Graphical Modeling of Hybrid Dynamics with Simulink and Stateflow

Akshay Rajhans
MathWorks
Natick, MA
akshay.rajhans@mathworks.com

Srinath Avadhanula
MathWorks
Natick, MA
srinath.avadhanula@mathworks.com

Alongkrit Chutinan
MathWorks
Natick, MA
alongkrit.chutinan@mathworks.com

Pieter J. Mosterman
MathWorks
Natick, MA
pieter.mosterman@mathworks.com

Fu Zhang
MathWorks
Natick, MA
fu.zhang@mathworks.com

ABSTRACT

Simulink and Stateflow are tools for Model-Based Design that support a variety of mechanisms for modeling hybrid dynamics. Each of these tools has different strengths. In this paper, a new modeling construct is presented that combines these strengths to enable graphical modeling of hybrid dynamics within a single Stateflow chart. A new type of Stateflow state that acts as a Simulink subsystem is developed to facilitate graphical modeling of continuous dynamics using Simulink blocks inside Stateflow. Remote textual and graphical state access using new state-accessor blocks enables continuous states to be used in transition guards and reset actions. Key features of this new formalism are illustrated using various examples with hybrid dynamics.

ACM Reference Format:
Akshay Rajhans, Srinath Avadhanula, Alongkrit Chutinan, Pieter J. Mosterman, and Fu Zhang. 2018. Graphical Modeling of Hybrid Dynamics with Simulink and Stateflow. In *HSCC '18: 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week), April 11–13, 2018, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3178126.3178152

1 INTRODUCTION

Model-Based Design of complex systems involves creation of mathematical models that serve as a basis for the design and analysis of the underlying system. Simulink® and Stateflow® tools provide a powerful framework for graphically modeling system dynamics by enabling users to drag and drop individual modeling elements—such as Simulink blocks or Stateflow states—that form the building blocks of system dynamics. These elements can be interconnected and hierarchically and parallely composed to model complex cyber-physical systems (CPS). Simulink and Stateflow tools, along with their auto-generated code, are routinely used in the automotive, aerospace, and other CPS domains. They have also been recently used successfully in such complex projects as the Pluto fly-by mission [7] and NASA's Orion spacecraft test flight project [13].

Simulink and Stateflow provide different modeling capabilities for modeling hybrid dynamics. There are a number of ways in which hybrid systems can be modeled using only Simulink, only Stateflow, or a combination of the two. This tool paper presents a new modeling syntax that combines the strengths of these tools together for modeling hybrid-dynamic systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HSCC '18, April 11–13, 2018, Porto, Portugal
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5642-8/18/04…$15.00
https://doi.org/10.1145/3178126.3178152

2 BACKGROUND

2.1 Dynamic system simulation in Simulink®

A Simulink block can be mathematically defined as follows.

Definition 2.1. **Simulink Block.** A Simulink *block* is a tuple $(x_{CT}, x_{DT}, f_{CT}, f_{DT}, g, u, y, x_0)$, where:

- x_{CT} and x_{DT} are (possibly-empty) continuous-time (CT) and discrete-time (DT) state vectors, together addressed as the continuous-valued state vector x;
- u and y are the input and output vectors;
- f_{CT} is the *derivative function*, i.e., $\dot{x}_{CT}(t) = f_{CT}(u(t), x(t))$, numerically integrated by an ODE solver at appropriate time points t based on the solver characteristics (e.g., number of internal states and error tolerances);
- f_{DT} is the *update function*, i.e., $x_{DT}(t+dt) = f_{DT}(u(t), x(t))$, with stepsize dt determined by the (specified or inferred) sample rate of the block;
- g is the *output function*, i.e., $y(t) = g(x(t), u(t))$; and
- x_0 is the initial value of the state vector x, i.e., $x(0) = x_0$, which can be either defined as a block parameter or read as an external input.

As a software implementation, each block defines *block methods* that correspond to the different equations defined above. Every block must define its *output method*, and may define *initialize*, *update* and/or *derivative methods* to realize the corresponding equations as applicable. Simulink's execution engine calls these methods in a pre-determined order in a loop, called the *simulation loop*, until the simulation stop time is reached [12]. Additionally, blocks that have a discontinuity in their output (e.g., switch blocks) also define a *zero-crossing method* that, if enabled, helps the Simulink engine precisely locate the time of the discontinuity [17].

Definition 2.2. **Block Diagram.** A Simulink *block diagram* is a (hyper-)graph (N_{SL}, E_{SL}), with Simulink blocks as nodes N_{SL}, and signal or communication lines representing the connectivity constraints as edges E_{SL}.

Each node $n_{SL} \in N_{SL}$ can either be an individual block or a *subsystem*, which is a hierarchical composition of blocks that has its own block diagram. Signal lines, message connections, and function-call lines are examples of the connectivity constraints $e_{SL} \in E_{SL}$.

Definition 2.3. **Stateflow Chart.** A Stateflow chart is a finite-state machine modeled as a (hyper-)graph (N_{SF}, E_{SF}), with nodes N_{SF} representing the states and edges E_{SF} representing the state transitions of the finite-state machine.

Each node $n_{SF} \in N_{SF}$ can be an individual state or a *subchart*, which encapsulates another Stateflow chart.

As a software implementation, a Stateflow chart is a Simulink block that defines all the relevant block methods based on its dynamics. Like any Simulink block, a chart can have external I/Os u and y, as well as internal state x. The *entry*, *exit* and *during* actions defined on the states, as well as transition actions, can modify x and y as a function of x and u. Transition guards can depend on u and x.

2.2 Existing alternatives for modeling hybrid dynamics in Simulink® and Stateflow®

(1) **Type I: Only Simulink.** Simulink provides a variety of blocks for modeling mode switches using explicit mechanisms such as switch blocks and conditionally-executed subsystems, or, implicit mechanisms such as saturation and external reset. The STARMAC quadrotor model from [10] and the Simulink example models `sldemo_bounce` [4] and `sldemo_clutch` [1] are examples of the Type I approach.

(2) **Type II: Only Stateflow.**
Since R2007b, a textual syntax for modeling continuous dynamics in the 'during' actions of Stateflow states allows the modeling of hybrid automata. The abstract powetrain control model from [11], the DC-DC converter from [15], and the Simulink bouncing ball example `sf_bounce` [3] use this Type II approach.

(3) **Type III: Mixed Simulink and Stateflow.** This approach uses Simulink blocks kept outside the Stateflow chart to model continuous dynamics; and signal lines between Simulink and Stateflow for (i) guard conditions in Stateflow that depend on state(s) of one or more Simulink blocks, (ii) control outputs from Stateflow that are used to drive the subsystem corresponding to the active mode, and (iii) reset values computed in Stateflow to drive the external initial condition (EIC) port of a Simulink block. Type III examples include: hybrid systems modeled using CheckMate [9], the cardiac cell model from [8], and the Simulink example model `sf_yoyo` [6].

2.3 Shortcomings of existing approaches

(1) **No central location for modeling mode switching.** In Type I models, the mode-switching blocks can be distributed throughout the model hierarchy. These blocks and their zero-crossings (ZCs) could potentially always be evaluated, although some might be infeasible in the current mode of operation. Models can be rearchitected to be efficient, but this requires additional effort on the modeler's part.

(2) **Need for external ports.** In Types I and III, transition guards and reset actions that read or change continuous state require the use of state ports and EIC ports. State ports provide an estimate of the state at the next time step without accounting for the reset; when used in a mutually-resetting context the simulation answer can be susceptible to block sorting. The use of output ports in place of state ports can cause algebraic loops that necessitate the use of memory blocks (e.g., in `sldemo_bounce` [4]), which introduces unnecessary artifacts.

(3) **Limitations of the textual syntax.** Type II modeling has recently seen adoption from the academic research community, but writing out complex continuous dynamics in a textual syntax in Stateflow does not scale to the large-scale industrial models. Long and complex textual equations may be hard to read and debug.

(4) **Signal lines between Simulink and Stateflow.** In the Type III approach, signals need to be passed back and forth between Simulink and Stateflow for the evaluation of guards and switching of the continuous dynamics, which may lead to diagram wclutter.

3 GRAPHICAL HYBRID AUTOMATA IN SIMULINK® AND STATEFLOW®

We present a new formalism, available starting release R2017b, for graphical modeling of hybrid dynamics in Simulink and Stateflow that overcomes the shortcomings of existing alternatives for modeling hybrid dynamics outlined in the previous section.

As a running example for illustrating the new formalism, we consider the lockup behavior of a clutch that consists of two plates that transmit torque between the engine and transmission [1]. When the clutch is disengaged (also called *slipping*), the two plates rotate freely. When the clutch is engaged (also called *locked*), the two plates rotate in a synchronized manner. The transitions between the engaged and disengaged modes depend on the relative velocities of the two plates and the kinetic friction between them.

Definition 3.1. **Graphical Hybrid Automaton.** A *graphical hybrid automaton* (GHA) is a Stateflow chart that is a graph (N_{GH}, E_{GH}), s.t.

- nodes N_{GH} are Stateflow states, called *Simulink based states* in Stateflow, which have an associated block diagram that graphically defines their dynamics; and
- E_{GH} is a set of transitions $e_{GH} = (s_{GH}, d_{GH}, g_{GH}, a_{GH})$, where
  - $s_{GH}, d_{GH} \in N_{GH}$ are the source and destination states;
  - g_{GH} is a transition guard that evaluates to a Boolean; and
  - a_{GH} is a transition action.

Figure 1 shows a GHA for modeling clutch dynamics using two Simulink-based Stateflow states `Slipping` and `Locked` and transitions between them. The guards on the transitions between the

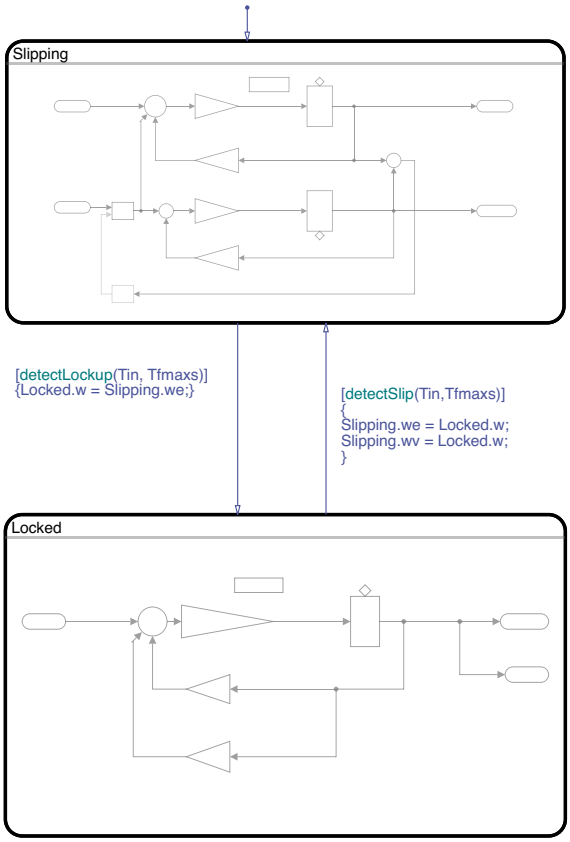

Figure 1: GHA of a clutch. © The MathWorks, Inc.

states are denoted by square brackets, and the transition functions are denoted by the curly braces, as per the Stateflow syntax.

3.1 Simulink® states in Stateflow®

We have developed a new kind of Stateflow state, called a *Simulink based state*, whose internal dynamics can be defined as a subsystem using Simulink blocks. This new state can be dragged from the Stateflow palette and dropped from onto the canvas, and double-clicking on it opens a Simulink canvas where users can graphically model dynamics as if it were a Simulink subsystem. Each Simulink based state by default has the same I/Os as the parent chart, but is allowed to have a subset of I/Os. Unused I/Os can either be terminated/grounded or simply deleted.

A Stateflow chart can compose Simulink based states along with other Simulink and/or regular states in the same chart in any combination as per the Stateflow syntax: sequentially using *exclusive (OR) decomposition*, or parallely[1] using *parallel (AND) decomposition* [5]. Hierarchical composition is supported just as in Simulink, e.g., by using subsystems, model blocks, or Stateflow charts inside a Simulink based state.

[1]Stateflow does not compute a product automaton, but executes parallel states based on a deterministic order [2].

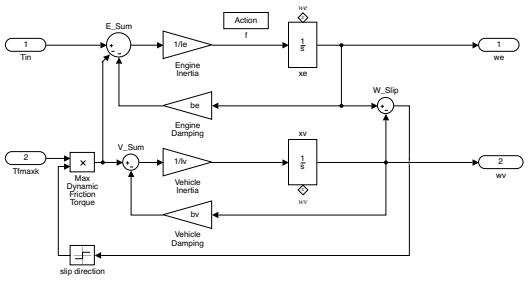

Figure 2: Dynamics inside the `Slipping` state from Fig. 1. © The MathWorks, Inc.

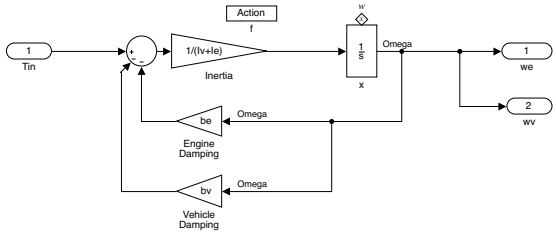

Figure 3: Dynamics inside the state `Locked` state from Fig. 1. © The MathWorks, Inc.

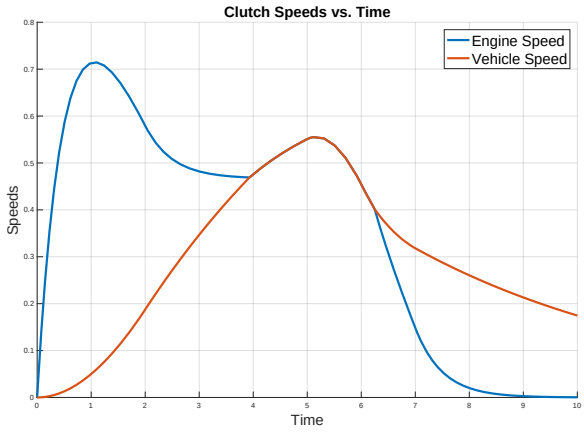

Figure 4: Simulation results for GHA from Fig. 1. © The MathWorks, Inc.

Figures 2 and 3 depict the dynamics of the Simulink based states `Slipping` and `Locked` from the GHA in Fig. 1. The former models the rotational dynamics of an unlocked clutch with two integrator blocks for two distinct angular velocities, and the latter models a locked clutch with only one integrator that models the joint angular velocity. The simulation results are plotted in Fig. 4, which shows regions of both overlapping and non-overlapping curves as a result of the mode switching.

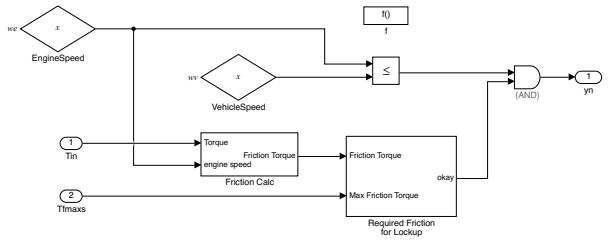

Figure 5: Contents of the Simulink® function `detectLockup` used as a transition guard in Fig. 1. © The MathWorks, Inc.

3.2 Remote access of continuous state

Unlike hybrid automata formalisms in the literature and the Type II approach, the graphical nature of GHA means that the continuous state is present inside individual Simulink blocks. For the purpose of accessing it in transition guards and reset actions, we present two approaches.

3.2.1 Textual remote access. On naming the state ('state name' parameter of a stateful block), the corresponding state is made available for direct access as a corresponding named variable in transition guards and actions in the chart. Names are resolved by dot notation, e.g., the read access `Locked.w` or the write access `Slipping.we` in Fig. 1.

3.2.2 Graphical remote access. For more complex computation that are either too cumbersome or impossible to be written as textual equations, we have developed two new types of Simulink blocks called *state reader* and *state writer* blocks, collectively called *state accessor* blocks. They remotely access the state of a uniquely identified *state owner* block, e.g., block `EngineSpeed` from Fig. 5 that reads the state from block `xe` from Fig. 2. When used as a guard, a Simulink function graphically computes a Boolean output based on the read states. When used as a transition reset action, it graphically computes values to be remotely written.

Block annotations with hyperlinks and edit-time graphical highlighting from the Stateflow symbols pane are available for bi-directional traceability between the state owner blocks and their corresponding graphical or textual remote access.

3.2.3 Advantages over port-based state access. Remote access of block state is superior to state- and EIC-port-based access. Unlike ports, they do not need to be wired up to their state owners, thereby avoiding sorting and algebraic loop problems. The Simulink engine maintains this association instead. Additionally, remote access is only allowed in explicitly ordered contexts. In a GHA, a Stateflow chart controls the execution order of various action subsystems, so that there is a deterministic order between competing access to the same state. Other permissible uses include explicitly ordered function-call subsystems.

4 EXAMPLES

4.1 Modeling a pole vault jump

This example is inspired by a blog post about modeling pole vault jumps using Simulink [16], and is intended to showcase the use of

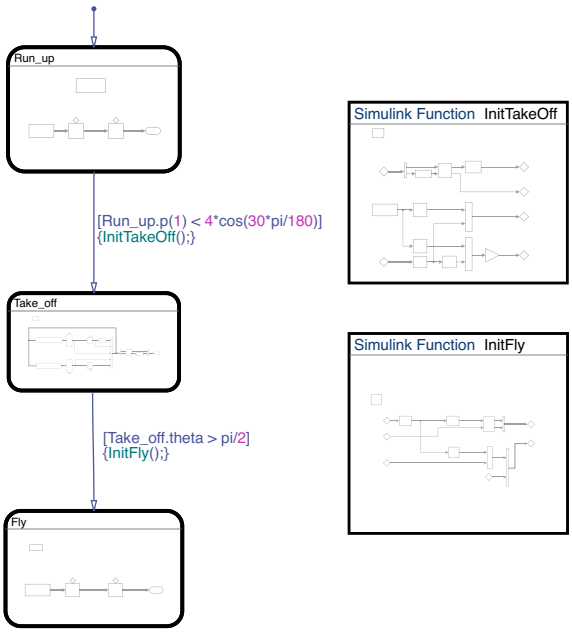

Figure 6: GHA of a pole vault jump. © The MathWorks, Inc.

arbitrarily different continuous states in different modes and **a new copy-paste workflow**.

A pole vault jump has hybrid dynamics with three distinct modes that model running in, lift off and free fall after release. The original Type I approach models the three cases in three Action Subsystems. The `Run_up` and `Fly` modes model **vector-valued** double-integrator dynamics (position (p_x, p_y) and velocity (v_x, v_y)) in the Cartesian coordinates. The `Take_off` mode uses polar $(r, \theta, \dot{r}, \dot{\theta})$ coordinates. This is made possible due to the distributed nature of the dynamics (i.e., individual blocks own their state and update/derivative methods)[2].

The detailed dynamics are as presented in [16], which we intend to reuse. In the GHA formalism, this existing Action Subsystem from a Simulink canvas can be directly copy-pasted onto a Stateflow canvas. This automatically generates a Simulink-based state in Stateflow, such as the three shown in the GHA in Fig. 6. `Run_up.p(1)` is a vector-valued textual state access.

4.2 Modeling a mode-switching controller

This example is intended to showcase **DT dynamics**, **built-in timers**, and **reuse**. Consider the air-fuel ratio controller hybrid automaton from the powertrain control verification benchmark [14] with `startup`, `power`, `sensor_fail` and `normal` modes. The normal mode uses a feedback PI control, while the rest all use a feedforward P control. The original model uses DT update dynamics for the states: estimated manifold pressure p_e, the integral term i of the PI controller, and the commanded fuel F_c. Due to the limitations

[2] In contrast, traditional hybrid automata maintain a single continuous state vector, so the use of different continuous variables for different mode necessitates constructing a union state vector and holding inactive states constant, but in process increasing the dimensionality of the model.

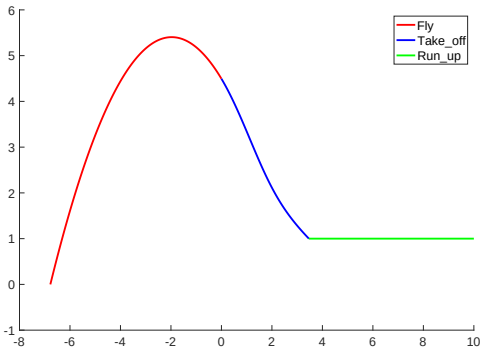

Figure 7: Pole vault jump xy plot. Three colors depict three modes. © The MathWorks, Inc.

of hybrid automata to capture DT hybrid dynamics, the model needs an auxiliary timer variable for periodic self loops to carry out the DT update. In contrast, Fig. 8 shows the GHA of the controller, which does not need self loops for DT dynamics: we simply set a discrete sample time on the chart instead, and use unit delay blocks rather than integrators to model DT states. Stateflow's temporal logic syntax '`after`' also simplifies the time-dependent transition out of `startup` and as such even that timer does not need to be handled manually. For this controller-only model, we model F_c is an output. We encapsulate the common P control law into a library as shown in Fig. 9, and reuse it as library links for modeling the states `startup`, `sensor_fail`, and `power`.

The controller GHA has the advantage that it contains the exact same structure as the original HIOA (Fig. 1 from [14]). Yet, it is also scalable to the real-world complexity using look-up tables, transport delays, or extended Kalman filters (EKF), which can all be conveniently used because the syntax supports the full modeling vocabulary.

4.3 Modeling a cardiac cell

This example is intended to showcase the **conciseness** of GHA. Consider a cardiac cell with four modes: `resting`, `stimulated`, `upstroke`, and `plateau`, with different continuous dynamics that determine the membrane voltage v of the cell in each mode [8]. The Type III model of the cell presented in Fig. 5(a) in [8] uses two Stateflow charts (`Event generator` and `Hybrid set`) and a Simulink subsystem `Subsystem3` that model trigger events, state transitions, and the continuous dynamics respectively. `Hybrid set` outputs a discrete variable `q` representing the mode, the reset voltage value `vreset`, and a flag `reset` that decides whether to reset. Switch and multi-port switch blocks inside `Subsystem3` model the mode switching based on `q`, `vreset`, and `reset`. All of this complicated model structure can be greatly simplified into a GHA as shown in Fig. 10. The GHA model combines the two Stateflow charts and a subsystem into one chart, as well as eliminates the need for: state and EIC ports, the initial condition, switch and multiport switch blocks, and their interconnections. Ultimately we simply get a GHA with four modes, exactly like original automaton (Fig. 2 in [8]), without any unecessary additional artifacts.

5 ADVANTAGES OF THE GHA FORMALISM

Improvement over existing approaches. The new GHA formalism overcomes the shortcomings from Sec. 2.3. GHA are intended to be semantically equivalent to the existing approaches, i.e., the simulation results match for equivalent dynamics. Elimination of the need for memory blocks to break algebraic loops (e.g., the second-order integrator flavor of sldemo_bounce [4]) is an instance of increase in simulation accuracy, but in general it is the same as before. Simulation speeds are comparable with those for existing approaches. ZCs are at least as efficient as before, and can be more efficient as GHAs evaluate entire transition guards at once and only when necessary [3].

Generality. The new approach is more general than the different flavors of hybrid auotmata found in the literature in the following manners. Given the graphical nature of the formalism, complex dynamics that include table lookups, filtering, etc., can also be modeled. Such dynamics cannot be modeled using hybrid automata as they require closed-form differential equations. Secondly, individual modes of GHA can have completely different continuous variables, as exemplified by the pole vault example.

Modularity and Reuse. The new formalism opens up the possibility of modular development. The continuous dynamics in different modes can be developed by various teams/vendors, possibly as libraries. Thanks to remote state access, the overall chart can be put together without having to edit the individual elements, which otherwise would have been necessary for exposing state or EIC ports and manually connecting signal lines or From/Goto blocks between them. On a similar note, the developer of the continuous dynamics model does not need to know about the ways in which the model would be used and therefore need not preemptively add unnecessary artifacts to account for all possible ways in which it might be used.

6 DISCUSSION

This tool paper presents a new framework for graphically modeling hybrid dynamics using Simulink and Stateflow. The new framework provides a clean separation between discrete and continuous dynamics, both of which can be graphically modeled using Stateflow and Simulink.

The new syntax has the look and feel of hybrid automata while still making full use of graphical modeling. We hope this can help bridge the gap between the ease of graphical modeling preferred by system designers in industry and a clean syntax preferred by the academic community for their analysis tools that work with hybrid automata.

ACKNOWLEDGMENTS

We thank our colleagues Dr. Mohamed Alimi, Dr. Huaizhong Han, Dr. Teresa Hubscher-Younger, Dr. Joshua Nasman, and Dr. Ramamurthy Mani for their contributions.

REFERENCES

[1] Building a clutch lock-up model. http://www.mathworks.com/help/simulink/examples/building-a-clutch-lock-up-model.html.

[3] In contrast, e.g., in the two-integrator bouncing ball model [4], the integrators each evaluate the same guard condition in each time step (shortcoming #1 from Sec. 2.3).

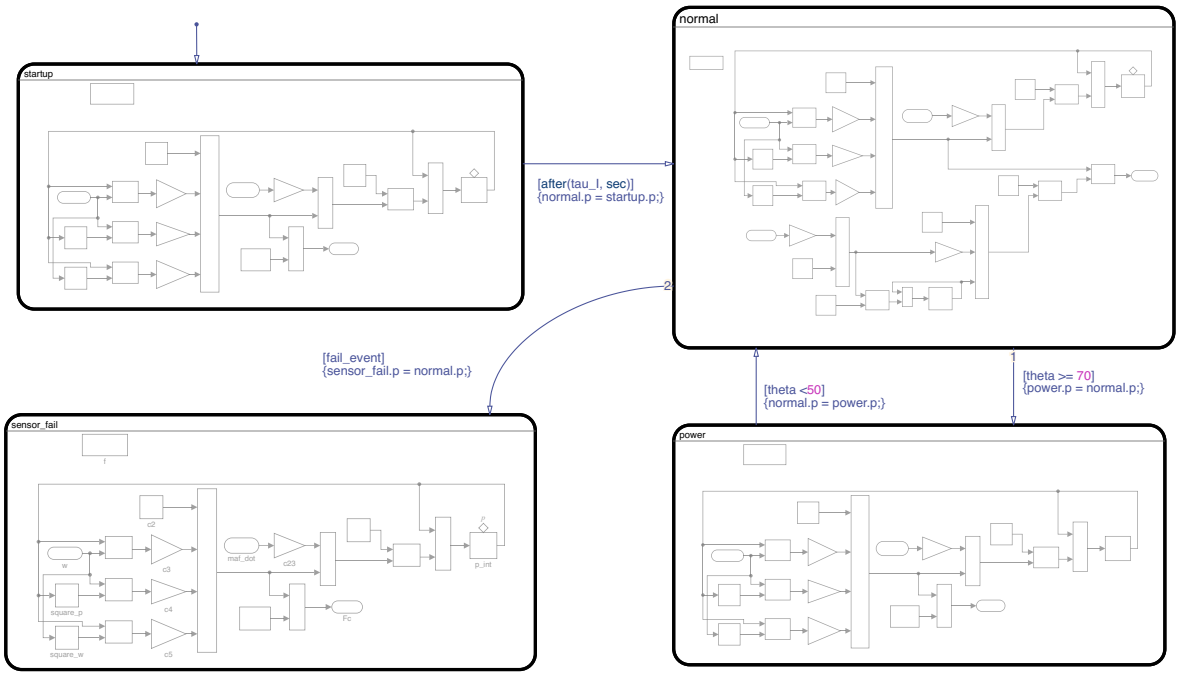

Figure 8: GHA of the air-fuel ratio controller HIOA from [14]. © The MathWorks, Inc.

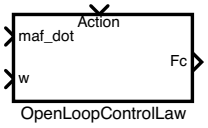

Figure 9: Library for reusing open-loop powertrain control. © The MathWorks, Inc.

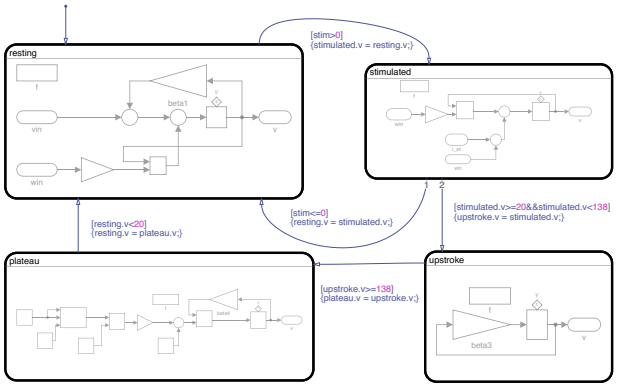

Figure 10: GHA of the cardiac cell from [8]. © The MathWorks, Inc.

[2] Execution order for parallel states. http://www.mathworks.com/help/stateflow/ug/execution-order-for-parallel-states.html.

[3] Modeling a bouncing ball. http://www.mathworks.com/help/stateflow/examples/modeling-a-bouncing-ball.html.

[4] Simulation of a bouncing ball. http://www.mathworks.com/help/simulink/examples/simulation-of-a-bouncing-ball.html.

[5] State decomposition in stateflow. https://www.mathworks.com/help/stateflow/ug/state-decomposition.html.

[6] Yo-yo control of satellites. http://www.mathworks.com/help/stateflow/examples/yo-yo-control-of-satellites.html.

[7] D. Artis, B. Heggestad, C. Krupiarz, M. Mirantes, and J. Reid. Messenger: Flight software design for a deep space mission. In *2007 IEEE Aerospace Conference*, pages 1–9, March 2007.

[8] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. A simulink hybrid heart model for quantitative verification of cardiac pacemakers. In *Proc. of HSCC 2013*, pages 131–136.

[9] A. Chutinan and B. H. Krogh. Computational techniques for hybrid system verification. *IEEE Transactions on Automatic Control*, 48(1):64–75.

[10] A. Donzé, B. Krogh, and A. Rajhans. Parameter synthesis for hybrid systems with an application to simulink models. In R. Majumdar and P. Tabuada, editors, *HSCC 2009*, volume 5469 of *LNCS*, pages 165–179. Springer Berlin Heidelberg.

[11] C. Fan, P. S. Duggirala, S. Mitra, and M. Vishwanathan. Progress on powertrain contrl verification challenge with C2E2. In *Applied Verification for Continuous and Hybrid Systems workshop (ARCH)*, 2015.

[12] Z. Han, P. Mosterman, J. Zander, and F. Zhang. Systematic management of simulation state for multi-branch simulations in Simulink. In *Proc. of the Symposium on Theory of Modeling and Simulation (TMS) 2013*, pages 84–89.

[13] M. C. Jackson and J. R. Henry. Orion GN&C model based development: Experience and lessons learned.

[14] X. Jin, J. V. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain contrl verification benchmark. In *Proc. of HSCC 2014*, pages 253–262.

[15] T. T. Johnson, S. Bak, and S. Drager. Cyber-physical specification mismatch identification with dynamic analysis. In *Proc. of ICCPS 2015*, pages 208–217.

[16] G. Rouleau. Olympic 2016 – pole vault. https://blogs.mathworks.com/simulink/2016/08/19/olympic-2016-pole-vault/, 2016.

[17] F. Zhang, M. Yeddanapudi, and P. Mosterman. Zero-crossing location and detection algorithms for hybrid system simulation. In *17th IFAC World Congress*, pages 7967–7972, 2008.