Rigorous Hybrid Systems Simulation with Continuous-time Discontinuities and Discrete-time Components

James H. Taylor and Jie Zhang
Department of Electrical & Computer Engineering
University of New Brunswick
Fredericton, NB CANADA E3B 5A3

Abstract— **Previous research in the modeling and simulation of hybrid systems led to the development of a general hybrid systems modeling language (HSML). In more recent work, we have implemented this concept in software. The standard MATLAB model framework and integration algorithms have been extended to support state-event handling in continuous-time components and to deal with embedded discrete-time components, with utmost accuracy and efficiency.**

In this paper we overview the algorithmic implementation of the HSML ideas and language constructs for dealing with state events and embedded discrete-time components in MATLAB. A practical example (a separately-excited DC motor coupled via a gear-train to a load driven by a digital controller) is presented to demonstrate the efficacy of these extensions.

I. Introduction

The hybrid systems modeling language (HSML), as described previously [1], [2], [3], was designed to support a broad definition of a hybrid system, which we may express informally as being an arbitrary interconnection of components that are arbitrary instances of continuous- and discrete-time subsystems. Requirements for HSML particularly focused on rigorous characterization and execution of "events", both discrete- and continuous-time, that cause discontinuous changes in system trajectories and/or the model structure itself.

One can rigorously model hybrid systems using certain other, extant languages (with limitations). For example, ACSL [4] can be used to model and simulate state events in a hybrid system with considerable generality; however many other packages (especially commercially-supported ones such as MATLAB) lack even the basic provisions for state-event handling. Also, the high-level features and strict semantics and syntax formulated for HSML facilitate and enforce a higher degree of rigor in hybrid systems modeling, thereby ensuring a greater probability of model correctness. For example, resetting the state after a state event can be done in ACSL, but not cleanly and reliably, and again, it cannot be done in MATLAB; here we use a strict protocol that permits state reset in a general yet reliable way. The ideas and algorithmic requirements underlying HSML can be translated into any modeling and simulation environments, assuming that a developer can gain access to the necessary internal "machinery", as demonstrated here.

This paper overviews our work to implement new features of HSML in the MATLAB environment [5]. It focuses on requirements, our approach and implementation, and it presents a detailed application involving significant state-event handling in the continuous-time part as well as dealing with a discrete-time part associated with a digital controller, thus providing a complete demonstration of the use and effectiveness of our algorithms, which are available on the first-named author's website.

II. HSML Overview

HSML is designed to be a rigorous and modular hierarchical scheme for modeling hybrid systems. At the lowest level HSML components are "pure" continuous-time components (CTCs) and discrete-time components (DTCs) [1]. These elements are assembled into composite components, and then systems.

Glossing over state events and state reset for the time being, we consider CTCs that may be represented as[1]:

$$\begin{aligned} \dot{x_c} &= f_c(x_c, u_c, m, t) \\ y_c &= g_c(x_c, u_c, m, t) \end{aligned} \tag{1}$$

where x_c is the continuous-time state vector, y_c is the output vector, u_c is a numeric input vector (continuous-time variables or, if discrete-time, sampled and held outputs of DTCs), m is comprised of a finite alphabet of numeric or symbolic input variables that characterizes the "mode" of the model, and t is the time. Of particular importance to the

[1]The specific class of CTC that can be modeled depends on the simulator's integration methods; standard MATLAB routines currently cannot handle differential algebraic equations (DAEs), so we restrict ourselves to ordinary differential equations and simplify the variable types in comparison with [1], [2].

present exposition, the **mode** input m is included to provide means of controlling the model's structure and coordinating its behavior with the numerical integration process in state-event handling, as described below.

State events may be characterized very generally in terms of *zero crossings,*

$$S(x_c, m, t) = 0 \tag{2}$$

A mode-change in a CTC can be classified as a `negative-going` event (i.e., one in which S changes from positive to negative), an `on-constraint` event (where S remains equal to zero until another state event occurs), or a `positive-going` event. Finally, we include provision for instantaneous reset of the model state variables at a state event:

$$r = x_c(t_e^+) = r(x_c(t_e^-), m, t_e^-) \tag{3}$$

where t_e is the event time. This feature is useful in resetting velocities after objects engage to conserve momentum, for example. In accordance with this scheme for state-event definition, we permit elements of m to take on the values $-1,\ 0,\ +1$.

In the present stage of development, a DTC is a general algorithm which we can characterize in terms of internal variables called "discrete states" and outputs that also change discretely (instantaneously) at each execution:

$$x_{d,k} = f_d(x_{d,k-1}, u_{d,k-1}, m, t_k) \tag{4}$$

$$y_{d,k} = g_d(x_{d,k}, u_{d,k}, m, t_k) \tag{5}$$

where $x_{d,k}$ is the discrete state vector, k is the index corresponding to the discrete time point t_k at which the state takes on the new value $x_{d,k}$, $u_{d,k}$ is a discrete-time input vector (which may contain outputs of CTCs sampled at time t_k), and $y_{d,k}$ is the DTC output. The update in Eqn. 4 can also be expressed as:

$$x_{d,k+1} = f_d(x_{d,k}, u_{d,k}, m, t_{k+1}) \tag{6}$$

as is conventional in discrete-time filters. The times t_k are usually - but not necessarily - uniformly spaced; in any case, we assume that the update times can be anticipated and thus programmed in the DTC model. Corresponding to this, we define the vector t_e which at any time is comprised of the next execution times for every DTC in the system.

III. HSML Event Handling

A. State-Events

The HSML features for modeling state events are designed to permit the accurate and efficient integration of CTCs that may exhibit discontinuous behavior such as relays switching and mechanical components engaging/disengaging. The nature of the problem and an approach for proper handling of such events has been detailed previously [6], [7]; in this context, it suffices to observe that blindly integrating a CTC by stepping from a point t before switching to $t + h$ after the discontinuity, where h is the integration step, usually produces results that are both inaccurate and inefficient (in the sense of consuming an inordinate amount of computation).

The appropriate handling of state events requires coordination between the model and simulation package. This is achieved in HSML via `flag` variables in the model (S in Eqn. 2), and the model input variable m that can be used to control model switching. State-event handling then proceeds as follows:

1) Integrate as usual as long as the `flag` variables do not change sign. Each integration point is treated as a "trial" point until the sign condition is checked; if no sign change occurred, the point is "accepted".
2) When a sign change is detected, the trial point is discarded and an iterative procedure is initiated (within the simulator) to find the exact step h^* such that the `flag` variable is zero (within a small tolerance ϵ "on the other side"). The model **does not switch** during this part of the procedure.
3) The integrator produces an accepted point just past the switching curve (Eqn. 2) and then signals the model to switch (e.g., by changing m from 1 to -1 or *vice versa* if the boundary is to be crossed, or to 0 if the trajectory is to be confined to the boundary until the next state event).
4) The integrator then calls the model to determine if state reset is required, and if so executes it.
5) Normal integration proceeds from that point until the next state event is encountered and handled in the same way.

B. Time-Events

The approach and conventions needed to handle time events are much simpler than those required for state events, since we are merely emulating the execution of a computer algorithm in a digital setting (but without actual real time considerations). Time-event handling proceeds as follows:

1) Each DTC "notifies" the higher-level system integration block (SIB, see section IV-A) about its next execution time at the beginning of the simulation and at every subsequent DTC execution.
2) The SIB determines the earliest of the anticipated time events (if there is more than on DTC), and signals the numerical integrator to stop at that time.

3) At such a stopping point the SIB is invoked and it proceeds to execute the appropriate DTC(s), handling priority issues as specified in the model.

At each DTC execution this process is updated and continued until the end of the simulation run.

IV. MATLAB Extensions for Event Handling

The above outline of HSML and it's approach for characterizing state and time events provides a clear set of requirements for implementation in a software package. We have focused on MATLAB for this purpose, since it is so readily extensible. Generalizations are needed in two major areas: modeling schemes and numerical integration methods.

A. Extended Model Schema

The model input/output framework from previous work in state-event handling [7] had to be extended to allow coordinated state event and time event handling. The original MATLAB schema was to create models in the form of functions with two inputs (t, x) and one output $(\dot{x})$, and for continuous-time systems with state events this was extended by adding the input m, mode, and outputs S, the state-event zero-crossing function, and r, the state reset.

To further extend the model input/output framework, we added four new input variables. The first new input variable is $x_{d,k}$, which is used to calculate $x_{d,k+1}$ when the corresponding DTC(s) is (are) updated. The second one is the DTC output $y_{d,k}$, which can be involved in the calculation of $\dot{x}$. The third is t_e, which is comprised of the next execution times for every DTC. The last one is n_{dtc}, which indicates which DTC(s) should be updated, if any.

In addition, three new output variables were added: $x_{d,k+1}$, the DTC state vector at t_{k+1} in Eqn. 6, $y_{d,k}$, the updated DTC output vector, and $t_{e,new}$, the updated next execution time variable.

Note that S and m may be vectors, to support multiple state event mechanisms (switching conditions), and t_e may also be a vector of length equal to the number of DTCs. This scheme is portrayed in Fig. 1 (at the end of the paper). In this diagram, observe that:

- The Numerical Integrator (NI) must now serve as the "memory" for the *aggregate discrete component state* $(x_{d,k})$, *the* DTC*'s output* $(y_{d,k})$ and for the DTC*'s times of next execution* t_e. The NI has the requirement of stopping exactly at t_n, the earliest of the elements of t_e, and the "System Integrator Block" (SIB) has the responsibility of executing the correct DTC(s) when $t = t_n$.
- If multiple DTCs are to be executed at t_n, then the SIB has the duty to call them in the correct priority order.
- The continuous-time dynamics can reside in the SIB if they are simple; for more complex systems it may be helpful to create one or several separate CTCs, as diagrammed in Fig. 1 (note that this necessitates CTCs having inputs and outputs that are defined at the interface as shown).

B. Extended MATLAB Integrators

Significant extensions must also be made in an appropriate MATLAB numerical integration algorithm such as `ode45`, which we selected for its outstanding numerics. There are three features needed to permit the MATLAB integration routines to deal with state and time events:

1) The Numerical Integrator must coordinate with the extended model to establish the initial values of the CTC modes, m, and DTC next execution times, t_e;
2) The routine must continuously test for the occurrence of events by:
    a) ensuring that t stops at time t_n defined by `t_n = min(t_e)` in order to execute the next time event(s) in the corresponding DTC(s), and/or
    b) watching for zero crossings in S, iterating to determine the exact switching point and then changing m according to the logic in the model; and
3) It must execute a state reset operation after a state event, if it is called for by the model.

To support this functionality, the following conventions are imposed: The value of m for initialization is "empty" ($m = [\ ]$). The model must return the appropriate value of S, based on the stipulated initial condition x_0. From this information, the integration routine will set $m = \text{sign}(S)$. During normal integration the value of m's elements will be $-1,\ 0,\ +1$. When a state event is detected and determined[2], the corresponding element of m is switched; then m is temporarily made complex and the model should respond by returning the reset value r (Eqn. 3) or $r = [\ ]$ if no reset is to be done. Finally, that element of m is returned to $-1,\ 0,\ +1$ and numerical integration is resumed with the indicated mode change. The value of n_{dtc} is set by the integrator to inform the system model which DTC(s) should be updated. The dimension of n_{dtc} is equal to the number of DTCs; elements of n_{dtc} are changed from 0 to 1 by the integrator for DTC(s) which should be updated. Then, the

[2]We determine zero crossings by embedding a modified version of MATLAB's `fzero` algorithm within the integrator; it determines h^* such that $S = 0$ within machine ϵ.

model should respond by executing the appropriate DTC updating section.

V. Example Application

The extensions to MATLAB outlined above were done in two stages: the first stage is the extension for handling state events [7]; the second is the extension for handling time events. The extension for handling state events was tested using a number of simple switching systems [6], [8]. The extension for handling time events was done recently, and it was also tested using a number of hybrid systems [9]. In addition, we will demonstrate the modeling and simulation of a more realistic (and difficult) application, a feedback speed-control system for a separately-excited DC motor coupled via a gear-train to a load, with a digital compensator. The model is illustrated in Fig. 2. The error signal is $e = r - y = \omega_{sp} - \omega_L$, where ω_L is the load angular velocity and ω_{sp} is the corresponding set point. The digital

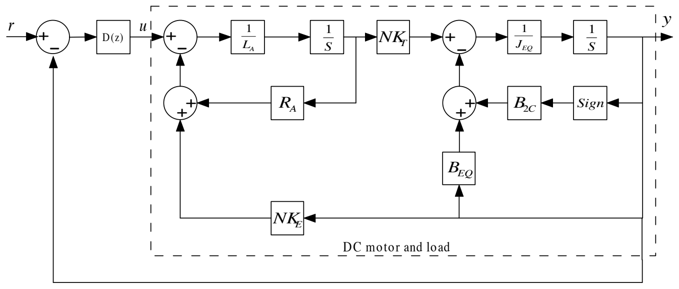

Fig. 2. DC Motor Control System Schematic

compensator is

$$D(z) = \frac{U(z)}{E(z)} = \frac{4z - 3.1}{z - 0.1}$$

and the sampling frequency is 10 Hz. The motor/load gear ratio is $N = 10$, the back emf coefficient is $K_E = 2$ *v-sec/rad*, and the torque coefficient is $K_T = 2$ *N-m/amp*. Other parameters are $R_A = 0.4\,\Omega$, $L_A = 0.02$ *H*, $J_{EQ} = 570$ *kg-m^2*, and $B_{EQ} = 280$ *N-m-sec/rad*. There is also a nonlinear effect informally modeled as $B_{2C}\text{sign}(y)$, where B_{2C} is the motor static friction coefficient (also called "stiction") that often cannot be neglected. Here we model it more rigorously using state events, where the motor "sticks" whenever the angular velocity goes to zero, i.e., the switching function is $S = y = \omega_L$, and it starts to move again only when there is sufficient torque to overcome stiction ($S = |\tau_e| - B_{2C}$ where τ_e is the motor torque, $\tau_e = NK_T i_A$). When the stiction term is large, for example $B_{2C} = 1500$ *N-m*, it will cause the simulation process to be extremely slow if the integrator does not include the state event handler, since the integration step is reduced to a very small value when the motor stops, and it remains infinitesimal until the torque can overcome the stiction. This problem is depicted in Fig. 3, which is the simulation result generated by SIMULINK using the informal model above; this simulation took 247 seconds and the "chattering" due to the extremely small step size is clearly evident.

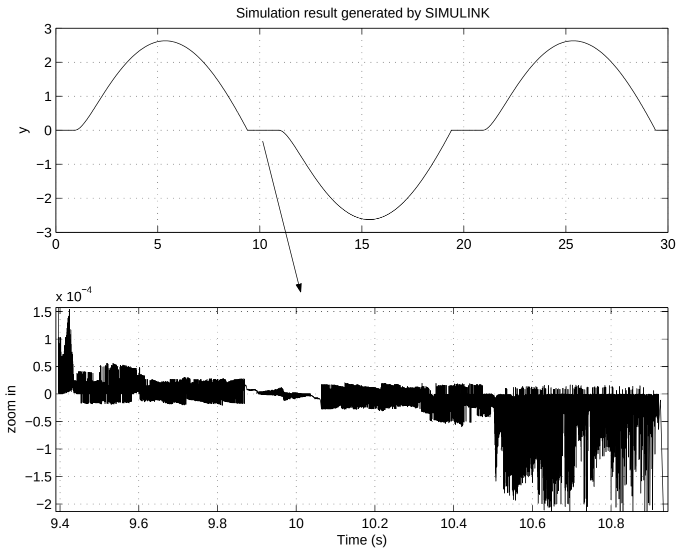

Fig. 3. DC Motor Control System Simulation Result Generated by SIMULINK

For this system, the extended integrator is much more efficient than SIMULINK, since it has a state event handler, which can recognize different stages of the motor motion (moving, stuck) and eliminate the chattering. Figure 4 shows the simulation result generated by the extended integrator; it only takes 2.4 seconds. More detailed models built for SIMULINK and the extended integrator can be found in [9].

VI. Conclusion

The MATLAB implementation presented above provides a demonstration of HSML in general and of the importance of careful time- and state-event handling in particular. Introducing the concept "mode" and the carefully prescribed "reset" protocol are both contributions toward making the modeling and simulation of switching in hybrid systems more systematic and rigorous. These features permit the study of systems that are well beyond the capabilities of the standard MATLAB integrators such as `ode45` and SIMULINK.

Extending this modeling approach and associated numerical integration routines can be pursued in several obvious ways, e.g., they can be inserted into more sophisticated modeling environments (like the SIMULINK framework

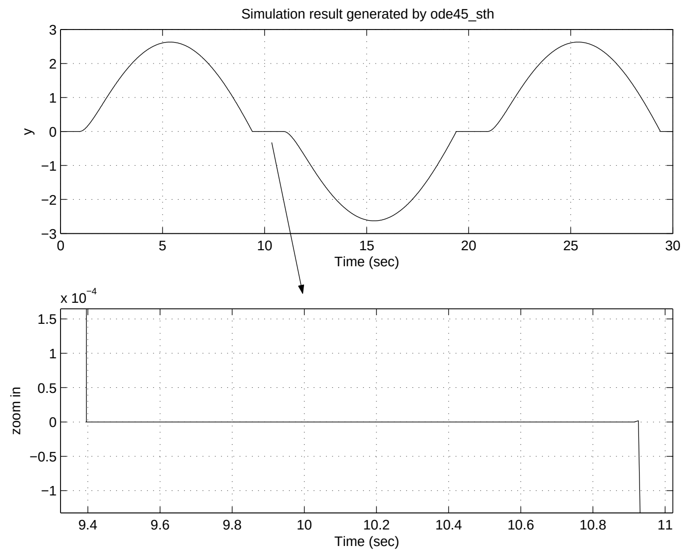

Fig. 4. DC Motor Control System Simulation Result Generated by the extended integrator

[10]). A more important extension would involve the development of a "HSML compiler", that would take the more rigorous HSML formulations and autocode extended MATLAB models.

REFERENCES

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

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

[3] Taylor, J. H. *A Rigorous Modeling and Simulation Package for Hybrid Systems*, US National Science Foundation SBIR Report, Award No. III-9361232, Odyssey Research Associates, Inc., June 1994.

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

[5] MATLAB *User's Guide*, The MathWorks, Inc., Natick, MA 01760

[6] Taylor, J. H., "Rigorous Handling of State Events in MATLAB", *Proc.* IEEE *Conference on Control Applications*, Albany, NY, 28-29 September 1995.

[7] Taylor, J. H. and Kebede, D., "Modeling and Simulation of Hybrid Systems", *Proc.* IEEE *Conference on Decision and Control*, New Orleans, LA, 13-15 December 1995.

[8] Taylor, J. H., "Rigorous Handling of State Events in MATLAB", *Proc.* IEEE *Conference on Decision and Control*, New Orleans, LA, 13-15 December 1995.

[9] Zhang, J. "A Creation of Hybrid System Modeling and Simulation Environment in MATLAB", *MScEng Thesis* University of New Brunswick, September 2005.

[10] SIMULINK *User's Guide*, The MathWorks, Inc., Natick, MA 01760.

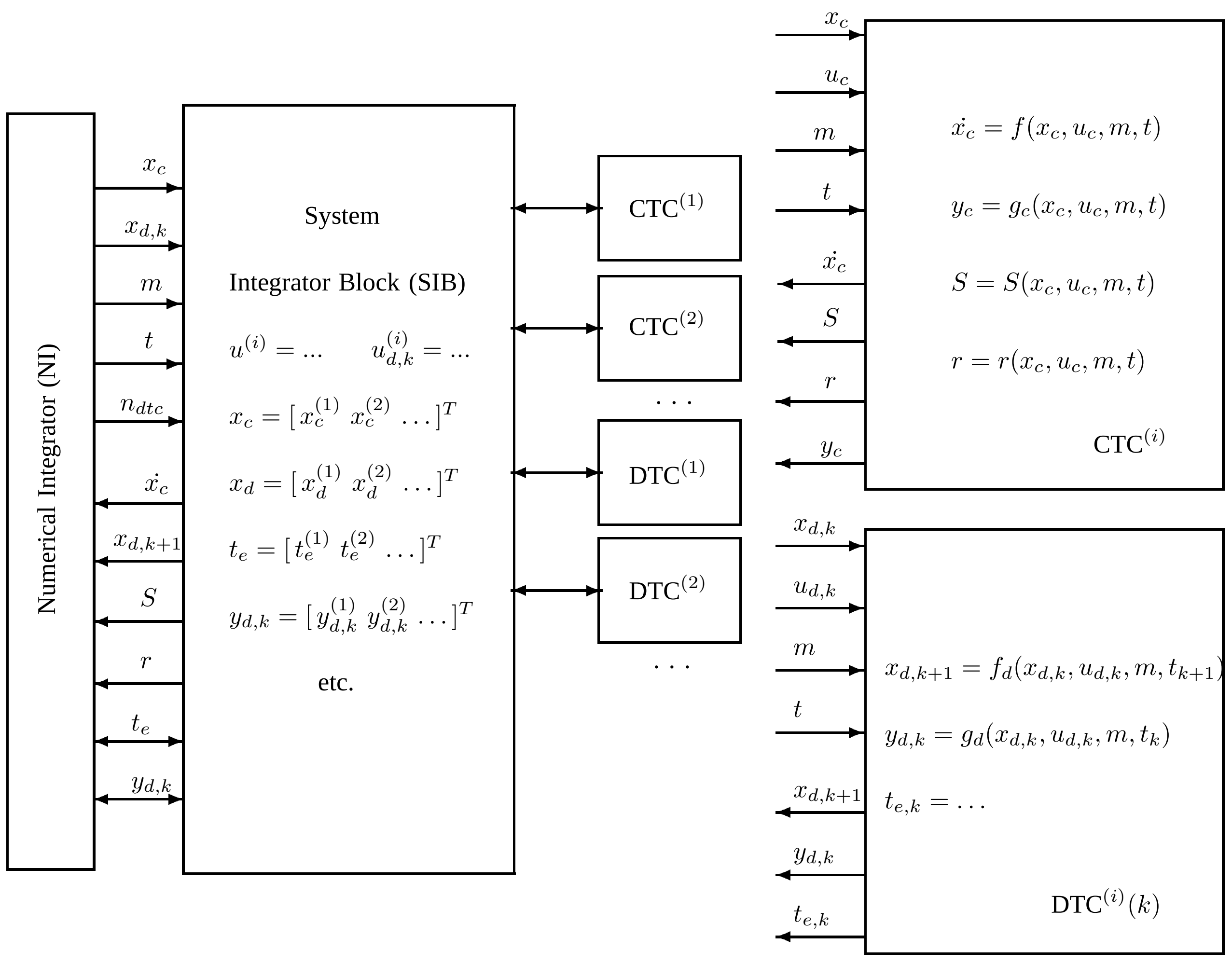
Numerical Integrator (NI)
x_c
$x_{d,k}$
m
t
n_{dtc}
$\dot{x}_c$
$x_{d,k+1}$
S
r
t_e
$y_{d,k}$
System
Integrator Block (SIB)
$u^{(i)} = \ldots \quad u_{d,k}^{(i)} = \ldots$
$x_c = [\,x_c^{(1)}\ x_c^{(2)}\ \ldots]^T$
$x_d = [\,x_d^{(1)}\ x_d^{(2)}\ \ldots]^T$
$t_e = [\,t_e^{(1)}\ t_e^{(2)}\ \ldots]^T$
$y_{d,k} = [\,y_{d,k}^{(1)}\ y_{d,k}^{(2)}\ \ldots]^T$
etc.
CTC$^{(1)}$
CTC$^{(2)}$
DTC$^{(1)}$
DTC$^{(2)}$
x_c
u_c
m
t
$\dot{x}_c$
S
r
y_c
$\dot{x}_c = f(x_c, u_c, m, t)$
$y_c = g_c(x_c, u_c, m, t)$
$S = S(x_c, u_c, m, t)$
$r = r(x_c, u_c, m, t)$
CTC$^{(i)}$
$x_{d,k}$
$u_{d,k}$
m
t
$x_{d,k+1}$
$y_{d,k}$
$t_{e,k}$
$x_{d,k+1} = f_d(x_{d,k}, u_{d,k}, m, t_{k+1})$
$y_{d,k} = g_d(x_{d,k}, u_{d,k}, m, t_k)$
$t_{e,k} = \ldots$
DTC$^{(i)}(k)$

Fig. 1. New MATLAB model component input/output structures