Hybrid Dynamic Systems: Modeling and Execution

Pieter J. Mosterman

The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, USA
pieter.mosterman@mathworks.com

Summary. Physical system modeling benefits from the use of implicit equations because it is often an intuitive way to describe physical constraints and behaviors. To achieve efficient models, model abstraction may lead to idealized component behavior that switches between modes of operation (e.g., an electrical diode may be on or off) based on inequalities (e.g., voltage > 0). In an explicit representation, the combination of these local mode switches leads to a combinatorial explosion of the number of global modes. It is shown how an implicit formulation can be used to formulate these mode switches, thereby circumventing the combinatorial problem. This leads to the use of differential and algebraic equations (DAE) for each of the modes. In case these DAEs are of high index, jumps in generalized state variables may occur. In combination with the inequalities that define mode switching, this leads to rich and complex mode transition behavior. An overview of this mode switching behavior and an ontology is presented.

1 Introduction

Model-Based Design is increasingly being adopted by industry to remain competitive. Computational models take a central position in Model-Based Design as illustrated in Fig. 1. The use of models throughout the design of engineered systems has a number of advantages. In Fig. 1, four of these are shown:

- Early evaluation of the appropriateness, consistency, rigor, unambiguity of requirements
- Automatic code generation eliminates the expensive manual coding which is labor intensive and error prone.
- Simulation allows quick design iterations while the use of models makes it possible to search a very large design space quickly.
- Verification of completeness and parsimony of the design can be done before implementation.
- Test vector sets can be composed in the early design stages, based on a system model, thus eliminating much of the lead time that is required if the real system has to be available instead.

From a business perspective, Model-Based Design allows a shorter design cycle to produce better products at a lower cost.

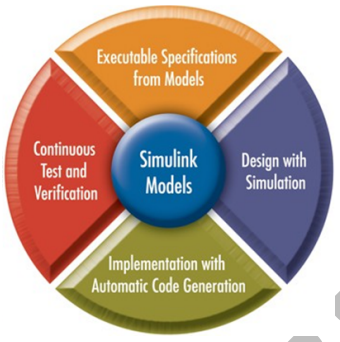


Fig. 1. Model-Based Design.

The models that are used throughout the design have to address many different aspects varying from structuring requirements, which may be best done by modeling scenarios and in an axiomatic manner, while the numerical models with an intensive data processing aspect are often best represented in a declarative manner, and execution models typically are best captured in an imperative manner.

Similarly, certain modeling tasks are best addressed by continuous-time models, while other tasks may be better performed with discrete-time or untimed models. For example, consider the power window in Fig. 2 [29]. Modeling the behavior of the window sliding up and down is often easiest using continuous-time differential equations, employing laws of physics such as Newton's second law of motion that the acceleration, a, of a mass, m, is determined by the force, F, applied to it, $F = m \cdot a$, where the acceleration is the time-derivative of velocity, v, $a = \frac{dv}{dt}$.

On the other hand, the desired behavior to start moving the window when the user presses a button is often best viewed as discrete in nature. An event *windowUp* then causes a state change from *neutral* to *moveUp*. At certain stages during the design, such an untimed model may be sufficient and even desired, as it leaves any implementation choices on how to achieve such control still open.

To implement the control of the power window, an electronic control unit that runs at a certain sample rate can be selected. In this implementation, the untimed events can be modeled to occur at points in time. During the fixed intervals of time in between, no changes of the variable values in the model occur. This leads to a discrete-time but periodic model.

Yet another task may consist of modeling the network that transmits the user command to move the window up to the controller that effects this command. In automobiles, these commands may be transmitted over a controller

Fig. 2. A power window.

area network (CAN) [1], that is also used to transmit other commands such as moving the headlights up and down, adjusting the outside mirrors, and opening the sunroof. At a certain level, the network traffic can be thought of as discrete-time but nonperiodic, since commands may be initiated with very different intervals of time in between.

When the integration of the different parts of a system is studied, it becomes necessary to employ the different models in concert [28]. This can be done in a number of ways, for example, the different types of models can be combined, integrated, and translated. Whichever approach chosen, the result is likely to include both continuous-time and discrete-event behavior, and the overall system is often referred to as a *hybrid dynamic system*, or *hybrid system* for short.[1] It is important to stress, at this point, that an engineered system is not inherently a 'hybrid system'. A hybrid system is a term for a mathematical representation, not a physical system. Whether an engineered system is modeled as a hybrid system depends on the problem that needs to be solved, the level of abstraction chosen, the phenomena the system embodies, and the background of the model designer [30].

This paper intends to provide a bird's eye view of the modeling and execution of hybrid systems. Some of the uses of hybrid systems will be presented, and the different ways of simulating them and their idiosyncrasies will be investigated. In Section 2, hybrid dynamic systems are presented; what they are, how they come about, and what different modeling perspectives exist. In Section 3, the behavioral perspective is discussed and the different behavioral aspects are introduced. In Section 4, the two possible implementations of an execution engine, time-driven or event-driven implementation, and possibly combinations are introduced. Section 5 then presents advanced topics with respect to numerical simulation. In Section 6 a number of pathological

[1] Note that omitting the adjective 'dynamic' may lead to confusion with hybrid systems such as mixed neural network/fuzzy logic systems and combined electrical/mechanical drivetrains.

behaviors that may arise in simulation are discussed. Section 7 presents the conclusions of this work.

2 Hybrid Dynamic Systems

2.1 Why Hybrid Systems?

A power window will serve to illustrate the application of hybrid systems in the design of engineered systems. This example has been studied in more detail in other work [29, 30] and is part of the Simulink automotive demos. The elements of interest in this context are illustrated in Fig. 3. On the right, it shows the window lift mechanism. A *dc* motor translates the electrical signals that reflect the controller commands into movement in the mechanical domain. A current sensor is used to measure the current drawn by the *dc* motor, which is used by the controller to determine whether the top or bottom of the window frame is reached.

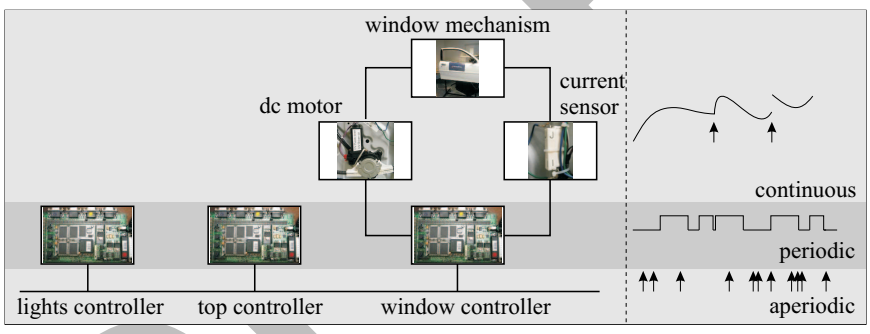


Fig. 3. A hybrid dynamic system with different behavior classes.

The controller receives the user input over a controller area network (CAN). This network provides communication between the electronic control units (ECUs) in an automobile and connects the ECU that interfaces with the window up and down button to the ECU that interfaces with the window lift mechanism.

The design of such a system typically proceeds by determining a set of requirements that it has to satisfy. Among these requirements [10], it may state that

1. `The window should be closed within 5 seconds.`
2. `The passenger command should always be overruled by the driver command.`
3. `The user commands should be acted upon within 200 miliseconds.`

To validate Requirement 1, a continuous-time model of the lift mechanism is best used. This could be a set of differential and algebraic equations (DAEs) that model how fast the window with mass, m_{window}, moves, v_{window}, given a voltage on the *dc* motor, u_{motor}, and the corresponding current that the motor draws, i_{motor}. The equations

$$\begin{aligned} F_{motor} &= r u_{motor} \\ F_{window} &= m_{window} \dot{v}_{window} \\ F_{lift} &= R_{lift} v_{window} \\ i_{motor} &= r v_{window} \\ F_{window} + F_{lift} &= F_{motor} \\ \dot{x}_{window} &= v_{window} \end{aligned} \tag{1}$$

model the conversion of the *dc* motor voltage by a parameter, r, and include linear friction, R_{lift}, that causes a friction force component, F_{lift}, for the lift mechanism. Solving this system of equations over time shows whether the window can rise quickly enough for certain input voltages, by evaluating the window position, x_{window}.

On the other hand, the control structure that Requirement 2 addresses is best modeled by a state machine that has hierarchical structure of its states. This allows the passenger commands to execute when the driver commands are neutral. When the driver issues a command, the passenger control structure is departed and overruled by the driver command. This is illustrated by the Stateflow [37] in Fig. 4.

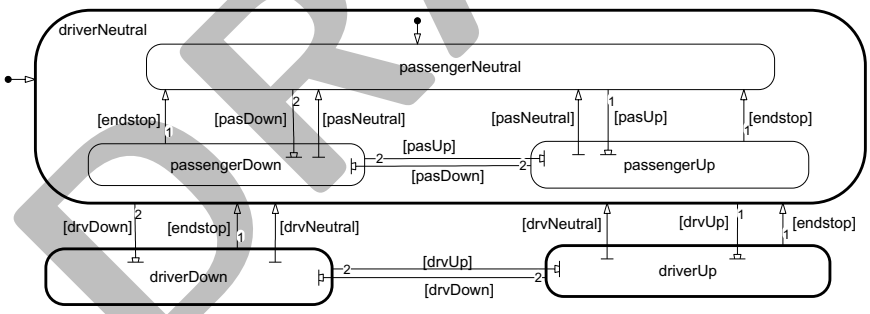


Fig. 4. A finite state machine in the form of a Stateflow chart.

In this Stateflow chart, there are three states for the driver control, *driverNeutral*, *driverDown*, and *driverUp*. In case *driverDown* is the active state, the window is moved down, and in case the *driverUp* state is active, the window is moved up. The *driverNeutral* state is of a hierarchical nature, and contains the control structure for the passenger, analogous to the control structure for the driver. In this manner, the driver control takes precedence over the passenger control, and the passenger may command the window only if the driver does not issue a command (i.e., a neutral command). The initial

states are indicated by a transition that is depicted by an arrow with a solid circle on one end and arrow head on the other end. Note that the ordering of transitions that could be activated simultaneously is indicated by the numbers associated with the respective transitions.

The events issued by the driver, *drvUp* and *drvDown* command the window to go up or down. In case any of the commands is released, the *drvNeutral* command is issued. Similar with the passenger related commands *pasUp*, *pasDown*, and *pasNeutral*. The *endstop* command is issued when the window has reached the top or bottom of the door frame.

To validate the response time as put forward in Requirement 3, the communication network needs to be modeled. Here, a server/queue type model is best used where the network is modeled as a server that aperiodically frees up and passes a packet from one ECU to another. A SimEvents [36] model of the packet transmitter is shown in Fig. 5. Here, the command as issued by the driver is assigned as the attribute of packets that are generated at a fixed rate. The packets further have an attribute that identifies their destination. They then are attempted to be put into a first-in-first-out (FIFO) queue and if this fails, they are lost as dropped packets. The FIFO queue is connected to another subsystem, which models the CAN bus behavior.

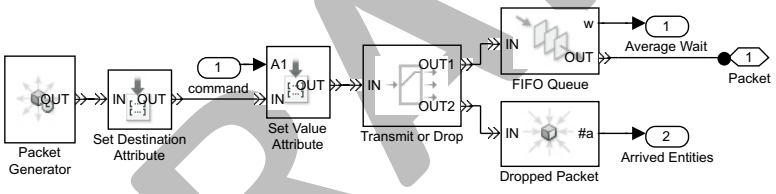


Fig. 5. An entity network model of a transmitter.

Stochastic server times may be included to model the variable arrival times of network requests from control systems, for example the head-lights control system, that communicate using the same CAN. This is a type of system integration test that often is performed in hardware because the different systems are provided by different suppliers, and the models used for the design of each of the separate ones are not shared among these suppliers.

Combining all these different types of models that are needed to verify the requirements necessitates handling both continuous-time differential equation models as well as discrete event state-transition models. This can be achieved, for example, by transforming the continuous-time models into discrete-event representations or the other way around. Alternatively, a computational model can be designed that can handle both the continuous-time and discrete-event behavior. This latter approach leads to the hybrid dynamic execution structure.

2.2 What Is a Hybrid Dynamic System?

A hybrid dynamic system is defined in behavioral terms, because this is model-structure agnostic, and, therefore, does not impose unnecessary and undesired modeling assumptions.

In a geometric sense, a hybrid dynamic system evolves continuously in time in a mode, α_i, according to a field, f_{α_i} [13, 26]. This field defines a relation $f_{\alpha_i}(\dot{x}, x, u, t) = 0$ between the state, x, its time derivative, $\dot{x}$, the input u, and the time, t. This is illustrated in Fig. 6, where the continuous-time behavior is shown as a solid directed line in the plane α_1.

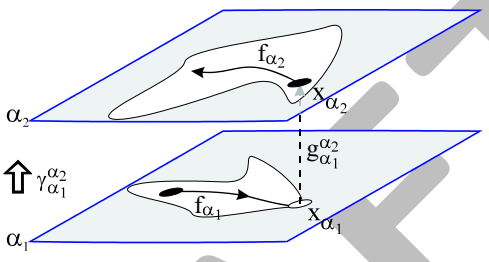


Fig. 6. Basic elements of a hybrid dynamic system.

The continuous-time behavior is often captured by an explicit representation as a set of differential equations

$$\dot{x} = f_{\alpha_i}(x, u, t), \tag{2}$$

where α_i is the mode of the model, x the continuous-time state vector, u the exogeneous input, and t is time. An alternative formulation is the implicit form

$$f_{\alpha_i}(\dot{x}, x, u, t) = 0, \tag{3}$$

often used in plant modeling, and the semi-explicit form that has an explicit representation of the time-derivatives, $f^d_{\alpha_i}$, combined with a set of implicitly formulated algebraic constraints, $f^a_{\alpha_i}$, [41]

$$\begin{aligned} \dot{x} &= f^d_{\alpha_i}(x, u, t), \\ 0 &= f^a_{\alpha_i}(x, u, t). \end{aligned} \tag{4}$$

In Fig. 6, the change to mode α_2 occurs when the state in mode α_1, x_{α_1}, reaches a threshold value. In general, a mode transition relation $\gamma^{\alpha_{i+1}}_{\alpha_i}(x, u, t) \geq 0$ defines the change from mode α_i to α_{i+1} when true.

The state space in a mode α_i consists of two parts: (i) the domain where f_{α_i} is properly defined and (ii) a *patch*, where $\gamma^{\alpha_{i+1}}_{\alpha_i}$ does not invoke a mode change. In Fig. 6, the patches are shown as white areas in the state space.

When the boundary of the patch in α_1 is reached, a mode transition as defined by $\gamma_{\alpha_1}^{\alpha_2}$ is invoked. In the new mode, a patch defined by $\gamma_{\alpha_2}^{\alpha_3}$ is entered in which the state can continue to evolve, now governed by the field f_{α_2}.

When a mode transition from α_i to α_{i+1} takes place, in general, the state x_{α_i} may change its value between α_i and α_{i+1}. Without loss of generality it is first assumed that the explicitly defined state transition function, $x_{\alpha_{i+1}} = g_{\alpha_i}^{\alpha_{i+1}}(x_{\alpha_i}, u_{\alpha_i}, t)$, is the identity function, i.e., $x_{\alpha_{i+1}} = x_{\alpha_i}$, as shown in Fig. 6 by the perpendicular dashed arrow.

2.3 How Are Hybrid System Models Designed?

The behavioral definition of hybrid dynamic systems left open the specific underlying model structure that is employed. There are two basic approaches to the design of hybrid dynamic system models: (i) an implicit approach and (ii) an explicit approach.

Implicit Models

In modeling for analysis, often an implicit approach is favored. This is best illustrated by considering the modeling of a physical device under control, the *plant*, such as the power window mechanism of the system in Fig. 3. The continuous-time behavior that models moving up the window is given in Eq. (1) in an implicit form. The equations are not converted in an explicit form that allows their solution yet. For example, the velocity of the window, v_{window}, is a state variable and so its time-derivative needs to be computed by the system of equations in Eq. (1). However, applying Newton's second law of motion leads to a formulation with the force, F_{window}, on the left-hand side. To arrive at an explicit computation of $\dot{v}_{window}$, the equation has to be reformulated into

$$\dot{v}_{window} = \frac{F_{window}}{m_{window}}. \tag{5}$$

The reason for the implicit formulation is modeling convenience. It is often easier to model physical constraints in an implicit, constraint-based, form such as Newton's second law of motion. Another such constraint is, for example, conservation of momentum,

$$\sum m_i v_i^+ = \sum m_i v_i^-, \tag{6}$$

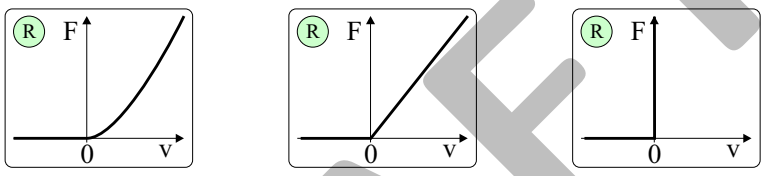
which states that the momentum, $m_i v_i^-$, of a body, m_i, with velocity, v_i^-, before a collision and after a collision, $m_i v_i^+$, should be the same. Which velocities are input to this constraint, and which ones are computed from the constraints is left implicit.

In the same vein, mode changes in models of physical system are often best represented in an implicit manner. For example, when the window reaches the top of its frame, further upwards movement is restricted by a rapid increase

of resistance to motion. This can be modeled in three different manners, illustrated in Fig. 7:

- By a *nonlinear* resistance that is distance dependent and rapidly increases when the window reaches the top of the frame position, Fig. 7(a).
- By a *piecewise linear* resistance that abruptly changes the resistance value when the window reaches the top of the frame position, Fig. 7(b).
- By an *ideal switch* of the resistance that enforces a hard stop when the window reaches the top of the frame position, Fig. 7(c).

These three classes of modeling approaches are typical in the modeling of physical systems with perceived discontinuities. The latter two lead to hybrid dynamic systems.



(a) Nonlinear Model (b) C^0 Hybrid Model (c) Hybrid Model

Fig. 7. Levels of Abstraction of a Cylinder End-stop

To illustrate how this could be formulated as a mathematical relation, the equation $F_{window} + F_{lift} = F_{motor}$ in Eq. (1) is replaced by

$$0 = L \cdot (F_{window} + F_{lift} - F_{motor}) + (L - 1) \cdot v_{window} \tag{7}$$

where L is a logical variable that switches based on the window position, x_{window}, where $L = x_{window} \leq x_{top}$. Such 'local' and 'implicit' switching of equations is intuitive and convenient in the domain of plant modeling.

Explicit Models

Controller modeling typically takes a different perspective because the modeled behavior is the explicit purpose of the design. As such, implicit modeling constructs such as algebraic loops that have a cyclic dependency, e.g., $x = -2x+3$, are not desired or even illegal (e.g., for real-time code generation).

The corresponding continuous-time formalism that is often used in control system design is explicit ordinary differential equations (ODEs). Whereas the DAEs used for implicit modeling are of the form $0 = f(\dot{x}, x, u, t)$, the explicit ODE form is $\dot{x} = f(x, u, t)$. This form is especially popular because amenable to control law synthesis, in particular when it is linear. For example, when

the behavior of the lift mechanism of the power window in Fig. 3 is available in an explicit linear ODE form, root-locus methods can be applied to design a controller that can achieve desired behavior such as the position overshoot in response to a step input [4].

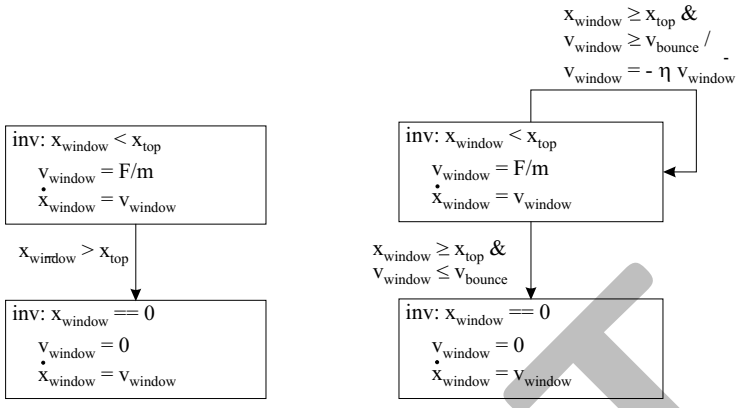
The discrete-event behavior is often modeled by finite state machines. For example, *state transition diagrams* are often used graphical finite state machine models. The different modes of behavior then correspond to the states of the transition diagram, and, consequently are explicitly available. In addition, the imperative nature of state transition diagrams makes that the state transition behavior becomes operational. The modeler has made the manner in which the transitions between states are made explicit. In the implicit formulation, constraints are provided on the modes, and a transformation to an operational form has to still be derived.

The combination of state transition diagrams and ODEs has been a popular modeling approach for control design as its explicit nature makes it amenable to reachability analyses (e.g., [19]). Such *hybrid automata* consist of discrete states with associated ODEs. When in a state, the associated ODE governs continuous-time behavior. The transitions between states are enabled by *guards* and when enabled the transition from one state to another *may* be taken. Each state has an *invariant* associated with it as well that cannot be violated while in that state. When the continuous-behavior in a state reaches a point where it would violate the corresponding invariant, an enable transition *must* be available and taken.

To illustrate, Fig. 8(a) shows a hybrid automaton for the power window behavior that is modeled in an implicit form in Section 2.3. The state (also called *mode* as it captures the mode of operation of the entire system) on the left-hand models the window moving up by the two equations for v_{window} and $\dot{x}_{window}$. The invariant of the state is $x_{window} < x_{top}$, which requires that the window can move up freely, i.e., it is not pushing against the top of the frame.

When the top of the frame is reached, $x_{window} \geq x_{top}$ becomes true and the transition into the right-hand state is enabled. Because the invariant $x_{window} < x_{top}$ becomes false at the same time, the transition is forced to be taken, and the system moves into a state where the window does not move anymore ($v_{window} = 0$).

In case some bounce-back effect would be modeled because of the window colliding with the top of the frame, the continuous-time state v_{window} could be re-initialized with a value $-\eta v_{window}$, where η is a coefficient of restitution. This re-initialization can be included on the enabled transition as an action $v_{window} = -\eta v^{-}_{window}$, as shown in Fig. 8(b). Note the $-$ superscript to indicate the left-hand limit value of the continuous-time behavior.



(a) No bounce back (b) With bounce back

Fig. 8. Hybrid automaton of power window endstop behavior.

3 Hybrid Dynamic System Behaviors

The different aspects of generating behaviors for hybrid systems are discussed and efficient methods to do so are presented.

3.1 An Operational Structure

To execute a hybrid dynamic system four different types of behavior need to be handled (see Fig. 9):

- Continuous-time behavior as specified by f needs to be generated.
- The crossing of thresholds of continuous-time variables needs to be detected and transitions between modes of behavior inferred, as specified by γ.
- The continuous state variables need to be initialized and possibly reinitialized (think of the window that bounces back, reversing its velocity), as specified by g.

3.2 Continuous-Time Behavior

Continuous-time behavior is typically modeled by differential equations, either as ODEs or DAEs. To generate behaviors, a *numerical solver* such as DASSL [34] can be used to integrate the time-derivative behavior.

Typically, the derivative with respect to time is computed at one or multiple points, after which a weighted mean is taken to infer the direction of

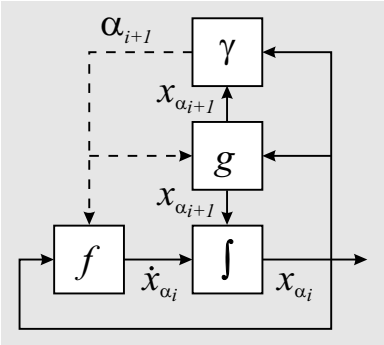


Fig. 9. Functional model of hybrid dynamic systems.

behavior. In its most straightforward form, called a *forward Euler* integration scheme, the time-derivative, $\dot{x}(t_k)$, at a time t_k is multiplied by the step in time, $h = t_{k+1} - t_k$, and the resultant is added to the current state, $x(t_k)$, to produce the state at t_{k+1}

$$x(t_{k+1}) = h\dot{x}(t_k) + x(t_k) \tag{8}$$

More sophisticated solvers rely on continuity constraints on the continuous-time behavior. For example, a second order Runge-Kutta algorithm is implemented by

$$\begin{aligned} m_1 &= hf(x(t_k), t_k) \\ m_2 &= hf(x(t_k) + 0.5m_1, t_k + 0.5h) \\ x(t_{k+1}) &= x(t_k) + m_2 \end{aligned} \tag{9}$$

This scheme is based on a Taylor expansion

$$x(t_k+h) = x(t_k)+hf(t_k, x(t_k))+\frac{h^2}{2}(\frac{\partial f(t_k, x(t_k))}{\partial t}+\frac{\partial f(t_k, x(t_k))}{\partial x}f(t_k, x(t_k)))+O(h^3), \tag{10}$$

and so it requires the integrated behavior to be continuous up to its third order for the error estimate to be valid. If this continuity constraint is not satisfied, the integration may still succeed, but without guaranteed error convergence behavior.

Note that often, in hybrid system literature, a forward Euler is applied to arrive at an overall discrete representation. Without incorporating continuity constraints, the hybrid nature of the original system is, therefore, removed.

3.3 Handling Mode Transitions

Event Detection and Location

The continuous-time behavior of a model typically affects the discrete-event part by events that are generated by continuous-time variables crossing thresh-

old values. For example, when the power window in Fig. 2 reaches the top of the door frame.

The exceeding of a threshold can be formulated as an inequality on 0 that either includes the boundary, $\gamma_{\alpha_i}^{\alpha_{i+1}}(x,u,t) > 0$, or not, $\gamma_{\alpha_i}^{\alpha_{i+1}}(x,u,t) \geq 0$. In general, multiple mode transition relations such as both a relation $\gamma_{\alpha_i}^{\alpha_{i+1}}(x,u,t) > 0$ as well as $\gamma_{\alpha_i}^{\alpha_{i+2}}(x,u,t) > 0$ may apply in any given mode, however, in case of deterministic models, only one of these should be active. Note that the behavior of the continuous-time states is best 'left-closed', i.e., each of the abutting intervals of continuous behavior includes its starting point to satisfy causality requirements [21].

An implementation of the event generation requires two parts:

- First it has to be *detected* whether the threshold has been exceeded when an integration step ΔT is about to be taken.
- Second, the values of the continuous states, input, and time for which the threshold was first exceeded have to be determined. In other words, the point in time at which the zero crossing event occurs has to be *located*.

A robust implementation of the event location can be done by means of a bisectional search to find when the first event (in general there are multiple events) in the ΔT interval occurs. In this case, if an event is detected, the stepsize is reduced from ΔT to δt_m, where δt_m is computed based on whether an event occurs, $\sigma = 1$, or not, $\sigma = 0$, in the interval δt_i as follows

$$\begin{aligned} \delta t_{i+1} &= \delta t_i + \Delta t_i(1-\sigma) \\ \Delta t_{i+1} &= \tfrac{1}{2}\Delta t_i \end{aligned} \tag{11}$$

The initial values for this iteration are $\delta t_0 = 0$ and $\Delta t_0 = \Delta T$, and the iteration terminates after a fixed number of *a priori* prescribed steps, m.

Other approaches to finding the point in time where the threshold is first exceeded such as *regular-falsi* and the *illinois* algorithm exist [20] and in practical simulation engines a combination of the different approaches tends to be employed.

Note that this requires the model not to change its mode until the zero-crossing is located. For example, the absolute value may not be effected while the zero crossing is located. This implies that a negative value may indeed be computed, and thus behavior is continuous.

Mode Transition Inferencing

The discrete events generated by the zero-crossing function may cause a mode change in the model. The relation $\gamma_{\alpha_i}^{\alpha_{i+1}}(x,u,t)$ transitions the model from a mode α_i to a mode α_{i+1}. It takes as arguments the continuous-time state vector, x, the exogeneous input, u, and time, t. The mode change is typically implemented as an instantaneous transition, which means there is no passage of time. As such, it is best modeled by an untimed formalism.

The discrete state transition behavior can be represented in two basic forms: (i) by *combinational* logic and (ii) by *sequential* logic. This is a very important distinction as it makes quite a difference in the complexity of analyses of the hybrid dynamic systems.

Combinational Logic

An important approach that applies combinational logic is complementarity modeling, which has proven very successful in the domain of collision modeling [18, 35] as well as, for example, power electronics [15]. Complementarity formulations are further employed in the work on mixed logical dynamical (MLD) systems [38]. A linear complementarity model is of the form

$$\begin{aligned} &y = Bz + b \\ &yz = 0 \\ &y \geq 0, z \geq 0 \end{aligned} \tag{12}$$

So, only one of the variables y and z can be positive while the other has to be 0. This is an intuitive representation for points of contact in mechanical systems, where there is either some distance larger than 0 between two bodies and no force acting, or there is no distance and a force larger than 0 acting.

Sequential Logic

In case the logic contains memory, a combinational representation does not apply and sequential logic is needed. This holds true, for example, for the modeling of sequences of collisions such as found in Newton's Cradle [24].

Sequential logic can be represented by a state machine, ϕ, often with a finite number of states [16]. Many graphical formalisms exist that are of a discrete state, sequential logic, nature. For example, there are state transition diagrams [16], statecharts (an example of these are given in Fig. 4) [14], and Petri nets [6, 32]. Computationally, a finite state machine can be represented by a five tuple

$$\phi = \langle \alpha, \alpha_0, \sigma, \delta, \nu \rangle, \tag{13}$$

in which the state transition function, δ, changes the active state, α, in response to events σ, while actions, ν, are generated. The initial state is given by α_0.

3.4 Re-initialization of State Variables

In response to a mode transition inferred by $\gamma_{\alpha_i}^{\alpha_{i+1}}$, the continuous-time state variables may be re-initialized, as governed by $g_{\alpha_i}^{\alpha_{i+1}}$. For example, in case of the power window bounce back, the window reverts its velocity upon impact with the frame and the corresponding state variable, v_{window}, needs to be reinitialized from a positive to a negative value. An important implication of

this is that the numerical solver may have to be reset. Sophisticated numerical solvers build up a history of time points and based on that history attempt to take larger steps in time to compute the next integration point. If an integrator state is reset, even if no mode transition occurs, this history becomes invalid and needs to be cleared. In this case, the integrator starts off with a minimial step size once continuous-time behavior resumes.

Note that in order to specify the re-initialization, semantics need to be defined for the two values around a discontinuity, the *a priori* and *a posteriori* values. In this work, if necessary, the *a priori* values are indicated by a '-' superscript, and the *a posteriori* values by a '+' superscript, see Eq. (6) for an example.

Finally, the number of continuous-time state variables may change between mode transitions. For example, while modeling a highway, vehicles may enter and leave, and, therefore, continuous-time states be included or discarded. This, again, will require a reset of the numerical solver, depending on the integration algorithm that is being used.

4 An Implementation

To generate behaviors for a hybrid dynamic system, two basic approaches exist: (i) time-driven execution and (ii) event-driven execution. The former has the execution driven by moving time forward, often by means of a numerical solver. The latter jumps in time in response to discrete event.

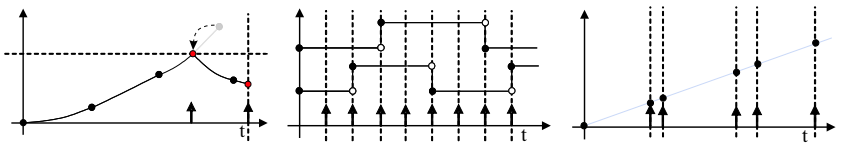
4.1 Classes of Events

For purposes of discussion, it is convenient to first identify two classes of events [3]: (i) *time events* and (ii) *state events*. A time event is an event that occurs at a given point in time, independent of the continuous-time state of the model, x, and the forcing function, u. Therefore, a time event is predictable. A state event, on the other hand, is generated based on the values of the continuous-time state and the forcing function.

4.2 Classes of Temporal Behavior

Depending on the particular type of behavior that is generated, one or the other may be more efficient. For the purposes of execution analysis, three categories of behavior over time can be distinguished, illustrated in Fig. 10. These behaviors correspond to those shown in the power window example in Fig. 3.

In Fig. 10(a), a behavior is shown that evolves continuously in time. This evolution is typically modeled by differential equations and the traces are generated using numerical solvers. In Fig. 10(a), there is a time event that



(a) Continuous time (b) Discrete time (c) Discrete event

Fig. 10. Different types of execution.

occurs at the point in time that is marked by the dashed vertical. A state event is generated at the point in time when the continuous-time behavior, the solid line, exceeds the dashed horizontal. When the threshold is exceeeded, it is backtracked to the earliest point in time where this occurred, indicated by the dashed arrow. The events are indicated by vertical solid arrows.

In Fig. 10(b), a behavior is shown that only contains time events that are of a periodic nature. This is typical in the design of an embedded controller, where the system operates at a fixed sample time, the *base rate*, and the different aspects of the control may execute at the fast base rate or any have a sample time that is an integer multiple of the base rate.

In Fig. 10(c), a behavior with only time events is shown, but the events are not periodic. Such behaviors are typical in network modeling, where the time a packet arrives is variable as well as the time it takes to move the packet through a network. Another application is the modeling of the task scheduling on a microprocessor network, where events occur when a task is completed. This time is typically variable and the execution may even be pre-empted by higher priority tasks, included further variability of the execution time.

4.3 Time-Driven Execution

In a time-driven execution, a numerical solver is applied that governs the advance of time. The numerical solver is typically given a differential equation, a start time, a set of initial states, a set of input, and a stop time. It then attempts to solve the differential equation to generate a trace of numerical values for the states from the start time to the stop time. This trace may consist of any number of steps and corresponding integration points.

The trace is generated by the numerical solver computing the size of a step in time to take, based on the time-gradients of the differential equations, as, for example, in Eq. (9). If there is a steep gradient because of fast changes in the continuous-state values, small steps are taken, whereas when the behavior is relatively slow, larger steps are taken.

The stop time is either taken to be the end time of the requested simulation, or, in case there are time events, the first of such an event is provided as the stop time. If only time events are present, and they are statically known,

then a schedule for execution can be pre-compiled. This is the case if all of the events are periodic in nature. The smallest common denominator of all the different sample times is determined and then each of the events are executed at an integer multiple of this base rate.

For aperiodic events, this approach may not be applicable, as the accuracy with which the time at which events can be effected degenerates to the period of the base rate, which is typically too coarse.

In case state events are present, a numerical solver with zero-crossing detection built-in is desirable. Such a numerical solver returns before the final time is reached and reports the time at which a zero-crossing was found.

4.4 Event-Driven Execution

Another approach to generating behavior takes an event-driven perspective. This is particularly efficient for time events that are aperiodic and often have a stochastic component to their time of occurrence.

Rather than having a numerical solver move time forward till each of the events occurs, time immediately jumps from one event time to another. To efficiently implement this, typically an event calendar is used that keeps track of all events that are scheduled to occur. An example of an event calendar is presented in Table 1. It shows how the first upcoming event to occur is at 20 (ms), followed by another at 2020 (ms).

Table 1. Event calendar

Time	**Event**
20 (ms)	*open_tonneau*
2020 (ms)	*move_top_up_cmd*
2150 (ms)	*move_down_window*
2250 (ms)	*stop_moving_window*

An event driven approach is much more efficient in handling aperiodic time events as it does not require a time-driven solver to move time forward by means of integration over time. Typically hundreds of thousands of events can be conveniently simulated in a matter of seconds.

To achieve such efficiency, it is critical that the event calendar be implemented in an efficient manner. In particular, efficient search of the event calendar needs to be facilitated, because new events must be inserted in the correct place, and events that were scheduled at one time may have to be located and retracted at a later time.

One implementation has the part of the event calendar as a doubly linked list list while the other part is a heap based priority queue. A doubly linked list is a list of items where each item has a link to the next item as well as the previous item. This allows each item in the list to be accessed in a forward as

well as backward manner, and, so, deleting any item can be done in constant time. A heap is a complete tree where every item has a key that is greater or less than the key of its parent. So, a heap based priority queue is a more efficient mechanism for managing a large number of entities.

This combines the benefit of quick insertion of events in the doubly-linked list where the heap is more efficient for a large calendar and used for the events beyond a given maximum number for the list part.

Note that an event-driven implementation can be exploited to generate behaviors for continuous-time models as well. In this approach, the numerical solver is modeled as a discrete event component and the computations at each time step are aperiodic events to be handled (e.g., [5]).

4.5 Combining the Execution Types

In many applications of modeling and simulation, in particular for hybrid dynamic systems, it is common to have an extensive model component that is of an aperiodic discrete event nature, as well as a component that is of a continous-time nature. For example, the power-window system in Fig. 3 may contain a detailed continuous-time model of the door with window, *dc* motor actuator, signal conditioning hardware and the like, which may contain some discontinuities modeled by local finite state machines. Similarly, the controller may operate in a discrete time manner because it executes with a given sample rate and implement an extensive signal processing component, data analysis computations, supervisory control and further elements. The CAN behavior may be modeled in detail by a discrete event model which captures the generation of packets, routing them through the network, possibly resending dropped packets, and like behavior.

To efficiently execute models that comprise all of these behaviors, it is desired to have dedicated solvers for each of the separate components. This requires combining the numerical solver for continuous time behavior with a scheduler for the discrete time behavior and an event calendar handler for the discrete event behavior.

Because of the predictability of the discrete time events, these are often conveniently integrated into the numerical solver by means of a *getNextEventTime* call that tells the solver the time up till which it should solve for the continuous-time behavior.

The discrete event part is a different matter, though. Because many events may have to be processed that have no bearing on the continuous-time behavior, it is more efficient to evaluate the discrete event part with a dedicated solver. Though this is more efficient, it requires coordination with the time-driven execution to ensure it is synchronized upon communication with the discrete-event part. This is illustrated in Fig. 11, which shows the time-driven part at the top and the event-driven part at the bottom. Events that affect the behavior in the part other than where they are generated are being communicated between the two parts.

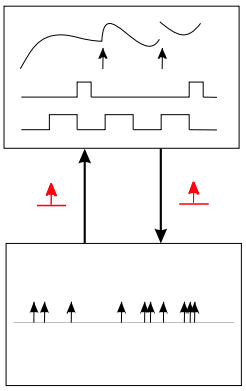


Fig. 11. Combining event-driven and time-driven execution.

The use of separate solvers requires efficient technologies for the synchronization. One approach is to operate the time-driven and event-driven parts in *locked step*. In this approach, neither one of the two parts is allowed to start leading the other. One step is made by either one of the parts and then the other catches up. This cancels any benefit of the separation in terms of execution efficiency.

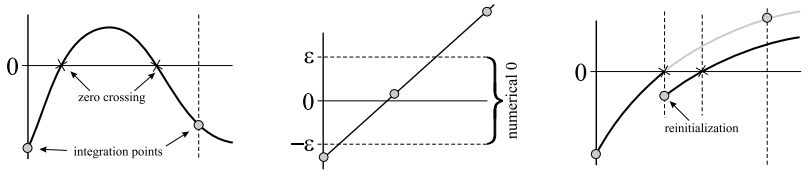
In case one of the two parts is allowed to lead, it may happen that a processed event or generated trajectory proves to be invalid after the other part catches up. This requires a *roll back* which can be achieved either by storing snapshots of the discrete or the continuous-time state. The actual state at which the interaction occurred can then be reconstructed from this.

5 Advanced Topics in Hybrid Dynamic System Simulation

5.1 Zero-Crossing Detection

The typical approach to zero-crossing detection compares the sign of a function result and if it changes, it has crossed zero. This approach may fail if the zero-crossing function has even zeros in the interval ΔT between the two evaluated points as determined by the numerical integration algorithm. This is illustrated in Fig. 12(a). In general the zero crossing function, z, is a function of the model state, but it does not contribute to its continuous dynamics, f. Therefore, numerical integration can proceed without taking the dynamics of z into account, and when these are faster than the dynamics of f, this situation may arise.

One solution to this is to include the dynamics of z in the model dynamics so the numerical solver adjusts its step-size when too large an error (this will be caused by even zeros) is found [33].



(a) Even roots

(b) Comparing a float

(c) Discontinuity sticking

Fig. 12. Difficulty in zero-crossing detection and location.

Alternatively, if k is the index of the points in time at which the numerical solver computes a value, t_k, then, if k is considered dense, rather than discrete, the step size between consecutive points, h, becomes $h(k) = \frac{dt}{dk}$. This step size can be chosen as

$$h(k) = -\eta \frac{z(x)}{\frac{\partial z}{\partial x} f} \tag{14}$$

to control the step size selection [8]. Given the sensitivity of the zero-crossing function, z, with respect to the step size

$$\frac{dz}{dk} = \left(\frac{\partial z}{\partial x} \frac{dx}{dt} \right) \frac{dt}{dk} = \left(\frac{\partial z}{\partial x} f \right) h(k) \tag{15}$$

this results in z behaving as

$$\frac{dz}{dk} = -\eta z \tag{16}$$

which gradually converges to 0 without actually crossing it. Instead, the solution to Eq. (16) is $g(k) = g(0)e^{-\eta k}$, and $z(k)$ approaches 0 exponentially as k tends to infinity. This requires a number of additional computations during numerical integration, though, such as the sensitivity, and, therefore, is computationally less attractive.

Other than exceeding the threshold value, another issue with zero crossing detection arises when the zero-crossing function g returns 0 exactly. This does not constitute a *crossing*, and, therefore, is not detected as such. Root-finding facilities of numerical solvers may require the function z to actually change sign between two integration steps to report that a crossing has occurred. For example, when z starts off at 0 and then moves away from it, no zero crossing event would be generated and consequently no mode change could occur.

One solution to this issue is the use of a zero-crossing function that is not at 0 exactly. Rather, it is chosen to be $-\epsilon$ when z is negative and ϵ when z is positive, with ϵ very small. When $-\epsilon < z < \epsilon$, both zero crossing functions, $z - \epsilon$ and $z + \epsilon$ are used.

One effect of this implementation is the need for a '*numerical* 0', a $\pm\epsilon$ band around 0 in which the zero crossing function is considered to be the 0

value of the sign function.

$$sign(x) = \begin{cases} -\ \forall_x(x < -\epsilon) \\ 0\ \ \forall_x(-\epsilon \leq x \leq \epsilon) \\ +\ \forall_x(x > \epsilon) \end{cases} \tag{17}$$

As long as the zero crossing function evaluates to a value within the $\pm\epsilon$ band, its sign is considered 0.

This approach has an important bearing on the analytical correctness of comparing for equality. If a simulation contains the comparison $x == 0$ when x is a continuous signal, a zero crossing function $z(x, u, t)$ is used to find the value of x that satisfies this equality within a certain tolerance. For this value, x', the zero crossing function has sign 0, whereas the strict equality $x == 0$ may not be satisfied. Figure 12(b) shows how the signal x may therefore cross 0 without having $x == 0$ evaluate to true because the analytical solution that satisfies this comparison is never evaluated by the numerical integration.

Another issue that is less critical but still causes inefficiencies occurs because of re-computation of the model variables after the zero crossing has been located. Because of numerical inaccuracies, even when no changes to the states are made that are used to compute a zero crossing variable, the return value of the function may still differ. This is illustrated in Fig. 12(c), which shows that re-starting the simulation results in a function return value that is slightly different from the computed value immediately before the zero crossing. Continuing simulation leads to the same zero crossing being detected and located again. This phenomenon has been referred to as *discontinuity sticking* [33].

5.2 Mode Changes

Re-initialization

An important phenomenon of the general DAE form $0 = f_\alpha(\dot{x}, x, u, t)$ is that the system may only be allowed to move in part of the *generalized* state space [42]. This is the case, for example, when the power window in Fig. 2 collides with an object, m_{object}, in a perfectly nonelastic manner. After the collision the window and object proceed to move with equal velocity, and this leads to the equations

$$\begin{aligned} &m_{object}\dot{v}_{object} + m_{window}\dot{v}_{window} + R_{lift}v_{window} = ru_{motor} \\ &v_{object} = v_{window} \\ &\dot{x}_{window} = v_{window} \\ &\dot{x}_{object} = v_{object} \end{aligned} \tag{18}$$

to described the dynamic behavior. In the form of a *matrix pencil* [7] with a forcing function, $E\dot{x} + Ax + Bu = 0$, this becomes

$$\begin{bmatrix} m_{window} & m_{object} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{v}_{window} \\ \dot{v}_{object} \\ \dot{x}_{window} \\ \dot{x}_{object} \end{bmatrix} + \begin{bmatrix} R_{lift} & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_{window} \\ v_{object} \\ x_{window} \\ x_{object} \end{bmatrix} + \begin{bmatrix} -r \\ 0 \\ 0 \\ 0 \end{bmatrix} [\, u_{motor} \,] = 0 \tag{19}$$

Here, because of the constraint that $v_{window} = v_{object}$, only the part of the state space for which this holds can be accessed. This implies that there is also a limitation on the reachability in the x_{window}, x_{object} space, but this is a non-holonomic constraint rather than that it is disallowed.

In a general sense, this can be represented in geometric terms shown in Fig. 13. Here, the system evolves in a mode α_1 till the boundary of a patch is reached. At this point in time, the system transitions into another mode, but the continuous-time state is not in the allowed space, which is marked by the thick solid line. In order to arrive at a consistent situation, the continuous-time state has to be in the allowed space, and the exact value is computed based on the *jump space*, which is the space in which the required instantaneous changes are allowed.

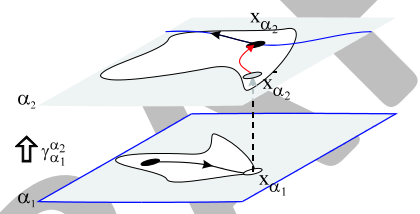


Fig. 13. A projection

In the linear case, this computation in the jump space corresponds to a projection. This projection can be computed in several ways [11, 12, 17, 22, 23, 40, 42]. To illustrate the method in [22], the Weierstrass normal form is derived for the velocity part of the equations in Eq. (19). The positions are geometric states that do not change discontinuously, and, therefore, are irrelevant for the example. This leads to the system of equations

$$\begin{bmatrix} m_{window} & m_{object} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{v}_{window} \\ \dot{v}_{object} \end{bmatrix} + \begin{bmatrix} R_{lift} & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} v_{window} \\ v_{object} \end{bmatrix} + \begin{bmatrix} -r \\ 0 \end{bmatrix} [\, u_{motor} \,] = 0 \tag{20}$$

After applying the following change of basis

$$\begin{bmatrix} v_{window} \\ v_{object} \end{bmatrix} = \begin{bmatrix} 1 & -\frac{m_{object}}{m_{window}} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\frac{m_{window}}{m_{window}+m_{object}} & 1 \end{bmatrix} \begin{bmatrix} \bar{v}_{window} \\ \bar{v}_{object} \end{bmatrix} \tag{21}$$

the following matrix pencil is arrived at

$$\begin{bmatrix} m_{window} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \dot{\bar{v}}_{window} \\ \dot{\bar{v}}_{object} \end{bmatrix} + \begin{bmatrix} 0 & \frac{m_{window}}{m_{window}+m_{object}} R_{lift} \\ 0 & \frac{m_{window}+m_{object}}{m_{window}} \end{bmatrix} \begin{bmatrix} \bar{v}_{window} \\ \bar{v}_{object} \end{bmatrix} + \begin{bmatrix} -r \\ 0 \end{bmatrix} \begin{bmatrix} u_{motor} \end{bmatrix} = 0 \tag{22}$$

The matrix $\begin{bmatrix} m_{window} & 0 \\ 0 & 0 \end{bmatrix}$ contains a finite space (top-left entry) $\begin{bmatrix} m_{window} \end{bmatrix}$ and infinite space (bottom-right entry) $\begin{bmatrix} 0 \end{bmatrix}$ Note that this is an *index 1* system of equations, as the infinite space is the null matrix. Therefore, its nilpotency is 1, and this is also referred to as the *index* of the system of equations [41]. The nilpotency is an indicator of how many stages of substitution are required to compute all infinite variables. In Eq. (22), because index 1, the infinite variable, $\bar{v}_{object}$, can be computed in one stage, i.e., $\bar{v}_{object} = 0$.

The finite part in Eq. (22) consists of $\bar{v}_{window}$ and this can be converted into a regular 1-dimensional ODE by inverting the matrix $\begin{bmatrix} m_{window} \end{bmatrix}$ and left-multiplying. Because a regular ODE, there is no discontinuous change in the variable $\bar{v}_{window}$, or

$$\bar{v}_{window} = \bar{v}^-_{window} \tag{23}$$

where the '-' superscript indicates the final value of $\bar{v}_{window}$ before the mode change (in case of the initialization before simulation starts, it is the user supplied initial value).

Now, from v^-_{window} and v^-_{object}, the value $\bar{v}^-_{window}$ can be computed using the inverse change of basis. Straightforward computations yield

$$\bar{v}^-_{window} = v^-_{window} + \frac{m_{object}}{m_{window}} v^-_{object} \tag{24}$$

which equals $\bar{v}_{window}$. With $\bar{v}_{object} = 0$ the change of basis can now be applied to yield

$$v_{window} = \frac{1}{m_{window} + m_{object}} (m_{window} v^-_{window} + m_{object} v^-_{object}) \tag{25}$$

and from $v_{window} = v_{object}$, v_{object} can be computed. Details on the derivation are available in previous work [22].

Sequences of Mode Changes

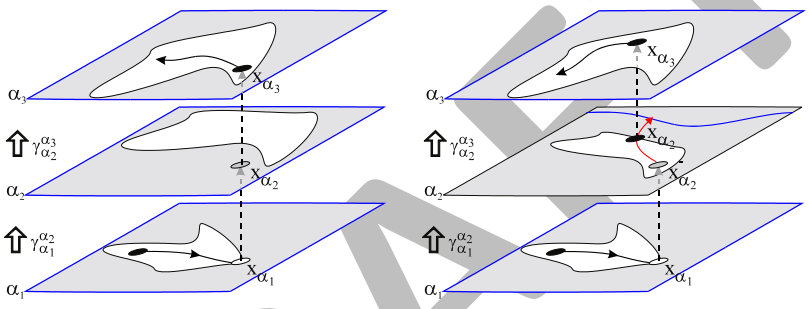
After one mode change, $\gamma^{\alpha_{i+1}}_{\alpha_i}$, and computing the initial values of the continuous-state variables in the new mode, a new transition, $\gamma^{\alpha_{i+2}}_{\alpha_{i+1}}$, may follow immediately as shown in Fig. 9 [25]. The mode change causes a new mode to be arrived at, after which re-initialization is performed again. This process repeats till no further mode changes occur and the system proceeds to evolve continuously again.

The values of the continuous-time state in modes where there is no continuous-time behavior can be of two types [25]:

- They do not change the value from the previous mode, and the mode is a so-called *mythical mode*.

- The re-initialization causes a change in value from the previous mode, which results in an isolated point, a so-called *pinnacle*, with no continuous-behavior in that mode.

In case of sequences of mode transitions, the left-closedness mentioned in Section 3.3 may (have to) be relaxed, though, in particular for sequences of pinnacles. For example, a pressure relief valve may move through a sequence of opening and closing cycles before the pressure has subsided to below the threshold for opening the relief valve. In a sufficiently detailed model, this sequence occurs over time, but when small physical phenomena are not included in the model, for example, to simulate it in real time, this sequence may occur at one point in time [23].



(a) State invariance (b) State re-initialization

Fig. 14. Sequences of Mode Transitions

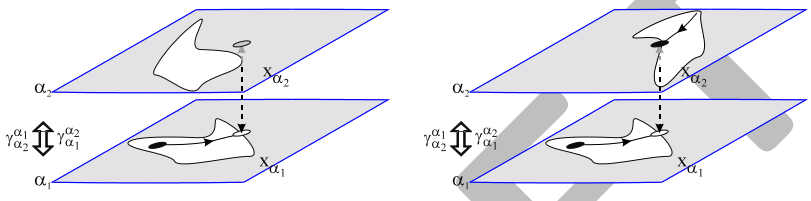
An important behavior that is not properly dealt with in simulation tools at present, is the crossing of the patch boundary in an intermediate mode such as in mode α_2 in Fig. 14(b). The proper value of the continuous state to be applied for initialization in mode α_3 appears to be the point at which the projection crosses the patch boundary. However, there may be physical phenomena that are best modeled with a different semantics. This is still subject of research.

6 Pathological Behavior Classes

Once sequences of mode changes may occur, models can be constructed that contain loops of mode changes, i.e., a previously visited mode is re-visited, without continuous-time behavior evolving in between.

Two classes of behavior are illustrated in Fig. 15. In Fig. 15(a), the pathological case is shown that violates the *divergence of time* principle [27]. Here,

the state is initialized inside of the patch in mode α_1. It evolves continuously till it reaches the patch boundary as defined by $\gamma_{\alpha_1}^{\alpha_2}$. When the state x_{α_1} is then transferred to mode α_2, it is outside of the patch as defined by $\gamma_{\alpha_2}^{\alpha_1}$ (note the exchange in subscripts of α). This causes the state to be transferred back to α_1 where it is outside of the patch as defined by $\gamma_{\alpha_1}^{\alpha_2}$. Thus, a loop of discrete changes between modes arises.[2] Because these are instantaneous, no time elapses, and, therefore, the model stops evolving in time. In other words, time does not diverge. Since this behavior is not observed in physical systems, such models are considered anomalous.



(a) Non-divergent. (b) Sliding.

Fig. 15. Sequences of mode transitions.

Similar but different behavior is illustrated in Fig. 15(b). Here, after reaching the patch boundary in α_1, the state transfers onto the patch boundary in α_2 as defined by $\gamma_{\alpha_2}^{\alpha_1}$ (note again the exchange in subscripts of α). Because it is the patch boundary, the state transfers back to α_1 after an infinitesimal step in time. This step results in a value x_{α_1} that may be immediately inside the patch in α_1 as defined by $\gamma_{\alpha_1}^{\alpha_2}$ and so another infinitesimal step will transfer the state back to α_2.[3]

Far-fetched and pathological as it may seem, this behavior, referred to as *chattering* or *sliding mode* behavior, is actually aimed for by robust control design methodologies [39] (e.g., it is used in anti-lock braking systems), as it is relatively insensitive to plant model parameter variations. Unlike the behavior in Fig. 15(a), here the state does continue to evolve in time and the divergence of time principle is satisfied. To efficiently derive the actual behavior along the *switching surface* as defined by the patches in mode α_1 and α_2 two methods exist: (i) equivalence of control [39] and (ii) equivalence of dynamics [9, 31]. Though there are classes of models for which these 'regularizations' result in the same behavior, in general they may differ.

[2]Note that a loop may involve any finite number of modes.

[3]Note how left-closedness is violated in this particular instance of behavior. In general, an infinitesimal 'hysteresis' effect may be present to guarantee left-closedness again.

Finally, another class of pathological behaviors can be identified, namely Zeno behavior.[4] Behaviors that are Zeno do progress in time by a non-infinitesimal value each time a mode transition occurs. However, this time reduces upon each transition as a converging series. For example, in case the time is halved upon each transition, the transition series converges to a limit value in time

$$t_f = \Sigma_i \frac{1}{2^i}. \tag{26}$$

that is never exceeded. In case the bounce back of the window in the hybrid automaton in Fig. 8(b) does not include the threshold clause, the bounce transition would be taken indefinitely, with shorter and shorter intervals of time in between. Depending on the coefficient of restitution, η, the new interval between bounces will be a fraction of the present one. The increasingly smaller intervals will converge to a limit point in time beyond which time would not progress. Therefore, though time diverges locally, it does not do so globally.

It is now possible to compare the three mode re-visiting behaviors.

- *Divergence of time*: infinitely many discrete steps in zero time. Time remains the same.
- *Chattering*: infinitely small time steps. Evolves past any value in time.
- *Zeno*: infinitely many time steps in a finite, nonzero, time interval. Does not evolve past a limit point in time.

Unfortunately, the general hybrid dynamic systems literature is loose in its use of these terms (e.g., behavior that is locally not divergent in time is often called Zeno as well).

7 Conclusions

All this exemplifies the richness and complexity of mode transition behavior in hybrid dynamic systems. It illustrates that though an explicit representation such as hybrid automata [2] may be a powerful vehicle for analyses, it is by no means trivial to design such a hybrid automata for complex physical system models. It means that the expressiveness of the formalism is achieved by a rather significant conceptual investment by the model designer. As such, part of the analyses burden is put squarely on the shoulders of the model designer, making it difficult to gain acceptance in communities where truly complex systems are being dealt with on a day to day basis.

To eliminate this reluctance, future research efforts should focus on automated model complexity reduction and transformation into an underlying hybrid automata representation that may be hidden from the user.

[4] Named after the Greek philosopher Zeno who studied the relation between points and intervals, i.e., is an interval an infinite collection of points.

8 Acknowledgements

The author wishes to acknowledge extensive discussions with Gautam Biswas and Feng Zhao on the topic.

References

1. CAN specification. Technical Report, 1991. Robert Bosch GmbH.
2. Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Lecture Notes in Computer Science*, volume 736, pages 209–229. Springer-Verlag, 1993.
3. François E. Cellier. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools*. PhD dissertation, ETH, Zurich, Switzerland, 1979.
4. Control System Toolbox. *Control System Toolbox User's Guide*. The MathWorks, Natick, MA, 2004.
5. Mariana C. D'Abreu and Gabriel A. Wainer. Defining hybrid system models using DEVS quantization techniques. In *Proceedings of the Winter Simulation Conference*, New Orleans, LA, December 2003.
6. René David and Hassane Alla. *Petri Nets & Grafcet*. Prentice Hall Inc., Englewood Cliffs, NJ, 1992. ISBN 0-13-327537-X.
7. James Demmel and Bo Kågström. Stably computing the Kronecker structure and reducing subspaces of singular pencils $A - \lambda B$ for uncertain data. In J. Cullum and R. A. Willoughby, editors, *Large Scale Eigenvalue Problems*. Elsevier Science Publishers B.V. (North-Holland), 1986.
8. Joel M. Esposito, Vijay Kumar, and George J. Pappas. Accurate event detection for simulating hybrid systems. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, pages 204–217, 2001. Lecture Notes in Computer Science.
9. A. F. Filippov. Differential equations with discontinuous right-hand sides. *Mathematicheskii Sbornik*, 51(1), 1960.
10. Jon Friedman and Jason Ghidella. Using model-based design for automotive systems engineering – requirements analysis of the power window example. In *Proceedings of the SAE 2006 World Congress & Exhibition*, pages CD–ROM: 2006–01–1217, Detroit, MI, April 2006.
11. F. R. Gantmacher. *Matrizenrechnung; Teil I Allgemeine Theorie*. Deutscher Verlag der Wissenschaften, Berlin, 1965.
12. Eberhard Griepentrog and Roswitha März. *Differential-Algebraic Equations and Their Numerical Treatment*. BSB Teubner, Leipzig, 1986. ISBN 3-322-00343-4.
13. John Guckenheimer and Stewart Johnson. Planar hybrid systems. In Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems II*, volume 999, pages 202–225. Springer-Verlag, 1995. Lecture Notes in Computer Science.
14. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

15. John G. Kassakian, Martin F. Schlecht, and George C. Verghese. *Principles of Power Electronics*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1991. ISBN 0-201-09689-7.
16. Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, Inc., New York, 1978.
17. F. L. Lewis. A tutorial on the geometric analysis of linear time-invariant implicit systems. *Automatica*, 28:119–137, 1992.
18. P. Lötstedt. Coulomb friction in two-dimensional rigid body systems. *Z. angew. Math. u. Mech.*, 61:605–615, 1981.
19. Nancy Lynch and Bruce Krogh, editors. *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.
20. Cleve Moler. Are we there yet? zero crossing and event handling for differential equations. *EE Times*, pages 16–17, 1997. Simulink 2 Special Edition.
21. Pieter J. Mosterman. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569, pages 164–177. Lecture Notes in Computer Science; Springer-Verlag, March 1999.
22. Pieter J. Mosterman. Implicit Modeling and Simulation of Discontinuities in Physical System Models. In S. Engell, S. Kowalewski, and J. Zaytoon, editors, *The 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems*, pages 35–40, 2000. invited paper.
23. Pieter J. Mosterman. HyBrSim – a modeling and simulation environment for hybrid bond graphs. *Journal of Systems and Control Engineering*, 216:35–46, 2002. special issue paper.
24. Pieter J. Mosterman. On the normal component of centralized frictionless collision sequences. *ASME Journal of Applied Mechanics*, 2005.
25. Pieter J. Mosterman and Gautam Biswas. A Formal Hybrid Modeling Scheme for Handling Discontinuities in Physical System Models. In *AAAI-96*, pages 985–990, Portland, Oregon, August 1996.
26. Pieter J. Mosterman and Gautam Biswas. Principles for Modeling, Verification, and Simulation of Hybrid Dynamic Systems. In *Fifth International Conference on Hybrid Systems*, pages 21–27, Notre Dame, Indiana, September 1997.
27. Pieter J. Mosterman and Gautam Biswas. A theory of discontinuities in dynamic physical systems. *Journal of the Franklin Institute*, 335B(3):401–439, January 1998.
28. Pieter J. Mosterman, Jason Ghidella, and Jon Friedman. Model-based design for system integration. In *Proceedings of The Second CDEN International Conference on Design Education, Innovation, and Practice*, pages CD–ROM, Kananaskis, Alberta, Canada, July 2005.
29. Pieter J. Mosterman, Janos Sztipanovits, and Sebastian Engell. Computer automated multi-paradigm modeling in control systems technology. *IEEE Transactions on Control System Technology*, 12(2):223–234, March 2004.
30. Pieter J. Mosterman and Hans Vangheluwe. Computer automated multiparadigm modeling: An introduction. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 80(9):433–450, September 2004.
31. Pieter J. Mosterman, Feng Zhao, and Gautam Biswas. Sliding mode model semantics and simulation for hybrid systems. In Panos Antsaklis, Wolf Kohn, Michael Lemmon, Anil Nerode, and Shankar Sastry, editors, *Hybrid Systems V*, pages 218–237. Springer-Verlag, 1999. Lecture Notes in Computer Science.

32. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
33. Taeshin Park and Paul I. Barton. State event location in differential-algebraic models. *ACM Transactions on Modeling and Computer Simulation*, 6(2):137–165, April 1996.
34. Linda R. Petzold. A description of DASSL: A differential/algebraic system solver. Technical Report SAND82-8637, Sandia National Laboratories, Livermore, California, 1982.
35. F. Pfeiffer and C. Glocker. *Multibody dynamics with unilateral contacts*. John Wiley & Sons, Inc., New York, 1996.
36. SimEvents. *SimEvents User's Guide*. The MathWorks, Natick, MA, 2005.
37. Stateflow. *Stateflow User's Guide*. The MathWorks, Natick, MA, 2002.
38. Fabio Danilo Torrisi and Alberto Bemporad. HYSDEL – a tool for generating computational hybrid models for analysis and synthesis problems. *IEEE Transactions on Control System Technology*, 12(2):235–249, March 2004.
39. V. I. Utkin. *Sliding Modes in Control and Optimization*. Springer-Verlag, 1992.
40. A. J. van der Schaft and J. M. Schumacher. The complementary-slackness of hybrid systems. *Math. Contr. Signals Syst.*, (9):266–301, 1996.
41. Johannes van Dijk. *On the role of bond graph causality in modelling mechatronic systems*. PhD dissertation, University of Twente, The Netherlands, 1994. ISBN 90-9006903-8.
42. George C. Verghese, Bernard C. Lévy, and Thomas Kailath. A generalized state-space for singular systems. *IEEE Transactions on Automatic Control*, 26(4):811–831, August 1981.