hybrid systems.

Over the last few decades these approaches and algorithms have been extended, often in an *ad hoc* fashion, to handle discontinuities and discrete-time components (e.g., computer algorithms embedded in a micro-processor). Added features permit interrupting the numerical integration of continuous-time dynamics for the processing of discrete-time events and other predictable discontinuities, but this does not answer the need for handling *unpredictable* state

events or digital operations with rigor. For example, integration algorithms with automatic step-size control do come closer to producing correct simulations of discontinuous behavior in continuous-time dynamics – however, the point of discontinuity is not captured accurately (making the prediction of chaos or limit cycles problematic), and there are computational problems (step-size becoming very small, resulting in an excessively slow "creeping simulation"). More recently, extensions to integration algorithms have been produced by experts in the field that handle broader classes of hybrid systems with high accuracy; however, these advances have not been reflected in extensions to modeling languages or in commercial simulation packages.

All of the trends outlined above establish a need for **rigorous hybrid system modeling and simulation environments**. Unfortunately, aside from research codes which are unsuitable for commercial use, these do not exist at the present time.

1.4 Research Findings and Results

Research accomplishments have been achieved in the following areas:

- Defining the class of hybrid systems that can be supported in a general-purpose modeling and simulation environment,
- Completing the semantics and nearly settling on the syntax of the corresponding hybrid system modeling language,
- Determining the methods and algorithms needed for modeling and simulation of hybrid systems, and deciding on utilization of existing codes and creation of new algorithms, and
- Developing a preliminary software development plan for QHSSE.

These areas are discussed in the following subsections **which need to be written!**

1.4.1 Defining the Class of Hybrid Systems

1.4.2 Hybrid SystemModeling Language Specification

1.4.3 Methods and Algorithms for Hybrid System Modeling and Simulation

1.4.4 Preliminary Software Development Plan

1.5 Potential Applications of the Research

The research performed under this grant will have wide-spread applicability in every domain involving modeling and simulation of computer-controlled dynamical systems. The following incomplete list serves as an illustrative catalog of this broad scope:

- Aerospace systems: flight control, propulsion control, integrated systems control, air traffic control, satellite guidance and control, deep-space mission control, landers and autonomous vehicles, remote sensing systems, ...

- Land vehicle systems: mass transit, people-moving systems, high-speed rail systems, intelligent vehicles and highways, ...
- Maritime systems: shipping vessels and fleets, fishing, military surface and submarine fleets, underwater autonomous vehicles for exploration and recovery, ...
- Electric power systems: fossil-fuel and nuclear power plants, wind and solar energy systems, power grids, ...
- Manufacturing systems: robotics, parts handling equipment, assembly systems, production control systems, numerically-controlled machine tools, ...
- Process control systems: chemical processes, materials processing, metallurgical, mining and mineral processing, ...
- Biotechnology: bioremediation systems, processing of bio-engineered materials, ...
- Biomedical systems: intensive care systems, prosthetics, automatic feedback control of drug administration, ...
- Information systems: CAD, CAM, CIM, CIME, CAE systems; integration of simulation and artificial intelligence, object oriented analysis and design, visualization technology, ...
- Virtual reality systems: multimedia, hypermedia, education and training, entertainment and gaming, ...

In addition, quality simulators of hybrid systems may be used for preliminary design, final design and verification, system maintenance and upgrade, documentation, education, training, and entertainment.

2 Hybrid Systems: The HSML Modeling Domain

This section summarizes our research in defining an appropriate *class of hybrid systems* to support in a new modeling and simulation environment. The resulting class is substantially broader than that handled by existing environments, and eliminates several long-standing problems.

2.1 Domain Overview

A precise definition of the term "hybrid system" must be provided before one sets out to define and implement a modeling and simulation environment for such systems. In basic terms, we assume that hybrid systems are composed by interconnecting **continuous-time components** (CTCs), **discrete-time components** (DTCs), and **logic-based components** (LBCs) in some arbitrary configuration. At the lowest level, every system "block" represents a component of one of these three types. At higher levels, these "pure" components may be aggregated to create a fourth category, the **composite component** (CC); CCs, in turn, serve as building blocks for still larger subsystems and systems. Thus, this hybrid systems representation approach supports hierarchical model building with complete generality.

A fifth "block" type, the **system driver component** (SDC), is introduced purely for the purposes of analysis and simulation. In almost every case, the behavior of a hybrid system model is studied by providing input signals to "drive" the system in various ways over a number of different scenarios. The addition of the SDC supports this functionality; in essence, the SDC provides all required inputs for one or more CCs and thus provides a closed system that is ready to simulate. As a technical matter, then, the definition of a hybrid system is not considered to be complete until there are no undefined inputs.

Note that the following "building-block model" of hybrid systems represents a limiting assumption. In particular, a component is assumed to have "inputs" which, when excited, cause a block to "respond" to produce "outputs" that vary in some fashion. This amounts to a causality assumption (wiggling an input causes a wiggle in the output) which does not lead to any loss in generality, but which in some contexts may be inconvenient; for example Dymola [19] and Omola [29] allow one to have a noncausal model of a resistor that may be used in a system model with either the causality $v = Ri$ or $i = v/R$, i.e., there is no need to specify which variable (v or i) is the input. At a higher level, a noncausal model of a rotating machine could be used either as a generator (applied torque produces voltage) or as a motor (applied voltage produces torque). This approach is called "object-oriented modeling" by its developers; in certain contexts (e.g., circuits and domains that can be treated by circuit-theoretic approaches such as bond graphs [28]) it is of obvious utility. In many domains object-oriented modeling is not needed, practical, or even possible, however, so we are using a block-oriented modeling paradigm to achieve more generality in other aspects, i.e., it permits the modeling and simulation of more general classes of hybrid systems given the state of the art in object-oriented modeling.

The remainder of this chapter deals with the "pure" components itemized above and their interconnection to create hybrid systems. In each case, we specify the most general class of component that can be supported, along with a discussion of limitations and technical rationale.

2.2 Pure Components – Mathematical Formulation

Each "pure" block of a hybrid system has inputs, outputs, and internal variables of correspondingly suitable form. Their distinguishing features involve the class of internal variables and the method of describing their evolution with time.

2.2.1 Pure Continuous-Time Components (CTCs)

Each CTC may have numeric, Boolean, and/or symbolic inputs; continuous-time numeric outputs; and continuous-time numeric internal variables called "states". The generic form of such a block is given by a set of differential-algebraic equations (DAEs) and output equations expressed as:

$$0 = F_c(x_c, \dot{x}_c, u_c, u_k, b_i, m_j, t) \quad (1)$$

$$y_c(t) = H_c(x_c, \dot{x}_c, u_c, u_k, b_i, m_j, t) \quad (2)$$

where x_c is the state vector, y_c is the output vector, u_c and u_k are the numeric input signals (continuous- and discrete-time, respectively), b_i is a Boolean input, 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. Note that 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. We also observe that the Jacobian $F_{\dot{x}_c} \triangleq \partial F_c / \partial \dot{x}_c$ is usually identically singular or the system in (1) may be treated as an ordinary differential equation set [6].

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 (with implicit digital-to-analog conversion), b_i might govern whether or not an actuator has failed, and m_j might define a higher-level condition such as 'engine-has-stalled'.

With respect to conditions that guarantee existence and uniqueness of solutions, the usual mathematical assumptions of smoothness, Lipschitz conditions, etc., are not imposed; in fact, the states themselves may change discontinuously (e.g., when two mechanical parts engage and their velocities change instantaneously to conserve momentum). In terms of well-posedness, it is up to the modeler to ensure that meaningful solutions exist, and up to the simulation environment to ensure that discontinuities are handled correctly. Also, we observe that the inputs b_i and m_j are included here primarily to allow for external means of controlling the model, and not as an actual input to the continuous-time dynamics themselves.

The generic form in (1) is often called a *fully implicit* DAE [6]; without imposing additional conditions, it generally cannot be solved by any existing numerical code. In fact, determining if such a model is solvable [6, 7, 10, 11, 36, 38] and arriving at consistent initial conditions [6, 12, 33, 35] is a complicated matter. To achieve a practical definition of the class of CTCs to be treated, we need to specialize the form in (1) in some manner:

- Most simply, we may replace the DAE form in Eqns. (1,2) with the following ordinary differential equation set:

$$\dot{x}_c(t) = f_c(x_c, u_c, u_k, b_i, m_j, t) \quad (3)$$

$$y_c(t) = h_c(x_c, u_c, u_k, b_i, m_j, t) \quad (4)$$

Models of this form are called **ordinary differential equation (ODE) components** and for simplicity designated $\mathcal{D}^0$ models; these have been the focus of most commercial modeling and simulation environments up to the current time [1, 18, 32, 40, 42, 50].

- Next most simply, we may replace the form in (1) with the following *constrained* ordinary differential equation set:

$$\dot{x}_c(t) = f_c(x_c, z_c, u_c, u_k, b_i, m_j, t) \quad (5)$$

$$0 = g_c(x_c, z_c, u_c, u_k, b_i, m_j, t) \quad (6)$$

$$y_c(t) = h_c(x_c, z_c, u_c, u_k, b_i, m_j, t) \quad (7)$$

where constraint variables z_c have been added along with constraint equations (6). Models of this form are called **semi-explicit DAE components**. For technical reasons (e.g., solvability [6, 11] and initializability [6, 33]) we impose the condition that the Jacobian of the constraint equations g_c with respect to the constraint variables z_c must be nonsingular:

$$|G_z| \triangleq \left|\frac{\partial g_c}{\partial z_c}\right| \neq 0 \quad (8)$$

Again, for simplicity we denote such models by $\mathcal{D}^1$; while these cannot be solved by most commercial modeling and simulation environments, there are codes such as DASSL and DASSLRT [34, 6] and COLDAE [3] that can supply the underlying numerical machinery. Semi-explicit DAEs that satisfy Eqn. (8) are said to be of *index 1* [6]; by differentiating the constraint equation once one may reduce an index 1 semi-explicit DAE to an ODE (i.e., solve for $\dot{z}_c$).

The question of generality versus availability of theory and algorithms is an important one, given the overall goal of this research. While ODE components are the staple of much modeling and simulation work today, there are many instances where a purely ODE approach is inadequate or frustratingly cumbersome. A well motivated discussion of this issue may be found in [6]; here we only mention that models of the semi-explicit DAE form arise in treating variational problems such as nonlinear optimal control, in eliminating parasitic effects from models via singular perturbations, and in solving systems of partial differential equations via discretization approaches such as the method of lines. In specific physical domains, DAEs arise naturally in the dynamics of mechanical systems such as robots and in realistic models of jet engines which include ODEs for slower states and "instantaneous" constraints based on energy and mass balance considerations that would be prohibitively difficult to model and simulate as ODEs. Thus the ability to handle semi-explicit DAEs represents a significant broadening over ODEs that is of substantial practical utility.

As a final comment regarding the class of CTC evolution equations that can be supported, we observe that there is substantial research currently being focussed on solving higher-index DAEs [8, 9, 12, 13, 24, 30, 31]. Much of this work is based on differential geometry and/or graph-theoretic approaches that lead to index-reduction methods which can be implemented using symbolic manipulation. For an overview, refer to [16]. We take the position that a preprocessor may be used to reduce a higher-index DAE into a model of form $\mathcal{D}^0$ or $\mathcal{D}^1$ and thus do not consider them further. Note, however, that it is often *not* a good practice to reduce a model of form $\mathcal{D}^1$ into class $\mathcal{D}^0$ [16], since the result may be numerically ill-conditioned (e.g., stiff) and/or the constraints may "drift" if not transformed properly.

The other area of generality to be addressed in this research concerns the existence of discontinuity in CTC component models and corresponding theory and algorithms for discontinuity handling. Here, too, there are a number of cases to consider:

1. Predictable discontinuities: In the simplest case, the CTC model may be continuous with respect to the continuous-time variables, but there are predictable discrete-time events that instantaneously change the right-hand sides of Eqns. (3) or (5). The only cases in which this behavior may be precluded involve CTCs that have only continuous-time input signals. We denote this case $\mathcal{N}^0$, i.e., $f_c \in \mathcal{N}^0$.

2. Switching nonlinearities: There may exist simple discontinuities with respect to the continuous-time variables, e.g., the CTC model may have switches or relays that abruptly change their output value in response to a smooth change in input through a threshhold value (e.g., a zero crossing). We denote this case $\mathcal{N}^1$.

3. State-changing nonlinearities: More complex situations exist when the dimension of the state space changes. A well known example is backlash; the modes here are "engaged_moving_clockwise", "disengaged" and "engaged_moving_counter-clockwise", and the number of states changes by two when the element changes mode. We denote this case $\mathcal{N}^2$.

4. Structure-changing nonlinearities: The most extreme case we consider involves discontinuities where the *nature* of the state space changes. Such a situation might involve a physical effect that causes motion in the usual spatial coordinates (x, y, z) to become constrained to some sub-manifold that is not a simple subspace (e.g., not a plane) and where the equations of motion are completely different. We denote this case $\mathcal{N}^3$.

We consider features in our proposed modeling and simulation environment that handle all these cases, with complete rigor.

2.2.2 Pure Discrete-Time Components (DTCs)

Each DTC may have numeric, Boolean, and/or symbolic inputs; discrete-time numeric outputs; and discrete-time numeric internal variables called "discrete states". The generic form of such a block is given by a set of difference equations expressed as

$$x_{k+1}(t_k) = f_k(x_k, u_c, u_k, b_i, m_j, k) \quad (9)$$

$$y_{k+1}(t_k + \delta_k) = h_k(x_{k+1}, u_c, u_k, b_i, m_j, k) \quad (10)$$

where x_k is the discrete state vector, k is the index corresponding to the discrete time point t_k at which the state takes on the new value x_{k+1}, 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 ($t_k = t_0 + k * T_s$ where T_s is the "sampling period"); in any case, we assume that the update times can be anticipated. There may be a computational delay δ_k between the sample time and the output 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 computations.

Example: A DTC may represent the discrete-time 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 may provide information for modifying the algorithm ('target-is-accelerating' might necessitate switching to a 9-state Kalman filter).

Concern for existence and uniqueness of solutions is unwarrented, since the DTC equations represent a given numerical algorithm rather than a DAE. We exclude component descriptions that include algebraic constraint equations; while in the CTC case these occur naturally (Eqn. 6, Section 2.2.1), in the context of DTCs it is not clear they are of any meaning. For example, if the states x_k in Eqn. (9) are to be constrained in some way, then these constraints should be expressed algorithmically within the framework provided above and not as an abstract "constraint equation" to be solved by the simulator. Note again that the inclusion of Boolean and symbolic inputs b_i and m_j is primarily for model control, as in the CTC case.

2.2.3 Pure Logic-Based Components (LBCs)

Each LBC may have numeric, symbolic, and/or Boolean inputs; Boolean and/or 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

$$m_{j+1} = \Phi_j(m_j, u_c, u_k, b_i, j) \tag{11}$$

$$b_{i+1} = \Psi_i(m_j, u_c, u_k, b_i, i) \tag{12}$$

where in Eqn. (11) m_j is the mode vector, j is the index corresponding to the discrete event triggering the LBC action, Φ_j denotes a completely undefined relationship, and u_c, u_k and b_i are as above. The Boolean variables and relations in Eqn. (12) are similar. Tentatively, the output of each LBC *is* the mode m_j or Boolean b_i, 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 situation in DTCs, we assume that the update times (mode changes) can*not* necessarily be anticipated. An LBC may be a model of a software component, or it may represent a high-level continuous-time artifact incorporated in the hybrid system model to handle complicated state-event situations. If the LBC is a software component, then 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.

Example: An LBC may represent a fuzzy-logic-based system 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 the complete reconfiguration of a hybrid system when a vehicle engine stalls.

Note that the DTC defined previously might be considered to be a special case of an LBC in some sense (e.g., both are realized in software); however, we use DTCs for well-defined and commonplace discrete-time *numerical* algorithms as in Eqns. (9, 10) and LBC for components that are primarily *logical* or *symbolic* in nature. The particular and specific format of the DTC so defined allows a very succinct formulation of such components, as in the case of CTCs, so it is worthwhile treating them separately. This also allows very detailed syntactic and semantic checking of each DTC, an extremely valuable support feature in a modeling

and simulation environment. In addition, certain operations (e.g., linearization) might be performed on a DTC that would make no sense in the context of an LBC.

2.3 Hybrid System Building

Given the taxonomy of components outlined in Section 2.1, a hierarchical hybrid system is defined by first developing models of its pure components and then specifying how they are connected. First we consider building a composite component (CC) using a set of "pure" components; next we outline how CCs can be embedded in a multi-level hybrid system and then interfaced with a signal generator to form a complete hybrid system.

2.3.1 Building a Composite Component

An illustrative composite component is portrayed in Fig. 1. It is composed of five pure components: `turret`, which represents the nonlinear continuous-time dynamics and electric drive system of the turret of a battle tank, `pid_control`, which applies a voltage to the turret electric drive to cause it to slew in a commanded direction, `az_filter`, which filters noisy measurements of the selected target azimuth position, `track_mgr`, which tells the controller which target to follow, and `engagement_mgr`, which processes threat priorities and track data to guide the track_mgr and pid_control modules.

There are five types of "pure" connections in this formalism, denoted as follows:

- Continuous-time signal: $\triangle\!\longrightarrow\!\triangle$ Example: CC input `disturbance` connected to `turret` input `load_dist`.
- Real number, transmitted at discrete time(s): $\bullet\!\longrightarrow\!\bullet$ Example: `track_mgr` output `theta_com` connected to `pid_control` input `ref`.
- Integer, transmitted at discrete time(s): $\square\!\longrightarrow\!\square$ Example: `engagement_mgr` output `which_one` connected to `track_mgr` input `threat_num`.
- Boolean variable, transmitted at discrete time(s): $\oplus\!\longrightarrow\!\oplus$ Example: CC input `engage` connected to `engagement_mgr` input `aim_it`.
- Character-string (message), transmitted at discrete time(s): $\diamond\!\longrightarrow\!\diamond$ Example: `engagement_mgr` output `stat` connected to CC output `track_status`.

In addition, there are two allowed types of "mixed" connections, denoted as follows:

- Continuous-time signal to real number, sensed and transmitted at discrete sampling time(s): $\triangle\!\longrightarrow\!\bullet$ Example: `turret` output `theta` connected (through a summing point) to `az_filter` input `theta`.
- Real number (transmitted at discrete time(s) and held constant until the next sample) to continuous-time signal: $\bullet\!\longrightarrow\!\triangle$ Example: `pid_control` output `command` connected to `turret` input `volts`.

These mixed connections are permitted to eliminate the need for explicit modeling of analog-to-digital and digital-to-analog converters. If there are reasons to model these transformations more rigorously, then one may define a component for each such conversion.

As a final significant architectural point, observe that there are two ports on the CTC component `turret` that are not connected to other components or the outside world: the input `Ksat`, which is a "knob" (notation: `k`) or gain that can be set in defining a simulation experiment, and the output `volt_lim`, which is a "view" variable (notation: `v`) which can be displayed but not connected to another port. The significance of such "secondary inputs and outputs" is that they *cannot* be used for connection purposes. The distinction between first- and second-class inputs and outputs is important when analyzing the topology of the system and performing operations such as linearization where unwanted inputs and outputs (artificially incorporated to serve the purposes of knob and view variables) would generally be bothersome.

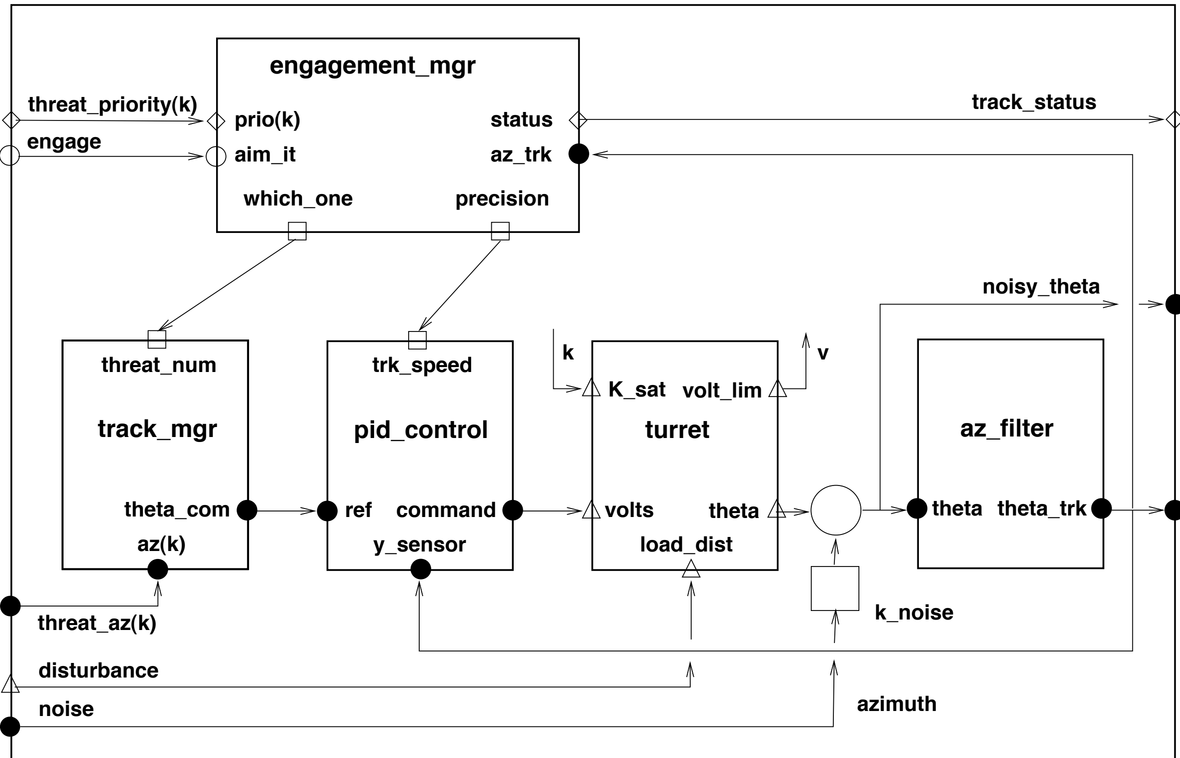

Figure 1: Illustrative Composite Component Model

2.3.2 Building Hybrid Systems

The CC above may be regarded as a part of a multi-level hybrid system. This involves a direct repetition of the component connection presented above, that is, the external inputs and outputs in Fig. 1 are treated as component inputs and outputs that are available for connection to other external components in the same manner. Each CC input and output inherits the type of the corresponding internal pure component variable (`engage` in Fig. 1 must be a Boolean, `azimuth` must be a real number transmitted at discrete times, etc.).

Consistency must be enforced at the time a composite component is assembled and again when it is incorporated into a larger hierarchical framework.

A composite component or collection of CCs may be "closed" to create a complete hybrid system by defining a *system driver component* (SDC) that generates the desired input signals; once this is done, the user may proceed to simulation and / or analysis. The SDC required for the CC in Fig. 1 must deliver the vector signals `threat_priority(k)` (a vector of string variables that may change values as a simulation proceeds) and `threat_az(k)` (a vector of reals defining the sensed azimuth angles of the threats), the scalar continuous-time signal `disturbance` and discrete-time signal `noise`, and the Boolean variable `engage`.

3 Simulation of Hybrid Systems

Section 2 defines the class of hybrid systems to be modeled and simulated in the context of this research. Here we define a *rigorous protocol* that governs the timing of events and the coordination of components that must be handled automatically in the modeling and simulation environment for this class and with *methods and algorithms* to be used for the evolution of continuous-time states.

3.1 Protocol for Simulating Hybrid Systems

The following rules for timing and coordination underlie the definition of the modeling and simulation environment for hybrid systems and the modeling language defined in Section 4:

- Every hybrid system component is invoked at the beginning of a simulation run (at $t = t_0$) to initialize it. All code in the component model is exercised during this process, including statements in an `initial` section that are executed *only* during initialization. The `initial` section provides a means for setting initial conditions on component state variables, evaluating constants or initial values of variables to be used in a component at the beginning of the run, etc., and arbitrary data processing within the DTCs and LBCs. It should also include a determination of the existence of time events or state events at $t = t_0$ that may cause a DTC or LBC to be executed immediately when the run is commenced; more specifically, the first time of invocation for each DTC should be established and the corresponding time event placed on the simulator's time-event stack[1]. Time does not advance during initialization.

- The first step after initialization is CTC state propagation (unless a DTC has specified that its first sample-time is $t = t_0$ or an LBC is triggered by an event at $t = t_0$; see first point above).

- CTC state propagation continues (by numerically integrating Eqns. $(3, 4)$ or Eqns. $(5, 7)$ subject to Eqn. (6) by suitable means) and time advances (by small increments determined by the integration algorithm) until a state event is encountered (in a CTC or LBC) or a time event (in a DTC or LBC) is triggered.

- The handling of state events internal to a CTC is described in Section 3.5.2; this does not require coordination among the components.

- Time stops when a DTC is triggered, and the inputs to the DTC are sampled for internal use. The DTC algorithm is executed to produce the updated state (Eqn. 9). The resulting updated output is made available to the other components instantaneously or with a specified time delay; if the time delay is not zero then the CTCs and / or LBCs and / or other DTCs are run until the delay time has elapsed at which point the simulation is stopped again and the DTC output is changed to its updated value (Eqn. 10).

[1] We use the term "time-event stack" informally; somehow, the simulation environment must keep track of time events that are scheduled to happen in the course of a simulation run.

- Triggering and execution of an LBC is similar to the DTC, except that an LBC may be invoked in response to state events as well as time events. Note that the definition of an LBC contains a number of mechanisms for emulating a digital module (computational delay, etc.); if an LBC is used as a continuous-time artifact (e.g., to manage state events) then these would not be utilized.

- In cases where more than one digital component is to be executed at the same instant, the simulator should check the *priority* of each operation and sequence them accordingly. If priority conflicts still exist (e.g., two modules of priority 1 are invoked simultaneously), then both will be executed with "old data" (i.e., if there are any data-dependencies among modules of equal priority, then data from the prior execution will be used).

Most of this coordination can and should be borne by the simulation environment; however, the modeling language must include mechanisms for the appropriate interplay to be specified, as outlined above. Existing commercial modeling and simulation environments incorporate many of these features [1, 18, 21, 32, 40, 42, 50]. In fact, it is generally fair to state that they can all handle these protocol elements with the exception of catching and handling state events properly (only ACSL [1] can do so) and rigorous prioritization of discrete-time events. In many cases where some of the remaining protocol elements are supported, the methods and language elements used to support them are cumbersome and/or unsupportive. The language defined in Section 4 rectifies these deficiencies.

3.2 Continuous-Time State Evolution - Overview

To reiterate the classes of CTCs that are under consideration, we have:

- Class $\mathcal{D}^0$ models:

$$\dot{x_c}(t) = f_c(x_c, u_c, u_k, b_i, m_j, t) \tag{13}$$
$$y_c(t) = h_c(x_c, u_c, u_k, b_i, m_j, t) \tag{14}$$

  which are called **ordinary differential equation (ODE) components**, and

- Class $\mathcal{D}^1$ models:

$$\dot{x_c}(t) = f_c(x_c, z_c, u_c, u_k, b_i, m_j, t) \tag{15}$$
$$0 = g_c(x_c, z_c, u_c, u_k, b_i, m_j, t) \tag{16}$$
$$y_c(t) = h_c(x_c, z_c, u_c, u_k, b_i, m_j, t) \tag{17}$$

  subject to the constraint

$$|G_z| \triangleq \left|\frac{\partial g_c}{\partial z_c}\right| \neq 0 \tag{18}$$

  which guarantees that these **semi-explicit DAE components** are of Index 1.

First, in Sections 3.3 and 3.4 we will consider the evolution of continuous-time states when the model is continuous ($f_c \in \mathcal{C}^0$ and $g_c \in \mathcal{C}^0$ at least, where in general $\mathcal{C}^m$ denotes the class of functions with up to m continuous derivatives and we ignore for the time being inputs

from discrete-time components if any) and when predictable discontinuities are present (e.g., f_c and/or g_c change abruptly at discrete times due to exercise of a DTC or LBC); then we will treat the handling of various types of discontinuous behavior, $f_c \in \mathcal{N}^0 \cdots f_c \in \mathcal{N}^3$ as categorized in Section 2.2.1.

3.3 Evolution of Continuous Class $\mathcal{D}^0$ Models

Reliable and robust methods and software for integrating Class $\mathcal{D}^0$ models with $f_c \in \mathcal{C}^m$ and $f_c \in \mathcal{N}^0$ (refer to Section 2.2.1) have been available for several decades, and serve as the core for the commercial modeling and simulation environments cited in this report. These are well established, and do not require extensive discussion or citation. The main issues in this area (aside from obvious considerations of quality algorithms, etc.) include "starting", "automatic step-size control", "stiffness" (see Gear [22] for a classical reference) and equation sorting (elimination of algebraic loops). In the present context, automatic step-size control and handling stiff ODEs are well known and do not need to be discussed; however, starting and equation sorting are worthy of consideration, as these will enter into other issues, such as restarting after discrete or state events and handling of Class $\mathcal{D}^1$ models.

3.3.1 Starting the Evolution of Class $\mathcal{D}^0$ Models

Starting the solution of a Class $\mathcal{D}^0$ CTC model requires an initial condition $x_c(t_0)$ and is often an iterative or exploratory process. First of all, the integration algorithm used for the first step(s) is often not that which will be most effective once a solution has been evolved for a few steps; for example, a predictor/corrector method may be the best algorithm for evolution, but it cannot be used for starting as it requires past points (points before $t = t_0$) for taking the first step, which is clearly infeasible. Therefore, an algorithm that does not involve prior points is used at first, e.g., one from the Runge-Kutta family.

Secondly, if the user has not specified an integration step the modeling and simulation environment must make some trials and perform error analysis to settle on an acceptable first step. This is a subtle process, especially if the model is stiff; however, it is well established and good approaches and algorithms are readily available in almost any reputable simulator.

Note that starting (restarting) is required every time a discrete or state event is handled. This is again due to the fact that past points (points before the event) have no bearing on the evolution of x_c after such a discontinuity.

3.3.2 Equation Sorting for Class $\mathcal{D}^0$ Models

Equation sorting is an important part of preparing a hybrid system model containing Class $\mathcal{D}^0$ CTC components for use in simulation. This step is significant for several reasons:

- The evaluation of Class $\mathcal{D}^0$ model expressions requires that all variables appearing on the right side of Eqn. (13) be already computed for the present value of time and state. If this is not so, then "stale" data is being used; use of old data corresponds to a one-step time delay and leads to erroneous results (perhaps even numerical instability). The

ability to create arbitrary interconnections of CTCs that may be developed independently makes it impossible to assume that the equation order provided by the model builder(s) is correct, so equation sorting is essential.

- The evaluation problem is solved, if possible, by equation sorting. To achieve this sorting, the complete set of hybrid system equations must be assembled and processed to see if it is possible to order them so that every variable on the right side of each expression has already been computed. This procedure is well-known and standard (see below for a brief overview).

- Equation sorting permits the detection of algebraic loops, that is, situations where it is impossible to order the model expressions so that all variables on the right side are already computed. A classic example of an algebraic loop is direct feedback around a standard nonlinear component model:

$$\begin{aligned} \dot{x} &= f(x,u) \\ y &= h(x,u) \\ u &= K \cdot (r(t) - y) \end{aligned} \tag{19}$$

  Trouble arises because one must evaluate y before u is evaluated which requires that y be evaluated, and so on, so the equations cannot be sorted.

- Restricting the topology of the system model to eliminate problems such as algebraic loops is much too stringent; instead, equation sorting pinpoints the problem exactly, making it easy to implement a solution. For example, it would be overly restrictive to forbid direct feedback in the above case; instead, the problem can be resolved either by using a DAE solver (see next section) or by eliminating the direct feed-through of u into y (make $y = h(x)$) by implementing a low-pass dynamic system model ("sensor dynamics") or a delay operator in the loop. The latter is standard and usually physically justified by the fact that a sensor for y cannot have infinite bandwidth; however such "fixes" may present their own problems (e.g., they may produce stiff system models).

The fact that a DAE solver is the most exact solution to the problem in Eqn. (19) can be appreciated by a trivial manipulation that reduces it to Class $\mathcal{D}^1$:

$$\begin{aligned} \dot{x} &= f(x,u) \\ 0 &= u + K \cdot (y - r(t)) \\ y &= h(x,u) \end{aligned} \tag{20}$$

For a succinct discussion of equation sorting, see [16]. Even more briefly, an effective procedure is based on graph-theoretic methods that begins by parsing the model relations to create a *structural incidence matrix* $\mathcal{S}$ defined by $S_{i,j} = 1$ if variable j appears in equation i, otherwise $S_{i,j} = 0$, and then finding (attempting to find) permutation matrices P and Q such that $P\mathcal{S}Q$ is lower triangular. A graph-theoretic algorithm of Tarjan [43] is very efficient for this part of the procedure. The result of this processing is:

- if $P\mathcal{S}Q$ can be rendered lower triangular, then the equations can immediately be sorted (by permutations based on P and Q) and the Class $\mathcal{D}^0$ model is ready to use;

- otherwise PSQ will be rendered lower *block* triangular, where each $m \times m$ block with $m > 1$ corresponds to a set of algebraic equations to be separated and included in the constraints, Eqn. (16).

We see from this discussion that including Class $\mathcal{D}^1$ models in the scope of the modeling and simulation environment eliminates any need to "fix" algebraic loops in ODE models. In essence, using equation sorting as outlined above makes it unnecessary to distinguish between them - the model may be processed and treated as one or the other, as appropriate.

3.4 Evolution of Continuous Class $\mathcal{D}^1$ Models

Discuss DASSL [6, 34] and COLDAE [3] as the most respected packages for handling Class $\mathcal{D}^0$ and $\mathcal{D}^1$ models when there are no discontinuous effects. Discuss initial conditions and starting. **Expand!**

3.5 Discontinuity (State-Event) Handling

Four types of discontinuous behavior are categorized in Section 2.2.1. These are: predictable discrete-time events, $f_c \in \mathcal{N}^0$; switch-type nonlinearities, $f_c \in \mathcal{N}^1$; state-changing nonlinearities, $f_c \in \mathcal{N}^2$; and structure-changing nonlinearities, $f_c \in \mathcal{N}^3$. All but the first case involve *unpredictable* occurrences (e.g., gears engaging and disengaging), which are called *state events.*

3.5.1 Handling Predictable Discontinuities

This most elementary type of discontinuity is handled by most commercial modeling and simulation environments, using a scheme roughly as follows:

- identify the times of upcoming discrete-time events, as outlined in Section 3.1;
- evolve CTC states using an appropriate numerical integration algorithm;
- at each integration step, determine if the earliest future discrete-time event will fall within the step or at its very end; if it will fall within the step, then reduce the step so integration stops precisely at that event; then execute the numerical integration step;
- process the discrete-time event(s);
- restart the evolution of the CTC states.

The only capability that is often weak or lacking in existing modeling and simulation environments is support for prioritizing discrete-time events; this is often done by adding small "epsilons" to the timing relations for lower-priority discrete-time components or other patchwork logic. In cases where components are taken from libraries designed for multiple use, this is a crude and inadvisable practice.

3.5.2 Handling Switch-Type Nonlinearities

Common examples of switch-type nonlinearities include Coulomb friction and an ideal relay, which are modeled by a change in sign of a friction term when (angular) velocity passes through zero or making or breaking a connection when a voltage crosses a threshhold value, respectively. Substantial errors in simulation may occur if such events are not "captured" correctly [14, 17], producing results that may be quite misleading when stability is marginal and which might even be confused with chaotic behavior in some cases.

Switching events can, of course, be substantially more complicated than illustrated in the examples mentioned above:

- In general, a switching event can be formulated in terms of an arbitrary boundary function or zero-crossing condition:

$$S(x_c, b_i, m_j) = 0 \tag{21}$$

  where S is a general expression involving the state and perhaps Boolean variables and modes of the CTC model. For convenience below we will associate a model variable `sgn` with the boundary relation, i.e., `sgn` $= 1$ if $S > 0$, `sgn` $= -1$ if $S < 0$.

- The result of a state event can also be more generally formulated; for example, a switching variable (such as `sgn`) may be set to ± 1 depending on the sign of S in the boundary function of Eqn. (21) and used in the evolution equations in arbitrarily complicated expressions of the form `if sgn > 0.0 then do ... else do ... endif;`.

To handle such an event, the simulation software must "watch" for sign changes in any S variable and, whenever they occur, adjust the integration step that resulted in the sign change until the event has "just happened" using a Newton-type "root-finding" algorithm to locate the zero-crossing. The event itself is not executed (e.g., `sgn` changed from –1.0 to +1.0) until this process is complete and we have an "accepted" next point on the trajectory; therefore, the event iteration can be done reliably because there can be no erratic points or integration steps taken with the discontinuous term switching back and forth in some arbitrary fashion.

The most straightforward and "safe" way to implement any unpredictable discontinuity is to insist that *unswitched dynamics must be smooth over the boundary.* In such cases, it is sufficient to define "just happened" to signify that S (Eqn. 21) is in the range (-ϵ, ϵ) where ϵ is an internal tolerance, and the iterative approach to finding such a point can involve using solution points on both sides of the boundary (e.g., using a Newton-type algorithm, as mentioned). When such a point is obtained numerical integration is halted, and `sgn` is changed from –1.0 to +1.0 or *vice versa* which mechanizes the required discontinuous change in the model; the evolution of the trajectory continues with `sgn` fixed until the next crossing is detected and handled. This behavior is directly implementable, as illustrated in Fig. 2.

We emphasize that the state event *cannot be allowed to occur* during an integration step; rather, each step must always be taken with the value of `sgn` fixed. In other words, the CTC model must be formulated so that it has a continuously-varying derivative during each step; an `event` handler is used to instantiate any discontinuity.

The above restriction eliminates a number of complications: If the model dynamics are undefined or unsmooth over the boundary without switching, then finding the switching point

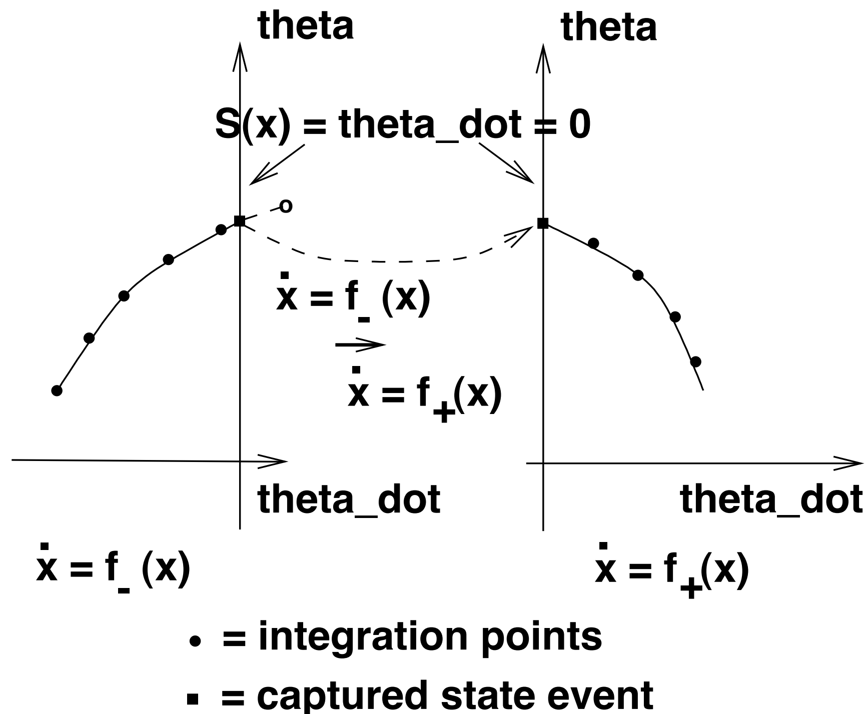

Figure 2: State-Event Simulation with Capture

cannot involve using solution points on both sides of the boundary, and "just happened" must be defined to mean that the boundary is strictly crossed (e.g., that S is in the range $(0, \epsilon)$ in the positive-going crossing case). The iterative approach to finding such a point cannot involve using solution points on both sides of the boundary; in fact, the point "on the other side" would have to be obtained by artificial means such as reflection across the boundary.

There is one additional feature that may be incorporated in state event handling: change of CTC state. Under normal circumstances, the state of the CTC should evolve only according to the dynamics or differential equations (3). The simulator should not allow expressions in the dynamical model to interfere with this evolution (e.g., an expression $x_c = \ldots$ would not be permitted). However, a state event may require a change of state as well as a change in the dynamics. For example, if two bodies collide and bounce apart there is an instantaneous change in the velocity of each body. Such changes should be allowed in an event handler.

The numerical algorithms for handling this class of discontinuity are readily available. First, ACSL [1] has the required zero-crossing detectors, although the modeling language support is weak; also, DASSLRT [6] has the root-finding algorithms that work in concert with the DAE solver to achieve the desired result for a broader class of models (including $\mathcal{D}^1$); however, there is *no* modeling language support since this software is only a subroutine library. Therefore, the primary need in this area is providing good modeling language for describing such events, as outlined in Section 4.

Warning: In handling this class of nonlinearity it is assumed above and in the software cited that the dynamics are such that a trajectory can always cross the boundary; in conflict situations (Section 3.5.4) this may not be so. However, the simulator can include a monitor to determine if this assumption is violated and warn the user accordingly (see Section 3.5.4)

so that the model can be corrected or reformulated with the offending nonlinearity treated as a structural one. No commercial modeling and simulation environment that we know of includes this vital feature.

3.5.3 Handling State-Changing Nonlinearities

This class of CTC includes elements $f_c \in \mathcal{N}^3$, i.e., involves nonlinear effects wherein the dimension of the state-space changes in a simple fashion. To outline a typical case (see [49]), assume a motor is coupled to a load through a gear train with backlash. Then there are three modes of operation: 'disengaged', 'engaged-turning-CW', 'engaged-turning-CCW'. When the mode is 'disengaged' there are two uncoupled second-order ODE sets describing the unrelated motions of the motor and load; when they are engaged we have $\theta_l = \theta_m \pm \delta$ where δ is one-half the backlash gap, and $\dot{\theta}_l = \dot{\theta}_m$. The most direct way to handle such a model is to prepare separate models for the two mechanical parts, including all torques acting on the motor and load, and add the constraint equations

$$\begin{aligned} 0 &= K_e \cdot (\theta_l - \theta_m \mp \delta) \\ 0 &= K_e \cdot (\dot{\theta}_l - \dot{\theta}_m) \end{aligned}$$

where K_e takes the values 0 if mode = 'disengaged' and 1 if the gears are engaged.

This approach handles a wide variety of common nonlinear effects, especially in systems with mechanical components. Another example is stiction (a friction effect wherein two objects stop sliding with respect to each other and become "stuck" together until a sufficient force or torque is applied to break them loose); a very elegant presentation of such a variable-order model may be found in [19].

3.5.4 Handling Structure-Changing Nonlinearities

The example below shows how to cope with nonlinear components where vector-field conflict situations arise, by which we mean the continuous-time dynamic equations produce derivative vector fields that are directed into a boundary on both sides. This condition indicates that the trajectory being evolved cannot simply cross the boundary, as assumed in discussing switching discontinuities (Section 3.5.2). Such situations most generally require changing to another dynamic model that governs motion *on the boundary*, in addition to the discontinuous model for motion on either side. This situation represents a structural change (reducing the order of the model for motion along the boundary) that may be fundamentally more difficult to handle than the state-changing case.

There are two possible approaches to resolving such situations:

- *Ad hoc:* Develop an approach to handle these situations with a "standard" method, based on mathematical principles or some other rationale.
- *Physics-based:* Require the modeler to develop a differential equation set describing the motion of the system along the boundary.

In either case, one must specify explicitly or implicitly conditions for moving off the boundary when the conflict disappears, and incorporate a handler to detect and execute this process.

An example of the first or *ad hoc* approach would be to declare that the trajectory in a conflict situation must evolve according to a specific rule such as the following: evaluate the vector field on each side of the boundary, denoted f_+ and f_-; project these vectors onto the boundary to obtain the tangential components denoted $f_{+,tan}$ and $f_{-,tan}$; let s denote motion along the boundary and evolve s according to

$$\dot{s} = f_{+,tan} + f_{-,tan} \tag{22}$$

This informal "recipe" produces a smoothed version of the solution obtained using first-order Euler integration with a fixed step h and allowing the solution to "chatter" along the boundary; a simple geometrical analysis shows this to be true in the limit as $h \to 0$ (Appendix A of [47]). A method due to Filipov recasts the vector conflict problem as a DAE where the boundary function becomes a constraint equation [20]. Filipov's method seems to be equivalent to the first recipe above; at least it has been shown to yield the same result in a simple test problem (also, see Appendix A of [47]).

The second or model-based approach is more general and satisfying from a physical or engineering point of view and is outlined here. Note that an automatic method for detecting conflicts and resolving them is proposed below, we believe for the first time; it is *not* assumed that the modeler knows that there is a potential for vector field conflict in the problem as specified and that it is handled appropriately.

The following simple example illustrates the issues and a proposed approach to handling such circumstances: Consider a second-order system with states x and y, with a boundary comprising the unit circle

$$\mathcal{B}: \quad S = x^2 + y^2 - 1 = 0 \tag{23}$$

and state differential equations

$$\dot{x} = \begin{cases} 1 & \text{if } (x,y) \subset \mathcal{B} \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

$$\dot{y} = \begin{cases} 0 & \text{if } (x,y) \subset \mathcal{B} \\ 1 & \text{otherwise} \end{cases} \tag{25}$$

In words, all trajectories evolve "to the right" inside $\mathcal{B}$ and "up" outside $\mathcal{B}$ when viewed in the usual (x, y) plane, as depicted in Fig. 3. Vector-field conflicts arise in quadrant 4, that is, trajectories are "into" $\mathcal{B}$ on both sides when $x > 0,\ y < 0,\ (x, y) \in \mathcal{B}$. In general, such a conflict situation requires a dynamical model that governs the motion of x, y for $(x, y) \in \mathcal{B}$; therefore, we arbitrarily characterize motion along $\mathcal{B}$ in polar coordinates as $\dot{\theta} = 1$ (a point on $\mathcal{B}$ moves counter-clockwise until the vector field conflict is resolved). A sketch of this system's phase portrait reveals that the only situation in which conflicts arise is when trajectories start from (x_0, y_0) such that $y_0 < 0$ and $-1 < x_0 < +1$ (cases **a** and **b** in Fig. 3). We observe that the resolution of the conflict according to Eqn. (22) would yield $\dot{\theta} = cos(\theta) - sin(\theta)$, so the model $\dot{\theta} = 1$ does not correspond to the "chattering" or Filipov solution.

We propose to handle situations such as that illustrated in Fig. 3 as follows:

1. Along trajectory **a** up to point $\odot$ we have no discontinuities except possibly predictable discrete-time events, and we integrate the states x, y as in Section 3.5.1.

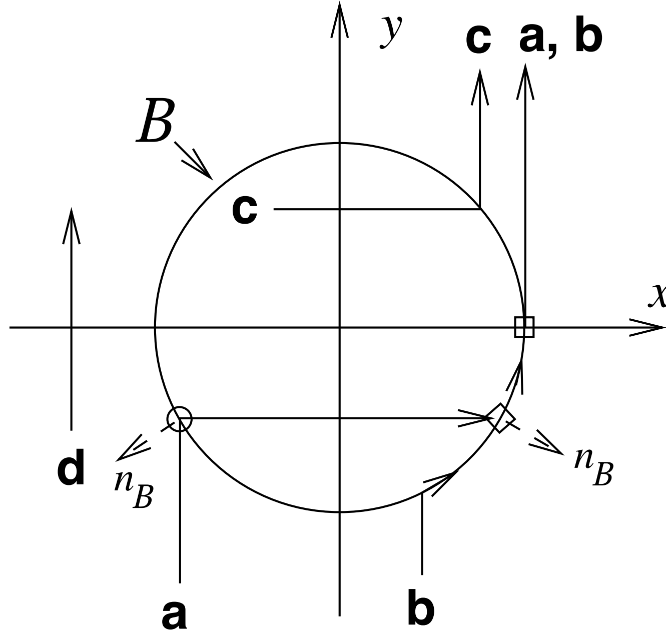

Figure 3: Vector Field Conflict Example

2. At point $\odot$ the state event is located as described in Section 3.5.2. The local normal n_B of the boundary $\mathcal{B}$ is evaluated by numerical differentiation of Eqn. (23)[2]. The model is switched to conform with $(x, y) \subset \mathcal{B}$, the state derivatives are calculated, and it is determined that the dot product of $\dot{x_c}$ and n_B is negative which permits the trajectory to cross the boundary, as shown.

3. From point $\odot$ to point $\diamond$ we again have a continuous case which we integrate as in Section 3.3, up to the second state event at point $\diamond$ which is handled as outlined in Item 2. The outcome is different, however: the dot product of $\dot{x_c}$ and n_B is positive, indicating a conflict that forces the dynamics to switch to the manifold $S = x^2 + y^2 - 1 = 0$, with reduced state description $\dot{\theta} = 1$.

4. We assume that the submodel for motion along $\mathcal{B}$ includes calculation of the position in the full state space x, y. As θ is integrated the conflict handler continually evaluates n_B, $\dot{x_c}$ for $(x, y) \subset \mathcal{B}$, and $\dot{x_c}$ for (x, y) outside $\mathcal{B}$ at each step. Both dot product conditions are checked; as long as they both indicate conflict integration along $\mathcal{B}$ continues, but as soon as the conflict disappears on one side or the other (at point $\square$) the model is switched (in this case, to that for $S > 0$).

The conditions for vector-field conflict are rigorously stated above in terms of gradients and dot products. Simpler tests (that would have to be carefully implemented in terms of numerical/algorithmic procedures) would involve evaluating $\dot{x_c}$ on both sides of the boundary, taking small integration steps based on each derivative evaluation, and seeing if either solution is consistent (moves into the corresponding region of the state space, e.g., does the numerical

[2]This can be done robustly using the method of [48] even when S does not possess continuous derivatives with respect to all variables.

integration using the model for $S > 0$ result in a trajectory that moves into the subspace $S > 0$?).

An automatic conflict handler such as that proposed above can alert the user if conflict situations are unanticipated, as well as handle them if they are expected (and a suitable submanifold model is provided). This feature, if enabled, would eliminate the possibility of obtaining meaningless solutions that arise from allowing the numerical integrator to chatter along the boundary; on the other hand, if the user accepts such solutions the handler can be disabled to restore integration efficiency. To the best of our knowledge, this is a novel feature.

4 Preliminary HSML Language Definition

Based on the class of hybrid systems defined in Section 2 and the simulation requirements outlined in Section 3 the following tentative definition is offered as a preliminary modeling language framework for hybrid systems.

We use the following partial BNF notation in all specifications:

Symbolism	Meaning
`< >`	delimits an arbitrary syntactic entity
`::=`	"denotes" or "is defined as"
`|`	exclusive OR (choose only one elements in list)
`[ ]`	delimits an optional element
{ }	delimits a compulsory element
`*`	repeat the marked element the appropriate number of times
`%`	the text that follows is a comment

The designation `<simple_variable>` refers strictly to the name of an arbitrary *scalar* entity (e.g., `circle = x*x + y*y - 1.0;`) and the element `<variable>` encompasses both `<simple_variable>`s as well as vectors of `<simple_variable>`s. It is generally assumed that `<variable>` names an element that changes as time evolves during a simulation run; the entity `<parameter>` is used to denote the name of an element that does *not* change with time but may be changed from one simulation run to another in the course of performing simulation experiments. An entity may be more specifically designated where useful to do so for convenient reference, for example, `<component_identifier>`, `<time_name>`, etc.

4.1 Describing Components in General

As mentioned in Section 2.1, at the lowest level HSML components are "pure" CTCs, DTCs and LBCs. These are assembled into composite components (CCs), and then systems. Every component has an *interface* and a *body*:

- its interface defines the entities that are accessible from and to the outside, as follows:
  - *Primary input/output variables:* `input` and `output` signals may be connected to other components' outputs and inputs, respectively.
  - *Secondary input/output variables:* `scope` and `knob` entities can *not* be connected to other components' inputs and outputs; rather `scope` variables may be stored and displayed, and `knob` parameters (constants) may be set/changed arbitrarily using a supporting simulation environment during the definition of simulation experiments.
- its body contains an arbitrary number of sections, a few of which are universal in occurrence (`declarations` for characterizing key elements of the component model and `assignments` for assigning values to parameters) and others which are particular

to the type of component being described (e.g., a CTC will have a `dynamics` section for encoding the state equations).

These basic properties are captured in the following general template for defining any type of component:

```
<Component_type> <Component_name> is
%
interface
  [ input(<name>,<type>,<range>); ]*
  [ output(<name>,<type>,<range>); ]*
  [ knob(<name>); ]*
  [ view(<name>); ]*
end interface;
%
body
  declarations
   . . . ;
  end declarations;
  section_one
    . . . ;
  end section_one;
    . . . ;
  assignments
    [ <parameter_name> : <value>; ]*
  end assignments;
end body;
%
end <Component_name>;
```

This general construct is particularized below for each hybrid system component under consideration.

4.2 Describing a Continuous-Time Component (CTC)

The following provides both a structural template and a tentative syntactic definition of a HSML continuous-time component:

```
{ CTC  <ctc_identifier> IS
%
{ INTERFACE
   [ INPUT(<variable>,<type>[,(<range> | <set>)]); ] *
   [ OUTPUT(<var_1>) = <var_2>; ] *
   [ KNOB(<parameter>); ] *
   [ VIEW(<var_1>) = <var_2>; ] *
end interface; }
%
{ BODY
   [ DECLARATIONS
```

```
      [ STATE(<var_1>,<var_2>[,<range>]); ] *
      [ LOCAL(<variable>[,<type>][,<range> | <set>)]); ] *
      [ TIME(<simple_variable>); ]
      [ TSWITCH(<simple_variable>); ] *
      [ FLAG(<simple_variable>); ] *
   end declarations; ]
%
   [ INITIAL
      [ Computation of initial values for state variables ]
      [ Computation of parameters ]
   end initial; ]
%
   [ EVENT(<flag_variable>)
      [ Computation of flag variable ]
      { POSITIVE-GOING
         [ Computation of model changes at event ]
         [ Computation of changes to state variables ]
      end positive-going; }
      [ ON-EVENT
         [ Computation of model "during" the event ]
         [ Computation of state variable transforms ]
      end on-event; ]
      { NEGATIVE-GOING
         [ Computation of model changes at event ]
         [ Computation of changes to state variables ]
      end negative-going; }
   end event; ] *
%
   { DYNAMICS
      [ Computation of auxiliary variables ]
      [ Computation of output variables ]
      [ Computation of derivatives ]
   end dynamics; }
%
   [ CONSTRAINTS
      [ Computation of auxiliary variables ]
      [ Computation of constraint equations ]
   end constraints; ]
%
   [ OUTPUTS
      [ Computation of auxiliary variables ]
      [ Computation of output variables ]
   end outputs; ]
%
   [ ASSIGNMENTS
      [ Parameter assignments ]
      [ Initial value assignments ]
   end assignments; ]
end body; }
```

```
%
END <ctc_identifier>; }
```

Note that the words in uppercase are reserved words; they cannot be used in other contexts (e.g., as variable names). The language itself will probably not be case sensitive; fom, FoM, and FOM would all refer to the same variable; also, DYNAMICS, dynamics, and DyNaMiCs would be equally acceptable section delimiters. The following constitutes a specification for the statement types in the above CTC syntax definition:

- INPUT: The first element <variable> is the internal name of a component input; the second element <type> may take on the values signal (a continuous-time variable), real (a sampled-and-held discrete-time variable), integer, boolean or string (also sampled-and-held); optionally a range <range> = (<min>,<max>) (end points excluded from range) or [<min>,<max>), (<min>,<max>], [<min>, <max>] (one or both end points included in range) may be provided for signal, real, or integer inputs where <min> and / or <max> may be a number or a parameter, or a <set> may be specified, e.g., {high, medium, low} for a string input. There may be any number of INPUT statements.

- OUTPUT: The first element <var_1> is the external name of a component output; it must be (in the same statement) identified in terms of an internal name (the second element <var_2>). There may be any number of OUTPUT statements.

- KNOB: The element <parameter> is the name of a *fixed parameter* (not a variable) that is to be varied from run-to-run as part of the definition of an experiment. There may be any number of KNOB statements. Note that this is a so-called "second-class input", which may not be connected to another component.

- VIEW: The element <var_1> is the external name of the internal variable <var_2> which can be displayed but not connected to another component (a so-called "second-class output"). There may be any number of VIEW statements.

- STATE: The first element <var_1> is the internal name of a component state variable x_c; the second element <var_2> is the internal name of the corresponding state derivative $\dot{x}_c$; optionally a range may be provided as for an INPUT. There may be any number of STATE statements.

- LOCAL: The element <variable> is the local name of a component internal variable; the second optional element <type> may take on the same values as for INPUT, above; optionally a range or set may be provided as for an INPUT. There may be any number of LOCAL statements.

- TIME: The element <simple_variable> is the name of the variable that corresponds to the current simulation time (e.g., to t for evaluating any function $g_c(x_c, t)$). There may be only one TIME statement.

- TSWITCH: The element <simple_variable> is the name of a time-point corresponding to a time event. The simulation should stop when TIME is equal to the quantity specified in <simple_variable>. There may be any number of TSWITCH statements.

- FLAG: The element `<simple_variable>` is the name of the variable that is evaluated to define a state event in terms of a zero-crossing (e.g., an event may occur when $S(x_c, b_i, m_j) = 0.0$). There may be any number of FLAG statements.

The functionality of the sections `INITIAL, EVENT,` and `DYNAMICS` is setting initial conditions, describing state events and modeling the dynamics (Eqns. 13, 15), respectively, as indicated by comments. We note only that the value of a state variable *cannot* be reset in the `DYNAMICS` section, where numerical integration varies the state according to the state variable differential equations; it can be reset in the `EVENT` section if necessary in state-event handling (e.g., to change the value of the ball velocity in a bouncing ball problem). The sections `CONSTRAINTS` and `OUTPUTS` are tentatively included in this definition; the former may be needed if the equation sorting approach outlined in Section 3.3.2 is not powerful enough to automatically separate out the constraint equations, and the latter is included in case the model contains extensive output equation computations which would be unnecessarily burdensome if evaluated multiple times during the numerical integration process.

The only section that may be repeated is `EVENT`; for each state event to be handled there must be a `FLAG(<simple_variable>);` declaration and corresponding `EVENT(<flag_variable>)` section. Finally, mandatory event subsections are provided to handle positive-going and negative-going transitions, and an optional subsection may be provided to handle structure-changing nonlinearities (Section 3.5.4). The details of the HSML for the computation of derivatives, parameter assignment, etc., are not fully defined at this time; the examples in Section 5 are suggestive.

4.3 Describing a Discrete-Time Component (DTC)

The following provides both a structural template and a tentative syntactic definition of a HSML discrete-time component:

```
{ DTC  <dtc_identifier> IS
%
{ INTERFACE
   [ INPUT(<variable>,<type>[,(<range> | <set>)]); ] *
   [ OUTPUT(<var_1>) = <var_2>; ] *
   [ KNOB(<parameter>); ] *
   [ VIEW(<var_1>) = <var_2>; ] *
end interface; }
%
{ BODY
   { DECLARATIONS
      [ STATE(<var_1>,<var_2>[,<range>]); ] *
      [ LOCAL(<variable>[,<type>][,<range> | <set>)]); ] *
      [ TIME(<time_name>); ]
      { TSAMPLE(<sample_T_name>); }
      [ TDELAY(<parameter>); ]
      [ PRIORITY(<integer>); ]
      [ BITS( (REAL | INTEGER),<integer>); ] *
   end declarations; }
```

```
%
   [ INITIAL
      [ Computation of initial values for discrete-time states ]
      [ Computation of parameters ]
   end initial; ]
%
   { UPDATE
      [ Computation of auxiliary variables ]
      [ Computation of output variables ]
      [ Computation of updated states ]
      { next DTC execution time specification --
          e.g. <sample_T_name> = <time_name> + <delta_T> }
   end update; }
%
   [ ASSIGNMENTS
      [ Parameter assignments ]
      [ Initial value assignments ]
   end assignments; ]
%
end body; }
%
END <dtc_identifier>; }
```

Note that the words in uppercase are reserved words; they cannot be used in other contexts (e.g., as variable names). The following constitutes a specification for all the statement types in the above DTC syntax definition:

- INTERFACE elements are defined precisely as in the CTC case.
- STATE: The first element `<var_1>` is the internal name of a component state variable x_k; the second element `<var_2>` is the internal name of the corresponding "new" state or updated value x_{k+1}; optionally a range may be provided as specified for CTC INPUTs. There may be any number of STATE statements.
- LOCAL: These elements are defined precisely as in the CTC case.
- TIME: The element `<time_name>` is the name of the variable that corresponds to the current simulation time (e.g., to t_k in any function $g_k(x_k, t_k)$). There may be only one TIME statement.
- TSAMPLE: The element `<sample_T_name>` is the name of the variable that corresponds to the simulation time at which the DTC is to be invoked for the next update. There may be only one TSAMPLE statement.
- TDELAY: The element `<parameter>` is the name of the parameter that corresponds to the simulation time offset between the time DTC inputs are sampled and DTC outputs are made available to the rest of the system. There may be only one TDELAY statement.
- PRIORITY: The element `(<integer>)` defines the precedence order of simultaneous discrete-time events; modules with lower values of `(<integer>)` should be executed before those with higher values. There may be only one PRIORITY statement.

- BITS: The first element, `real` or `integer` is an accepted variable type; `<integer>` specifies the number of bits used in representing that variable type in the DTC (of course, the simulation environment would have to support this feature).

Again, the details of HSML for the computation of updated states, etc., are not fully defined at this time.

4.4 Describing a Logic-Based Component (LBC)

Only the shell of a logic-based component can be specified at this time. The syntax of a logic-based component shell definition is as follows:

```
{ LBC  <lbc_identifier> IS
%
{ INTERFACE
   [ INPUT(<variable>,<type>[,(<range> | <set>)]); ] *
   [ OUTPUT(<var_1>) = <var_2>; ] *
   [ KNOB(<parameter>); ] *
   [ VIEW(<var_1>) = <var_2>; ] *
end interface; }
%
{ BODY
   { DECLARATIONS
      [ MODE(<simple_variable>); ] *
      [ LOCAL(<variable>[,<type>][,<range> | <set>)]); ] *
      [ TIME(<simple_variable>); ]
      [ TSWITCH(<parameter>); ] *
      [ TDELAY(<simple_variable>); ]
      [ FLAG(<simple_variable>); ] *
      [ INTERRUPT(<simple_variable>); ] *
      [ PRIORITY(<integer>); ]
   end declarations; }
%
   [ INITIAL
      [ Computations to initialize the LBC ]
      [ Computation of parameters ]
   end initial; ]
%
   [ EVENT(<variable>)
      [ Computation of flag variable ]
      { positive-going
         [ Computation of model changes at event ]
         [ Computation of changes to state variables ]
         [ Invocations of an LBC ]
      end positive-going; }
      { negative-going
         [ Computation of model changes at event ]
         [ Computation of changes to state variables ]
         [ Invocations of an LBC ]
```

```
      end negative-going; }
   end event; ] *
%
   { LOGIC
      [ simple logical expressions |
        *** logic/expert system code/interfaces here *** ]
   end logic; }
%
   [ ASSIGNMENTS
      [ Parameter assignments ]
      [ Initial value assignments ]
   end assignments; ]
end body; }
%
END <lbc_identifier>; }
```

Note that the words in uppercase are reserved words; they cannot be used in other contexts (e.g., as variable names). The following constitutes a specification for some of the statement types in the above LBC syntax definition:

- INTERFACE elements are defined precisely as in the CTC case.
- DECLARATIONS elements are defined precisely as follows: FLAG, LOCAL, TIME and TSWITCH are defined as in the CTC case; TDELAY and PRIORITY are as for DTCs; and MODEs (m_j, Eqn. 11) and INTERRUPTs (signals that the LBC must be invoked) are unique to LBCs (usage is obvious).

The details of the HSML for the symbolic and / or numerical computations required to implement an LBC are not fully defined at this time.

4.5 Describing a Composite Component

The following provides both a structural template and a tentative syntactic definition of a HSML composite component:

```
{ CC  <cc_identifier> IS
%
{ INTERFACE
   [ INPUT(<variable>,<type>[,(<range> | <set>)]); ] *
   [ OUTPUT(<var_1>) = output(<compt_ident>.<var_2>) |
      input(<compt_ident>.<var_3>); ] *
   [ KNOB(<parameter>); ] *
   [ VIEW(<var_1>) = output(<compt_ident>.<var_2>) |
      input(<compt_ident>.<var_3>); ] *
end interface; }
%
{ BODY
   { CONNECTIONS
      [ INPUT(<component_identifier>.<variable>) = permitted
            expression involving <inputs> and/or
```

```
         output(<compt_ident>.<variable>); ] *
   end connections; }
   { COMPONENTS
      [ <instance_name>(<file_id>); ] *
   end components; }
   [ ASSIGNMENTS
      [ Parameter assignments ]
      [ Initial value assignments ]
   end assignments; ]
end body; }
%
END <cc_identifier>; }
```

Note that the words in uppercase are reserved words; they cannot be used in other contexts (e.g., as variable names). The following constitutes a specification for all the statement types in the above hybrid system or composite component syntax definition:

- INTERFACE elements are defined precisely as in the CTC case.
- CONNECTIONS: Each input of each component (in global variable notation) is identified in terms of component outputs (in the same notation) and/or CC inputs. The use of operators is not definitive at this time; however, the use of addition and multiplication by a gain is likely to be permitted:

  ```
  input(az_filter.theta) = output(turret.theta) + k_noise*noise;
  ```

  However, the specification of "permitted expressions" is still open.

4.6 Describing a Hybrid System

The following provides both a structural template and a tentative syntactic definition of a HSML system driver component (SDC):

```
{ SDC  <sb_identifier> IS
%
{ INTERFACE
   [ KNOB(<parameter>); ] *
   [ VIEW(<var_1>) = output(<compt_ident>.<var_2>) |
      input(<compt_ident>.<var_3>); ] *
end interface; }
%
{ BODY
   { DECLARATIONS
      [ TIME(<simple_variable>); ]
      [ TSWITCH(<parameter>); ] *
   end declarations; }
%
   { CONNECTIONS
      [ INPUT(<component_identifier>.<variable>) = <variable>; ] *
   end connections; }
```

```
%
   { DRIVERS
      [ Computation of CC input variables ]
   end drivers;
%
   { COMPONENTS
      [ <instance_name>(<file_id>); ] *
   end components; }
%
   [ ASSIGNMENTS
      [ Parameter assignments ]
      [ Initial value assignments ]
   end assignments; ]
end body; }
%
END <sb_identifier>; }
```

Note that the words in uppercase are reserved words; they cannot be used in other contexts (e.g., as variable names). This construct is similar to the definition of a CC, as CONNECTIONS and COMPONENTS are defined in a SDC module; it also has characteristics of a CTC in that continuous-time signals and time events may be modeled in the DRIVERS section. The most significant departure relative to the four basic component types is that an SDC cannot have INPUT or OUTPUT variables in the INTERFACE section.

4.7 Component Encapsulation

We emphasize that each component is, with very few and carefully circumscribed exceptions, *rigorously encapsulated*. The goal is that a component's internal variables and parameters must not be *confused with* or *inappropriately influenced by* those of other components. This property is critical if components are to be interchangeable, reusable, and arbitrarily interconnectable without having to worry about "side effects". For example, the fact that the states in `turret` (see Section 5.2.1) are called `theta` and `theta_dot` for θ and $\dot{\theta}$ does not rule out using the same names in other components; if this is done, then no undesirable side-effects should occur.

The two instances where internal variables in one component may be influenced by those in another are as follows:

- The state of one component may depend on that of another *in an* `initial` *section.* In fact, it is assumed in the `initial` section of `turret` (Section 5.2.1) that the turret is coupled to the tank in some way, and that the `tank_body` component also contains an angle variable called `theta`; the angle state of `turret` is initialized to be offset from that of `tank_body` by the amount `del_theta`. To support this permitted violation of encapsulation (see Section 4.7), we use the unambiguous "global-variable" notation for external variables introduced in Section 5.1, i.e., `tank_body.theta`, as shown in that example.

- The state of one component may depend on that of another *in an* `event` *section.* For example, if two objects collide then each velocity after collision depend on the precol-

lision velocities of both objects. This intercomponent interaction is also implemented using global variable notation.

The reason that these influences are not modeled at the interface (using component input/output variables) is that these effects are qualitatively different, that is, the influence is instantaneous rather than dynamic. Also, to add the input/output "hooks" to implement these effects via the interface would create unnecessary complexity as well as topological confusion in other domains (e.g., linearization and stability analysis).

5 HSML Modeling Example

It is useful to illustrate the basic ideas and approach for modeling hybrid systems by introducing a moderately comprehensive example. This illustration is developed by presenting low-order components to flesh out parts of the composite component depicted in Fig. 1 (Section 2.3.1). The block diagram of Fig. 1 is first translated into an HSML composite component; then the pure components `turret`, `pid_control`, and `engagement_mgr` are very simply instantiated.

5.1 A Composite Component Connection Definition

The block diagram example in Fig. 1 defines a composite component, which may be described in textual form as follows:

```
CC turret_azimuth_control is
%
interface    %  declare CC interface variables:
  input(threat_priority,message);   % threat priorities input
  input(engage,boolean);            % set ‘true‘ to engage target
  input(threat_az,real);            % threat azimuth angles
  input(disturbance,signal);        % external disturbance
  input(noise,real);                % sensor noise source
  knob(k_sat);                      % two ’knob’ parameters to
  knob(k_noise);                    % ... experiment with
  output(track_status) = output(engagement_mgr.stat);
  output(noisy_theta) = input(az_filter.theta);
  output(azimuth) = 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(engagement_mgr.prio) = threat_priority;
    input(engagement_mgr.aim_it) = engage;
    input(engagement_mgr.az_trk) = output(az_filter.theta_trk);
    input(track_mgr.az) = threat_az;
    input(track_mgr.threat_num) = output(engagement_mgr.which_one);
    input(pid_control.ref) = output(track_mgr.theta_com);
    input(pid_control.y_sensor) = output(az_filter.theta_trk);
    input(turret.volts) = output(pid_control.command);
    input(turret.load_dist) = disturbance;
    % make turret.Ksat a ’knob’ variable:
    input(turret.Ksat) = k_sat;
    input(az_filter.theta) = output(turret.theta) + k_noise*noise;
  end connections;
  components  %  identify sources for component descriptions
    turret(turret.mdl);
    az_filter(/library/filters/kalman.mdl);
```

```
    pid_control(/library/controllers/d_pid.mdl);
    track_mgr(track_manager.mdl);
    engagement_mgr(engagement_manager.mdl);
  end components;
  assignments
    k_noise: 1.0;                    %  default noise gain
    k_sat: 2.0;                      %  [v**-1] (default)
  end assignments;
end body;
end turret_azimuth_control;
```

The graphic representation in Fig. 1 is related to the above composite component textual description as follows:

- 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, and in general it encourages good in-line documentation.
- Every component has an `interface` and a `body`. The interface section specification is the same for all component types, while the contents of the body differ from case to case.
- Primary inputs and outputs in the interface correspond to those data-flow arrows that enter and exit the dashed box defining the scope of the CC in Fig. 1; these define the variables that may be connected to in order to build a hierarchical hybrid system using this component.
- Each CC input and output inherits the type of the associated lower-level component variable; in fact, *every* variable incorporated in the system description is typed in the lowest-level component where it is used.
- CC inputs named in the `interface` section must be used to specify component inputs within the system. In the example in Fig. 1, `threat_priority(k)` is a message-vector from an external module telling the `engagement_mgr` module the importance of each potential target, `engage` is a Boolean signaling the system when it is appropriate to slew and track the most threatening target, `disturbance` is a signal input for the `turret` component, `noise` corrupts the signal processed by the `az_filter`, and `threat_az(k)` represents the azimuth angles (a vector) of the various threats being tracked.
- Inputs may be instantiated by connecting this CC to a "driver" (system block) or by using this module as a composite component of a larger system. Such a connection defines the specific nature of the inputs (e.g. how `disturbance` varies with time, the dimension of `threat_az(k)`, when `engage` changes from `false` to `true`, etc.). Specifically note that the dimension `(k)` of vector inputs *may be* established by external definition (system model drivers) if permitted within the component. That is, a component may allow `(k)` to range between `K_min` and `K_max`, or it may specify a specific value. Consistency between the input source and the component constraints would be checked as part of validating the CC when it is assembled prior to simulation.
- Outputs named in the `interface` section must be (in the same statement) connected to an appropriate internal component input or output. In the example in Fig. 1, `azimuth`

is the azimuth angle of the current target being tracked, `noisy_theta` is the noisy input signal to the az_filter, and `track_status` is a message from the `engagement_mgr` indicating track quality. 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.

- The example in Fig. 1 shows two simple operations beyond connection, i.e., multiplication by a gain factor `k_noise` and then addition to inject the signal into the `az_filter` input `az_filter.theta`. It is an open question whether to allow more generality; for example, 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).

- Secondary CC inputs and outputs (not shown in the graphical rendition) are "knobs", usually parameters that can be changed in the course of performing simulation experiments, and "views", variables that can be displayed (e.g., plotted after simulation). These secondary inputs and outputs *cannot* be used for connection purposes. Note that knobs and views may be designated at the component level as well; at whatever level in the model hierarchy an element is designated as a knob or view, it cannot be used as a first-class input or output above that level. Also, note that a component's knob and view variables are always accessible, even if it is embedded in higher-level composite components.

- 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`. As a syntactical matter, it is undecided whether or not to permit unique identifiers to be used without the global variable "dot notation" in defining connections.

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

- Components comprising the CC are named and their source files identified in the `components` section. Note that the component *names* must be unique, but source files may be used repeatedly; thus, for example, the module `/library/filters/kalman.mdl` may be employed in several filtering operations in a composite component or complete hybrid system model.

Note that the above interface structure is common to all component types. Also, the distinction between first- and second-class inputs and outputs is important when analyzing the topology of the system and performing operations such as linearization where unwanted inputs and outputs (artificially incorporated to serve the purposes of knob and view variables) would generally be bothersome.

5.2 Pure Component Model Definitions

The HSML requirements for CTCs and DTCs are defined by proposing suitable extensions to various existing modeling languages. The modeling framework for LBCs is less clear, since

this is a new area for most conventional simulation environments. Appropriate paradigms for LBCs may be discrete-event systems in petri-net or finite-state machine form, expert systems, neural nets, fuzzy logic, or other logic-based discrete-time software; since the internal form of an LBC is open, we merely provide an interface or "shell" for such components.

The basic schema of HSML component modeling can be illustrated by the following examples representing three of the modules that comprise the sample CC depicted in Fig. 1 plus a corresponding system block to act as the signal generator. We focus on the turret drive (represented by the continuous-time component or CTC `turret`), a discrete-time component (DTC) `pid_control` that instantiates a PID (proportional-integral-derivative action) control algorithm, and a (mindless) logic-based component (LBC) shell `engagement_mgr`. The final hybrid system is then assembled using these components plus an SDC `azimuth_control_system` to provide system inputs.

5.2.1 `turret` – A Continuous-Time Component

The following HSML model is a simple example of a nonlinear physical "plant" that might be used for control systems analysis and design. There is one elementary state event, modeling Coulomb friction and handled inside the CTC component where it occurs:

```
CTC turret is
%
% an electric-drive 'turret' example
%
interface
  input(volts,signal,(-vl,vl));  %  voltage-source input
  input(load_dist,signal);       %  load disturbance torque
  input(Ksat,signal);            %  permit varying this "parameter"
  output(theta) = theta;         %  turret azimuth angle
  output(volt_lim) = v_lim;      %  another output variable
  knob(del_theta);               %  permit use of different offsets
  view(current) = curr;          %  this variable may only be displayed
end interface;
%
body
%                    declare internal variables:
  declarations
    state(theta,theta_dot);        %  first state and derivative names
    state(theta_dot,moment);       %  second state and derivative names
    local(moment,(-m_max,m_max));  %  "moment" has a limited range
    flag(theta_dot);               %  declare flag for state event
  end declarations;
%                    now handle initialization:
  initial         %  (this section is executed at start of simulation):
    theta = tank_body.theta + del_theta;  % init turret/tank offset
    sgn = if theta_dot < 0.0 then -1.0
          else if theta_dot > 0.0 then 1.0
          else sign(moment);              % init friction sign
  end initial;
```

```
%                    now handle the state event:
  event(theta_dot)
    positive-going
      sgn = 1.0;
    end positive-going;
    negative-going
      sgn = -1.0;
    end negative-going;
  end event;
%                    now handle system dynamics:
  dynamics
    v_lim = volts/(1.0 + Ksat*abs(volts)); %  input is softly saturated
    curr = (v_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;
%                ... and finally assign parameter values:
  assignments
    vl: 15.0;          %  [volts]
    del_theta: 0.1;    %  [rad]
    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 additional features of the HSML language:

- Interface variables must be identified in the `interface` section, where, as in Section 5.1, they are categorized as `input, output, knob,` and `view`; again, the first pair are *first-class inputs and outputs* that can be connected to other first-class I/O variables, and the second pair are secondary variables that provide access from outside the component but are not connectable. Note that the modeler allowed `Ksat` to be a variable input to the CTC; it is declared to be a knob at the higher level defined in `CC turret_azimuth_control`. In a similar way, `volt_lim` is an output here, but is demoted to being a view variable in the CC.

- The roles of key internal variables must be declared in the `declarations` section, where in this model we have examples of `state, local,` and `flag` variables. The state variables correspond to the dynamic states in the CTC, as in Eqn. (3); the first entity is the state x (e.g., `theta`) and the second its derivative $\dot{x}$ (e.g., `theta_dot`). The purposes of `local` and `flag` variables are described below. Two additional declared variable types are `time` (the scalar variable t in Eqn. 3) and `Tswitch`, used to define the instant a time event occurs; both are illustrated in the SDC component `azimuth_control_system`,

Section 5.3. It is not necessary to declare all internal variables (e.g., `frict` is undeclared).

- The interface variables `input` and `knob` and the internal variables `state` and `local` may be specified to be restricted in range, as shown; for example, `volts` must be in the range (`-vl`, `vl`); the simulator must support checking such restrictions and either warning the user or stopping the simulation if they are violated.

- Introducing the `local` variable classification permits the specification of internal variable characteristics (e.g., type and range); this allows strict control over internal variables without making them external, and makes it unnecessary to add range specifications to `output` and `view` variables.

- Provision could be made for the specification of the precision to be used in representing the continuous-time signals in a CTC; most simulators, however, do not support this level of specification and it is not clear that it would be of sufficient benefit to be worth the extra detail. Modern computers have a single-precision arithmetic that is usually adequate; also, automatic step-size-control algorithms generally reduce or eliminate the need for double-precision arithmetic.

- Support for specifying units of key physical variables might also be provided. This feature 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 not a simple 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. We are presently undecided about this area of support, but tending to be negative.

- A designated section provides for initial condition calculations and other "set-up" evaluations. Here we have specified that the turret angle is initially offset from the `tank_body` center-line 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` ($\ddot{\theta}$). Observe that we are permitting prescribed cases where absolute encapsulation is violated; we introduce a global variable notation to support this.

- A designated section provides for state-event handling (change of sign in the 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 positive-going and negative-going transitions. There would be a separate `event` section for each state event if there were more than one. A detailed discussion of state-event handling is provided in Section 3.5.

- Note that the state-event handling mechanism illustrated here is the internal or "encapsulated" form; this usage assumes that no other module needs to "know" the status of this event (whether the friction term is positive or negative).

- We believe that support for a class of differential-algebraic equations (DAEs), that is, 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 may 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 idea is still tentative. Of course, the simulator would have to support nonlinear equation solving in coordination with numerical integration; DASSL [6] is the best-reputed solver for this class of DAEs. For a more general and rigorous discussion of DAE solvers, see [9, 30].

- The efficiency of the CTC "code" could be improved by introducing an additional section for calculations not required in derivative evaluation. For example, if the output equations (Eqn. 2) are substantial and unrelated to evaluating $\dot{x}$, then they need to be performed only after the numerical integration process is complete rather than at every invocation of the `dynamics` section. We are undecided about adding this feature.

- The differential equations are simply and naturally rendered. Here we have a "chain" of integrators, that is, the derivative of state 1 (`theta`) is state 2 (`theta_dot`); then the derivative of state 2 is `moment`; in mathematical notation, $\ddot{\theta} = M$ where $M =$ `moment` denotes the total moment acting on the turret.

- We have minimized superfluous and / or redundant programming to the extent possible, and have tried to keep the language simple. Simplicity is important because it allows the incorporation of model checking logic in the compiler or simulator that can perform equation sorting, detection of algebraic loops, checking for topological consistency, type constraints, semantic rigor, etc. (see Sections 3.1 and 3.3.2).

- We have designed the tentative syntax in this report to be moderately terse rather than verbose. A more readable or novice-friendly notation might use key words (as in Ada); for example,

```
input(name=>volts,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. Construction of a verbose variant of the HSML delineated here would be a straightforward task. On the other hand, the language is not cryptic, either; for example, the event subsection delimiter `negative-going` could be `ng`, but that would be a nuisance to remember.

5.2.2 `pid_control` – A Discrete-Time Component

The representation of a discrete-time component in HSML is similar to that in the continuous-time case. This similarity is a natural outcome of the fact that DTCs are defined to be the digital analog of a CTC (in the sense that a difference equation is the discrete-time analog of a differential equation). Note that some of the complications arising in the modeling and simulation of continuous-time components do not occur in treating DTCs; for example, there are no state events to worry about. The following is a digital implementation of a PID algorithm[3]:

[3]Note that this controller does not include the logic needed to handle targets that pass through the singularity at $\theta = \pm\pi$; protection is provided by specifying appropriate ranges for the angular inputs.

```
DTC pid_control is
%
% PID controller
%
interface
  input(ref,real,(-pi,pi));       %  reference input
  input(y_sensor,real,(-pi,pi));  %  feedback signal
  input(trk_speed,integer,[1,2]); %  "fast track" indicator
  output(command) = comm;         %  controller output
  knob(K_P);                      %  provide access to vary K_P
  view(deriv) = D;                %  .. and to see the deriv term
end interface;
body
  declarations
    state(integral,new_int,real);   %  numerical integrator state
    state(old_y,y_sensor);          %  "old sensor data" state
    time(T);                        %  current time
    Tsample(Ts);                    %  time of next sample
    Tdelay(Td);                     %  delay time from sample to output
    Priority(2);                    %  priority 1 modules execute first
  end declarations;
%                    now handle initialization:
  initial              %  (this is done at beginning of simul):
    old_y = y_sensor;  %  set the derivative est = 0.0 at first sample
  end initial;         %  ends INITIAL section
%                    begin difference equations:
  update
    error = ref - y_sensor;         %  error signal (summing junction)
    ratio = 1.0 / real(track_speed);
    DT = DT_nom * ratio;            %  sampling time depends on 'speed'
    P = K_P * error;                %  proportional term
    D = T_D * (y_sensor - old_y)/DT; %  derivative term (rate feedback)
    new_int = integral + error*DT/T_I; %  update integral term, and ...
    Ts = T + DT;                    %  update time for next sample
    comm = P + new_int + D;         %  controller output voltage
  end update;
%                   ... and finally assign parameter values:
  assignments
    pi: 3.14159;
    K_P: 1.0;              %  `proportional' gain
    T_D: 0.3;              %  `derivative' gain times DT
    T_I: 1E1;              %  1/`integral' gain
    DT_nom: 0.1;           %  nominal sampling interval
    Td: 0.0125;            %  output delay
  end assignments;
end body;
end pid_control;
```

The elements of this DTC are similar to those of CTCs, as illustrated in the previous section. The in-line comments provide a guide to many important features. Note that the HSML description rigorously distinguishes between the 'state' x_k and the 'new state' x_{k+1}, which are the first and second arguments of the `state` statements, respectively. This feature (taken directly from Simnon [18, 21]) prevents a common problem of updating higher-numbered states on the basis of lower-numbered states that have already been updated, which destroys the integrity of the algorithm (Eqns. 9, 10). In addition, we note the following features:

- The DTC component `interface` section is based on exactly the same prescription as in the CTC case.
- Internal declared variables include `state` (x_k, Eqn. 9); `time` or t; `Tsample` or time of next sample (internally specified in the component); `Tdelay` or δ_k, Eqn. (10); and `Priority`, which establishes the precedence ordering of discrete-time modules that might be invoked "simultaneously". In this example, other Priority 1 LBC or DTC interrupts are processed before `pid_control`.
- The provision for writing the DTC difference equations is also simple and natural, as in the CTC case. The fact that the "new" value of the state `old_y` is declared to be `y_sensor` (a module input from the component `az_filter`) eliminates the need to specify in an explicit update statement that `old_y = y_sensor`.
- The key variables may be restricted in value, as shown; for example, `ref` must be in the range $(-\pi, \pi)$ and the integer `trk_speed` may take on values only in the range $[1, 2]$. Again, the simulator would have to support checking such restrictions.
- Each DTC has its own built-in "timer"; each time a DTC is invoked, it tells the simulator when it is to be called again (via the `Tsample` variable `Ts`), placing the corresponding event time on the simulator's "time-event stack". Time between samples may be variable, as in this example.
- The DTC makes provision for a constant time delay, as outlined in Section 3.1. The data is sampled at each time specified by `Ts` and used to calculate the next `command`; the `pid_control` output is only made available to the rest of the hybrid system (i.e., to `turret`) after a fixed delay of `Td` $= 0.0125$ seconds. Note that the simulator should check that the delayed output should be available before the time of the next sample, to avoid a pathological condition.

5.2.3 `engagement_mgr` – A Logic-Based Component

The last component type in the multi-block HSML model portrayed in Fig. 1 is the logic-based component (LBC). As mentioned previously, the internal representation of such a component is not specified at this time. However, we can define the **shell** for such a component and leave the representation of the internal behavior arbitrary:

```
LBC engagement_mgr is
%
%   provide a shell for a logic-based component
%
interface
```

```
  input(prio(1..k_max),string,{high,medium,low}); %  threat priorities
  input(aim_it,boolean);                   %  flag to engage a target
  input(az_trk,real);                      %  turret azimuth angle
  output(which_one) = worst_threat;        %  ID of target to track
  output(precision) = prec;                %  track precision flag
end interface;
body
  declarations
    flag(trigger);                         %  condition for execution
    Tdelay(Td);                            %  time-delay for execution
    Priority(1);                           %  interrupts processed first
  end declarations;
%                      now handle initialization:
  initial
    %  do whatever preliminary things need to be done ...
  end initial;
%                      now perhaps handle a state event:
  event(trigger)
    %  express invocation condition as a state-event zero-crossing:
    trigger = . . . .
    %  one may specify a delay-time (Td) as the time between the
    %  event and the delivery of a result.
  end event;
%                      and now begin logic here:
  logic
    %  we leave it up to your imagination how this module decides to
    %  pick and engage a target, and determines track precision:
    worst_threat = . . .;
    prec = . . .;
  end logic;
  assignments
    k_max: 20;          %  logic handles no more than 20 targets
    Td: 0.0375;         %  takes 0.0375 seconds to deliberate
  end assignments;
end body;
end engagement_mgr;
```

5.3 Completing a Hybrid System Model

The following HSML SDC module provides the input variables necessary to run the CC model `turret_azimuth_control` and thus can be used to complete the definition of a simple hybrid system:

```
SDC azimuth_control_system is
interface
  view(azimuth) =
      output(turret_azimuth_control.azimuth);
  view(noisy_theta) =
```

```
      input(turret_azimuth_control.noisy_theta);
  view(track_status) =
      output(turret_azimuth_control.track_status);
  knob(T_aim);
  knob(disturb_1);
  knob(threat_az_1(2));
end interface;
body
  declarations
    time(T);
    Tswitch(T_aim);                %  these define time events
    Tswitch(T_dist);               %  (see below)
  end declarations;
  connections
    % attach the SDC to the CC turret_azimuth_control:
    input(turret_azimuth_control.threat_priority(k)) = t_p(5);
    input(turret_azimuth_control.engage) = engg;
    input(turret_azimuth_control.disturbance) = dist;
    input(turret_azimuth_control.noise) = noise;
    input(turret_azimuth_control.threat_az(k)) = t_a(5);
  end connections;
  drivers
    t_p = [ low, high, medium, low, medium ];
    engg = if T < T_aim then false else true;
    dist = if T < T_dist then disturb_0 else disturb_1*sin(T);
    noise = gaussian_random_process(0.0,0.25,0.01);
    for k in 1..5 loop
      t_a(k) = threat_az_0(k) + threat_az_1(k)*T;
    end loop;
  end drivers;
  assignments
    T_aim: 2.34;
    T_dist: 4.32;
    disturb_0: -.123;
    disturb_1: .231;
    threat_az_0: [ -1.23, -.5, 0., .76, 1.92 ];
    threat_az_1: [ -.13, .15, 0., -.15, .02 ];
  end assignments;
  components
    turret_azimuth_control(az_ctrl.mdl);
  end components;
end body;
end azimuth_control_system;
```

Much of the above syntax and structure is tentative; however, note the following:

- An SDC may not contain inputs or outputs; knobs are permitted (to define simulation experiments), as are views (to allow display of simulation data). Note that knob and view entities of lower-level components are also accessible within the simulation

environment.

- The SDC `drivers` should correspond exactly to the highest-level composite component(s) `input` list(s). The type of each SDC driver is checked against the corresponding component input where it is ultimately consumed.

- The SDC completely instantiates the hybrid system. In addition to supplying scalar and vector signals to drive the composite component(s), it defines the *dimension* of the vectors (subject, in this example, to the LBC's internal specification that `k` must not exceed 20).

- The above assumes the existence of a gaussian noise generator with user-supplied specifications for mean, rms level, and sample time, respectively.

- Time-varying outputs may contain time events; these events are made known to the simulator via the `Tswitch` entities. The simulator must stop at these times exactly and exercise the SDC to change the associated variable. The significance and handling of `Tswitch` elements is thus exactly similar to the `Tsample` elements in a DTC; note, however, that the SDC is generally operated as a continuous-time component.

- Handling of vector signals above is highly tentative.

6 Conclusions

A quality hybrid system simulation environment (QHSSE) will have broad usefulness in both commercial and government applications, as outlined in Section 1.3.1. In fact, it is obvious that the use of modeling and simulation is not just confined to controls applications, although that is the application with which ORA is most familiar and we would be satisfied with commercial viability on the basis of that market alone. The surveys of user needs and available methods and software for modeling and simulation as outlined in the previous Task Statements will be designed to permit us to understand and develop the broadest possible user base; the breadth of this marketplace is a major consideration in our strong positive assessment of the commercial importance of this proposal.

We expect to have decided either to choose a partner or proceed alone during Phase I. If possible, and if a collaboration makes sense, we anticipate that at least an outline of such a partnership will be established. We believe tentatively that collaboration would give us several advantages in comparison with a solo approach, because

1. it is easier and less risky to increase an existing market share instead of starting from zero, and
2. it is conceivable that a limited Version 1.0 of QHSSE might be available by or just after the end of Phase II.

For these reasons, unless we fail to achieve an equitable partnership or an unforeseen technical or business reason negates these positive considerations, we plan to form an alliance in Phase II and proceed into Phase III with internal ORA support, support in kind from our commercial partner, and perhaps a modest revenue stream from an extended version of an existing commercial simulation package.

7 References

[1] *Advanced Continuous Simulation Language (ACSL), Reference Manual.* Mitchell & Gauthier Associates, Concord, MA 01742.

[2] Amsterdam, J., "Automated Modeling of Physical Systems", ASME Winter Annual Meeting, Anaheim, CA, November 1992; in *Automated Modeling*, ASME Publication DSC-Vol. 41.

[3] Ascher, U. M. and Spiteri, R. J., "Collacation Software for Boundary Value Differential-Algebraic Equations", *SIAM Journal of Scientific Computation*, to appear.

[4] 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.

[5] *AutoCode User's Guide*, Integrated Systems, Inc., Santa Clara, CA 95054.

[6] Brenan, K. E., Campbell, S. L. and Petzold, L. R., *Numerical Solution of Initial Value Problems in Differential-Algebraic Equations*, North Holland, 1989.

[7] Campbell, S. L., "A General Form for Solvable Linear Time Varying Singular Systems of Differential Equations", *SIAM J. on Mathematical Analysis*, Vol. 18, pp. 1101-1115, 1987.

[8] Campbell, S. L., "Least Squares Completions for Differential Algebraic Equations", *Numerische Mathematik*, Vol. 65, pp. 77-94, 1993.

[9] Campbell, S. L., "High Index Differential Algebraic Equations", *J. of Mechanical Structures and Machines*, to appear.

[10] Campbell, S. L. and Gear, C. W., "The Index of General Nonlinear DAEs", preprint/report, Dept. of Math, North Carolina State University, Raleigh, NC 27695-8205, November 1993; *slc@math.ncsu.edu.*

[11] Campbell, S. L. and Griepentrog, E., "Solvability of General Differential Algebraic Equations", *SIAM Journal of Scientific and Statistical Computation*, to appear.

[12] Campbell, S. L. and Moore, E., "Progress on a General Numerical Method for Nonlinear Higher Index DAEs II", *Circuits, Systems and Signal Processing*, Vol. 13, No. 2-3, pp. 123-138, 1994.

[13] Campbell, S. L. and Moore, E., "Constraint Preserving Integrators for General Nonlinear Higher Index DAEs", *Numerische Mathematik*, to appear.

[14] Cellier, F. E., "Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools", PhD Thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, Number ETH 6438, 1979.

[15] Cellier, F. E., "Combined Continuous/Discrete Simulation - Applications, Techniques and Tools", *Proc. Winter Simulation Conference*, Washington, DC, pp. 24-33, 1986.

[16] Cellier, F. E., *Continuous System Modeling*, Springer Verlag, 1991.

[17] Cellier, F. E., Elmqvist, H., Otter, M. and Taylor, J. H., "Guidelines for Modeling and Simulation of Hybrid Systems", *Proc. IFAC World Congress*, Sydney, Australia, 18–23 July 1993.

[18] Elmqvist, H., "SIMNON - An Interactive Simulation Program for Non-Linear Systems", *Proc. of Simulation '77*, Montreux, France, 1977.

[19] Elmqvist, H., Cellier, F. E. and Otter, M., "Object-Oriented Modeling of Hybrid Systems", *Proc. ESS'93, SCS European Simulation Symposium*, Delft, The Netherlands, 1993.

[20] Filipov, A. F., *Differential Equations With Discontinuous Right Hand Side*, Kluwer Academic Press, 1988. (See also article of the same name, AMS Translations, 42:199-231, 1964.)

[21] Frederick, D. K. and Taylor, J. H., "SIMNON Reference Manual", GE Corporate Research and Development, 1989.

[22] Gear, C. W., *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, 1971.

[23] Gear, C. W., "The Simultaneous Numerical Solution of Differential-Algebraic Equations", *IEEE Transactions on Circuit Theory*, Vol. TC-18, pp. 89-95, 1971.

[24] Gear, C. W., "Differential Algebraic Equation Index Transformations", *SIAM J. of Scientific and Statistical Computation*, Vol. 9, pp. 39-47, 1988.

[25] Gear, C. W. and Petzold, L. R., "ODE Methods for the Solution of Differential/Algebraic Systems", *SIAM J. of Numerical Analysis*, Vol. 21, pp. 367-384, 1984.

[26] Hassard, B. D., Kazarinoff, N. D. and Wan, Y-H., *Theory and Applications of Hopf Bifurcations*, Cambridge University PJress, 1981.

[27] *ISEE Interface to Simnon*, A. Marttinen, Control CAD, Espoo, Finland.

[28] Karnopp, D. C., Margolis, D. and Rosenberg, R. C., *System Dynamics: A Unified Approach*, John Wiley & Sons, New York, 1990.

[29] Mattsson, S. E. and Andersson, M., "The Ideas Behind Omola", *Proc. CACSD'92, IEEE Computer-Aided Control Systems Design Conference,* Napa, CA, pp. 218–224, March 17–19, 1992.

[30] 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, CA, pp. 218–224, March 17–19, 1992.

[31] Mattsson, S. E. and Söderlind, G., "Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives", *SIAM J. of Scientific and Statistical Computation*, Vol. 14, pp. x-x, 1993.

[32] *Model-C User's Guide*, Systems Control Technology, Inc., Palo Alto, CA 94303.

[33] Pantelides, C. C., "The Consistent Initialization of Differential-Algebraic Systems", *SIAM Journal of Scientific and Statistical Computation*, Vol. 9, No. 2, pp. 213–231, 1988.

[34] Petzold, L. R., "A Description of DASSL: A Differential/Algebraic System Solver", *Proc. 10th IMACS World Congress*, Montreal, August 8-13, 1982.

[35] Potra, F. A. and Rheinboldt, W. C., "Differential-Geometric Techniques for Solving Differential Algebraic Equations", in *Real-Time Integration Methods for Mechanical System Simulation*, Ed. by E. J. Haug and R. C. Deyo, Springer-Verlag Computer & Systems Sciences Series, Vol. 69, pp. 155-191, 1991.

[36] Rabier, P. J. and Rheinboldt, W. C., "A General Existence and Uniqueness Theorem for Implicit Differential Algebraic Equations", *Diff. Int. Eqns.*, Vol. 4, pp. 563-582, 1991.

[37] Rasband, S. N., *Chaotic Dynamics of Nonlinear Systems*, John Wiley & Sons, 1990.

[38] Reich, S., "On an Existence and Uniqueness Theory for Differential Algebraic Equations", *Circuits, Systems and Signal Processing*, Vol. 10, pp. 343-359, 1991.

[39] Redfield, R. C., "Bond Graphs as a Tool in Mechanical System Conceptual Design", ASME Winter Annual Meeting, Anaheim, CA, November 1992; in *Automated Modeling*, ASME Publication DSC-Vol. 41.

[40] *SimuLink User's Guide*, The MathWorks, Inc., Natick, MA 01760.

[41] Steward, D. "Partitioning and Tearing Systems of Equations", *SIAM J. on Numerical Analysis*, Series B, Vol. 2, No. 2, pp. 345-365.

[42] *SystemBuild User's Guide*, Integrated Systems, Inc., Santa Clara, CA 95054.

[43] Tarjan, R. E., "Depth First Search and Linear Graph Algorithms", *SIAM J. on Computing*, Vol. 1, pp. 146-160, 1972.

[44] Taylor, J. H., *User's Guide for MEAD Computer Program Version 1*, GE Corporate R & D, 29 January 1989. See also: J. H. Taylor, D. K. Frederick, C. M. Rimvall, and H. A. Sutherland, "The GE MEAD Computer-Aided Control Engineering Environment", *Proc. IEEE Symposium on CACSD,* Tampa, FL, December 1989.

[45] Taylor, J. H., "Toward a Modeling Language Standard for Hybrid Dynamical Systems", *Proc. 32nd IEEE Conference on Decision and Control,* San Antonio, TX, 15-17 December 1993.

[46] Taylor, J. H., "A Modeling Language for Hybrid Systems", *Proc. CACSD94* (IEEE/IFAC Symposium on Computer-Aided Control System Design), Tucson, AZ, 7-9 March 1994.

[47] Taylor, J. H., *A Rigorous Modeling Language for Hybrid Dynamical Systems*, ORA Report TM-94-001, ARPA/US Army Contract No. DAAA21-92-C-0013, February 1994.

[48] J. H. Taylor and A. J. Antoniotti, "Linearization Algorithm for Computer-Aided Control Engineering", *IEEE Control Systems Magazine, Vol. 13,* No. 2, pp. 58-64, April 1993.

[49] J. H. Taylor and J. Lu, "SIDF-Based Nonlinear Control System Synthesis for an Electro-Mechanical Pointing System", accepted for Special Issue on Motion Control Systems, *Journal of Systems Engineering*.

[50] *VisSim User's Guide*, Visual Solutions, Inc., Westford, MA 01886.