

Simulation of Hybrid Systems

Michael S. Branicky* and Sven Erik Mattsson**

Abstract. Hybrid systems—those composed of the interaction of discrete and continuous inputs, outputs, states, and dynamic equations—are an important class of models of complex, real-world phenomena. However, the simulation tools currently available seem to be (1) *ad hoc* retrofitting of existing packages, (2) hastily-built new languages, or (3) specialized software for particular subclasses (e.g., piecewise-constant dynamics). Our goal is to produce fast, high fidelity simulations of (networks of) a very broad class of hybrid systems in a user-friendly environment. In this paper, we first review expertise in the mathematical modeling of hybrid systems, viz. the hybrid dynamical systems of Branicky (HDS). Also, we discuss the object-oriented modeling and simulation of combined discrete/continuous systems using the Omola modeling language and Omsim simulation environment developed over the last eight years at Lund. Leveraging these, we are led to our main contribution: a general set of hybrid systems model classes which encompass HDS and hence several other models popularized in the literature that combine finite automata and discrete event systems with ordinary differential (ODEs) and differential algebraic equations (DAEs). These Omola model classes may be viewed as “templates” or “macros” for quick and easy entering of hybrid systems for subsequent analysis and numerically-sophisticated simulation using Omsim.

1 Introduction

Colloquially, a hybrid system has come to mean a system which is an amalgamation of continuous and discrete inputs, outputs, states, and dynamic equations. See Figure 1. Hybrid systems are those in which a melding of two worlds—the analog and the digital—exists, and they are intertwined to the extent that a “one-world” description is not desired, not tractable, or not possible. Such systems seem to naturally arise in a variety of applications due to autonomous or controlled phenomena.

In the autonomous case, the system evolution itself may fall naturally into a finite number of different phases (a.k.a. modes or epochs), between which abrupt changes in continuous dynamics (switching) or continuous states (jumps or resets) occur. In the controlled case, a simple finite state machine may be used to regulate a physical process, such as may arise even in a simple thermostat.

* Dept. of Electrical Eng. and Applied Physics, Case Western Reserve U., 10900 Euclid Ave., Glennan 515B, Cleveland, OH 44106-7221 USA. branicky@eeap.cwru.edu

** Dept. of Automatic Control, Lund Institute of Technology, PO Box 118, S-221 00, Lund, SWEDEN. svenerik@control.lth.se

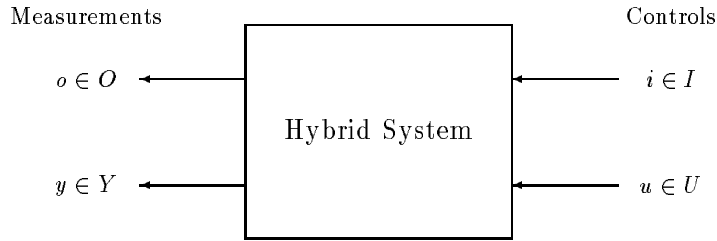


Fig.1. Hybrid Control System.

In more complicated situations, a mixture of autonomous and controlled phenomena may be present. See Figure 1. Alternatively, a hybrid dichotomy may be invoked as a means of dealing with complexity through abstraction, e.g., using logic to switch among various continuous controllers (each with predictable behavior when used alone) in order to accomplish more global objectives, as in mode-switched aircraft. Of course, such abstractions may occur multiple times, leading to layered or hierarchical control schemes seen in robotics and communication networks.

We have already mentioned some real-world examples of hybrid systems. Application areas include power electronics (state-dependent circuit switching), motion control (disk drives, transmissions, stepper motors, position encoders), robotics (constrained robots, flexible manufacturing, interacting agents), intelligent transportation systems (automated highway systems, personal rapid transit), and aerospace (mode-switched flight, vehicle management systems, air traffic control). The reader may imagine more or consult the literature [8, 12, 3, 1].

Researchers in hybrid systems have been trying to build a theory for such systems in recent years. However, it is still a challenge to even accurately simulate them because of the sophistication needed to do mixed continuous/discrete simulation in a timely manner. Since these are important areas and examples, we begin here to develop a framework and a collection of computer tools to meet the hybrid systems simulation challenge.

Simulation is not a means unto itself, though, but a means to understanding. Backtrack a moment and consider a slightly more refined definition of a hybrid system which consists of a finite automaton or discrete-event system “supervising” the action of a collection of ODEs or DAEs by giving commands for when to switch between them, and how to update variables upon switching. See Figure 3. It is an instantiation of the digitally-regulated plant and two-layer control schemes mentioned above. However, it has been shown that even low-dimensional systems of this type possess rich behavioral possibilities, including the power of universal computation in as little as three dimensions [8]. Further, most engineering insight and tools have been developed for “one-world” scenarios.

Thus, with hybrid systems, we are also faced with low intuition and high complexity. In such a situation, we believe that simulation is an important tool

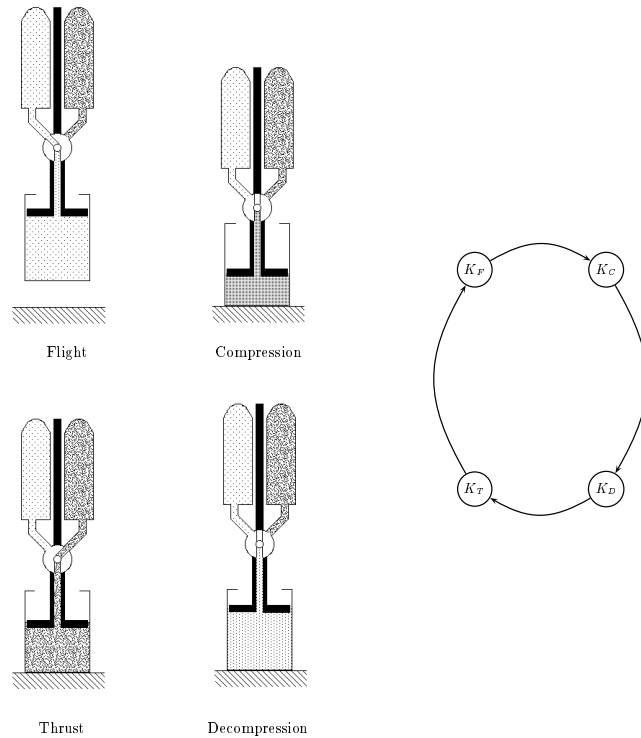


Fig. 2. Raibert's hopping robot: dynamic phases and controller.

to gaining intuition about, and a stepping stone to the automatic analysis of, hybrid systems. Further, we believe that many of the simulation tools currently available are lacking in some respect. Without classifying tools, they seem fall into one of the following three categories: (1) *ad hoc* retro-fitting of existing “one-world” packages, (2) hastily-built new languages, or (3) specialized software for particular subclasses (e.g., piecewise-constant dynamics). Therefore, it is the goal of this research to *produce fast, high fidelity simulations of (networks of) a very broad class of hybrid systems in a user-friendly environment*.

Our approach builds on a modeling language and simulation package, Omola and Omsim resp., designed from inception to handle mixed discrete/continuous simulation. Omola/Omsim uses a sophisticated mix of symbolic equation manipulation, solution approximation, zero-finding, and ODE/DAE solvers to accurately perform such mixed simulations. Omola/Omsim has been under research and development—and test through numerous projects, theses, and users—since 1989 in the CACE group in the Department of Automatic Control at Lund Institute of Technology. In this work, we have added a library of hybrid sys-

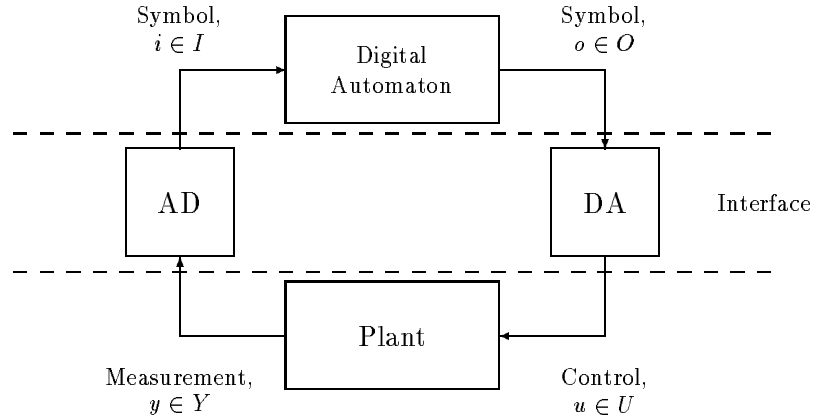


Fig. 3. Hybrid System.

tems *model classes*, which may be thought of as a set of templates or “macros,” designed for the quick specification and simulation of a broad class of hybrid systems, viz. Branicky’s hybrid dynamical systems (HDS). In previous work [8], HDS were shown explicitly to encompass various hybrid phenomena as well as six popular models of hybrid systems previously proposed in the literature [4, 6, 10, 13, 16, 17]. Given this, our macros may be used to easily simulate these systems as well. Indeed, we give explicit *model subclasses* suited to the purpose of simulating several interesting subclasses of hybrid systems, including those with autonomous switching [8], that due to Brockett [10], and a new one due to Artstein [5].

The paper is organized as follows. In the next section, we review HDS and present some examples used throughout. In Section 3, we discuss Omola and Omsim, introducing some of their philosophy, syntax, and capabilities. Section 4 presents our main contributions: two different instantiations of HDS model classes, as well as the model subclasses and examples mentioned above. Section 5 gives conclusions.

2 Hybrid Dynamical Systems

2.1 Background and Motivation

Hybrid systems involve both continuous-valued and discrete variables. Their evolution is given by equations of motion that generally depend on both. In turn these equations contain mixtures of logic, discrete-valued or *digital* dynamics, and continuous-variable or *analog* dynamics. The continuous dynamics of such systems may be continuous-time, discrete-time, or mixed (sampled-data), but is generally given by differential equations. The discrete-variable dynamics of hybrid systems is generally governed by a *digital automaton*, or input-output

transition system with a countable number of states. The continuous and discrete dynamics interact at “event” or “trigger” times when the continuous state hits certain prescribed sets in the continuous state space. See Figure 3. *Hybrid control systems* are control systems that involve both continuous and discrete dynamics and continuous and discrete controls. The continuous dynamics of such a system is usually modeled by a controlled vector field or difference equation. Its hybrid nature is expressed by a dependence on some discrete phenomena, corresponding to discrete states, dynamics, and controls. The result is a system as in Figure 1.

In this paper, our focus is on the case of hybrid systems where the continuous dynamics is modeled by a differential equation³

$$\dot{x}(t) = \xi(t), \quad t \geq 0. \quad (1)$$

Here, $x(t)$ is the *continuous component* of the state, taking values in some subset of a Euclidean space. $\xi(t)$ is a *controlled vector field* that generally depends on $x(t)$, the *continuous component* $u(t)$ of the control policy, and the aforementioned discrete phenomena. We have classified the discrete phenomena generally considered into four types [9]:

1. autonomous switching, where the vector field $\xi(\cdot)$ changes discontinuously when the state $x(\cdot)$ hits certain “boundaries”;
2. autonomous impulses, where $x(\cdot)$ jumps discontinuously on hitting prescribed regions of the state space;
3. controlled switching, where $\xi(\cdot)$ changes abruptly in response to a control command; and
4. controlled impulses, where $x(\cdot)$ changes discontinuously in response to a control command.

Two examples that we will use throughout are

Example 1 (Hysteresis). Consider a system with hysteresis: $\dot{x} = H(x)$, where H is shown in Figure 4.

Note that this system is not just a differential equation whose right-hand side is piecewise continuous. There is “memory” in the system, which affects the vector field’s value. Indeed, the system naturally has a finite automaton associated with the function H , as shown in Figure 5.

Example 2 (Planar Autonomous Switching).

$$\dot{x}(t) = M_{q(t)}x(t), \quad x(t) \notin A_{q(t)}, \quad (2)$$

$$q(t^+) = \nu(x(t), q(t)), \quad x(t) \in A_{q(t)} \quad (3)$$

where $q \in \{1, 2, \dots, N\}$, $M_q \in R^{2 \times 2}$, and $A_q \subset \mathbf{R}^2$ for each q .

A particular instantiation is shown in Figure 6.

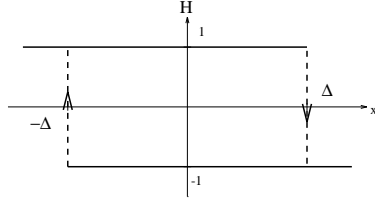


Fig. 4. Hysteresis Function.

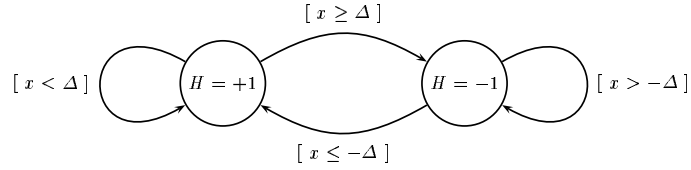


Fig. 5. Finite Automaton Associated with Hysteresis Function.

We also briefly describe two nice classes of hybrid systems from the literature. For pedagogic reasons, we have simplified the models by making them autonomous and changed notation.

Example 3 (Brockett's Model).

$$\dot{x} = f(x, p, z), \quad (4)$$

$$\dot{p} = r(x, p, z), \quad (5)$$

$$z[p] = \nu(x, p, z[p]). \quad (6)$$

where $x(t) \in X \subset \mathbf{R}^n$, $p(t) \in \mathbf{R}$, and the *rate equation* r is nonnegative for all arguments.

Brockett has mixed continuous and “symbolic” controls by the inclusion of the special *counter* variable p . The last equation means that z is updated whenever p passes through integer values.

Example 4 (Artstein's Model). Artstein's model consists of a finite number, N , of modules each of which executes its own dynamics, $\dot{x} = f_q(x)$ for a given time T_q . After timeout, execution switches to a different module depending on whether a test function, $\psi_q(x)$ is non-negative or not. In equations, one might write

$$\begin{aligned} [\dot{x}(t), \dot{\tau}(t)]^T &= [f_{q(t)}(x(t)), 1]^T, & 0 \leq \tau \leq T_{q(t)}, \\ q(t^+) &= \begin{cases} G_p(q(t)), & \psi_q(t)(x(t)) \geq 0, \\ G_n(q(t)), & \psi_q(t)(x(t)) < 0. \end{cases}, & \tau = T_{q(t)}, \\ \tau(t^+) &= 0, & \tau = T_{q(t)}. \end{aligned}$$

where $x(t) \in X \subset \mathbf{R}^n$, and $q(t)$, $G_p(q(t))$, and $G_n(q(t)) \in \{1, 2, \dots, N\}$.

³ Please refer to [8] for background, references, and details.

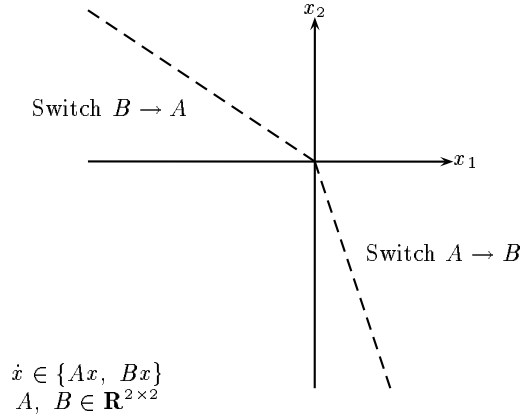


Fig. 6. Example planar switching rule; switches occur when state hits dotted lines.

2.2 A Mathematical Model

We now introduce a mathematical model of hybrid systems that is general enough to

- cover significant phenomena found in real-world examples and
- encompass other reasonable models,

but is specific enough

- to build on previous insight and results
- to be able to derive new theory.

We build on the much-studied notion of dynamical system. Briefly, a *dynamical system* [14] is a system

$$\Sigma = [X, \Gamma, \phi],$$

where X is an arbitrary topological space, the *state space* of Σ . The *transition semigroup* Γ is a topological semigroup with identity. The *extended transition map* $\phi : X \times \Gamma \rightarrow X$ is a continuous function satisfying the identity and semigroup properties [15]. We will also denote by dynamical system the system

$$\Sigma = [X, \Gamma, f],$$

where X and Γ are as above, but the *transition function* f is the *generator* of the extended transition function ϕ .

In this paper, we restrict our attention to the case where X is a subset of \mathbf{R}^n for some $n \in \mathbf{N}$, $\Gamma = \mathbf{R}_+$, and the dynamics are given by vector fields: $\dot{x} = f(x)$.

Example 5. Recall a linear system, $\dot{x} = Ax$, has $\phi(x, t) = e^{At}x$.

Briefly, a hybrid dynamical system is an indexed collection of dynamical systems along with some map for “jumping” among them (switching dynamical system and/or resetting the state). This jumping occurs whenever the state satisfies certain conditions, given by its membership in a specified subset of the state space. Hence, the entire system can be thought of as a sequential patching together of dynamical systems with initial and final states, the jumps performing a reset to a (generally different) initial state of a (generally different) dynamical system whenever a final state is reached.

Formally, a *controlled hybrid dynamical system (CHDS)* is a system $H_c = [Q, \Sigma, \mathbf{A}, \mathbf{G}, \mathbf{C}, \mathbf{F}]$, with constituent parts as follows.

- $Q \subset \mathbf{N}$ is the set of *index or discrete states*.
- $\Sigma = \{\Sigma_q\}_{q \in Q}$ is the collection of controlled dynamical systems, where each

$$\Sigma_q = [X_q, \mathbf{R}_+, f_q, U_q, Y_q]$$

is a controlled dynamical system. Here, the $X_q \subset \mathbf{R}^{d_q}$, $d_q \in \mathbf{N}$, are the *continuous state spaces* and the vector fields $f_q : X_q \times U_q \rightarrow \mathbf{R}^{d_q}$ represent the (controlled) *continuous dynamics*. $U_q \subset \mathbf{R}^{m_q}$ and $Y_q \subset \mathbf{R}^{p_q}$ are the *input and output spaces*, resp., with $m_q, p_q \in \mathbf{N}$.

- $\mathbf{A} = \{A_q\}_{q \in Q}$, $A_q \subset X_q$ for each $q \in Q$, is the collection of *autonomous jump sets*.
- $\mathbf{G} = \{G_q\}_{q \in Q}$, where $G_q : A_q \rightarrow S$ is the *autonomous jump transition map*, said to represent the *discrete dynamics*.
- $\mathbf{C} = \{C_q\}_{q \in Q}$, $C_q \subset X_q$, is the collection of *controlled jump sets*.
- $\mathbf{F} = \{F_q\}_{q \in Q}$, where $F_q : C_q \rightarrow 2^S$, is the collection of *controlled jump destination maps*.

Thus, $S = \bigcup_{q \in Q} X_q \times \{q\}$ is the *hybrid state space* of H_c . Roughly, the dynamics of H_c are as follows. The system is assumed to start in some hybrid state in $S \setminus A$, say $s_0 = (x_0, q_0)$. It evolves according to $f_{q_0}(\cdot, u)$ until the state enters—if ever—either A_{q_0} or C_{q_0} at the point $s_1^- = (x_1^-, q_0)$. If it enters A_{q_0} , then it *must* be transferred according to transition map $G_{q_0}(x_1^-)$. If it enters C_{q_0} , then we *may* choose to jump and, if so, we may choose the destination to be any point in $F_{q_0}(x_1^-)$. Either way, we arrive at a point $s_1 = (x_1, q_1)$ from which the process continues. See Figure 7.

The case where the sets U_q , \mathbf{C} , and \mathbf{F} above are empty is simply a *hybrid dynamical system (HDS)*: $H = [Q, \Sigma, \mathbf{A}, \mathbf{G}]$. In equations, it might look as follows:

$$\left. \begin{aligned} \dot{x}(t) &= f(x(t), q(t)), & x(t) &\notin A_{q(t)} \\ q(t^+) &= G_q(x(t), q(t)) \\ x(t^+) &= G_x(x(t), q(t)) \end{aligned} \right\}, \quad x(t) \in A_{q(t)}$$

Note: The case of GHDS with $|Q|$ finite is a coupling of finite automata and differential equations and includes several previously posed hybrid systems models [4, 6, 10, 13, 16, 17], systems with impulse effect [7], and hybrid automata [11]. For more details, refer to [8].

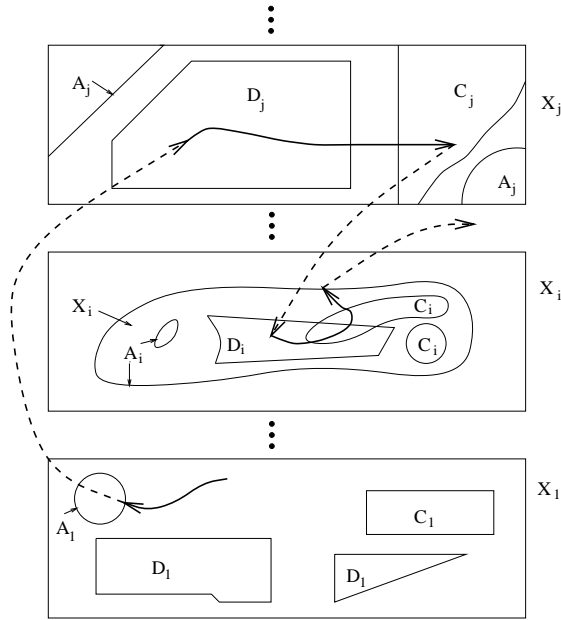


Fig. 7. Example dynamics of controlled hybrid dynamical system.

3 Omola and Omsim

3.1 Omola

Omola stands for “Object-oriented Modeling Language.” It has been developed in the CACE group of the Dept. of Automatic Control at the Lund Institute of Technology. Development started in 1989. A recent, comprehensive description may be found in [2], which is the definitive reference for this summary.

Omola is a language for describing dynamic models. It is

- user-oriented, that is, it is intended that engineers specify models in Omola, not that computer scientists program them.
- high-level and textual.
- supports reuse, through an object-oriented philosophy.
- directly usable in simulation and analysis tools, e.g., Omsim.

Omola supports mixed continuous evolution and discrete events.

Omola has an object-oriented modeling philosophy. Models are to be perceived as *declarative*, encoding facts and relations. They are not thought of as procedures with inputs, outputs, and flow of information. Omola also supports acausal physical models. Thus,

$$\begin{aligned}
 R \times I &= V, \\
 R &= V/I, \\
 V/R &= I,
 \end{aligned}
 \tag{7}$$

are all equally valid descriptions of a resistor.

Omola supports *modularity*. An Omola model encapsulates state and behavior much like an object in C++ encapsulates data and procedures.

Omola supports *abstract interfaces*, allowing one to use a model (as part of a bigger one) without knowing all details about its definition. However, there is no strict information hiding in Omola; access to all internal variables is always possible.

It is the view of Omola that *models are classes*. That is, *models are descriptions of a system type, not a representation of a particular system*. Going back to our resistor example, any one of the equations in (7) is a description of a system type, viz. **Resistor**. A particular resistor would be an instantiation of this model that must add the fact that, for example, $R = 2\text{k}\Omega$. Note that before a model can be simulated, all variables must be fully instantiated (this is checked automatically by Omsim).

Omola supports model *inheritance* as a mechanism for information sharing and reuse. We note that inherited attributes may be selectively over-ridden. Also, strictly speaking, Omola only supports single inheritance. However, a composite model may be composed of two parts, each of which inherits structure from other models. We call this *indirect inheritance* and will see an example below.

An inheritance diagram for an electric component library appears in Figure 8. There, **Class** and **Model** are basic Omola classes, which the user may think of as “empty slots.” Omola has several predefined terminal classes, including **SimpleTerminal** for connecting *across* or *effort* variables like voltage; **ZeroSumTerminal** for *through* or *flow* variables like current; and **RecordTerminal** which consists of several terminals, in order to represent an aggregate of interaction variables.

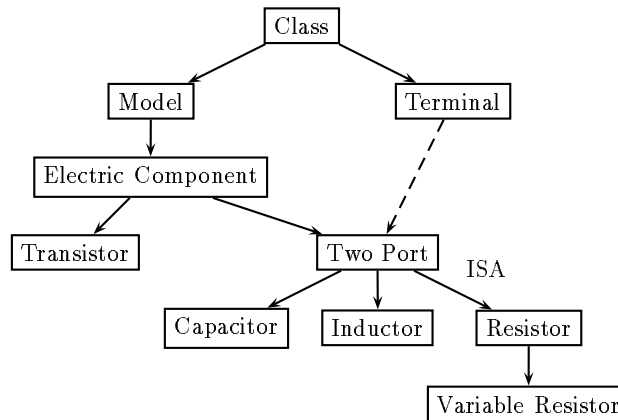


Fig. 8. Class hierarchy for electrical components.

Example 6 (Two Port). The `TwoPort` class in Figure 8 might include as subclasses two `RecordTerminal`'s, each composed of a `SimpleTerminal` (for `V1` and `V2`) and a `ZeroSumTerminal` (for `I1` and `I2`).

The former example uses the aforementioned mechanism of indirect inheritance since the `TwoPort` inherits both from the `ElectricComponent` class (as a subclass) and some `Terminal` classes (used as subcomponents), even though, strictly speaking, multiple inheritance has not been invoked.

Here is a simple fragment of Omola code for a resistor viewed as a two-port device (see Figure 8).

Omola 1 (Resistor Example)

```
Resistor ISA TwoPort WITH
  R ISA Parameter WITH
    default := 1.0;
  END;

  V1 - V2 = R * (I1 - I2);
  I1 = I2;
END;
```

Reserved words are in all capitals. Inheritance is accomplished through the `ISA` (and `ISAN`) command, refinements are given after `WITH`. The reader may guess from this that `Parameter` is an Omola base class consisting of a real value plus a default value. The symbol `:=` is used to denote equality with causality specified, viz. the left-hand side is dependent on the right-hand side. The equality sign denotes an acausal relation. The relations should be self-explanatory.

3.2 Omsim

Omsim stands for “Omola Simulation” package. Again, our discussion is brief, with the reader directed to [2] for more details. Omsim

- provides a graphical model editor, used for input, visualization, and editing of models.
- compiles Omola to simulation code, performing consistency checks, sorting equations, determining causality, etc., and solving initialization problems.
- provides simulation tools, including plotters, access windows for changing parameters and initial conditions; output routines for exporting to files, printers, or Matlab; a variety of DAE and ODE solvers.
- may be controlled using “ocl” (Omsim Command Language) files instead of point-and-click menus for easy repetition of experiments or reuse of simulation environments.

Continuous Dynamics. Omsim can handle continuous-time dynamics given by general DAEs:

$$\dot{x} = f(x, y, t), \tag{8}$$

$$0 = g(x, y, t). \tag{9}$$

with the usual semantics from mathematics. It also allows $g = g(\dot{x}, x, t)$. Omsim does sophisticated pre-processing of models in compiling them into simulation code, including symbolic manipulation. Very roughly, the procedure is as follows.

1. Divide variables (into four classes of increasing complexity from constants, parameters, and discrete (defined below), to continuous-time varying).
2. Divide equations (according to the most complex of the four variable classes appearing therein).
3. Check consistency (are all variables defined, of correct type, etc.)
4. Sort equations into a block-lower triangular form. This includes determining causality (necessary not for modeling but for simulation!), eliminating derivatives, dividing blocks into the four classes, performing index reduction on any DAEs, determining which solver should be used (if not prescribed by the user), and outputting equations into a form required for the chosen DAE/ODE solver.

Discrete Events. Here, we discuss mostly Omola syntax for discrete events, postponing some semantics to the next section. Please see [2] for details. The fragment

```
x TYPE DISCRETE Real;
```

means that the value of \mathbf{x} changes only at discrete times, i.e., $\dot{x} = 0$. Integers and booleans are automatically **DISCRETES**.

Events, their conditions, and their side effects are specified using a template as follows:

```
<event name> ISAN Event WITH
  CONDITION := <logical expression>;
  <body of actions>
END;
```

Here, the logical expression can depend on a continuous variable, e.g., $(\mathbf{y} > 0)$, for \mathbf{y} **TYPE Real**; a discrete variable, e.g., $(\mathbf{i}=1)$ for \mathbf{i} **TYPE Integer**; or be edge-triggered, e.g., $\wedge(\mathbf{y} > 0)$. The latter condition is true only when \mathbf{y} first becomes positive; it will not be true again until \mathbf{y} becomes nonpositive and then positive again. The body of actions may, for example, reset variables based on other variables' or their own values. In the later case, the **NEW** operator is used to distinguish previous from new values of the variable. For example,

```
NEW(x) + x = 0;
```

will change the sign of \mathbf{x} upon the event's firing. Events may also be scheduled to occur after some delay; scheduled events may be descheduled:

```
schedule(EventName, Delay);
deschedule(EventName);
```

Finally, event propagation can be specified with the following self-evident template:

```
WHEN <condition> [CAUSE <event list>]
  [DO <actions>] END;
```

The following is Omola code for the automaton of Figure 9.

Omola 2 (Finite State Automaton)

```
Server ISA Model WITH
  State TYPE DISCRETE (Passive, Active) % textual names are allowed
  Start, Ready, Init ISAN Event;

transitions:
  WHEN State=='Passive AND Start DO      % they are referenced with '
    NEW(State) := 'Active;
  END;
  WHEN State=='Active AND Ready DO
    NEW(State) := 'Passive;
  END;

initialization:
  WHEN Init DO
    NEW(State) := 'Passive;
  END;
END;
```

The headings `transitions` and `initialization` are just that; they are arbitrary and do not affect the equations. Everything on a line after a `%` is a comment.

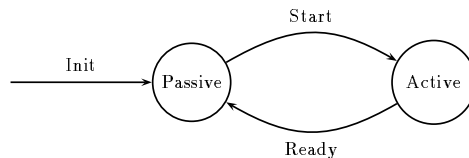


Fig. 9. Example finite state machine.

Detecting State-Dependent Events. We have mentioned that Omsim uses sophisticated techniques for dealing with mixed continuous/discrete events. The full discussion is quite involved and may be found in [2, pp. 200–208]. We try to give the reader a brief hint here. Those not interested in numerical details, however, may skip ahead to the next section.

First a word about ODE/DAE solvers. If there are no implicit equations, RKF45 is used, else an implicit Runge-Kutta scheme, Radau5, is used. For general DAEs, DASSL is used. DOPRI(4)5 is also available.

All boolean event conditions which refer to values of continuous variables are translated so that they occur on zero crossings, e.g., $\mathbf{x} > \mathbf{xmax}$ becomes $\mathbf{x} - \mathbf{xmax}$. Compound conditions are dealt with by introducing multiple event conditions.

Example 7. $(x > x_{\max})$ AND $(y > y_{\max})$ is not replaced by the non-differentiable $\min(x - x_{\max}, y - y_{\max})$. Instead, it is replaced by two event conditions.

We assume that purely discrete event conditions can be dealt with effectively (see [2] for details). However, continuous-time event conditions must be evaluated along with the continuous time solution. The root-finding program DASSRT is used in conjunction with DASSL to find the zero-crossings associated with these. An algorithm by Brankin et al., based on ALEVNT (herein, “A”), is used with Radau5. This algorithm is used with integration schemes that provide solutions between steps, so-called “dense output.” This is done by returning a polynomial interpolation that is valid over the next interval. Algorithm A uses Sturm sequences to determine if a zero is present in the interval. If yes, Sturm sequences and bisection are used to locate the subinterval of the first root. Then, standard bisection and the secant algorithm are used to locate the root precisely. This will work well if the event functions are linear in the unknown variables. However, since Omsim allows nonlinear event functions, more must be done. Briefly, in general Omsim uses Algorithm A to approximately locate the event based on the approximate event function (evaluated using the approximated solution) and then iterates using bisection or secant based on the event function itself.

4 Simulating HDS

4.1 Two Approaches

We first discuss simulating hybrid dynamical systems (HDS) using Omsim, saving the more general case of Controlled HDS (CHDS) for later. There are two main approaches for representing HDS: distributed state and global state.

In the *distributed state* approach, one views the HDS as consisting of N different dynamic systems, each of which always exists and is either “active” (controlling the dynamics) or “inactive.” Here, the state of the simulation is given by the concatenation of the states of each of the N systems, $[_1x, \dots, _Nx]^T$ plus an integer, q , representing the currently active one. If the i th system is active, its dynamics is given by $_i\dot{x} = f_i(_ix)$, and it becomes inactive when $_ix$ hits A_i ; if it is inactive, its state does not change (which is represented by setting its vector field to zero in that case), and it becomes active whenever $G(x, q) = (_i\xi, i)$. See Figure 10(a).

In the *global state* approach, we let $q(t)$ denote the active system and dynamically redefine the vector field to be given by $f_{q(t)}$, the test set by $A_{q(t)}$, and the transition map by $G_{q(t)}$. Here, we use one global continuous state, $x(t)$, which must be given a size equal to the largest dimension of the $_qx$. See Figure 10(b).

4.2 Distributed State HDS

We now build up a distributed state HDS model class, piece by piece,. We start with a dynamic system (DS) given by an ODE, represented in Omola 3.

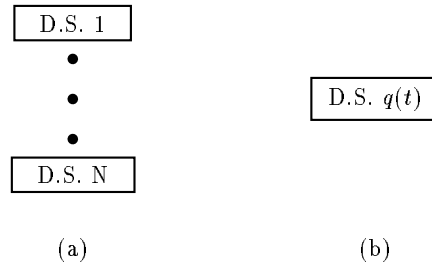


Fig. 10. Two approaches of storing state: (a) distributed, (b) global.

Omola 3 (DS class)

```

DS ISA Model WITH

parameters:
  d TYPE Integer;      % model order
  fx TYPE Column [d];

state:
  x TYPE Column [d];

behavior:
  x' = fx;
END;

```

The prime denotes derivation. Hence, this model simply says that the state has dimension d and the dynamics are given by a d -dimensional vector field $\mathbf{f}\mathbf{x}$. Hence, $\dot{x} = f(x)$, $x \in \mathbf{R}^d$.⁴

Now, we add the capability of switching the ODE between **Active** and **Inactive** modes. Recall that when in the inactive mode, the vector field is set to zero.

Omola 4 (Switched DS class)

```

SwitchedDS ISA DS WITH
variables:
  A TYPE Real;
  mode TYPE DISCRETE (Active, Inactive);
  Init, Restart ISAN EventInput;
  AutoJump ISAN EventOutput;
  f TYPE Column [d];

equations:
  fx = IF mode=='Inactive' THEN 0.0 ELSE f;

```

⁴ Columns, Rows, and Matrixs are only real-valued; indexing conventions are those widely used, viz. as in Matlab.

```

events:
  WHEN mode=='Active and A >= 0 CAUSE AutoJump DO
    NEW(mode) := 'Inactive;
  END;
  WHEN Restart DO
    NEW(mode) := 'Active;
  END;
END;

```

Note we have encoded $x \in A$ by a real value A 's (whose functional form will be specified in instantiated models) being nonnegative.

What we would want now is to have a hybrid model that is a vector of `SwitchedDS`'s, with size = N . For example,

Omola 5 (Distributed-State HS)

```

HybridSystem ISA Model WITH
  N, Q TYPE Integer;      % number of discrete states, active state
  Sigma TYPE SwitchedDS Column [N];
END;

```

Unfortunately, Omola does not currently support vectors of models—though this capability is being actively pursued. Of course, one can define templates—just once, then stored in a library—for each size of hybrid system. One can save some typing by doing this in an inductive manner using inheritance. However, below is an explicit example of a hybrid system with two discrete states.

Omola 6 (Two Distributed-State HS)

```

HybridSystem2 ISA Model WITH
  Q,d TYPE Integer;      % discrete state, cont. state's dim.
  x, Gx1, Gx2 TYPE Column[d]; % continuous states and resets

components:
  Q1, Q2 ISA SwitchedDS;

equations:
  x = IF Q==1 THEN Q1.x ELSE Q2.x;

  WHEN Q1.AutoJump CAUSE Q2.Restart DO
    new(Q2.x) := Gx1; END;
  WHEN Q2.AutoJump CAUSE Q1.Restart DO
    new(Q1.x) := Gx2; END;
END;

```

Of course, the case for higher N is a little more complicated, but we think the reader can figure out the general case readily.

We now give an Omola model of our hysteresis example using the above model classes.

Omola 7 (Hysteresis Example)

```
Mode1 ISA SwitchedDS WITH
  Delta ISA Parameter WITH default := 0.1; END;
  d = 1;
  f = 1;
  A = x-Delta;
END;

Mode2 ISA SwitchedDS WITH
  Delta ISA Parameter WITH default := 0.1; END;
  d = 1;
  f = -1;
  A = -x-Delta;
END;

Hysteresis ISA HybridSystem2 WITH
  d = 1;
  Q1 ISA Mode1;
  Q2 ISA Mode2;

  Gx1 := Q1.x;
  Gx2 := Q2.x;
END;
```

Each mode has dimension one, with dynamics given by either plus or minus one. Switching occurs from **Mode1** to **Mode2** whenever $x \geq \Delta$, which is equivalent to whenever $-x - \Delta \geq 0$. Here, we have specified Δ to be a parameter, **Delta**, with a default value of 0.1. In Omsim, we might look at simulation results with different values of the parameter, which could be changed using the aforementioned access windows. Finally, note that the reset maps are the identity since only the dynamics, and not the continuous state, is reset upon changing modes.

4.3 Global State HDS

Here is a global state model class of HDS. It parallels the equations of Section 2 closely enough that the reader should be able to follow the code, perhaps using the comments.

Omola 8 (Global State HS)

```
HDS ISA Model WITH

states:
  d TYPE Integer;           % max dim. of constituent state spaces
  x TYPE Column [d];       % continuous state
  N TYPE Integer;          % number of discrete states
  q TYPE DISCRETE Integer; % discrete state
```

```

continuous_dynamics:
  f TYPE Matrix [d,N];  % vector fields

  x' = f[1..d,q];

discrete_transitions:
  % autonomous switching functions
  Ajump TYPE Row [N];

  % autonomous switching map, discrete and continuous parts
  Gq TYPE Row [N];
  Gx TYPE Matrix [d, N];
  ModeChange ISAN Event;

  WHEN Ajump[q] >= 0 CAUSE ModeChange DO
    new(q) := Gq[q];
    new(x) := Gx[1..d,q];
  END;
END;

```

Note that we have added a **ModeChange** event. This was done for fun and could be used by Omsim to count events or change plotter colors (or execute some other actions) upon mode changes.

Now here is an Omola model of our hysteresis example using the HDS model classes.

Omola 9 (Hysteresis Example)

```

Hysteresis ISA HDS WITH
  Delta ISA Parameter WITH default := 0.1; END;
  N = 2;
  d = 1;
  f[1,1] = 1;
  f[1,2] = -1;
  Ajump[1] = x - Delta;
  Ajump[2] = -x - Delta;
  Gq[1] = 2;
  Gq[2] = 1;
  Gx[1,1] = x;
  Gx[1,2] = x;
END;

```

Again, note that the reset maps are the identity.

For the remainder of this paper, we will talk only of the global state model class HDS above and subclasses thereof.

4.4 Subclasses and Examples

In this section we show how several classes of hybrid systems models may be easily recovered as subclasses of the global state HDS above. An autonomous

switched hybrid system is one where the continuous state does not change at autonomous jump times, that is, where the projection of the reset map G onto the continuous state is the identity: $G_x(x, q) = x$ for each discrete state q .

Omola 10 (Autonomous-switched HS)

```
AutoSwitch ISA HDS WITH
  Gx=x*ones(1,N);
END;
```

Here is a special case of an HDS, the model due to Zvi Artstein (see [5] and Example 4), which can naturally be represented as a subclass of HDS:

Omola 11 (Artstein's HS)

```
Artstein ISA HDS WITH
  dim TYPE Integer;
  T TYPE Row [N];

  d = dim + 1;

  f[d,1..N] = ones(1,N);

  Ajump = x[d]*ones(1,N) - T;

  Gx[1..d-1,1..N] = x[1..d-1]*ones(1,N);
  Gx[d,1..N] = zeros(1,N);
END;
```

Note that if the state has dimension \mathbf{dim} , then the model has one more dimension, with $\mathbf{x}[d]$ representing the timer value τ in Example 4. The vector \mathbf{T} denotes the vector of timeout values, $[T_1, \dots, T_N]^T$. As expected, jumps occur whenever the timer, $\mathbf{x}[d]$ is greater than or equal to the timeout value. Note also that this variable is reset to zero—and the continuous state otherwise does not jump—upon timeout, no matter what the discrete state.

Brockett's model (see [10] and Example 3) may also be recovered as a subclass of the HDS model class:

Omola 12 (Brockett's HS)

```
Brockett ISA HDS WITH
  dim TYPE Integer;
  p, pfrac TYPE Real;          % p and its fractional part
  r TYPE Row [N];
  z TYPE DISCRETE Integer;

  d = dim + 2;

  p = x[d-1];
  pfrac = x[d];
  z = q;
```

```

f[d-1,1..N] = 0;
f[d,1..N] = r;

Ajump = x[d]*ones(1,N) - ones(1,N);

Gx[1..d-2,1..N] = x[1..d-2]*ones(1,N);
Gx[d-1,1..N] = (x[d-1]+1)*ones(1,N);
Gx[d,1..N] = zeros(1,N);
END;

```

Note that if the state has dimension `dim`, then the model has two more dimensions, with `x[d-1]`, `x[d]` now representing the counter, p , and its fractional part, resp. Note, however, that here jumps occur whenever the counter's fractional part, `x[d]`, is equal to one (and not at integer points); this is compensated for by resetting it to zero at jump times, yielding an equivalent description. Finally, note the portion of the continuous state corresponding to Brockett's original x , `x[1..d-2]` does not jump.

Below is a model class for switching among two linear systems in the plane. Note that it is autonomous switching, so that we can make it a subclass of `AutoSwitch`. Also note that the switching surfaces, given by the `Ajump` vector (inherited from `HDS` through `AutoSwitch`), are still unspecified.

Omola 13 (Switched Planar HDS)

```

Switcher ISA AutoSwitch WITH
  N=2;   d=2;
  A TYPE MATRIX [2, 2] := [1, -100; 10, 1];
  B TYPE MATRIX [2, 2] := [1, 10; -100, 1];

  f[1..2,1] = A*x;
  f[1..2,2] = B*x;

  Gq[1] = 2;
  Gq[2] = 1;
END;

```

Using the above template, we can now investigate the effect of different switching rules for the same underlying hybrid system. Here are two particular examples followed by a special model discussed below.

Omola 14 (Two Switching Systems)

```

SwitchQuadrant ISA Switcher WITH
  Ajump[1] = IF (x[1]>=0 AND x[2]<=0) THEN 1 ELSE -1;
  Ajump[2] = IF (x[1]<0 AND x[2]>=0) THEN 1 ELSE -1;
END;

SwitchStretch ISA Switcher WITH
  Ajump[1] = IF (x[1]>=0.75*-x[2] AND x[2]<=0) THEN 1 ELSE -1;
  Ajump[2] = IF (x[1]<=0 AND x[2]>=0.75*-x[1]) THEN 1 ELSE -1;
END;

```

Omola 15 (Augmented Switching System)

```
SwitchStretchPlus ISA SwitchStretch WITH
  Stop ISAN Event;
  xdum TYPE Real;
  WHEN ModeChange CAUSE Stop;
  xdum = 10;
END;
```

The last model above simply adds an event and a dummy variable to the `SwitchStretch` model. `Stop` is a special Omsim event which causes the simulation to halt at mode changes. Below is an example of ocl code used to simulate the model. We take advantage of the `Stop` event to count mode changes (allowing a maximum of 60) and the extra dummy variable to hack a color change in the plotter upon switching. Details are not important, but the interested reader may browse the comments. See Figure 11.

ocl 1 (Switching Example)

```
BEGIN
  Hybrid::SwitchStretchPlus m; % create the model

  Simulator sim(m);           % make a simulator for it
  sim.display(10, 450);       % display the simulator
  sim.stoptime := 1;
  sim.outputstep:=0.005;

  Plotter Response(m);       % make a plotter
  Response.display(10, 150);
  Response.xrange(-3, 1);    % set the scales
  Response.yrange(-3, 1);
  Response.x(x[1]);          % define the variables to plot
  Response.y(x[2],xdum);     % define the variables to plot

  Plotter R(m);              % make a plotter
  R.display(260, 150);
  R.xrange(0, 1);           % set the scales
  R.yrange(0.95, 2.1);
  R.y(q);                   % define the variables to plot

  m.q := 1;
  m.x := [0.3; 0.826];
  sim.start;

  for eventnum := [1:30] BEGIN % simulate for 30 events
    Response.y(xdum,x[2]);    % change color on each event
    sim.continue;
    Response.y(x[2],xdum);    % do this by swapping variable order
    sim.continue;
  END;
END;
```

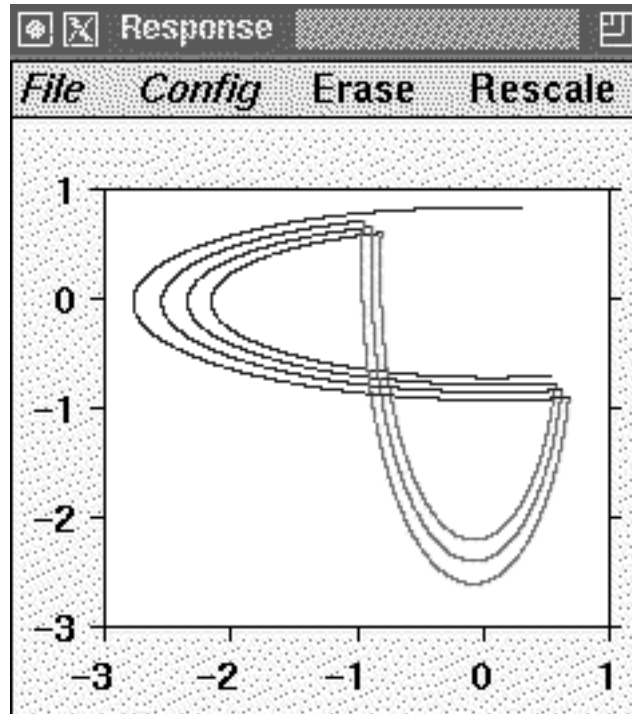


Fig. 11. Example planar switching system (screendump of Omsim plot window).

4.5 Adding Control

We hope the reader is seeing the power of Omola's object-oriented philosophy and the ease of reuse via subclasses by now. For completeness, though, we add controlled versions of DS's, HDS's, and Artstein's model. Then we give a model of Pait's two-state stabilizer for the simple harmonic oscillator (S.H.O.) (see Figure 12 and [5]).

A Controlled HDS (CHDS) is simply a subclass of HDS with inputs, outputs, and controlled switching added.

Omola 16 (CHDS)

```
CHDS ISA HDS WITH
```

```
inputs_outputs:
```

```

  m TYPE Integer;           % max dim. of constituent control spaces
  u TYPE Column [m];       % continuous control
  p TYPE Integer;          % max dim. of constituent output spaces
  y TYPE Column [p];       % continuous output
```

```

controlled_transitions:
  % controlled switching functions
  Cjump TYPE Row [N];

  % controlled switching map, discrete and continuous parts
  Fq TYPE Row [N];
  Fx TYPE Matrix [d, N];

  WHEN Cjump[q] >= 0 CAUSE ModeChange DO
    new(q) := Fq[q];
    new(x) := Fx[1..d,q];
  END;
END;

```

A controlled version of Artstein's HS is now a subclass of CHDS much as its autonomous version was one of HDS.

Omola 17 (Controlled Artstein)

```

CArtstein ISA CHDS WITH
  dim TYPE Integer;
  T TYPE Row [N];

  d = dim + 1;

  f[d,1..N] = ones(1,N);

  Ajump = x[d]*ones(1,N) - T;

  Gx[1..d-1,1..N] = x[1..d-1]*ones(1,N);
  Gx[d,1..N] = zeros(1,N);
END;

```

The above is a case where multiple inheritance would have been nice, that is, from both the `CHDS` and `Artstein` classes. Indeed, the reader may find it instructive to construct a `CArtstein` class as a subclass of `Artstein`, and then compare it with the above.

The code below instantiates the system in Figure 12. At this point, the reader should be able to understand it without comment. Note: do not confuse the S.H.O.'s velocity, y , with the Omola 18 variable \mathbf{y} ; the latter is a generic output measurement, which for the S.H.O. is the position, x a.k.a. $\mathbf{x}[1]$. Note also, though, how easy it is to connect the two subcomponents in making `Full12`. Thus, using our macros, one can simulate interacting networks of controlled hybrid systems. Figure 13 shows the output of the Omsim simulation of the system shown in Figure 12 and specified in Omola 18. This time, we plotted a PostScript file generated by Omsim instead of a screendump. The two plot windows, from left to right, show the S.H.O.'s phase plane (with $\mathbf{x}[1] \simeq x$ as the abscissa and $\mathbf{x}[2] \simeq y$ as the ordinate) and the controller's discrete state ($q \simeq q \in \{1, 2\}$) versus time.

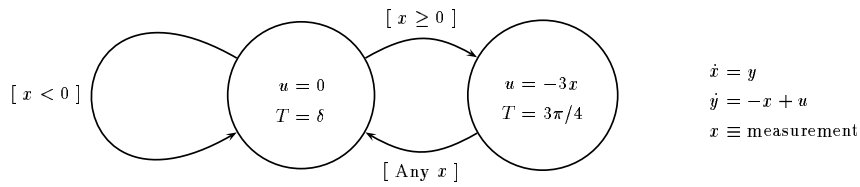


Fig. 12. Pait's two-state hybrid S.H.O. stabilizer and S.H.O. equations.

Omola 18 (S.H.O.)

```

SHO ISA CDS WITH
  d = 2;   m = 1;   p = 1;

  f[1] = x[2];
  f[2] = -x[1]+u[1];

  y[1]=x[1];
END;

SHOCont2 ISAN Cartstein WITH
  delta ISA Parameter WITH default := 0.1; END;

  N = 2;   dim = 0;   m = 1;   p = 1;

  y[1]=IF q==1 THEN 0 ELSE -3*u[1];

  Gq[1] = IF u[1] >=0 THEN 2 ELSE 1;
  Gq[2] = 1;

  Fq[1] = 1;
  Fq[2] = 1;

  Fx[1,1] = x;
  Fx[1,2] = x;

  Cjump[1] = -1;
  Cjump[2] = -1;

  T[1] = delta;
  T[2] = 2.3562;
END;

Full2 ISA Model WITH
  Plant ISA SHO WITH u[1]=Controller.y[1]; END;
  Controller ISA SHOCont2 WITH u[1]=Plant.y[1]; END;
END;

```

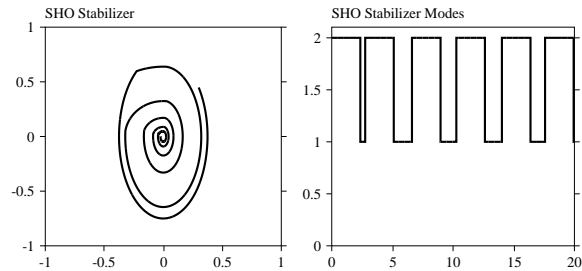


Fig. 13. Omsim simulation of Pait’s SHO stabilizer (PostScript output).

5 Conclusions

Our goal is to produce fast, high fidelity simulations of (networks of) a very broad class of hybrid systems in a user-friendly environment. In this paper, we first reviewed a very broad class of hybrid systems: hybrid dynamical systems (HDS) [8]. We briefly discussed the object-oriented modeling and simulation tools for combined discrete/continuous systems: Omola and Omsim [2]. Leveraging these, we gave a general set of hybrid systems model classes which encompass HDS. HDS encompass several other hybrid systems models popularized in the literature; we made this explicit by representing them as subclasses of our HDS model class. These Omola model classes may be viewed as “templates” or “macros” for quick and easy entering of hybrid systems for subsequent analysis and numerically-sophisticated simulation using Omsim. They may be used to simulate networks of interacting dynamical and hybrid systems, a simple example of which was shown.

6 Acknowledgments

The first author did most of this work while visiting the Department of Automatic Control at the Lund Institute of Technology in May/June 1996. His thanks must go to Profs. Karl J. Astrom (Lund) and Sanjoy K. Mitter (MIT) for making the visit possible and to the lab members’ hospitality and expertise for making it enjoyable and productive, resp. Omola/Omsim may be freely downloaded from Lund (<http://www.control.lth.se/~cace/omsim.html>).

References

1. R. Alur, T. A. Henzinger, and E. D. Sontag, editors. *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer, New York, 1996.
2. M. Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Lund Institute of Technology, Dept. of Automatic Control, 1994.
3. P. Anstaklis, W. Kohn, A. Nerode, and S. Sastry, editors. *Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*. Springer, New York, 1995.
4. P. J. Antsaklis, J. A. Stiver, and M. D. Lemmon. Hybrid system modeling and autonomous control systems. In Grossman et al. [12], pages 366–392.
5. Z. Artstein. Examples of stabilization with hybrid feedback. In [1] pages 173–185.
6. A. Back, J. Guckenheimer, and M. Myers. A dynamical simulation facility for hybrid systems. In Grossman et al. [12], pages 255–267.
7. D. D. Bainov and P. S. Simeonov. *Systems with Impulse Effect*. Ellis Horwood, Chichester, England, 1989.
8. M. S. Branicky. *Studies in Hybrid Systems: Modeling, Analysis, and Control*. ScD thesis, Massachusetts Institute of Technology, Dept. of Electrical Eng. and Comp. Science, June 1995.
9. M. S. Branicky, V. S. Borkar, and S. K. Mitter. A unified framework for hybrid control. In *Proc. IEEE Conf. Decision and Control*, pages 4228–4234, Lake Buena Vista, FL, Dec. 1994.
10. R. W. Brockett. Hybrid models for motion control systems. In H. L. Trentelman and J. C. Willems, editors, *Essays in Control: Perspectives in the Theory and its Applications*, pages 29–53. Birkhäuser, Boston, 1993.
11. A. Deshpande. *Control of Hybrid Systems*. PhD thesis, Univ. of California at Berkeley, 1994.
12. R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, New York, 1993.
13. A. Nerode and W. Kohn. Models for hybrid systems: Automata, topologies, stability. In Grossman et al. [12], pages 317–356.
14. K. S. Sibirsky. *Introduction to Topological Dynamics*. Noordhoff International Publishing, Leyden, The Netherlands, 1975. Translated by Leo F. Boron.
15. E. D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*, volume 6 of *Texts in Applied Mathematics*. Springer, New York, 1990.
16. L. Tavernini. Differential automata and their discrete simulators. *Nonlinear Analysis, Theory, Methods, and Applications*, 11(6):665–683, 1987.
17. H. S. Witsenhausen. A class of hybrid-state continuous-time dynamic systems. *IEEE Trans. Automatic Control*, 11(2):161–167, 1966.