# Formal Semantics and Analysis Methods for Simulink Stateflow Models

A. Tiwari

*Abstract*—**Embedded control systems typically comprise continuous control laws combined with discrete mode logic. The Simulink graphical environment of MathWorks' tool suite is a popular choice for modeling and designing embedded controllers. Mode logic in Simulink models is described in terms of hierarchical state machines specified in a variant of Statecharts called Stateflow. The semantics of Stateflow is quite complex and it is valuable if these designs can be formally analyzed for both early error detection and positive assurance.**

**It is important that formal analysis should be unobtrusive and acceptable to engineering practice. We motivate a methodology called "invisible formal methods" that provides a graded sequence of formal analysis technologies ranging from extended typechecking, through approximation and abstraction, to model checking and theorem proving. As an instance of invisible formal methods, we describe the formal semantics of a fragment of Stateflow based on a modular representation called** *communicating pushdown automata***. We show how this semantics can be used to analyze simple properties of Stateflow models.**

*Keywords*—**Hybrid dynamical systems, Invariant, Symbolic Simulation.**

## I. Introduction

HYBRID systems involve a combination of discrete and continuous dynamics and are used for modeling embedded control systems. Many of the embedded control systems are safety critical and require formal guarantees of safe operation. Formal design and analysis of hybrid system models has received much attention in the research community recently, from both the computer science and control theory worlds.

The systems that have been traditionally studied in the computer science community have been discrete. Such systems evolve in discrete time steps. Moreover, given a current state of the system, the state in the next discrete time instance is assumed to come from a *finite* (and countable, at worst) set of states. Good advances have been made in the techniques and tools for analyzing discrete systems [9,24]. Some of the most effective techniques include *model checking* and *abstraction*. Abstraction is typically used to reduce the possibly infinite state space system into a finite state space abstract system, and model checking is subsequently used to exhaustively search through all behaviors of the finite abstraction.

Hybrid systems differ from purely discrete systems in that they also contain a continuous component. Such systems evolve in continuous time with discrete jumps at particular time instances. The techniques developed for discrete systems are thus not directly applicable. First, the state space now is uncountably infinite. Second, from a given state, a hybrid system can make a transition such that the next state comes from an *uncountable* set of states. Furthermore, certain undecidability results can be established for checking reachability in even simple classes of hybrid systems (for example, systems whose continuous dynamics involves variables that proceed at two constant slopes [18]).

One approach to overcoming undecidability involves restricting the continuous dynamics of the hybrid system so that suitable abstractions can be successfully applied to yield conservative discrete transition systems. Timed automata [2], multirate automata [1], and rectangular automata [18] are some such examples. Another approach is to restrict the discrete transitions and the continuous flows so that finite abstractions can again be constructed. The idea of o-minimal hybrid systems [25] is motivated by this.

Due to the undecidability results for general systems and known decidability results for rather restricted classes of hybrid systems, there is a huge gap between the interesting large and complex systems that are typically used in practice, and the restricted and simple systems that the analysis techniques can handle. Furthermore, the available techniques are still far removed from the tools engineers most often use in practice to design embedded control systems.

One of the most extensively used tools for modeling, simulation, and rapid prototyping of control designs for embedded applications is the Simulink/Stateflow development suite provided by MathWorks Inc. The addition of formal analysis capabilities to such a tool would offer benefits in early error detection and more complete assurance of the designs. But this dream is hampered by the lack of formal and rigorous semantics for the modeling language of this tool. It is potentially valuable, therefore, to provide formal semantics and to develop formal analysis techniques for important features of the modeling language provided by the MathWorks tool.

MathWorks' Simulink/Stateflow development suite consists of two modeling languages: Simulink is used to model the continuous dynamics and Stateflow is used to specify the discrete control logic and the modal behavior of the system. The first part of this paper discusses semantic issues of Simulink/Stateflow models. Section II provides a formal semantics to the Stateflow modeling language. We achieve this by translating a Stateflow model into a set of *communicating pushdown automata*. The resulting pushdown automata are then translated into a transition system specification language, called SAL, for which many formal analysis tools are available. SAL is described in some detail in Section V-A.

The Stateflow modeling language is based on hierarchical state machines with discrete transitions between states. Hence, it is not surprising that we can translate a Stateflow model into a guarded transition formalism. In Section III, we show how to deal with *arbitrary* continuous dynamical systems. The approach is based on using the simulation semantics to discretize the differential equations into difference equations. The dis-

Research supported by DARPA under the MoBIES program administered by AFRL under contract F33615-00-C-1700 and NASA Langley Research Center contract NAS1-00108 to Rannoch Corporation.

All authors are with the Computer Science Laboratory, SRI International, 333 Ravenswood Ave, Menlo Park, CA, U.S.A. E-mail: {tiwari}@csl.sri.com

cretization parameter is left symbolic. The resulting difference equations are easily cast into the guarded transition system formalism over which we build our analysis tools.

The second part of the paper describes formal analyses techniques that are based on the semantics and translations described in the first part. We can statically determine useful properties of Stateflow charts such as absence of undesirable cycles of broadcast events. Analysis tools for a full Simulink/Stateflow model include symbolic simulation, invariance checking, typechecking, abstraction, and model checking. This tool set provides a graded sequence of formal analysis technologies. On one end are completely automated techniques that determine bounds on recursive event calls and perform extended typechecking. Although such analysis helps in early error detection, it does not provide full verification. Abstraction and invariant generation is used to make the model amenable for exhaustive search. Thus, complete assurance can be provided using theorem proving and model checking. Case studies are used to illustrate some of these techniques.

Currently available versions of the Simulink modeling tool are lacking in static analysis capabilities. They only offer simple simulation facilities that show how the model would behave under a particular input vector. A hybrid dynamical system evolves in time via different trajectories through its state space. In a deterministic system, a fixed given input induces a unique behavior. The complete behavior of the system is given by the set of simulations of the system under all possible input vectors. Exhaustive simulation, however, is not feasible, as the number of possible inputs is usually infinite (even uncountable). In Section V-C, we generalize the notion of simulation to symbolic simulation and show how it can be useful.

Symbolic simulation forms the basis of our tool suite. It is used in different ways to do forward and backward propagation, reachability computation, invariance checking, and typechecking. We illustrate this using simple examples from the hybrid systems literature.

## II. Stateflow Semantics

We provide semantics for Stateflow diagrams via a two-step translation: First, a Stateflow chart is transformed into a set of communicating pushdown automata. Subsequently, the communicating pushdown automata is straightforwardly mapped into a SAL module. The translation from a Stateflow model into SAL preserves the modularity of the original design, thereby allowing analysis of subcomponents of the original design while suitably abstracting other components and the environment.

The communication between different pushdown automata allows for passing of control between any two automata and not only between automata adjacent in the *hierarchy* specified in the original model. This is required for the translation of important features like *supertransitions* and *directed event broadcasting* in the Stateflow language. In addition, the automata share a global pushdown stack that is used to keep track of events that have been broadcast.

### A. Stateflow Charts

Mode control logic in Simulink models is described in terms of hierarchical state machines specified in a variant of Statecharts called Stateflow. A Stateflow chart is described by a tuple $SF = (D, E, S, T, f)$, where
(i) $D = D_I \cup D_O \cup D_L$ is a finite set of typed variables that is partitioned into input variables $D_I$, output variables $D_O$, and local variables $D_L$;
(ii) $E = E_I \cup E_O \cup E_L$ is a finite set of events that is partitioned into input events $E_I$, output events $E_O$, and local events $E_L$;
(iii) $S$ is a finite set of states, where each state is a tuple consisting of three kinds of *actions*: *entry*, *exit*, and *during*; an *action* is either an assignment of an expression to a variable (as in imperative programming languages) or an *event broadcast*;
(iv) $T$ is a finite set of transitions, where each transition is given as a tuple $(src, dst, e, c, ca, ta)$ in which $src \in S$ is the source state, $dst \in S$ is the destination state, $e \in E \cup \{\epsilon\}$ is an event,[1] $c \in WFF(D)$ is a condition given as a well-formed formula in predicate logic over the variables $D$, and $ca, ta$ are set of actions (called condition actions and transition actions, respectively);
(v) $f : S \mapsto (\{and, or\} \times 2^S)$ is a mapping from the set $S$ to the cartesian product of {and, or} with the power set of $S$ and satisfies the following properties: (a) there exists a unique root state $s^{root}$, i.e., $s^{root} \notin \cup_i descendants(s_i)$, where $descendants(s_i)$ is the second component of $f(s_i)$, (b) every nonroot state $s$ has exactly one ancestor state, that is, if $s \in descendants(s_1)$ and $s \in descendants(s_2)$, then $s_1 = s_2$, and (c) the function $f$ contains no cycles, that is, the relation $<$ on $S$ defined by $s_1 < s_2$ iff $s_1 \in descendants(s_2)$ is a strict partial order. If $f(s) = (and, \{s_1, s_2\})$, then the state $s$ is an AND-state consisting of two substates $s_1$ and $s_2$. If $f(s) = (or, \{s_1, s_2\})$, then $s$ is an OR-state with substates $s_1$ and $s_2$.[2]

A *configuration* $c \in 2^S \times \mathbf{D}$ of a Stateflow chart is a tuple containing the set of *active* states and a valuation for all the variables in $D$. The set of all valuations of a variable set $D$ will be denoted by $\mathbf{D}$. If a nonleaf OR-state is active, then exactly one of its descendant substates should be active, and if a nonleaf AND-state is active, then every descendant substate should be active. The set of all configurations that satisfy these conditions, denoted by $\mathcal{C}$, is called the set of *valid* configurations. The formal Stateflow semantics is given by a function $|SF| : \mathcal{C} \times \mathbf{D_I} \times E_I \mapsto \mathcal{C}$. This function maps a configuration, a valuation of the input variables, and an input event to a new configuration.

This semantics function is specified informally through examples in the Matlab documents. Broadly speaking, an input event $e$ causes execution of the root state. A state *executes* by firing any of its transitions that can be fired. If none of the transitions can be fired, the state causes execution of its (either one or all, depending on if it is an OR-state or AND-state) descendants. A transition $t = (src, dst, e, c, ca, ta)$ can fire if event $e$ is present, condition $c$ is true, and state $src$ is active. A transition $t$ *executes* by *preempting* the source state $src$, *executing* condition actions $ca$, *entering* the destination state $src$, and finally *executing* the transition actions $ta$. Executing an assignment action $x := expr$ updates the value of the variable $x$ to $expr$. Executing an event broadcast action is similar to doing a function call and involves executing the target state of the broadcast event. A state is *pre-*

[1] If $e$ is $\epsilon$, then the transition can fire on any event.
[2] In the syntactic description of a Stateflow chart, we have ignored here objects called *junctions* for simplicity.

*empted* by first recursively preempting all its substates and finally marking the state inactive. A state is *entered* by executing its default transitions.

The informal semantics of Stateflow is clearly different from the semantics of Statecharts. Stateflow works only on one event at a time and there is no notion of "maximal and non-conflicting" transitions. Event broadcasting is recursive. Moreover, after an event is processed, control needs to return to the state that generated that event. We need a stack to store this additional information.

### B. Pushdown Systems

We provide formal semantics for Stateflow diagrams via a translation to communicating pushdown systems. Let $A^i = (\Sigma^i, Q^i, \Gamma, s_0^i, \Delta^i)$ be a pushdown system, where $\Sigma^i$ is the input alphabet, $Q^i$ is the set of states of $A^i$, $\Gamma$ is the stack alphabet, $s_0^i$ is the initial state, and $\Delta^i \subseteq (Q^i \times \Gamma^i) \times \Sigma^i \times (Q^i \times \{push(\Gamma^i), pop, \epsilon\})$ is the transition relation. Based on the current control state, the symbol on the top of the stack, and the input symbol, the machine can update the control state and either push a symbol on top of the stack, pop the top symbol, or leave the stack unchanged. The set $Q^i$ consists of valuations of a finite set of typed variables $V^i = V_I^i \cup V_O^i \cup V_L^i \cup V_G$, which are usually classified as input, output, local (state), and global[3] variables. Thus, the set $Q^i$ of states could possibly be uncountable. The input alphabet consists of valuations for the input variables $V_I^i$.

Let $F = \{A_1, \ldots, A_n\}$ be a finite set of pushdown systems with mutually disjoint local variable sets. A function $f$ from the combined set of input variables to the combined output variable set of $F$, that is,

$$f : \cup_i V_I^i \mapsto \cup_i V_O^i$$

is a renaming function. A communicating pushdown system is a tuple (F, f) where $F$ is a finite set of pushdown automata and $f$ is a renaming function. Note that all the automata share the same stack, stack alphabet, and global variables. A communicating pushdown system itself can be flattened into a single pushdown automata.

### C. Mapping Stateflow Charts to Pushdown Systems

A Stateflow model $SF = (D, E, S = \{s_1, \ldots, s_n\}, T, f)$ is translated into a communicating pushdown system $(\{A_1, \ldots, A_n\}, f)$ by transforming every Stateflow state $s_i \in S$ into a pushdown automata $A_i = (\Sigma^i, Q^i, \Gamma, s_0^i, \Delta^i)$.

- Each pushdown automata $A_i$ inherits as global, input, and output variables the Stateflow chart $SF$'s local, input, and output variables, respectively.
- Each pushdown automata $A_i$ contains additional boolean input variables that are unique to it: $\mathbf{defaultPort}_i$ and $\mathbf{controlFPort}_i$, $\mathbf{resultPort}_i$ and $\mathbf{controlBPort}_i$ (only nonleaf states), and $\mathbf{preemptionPort}_i$ (only nonroot states). The **controlFPort** of an automata is true only if the control is with this automata and the control was passed on to this automata from the "parent" automata. The **controlBPort** of an automata is true only if the control is with this automata and the control was passed on to this automata from one of its "descendants" or from the destination of a directed event broadcast. An automata can be requested to preempt itself by setting its **preemptionPort** to true. An automata can be requested to activate itself by setting its **defaultPort** to true.
- Each pushdown automata $A_i$ contains a unique boolean output variable called $\mathbf{state}_i$. The state variable of an automata indicates if the corresponding Stateflow state is marked active or not.
- Each pushdown automata $A_i$ contains a unique boolean local variable $\mathbf{tempVar}$ of type **StackAlphabet** that is used to keep a copy of the event (present on the top of the stack). An AND-state uses this for passing the event to all its descendants.
- Each pushdown automata $A_i$ contains a local variable corresponding to every transition $t \in T$ whose source $src$ and destination $dst$ states are such that $s_i = lca(src, dst)$. We define $lca(src, dst)$ to be the least common ancestor of states $src$ and $dst$ if $src \neq dst$. In case $src = dst$, $lca(src, dst)$ is defined to be the parent of $src$ and $dst$.[4] The local variable corresponding to $t$ is true whenever it is $t$'s turn to execute.
- All events (local, input from Simulink, output from Simulink) are declared as stack alphabet symbols. The stack alphabet type also contains a unique identifier corresponding to each nonleaf state. This identifier serves as the return address passing control back. There is also a unique symbol in the stack alphabet type that marks the end of the stack.

Now consider a transition $t = (src, dst, e, c, ca, ta) \in T$ in the Stateflow chart $(D, E, S, T, f)$. First assume that $src$ and $dst$ are descendants of $lca = lca(src, dst)$.[5] The transition $t$ gives rise to the following transitions in the automata $A_{lca}$:

1. *If state $src$ is active, event $e$ is present on the top of the stack, and variable $t$ is true, then perform condition actions*: Note that assignment actions are easily performed. In case of an event $e1$ broadcast action, a new stack symbol, say $e_{t1}$, is pushed on the stack, followed by pushing a symbol representing state $lca$ (return address) and $e1$ on the stack. The automata $A_{lca}$ detects the completion of the broadcast event action processing by testing if its **controlBPort** is true and the top of stack is $e_{t1}$. In this way, all condition actions are processed.
2. *If all condition actions have been completed, then push a new symbol, say $e_{t2}$, on the stack and preempt $src$*: The automata $A_{src}$ is preempted by setting $A_{src}$'s **preemptionPort** and **controlFPort** to true and setting $A_{lca}$'s **controlFPort** to false.
3. *If state $src$ has preempted, then perform transition actions*: Automata $A_{lca}$ detects if $src$ has preempted by checking if top of stack is $e_{t2}$ and its own **controlBPort** is true. Transition actions are performed in the same way as the condition actions.
4. *If the transition actions have been completed, then push a new symbol, say $e_{t3}$, on the stack and activate the destination state $dst$*: The destination state $dst$ is activated by setting its **controlFPort** and **defaultPort** to true.
5. *If the destination state $dst$ has been activated then indicate that the transition $t$ has been successfully completed*: We test if $dst$ has been activated by checking if the top of the stack is $e_{t3}$

[3] Global variables are considered to be both input and output.

[4] We are ignoring inner transitions here.

[5] We add to $lca$ an input variable that indicates if $src$ is active and four output variables that, respectively, set the **defaultPort** and **controlFPort** of $dst$ and **preemptionPort** and **controlFPort** of $src$.

and if $A_{lca}$'s **controlBPort** is true. We indicate that transition $t$ was successfully completed by setting **tempVar** and **resultPort** accordingly.

Next consider the case where the least common ancestor $lca$ of the $src$ and $dst$ states is such that $src$ and $dst$ are not both immediate descendants of $lca$. In this case, additional transitions are required to make sure that the correct destination state is activated. This can be done in a straightforward way, though it involves addition of extra transitions into certain automata. More specifically, if $(s_0 = lca, s_1, \ldots, s_k = dst)$ is the path from the state $lca$ to the state $dst$, then new transitions and new input/output variables must be added to the automata $A_{s_0}, A_{s_1}, \ldots$.

Note that entry and exit actions that are associated with a state in a Stateflow chart can be suitably included in the transitions that are activated by the **defaultPort** and the **preemptionPort**, respectively. The during action can be similarly performed when control is passed on to an already active state.

Finally, additional transitions are required for capturing other aspects of the semantics of Stateflow.

*Hierarchy* : Whenever an automata $A_s$ corresponding to an AND-state $s$ is activated, it explicitly (through additional transitions) activates each of its descendant automata. Similarly, whenever automata $A_s$ is preempted, it explicitly preempts each of its descendant automata.
*Priority between transitions* : Some transitions are tested before others in a Stateflow chart. This is captured by additional transitions that set the local variables corresponding to a transition $t$ to true or false based on whether or not they are allowed to fire. If a transition is unable to fire, then the local variable for the next transition is set to true.
*Control transitions* : In Stateflow semantics, control is passed starting from the root down the hierarchy to the leaf states, and back the opposite way. This is violated only in the case of directed event broadcast actions. Thus, we need transitions that read the return address from the top of the stack and pass control back to the respective state.
When an active state gets control, it checks to see if any of its transitions can be fired. This is done in order of the priority of the transitions. In case a state is unable to fire any transitions locally, it passes control down the hierarchy. On return, the descendant state passes the control back to its parent indicating whether or not it was able to "use" the event (through the **resultPort** variable).
*Preemption transitions* : If the **preemptionPort** and **controlFPort** variables are true, then the automata needs to be preempted (i.e., made inactive). Preemption involves recursively preempting all the descendants, marking oneself inactive, and passing control back to the state on top of the stack.
*Default transitions* : If the **defaultPort** and **controlFPort** variables are true, then the automata fires its default transitions and marks itself active before returning control.
*Junctions and flow graphs* : Junctions are translated into local boolean variables and the special semantics associated with transitions over junctions are again encoded explicitly. We skip the details here.

### D. Analysis of the Pushdown System

The Stateflow chart represented as a pushdown system can be statically analyzed. By using the algorithm for reachability in pushdown systems [10], it can be determined if a pushdown system requires a bounded or an unbounded stack depth. An unbounded stack depth corresponds to infinite recursive event broadcasting in the Stateflow charts. The Simulink/Stateflow tool detects loops in event broadcasting at simulation time. So that the bounded stack depth analysis can be performed, all non-boolean data variables are abstracted and the analysis is performed on a finite state abstraction. The pushdown system can also be analyzed to detect any nondeterminacy in the Stateflow chart and other such properties. All of this analysis can be performed in an completely automated and non-intrusive way. For some theoretical results on analysis of recursive state machines, the reader is referred to [3].

## III. Continuous Dynamical Systems

In this section, we consider the other extreme of a hybrid system, that is, one in which there is no discrete component. Simulink provides a rich language for modeling such systems. For the purpose of discussion in this paper, we assume that Simulink models are purely continuous dynamical systems. A continuous dynamical system is a tuple $(X_C, U_C, Y_C, I, f, h)$ where $X_C, U_C, Y_C$ are, respectively, open subsets of $\mathbb{R}^n$, $\mathbb{R}^m$, $\mathbb{R}^p$, for some finite values of $n, m, p$, $I \subset X_C$, $f : X_C \times U_C \mapsto TX_C$ and $h : X_C \times U_C \mapsto Y_C$. Here $TX_C$ denotes the tangent space of $X_C$. We assume that $f$ satisfies the standard assumptions for existence and uniqueness of solutions to ordinary differential equations.

The semantics of a continuous dynamical system $(X_C, U_C, Y_C, f, h)$ over an interval $\tau = [t_i, t_f]$ is a collection of tuples $(x, y, u)$ with $x : \tau \mapsto X_C$, $y : \tau \mapsto Y_C$, and $u : \tau \mapsto U_C$ satisfying
*(a)* initial condition: $x(t_i) \in I$,
*(b)* continuous evolution: for all $t \in [t_i, t_f]$, $\dot{x}(t) = f(x(t), u(t))$, and
*(c)* output evolution: for all $t \in [t_i, t_f]$, $y(t) = h(x(t), u(t))$.

Note that Simulink models can describe much richer systems, too. For instance, a model could contain discrete blocks, as well as a mixture of discrete and continuous blocks. (The discrete blocks introduce discrete states $X_D$ into the state space $X = X_D \cup X_C$, and other definitions need suitable modification). Furthermore, Simulink blocks can also express algebraic constraints via zero-time loops.

*Example 1:* As an example of a purely continuous system, we consider simplified leader control from the design of automated highway systems [34]. Suppose vehicle A is following vehicle B in a lane. Let $gap$ denote the distance between the two vehicles, $v0$ be the velocity of vehicle A, and $v1$ be the velocity of vehicle B. In the leader control mode, vehicle A follows vehicle B by suitably adjusting its velocity based on the sensor reading giving the gap (and the rate of change of gap) between the two vehicles. Let us assume that the dynamics of the system are given by the following equations:

$$\begin{aligned} \dot{v1} &= a1 \\ \dot{v0} &= a1 + v1 - v0 \\ \dot{gap} &= v1 - v0 \end{aligned}$$

Let us say that the initialization condition, or equivalently the condition under which this control mode is triggered, is given by $gap \geq 2$ and $v1 \geq v0$. The problem is to show that the rear car does not crash into the car in front, that is, $gap \geq 0$ at all times.

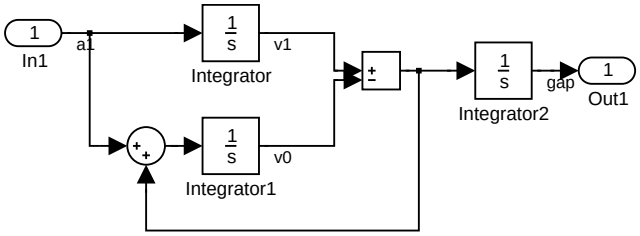


Fig. 1

SIMULINK MODEL OF THE LEADER CONTROL MODE IN AHS

This leader control dynamic system can be modeled in Simulink as shown in Figure 1. However, the initial conditions $gap \geq 2 \ \wedge \ v1 \geq v0$ cannot be represented in Simulink. Similarly, the acceleration $a1$ of the car in front cannot be left unspecified, and a specific funtion is required for performing a simulation. See Example 4 for an analysis using symbolic techniques that reason about the complete state space.

### A. Simulink Simulation Semantics

Simulink computes the values of functions $x : \tau \mapsto X_C$ and $y : \tau \mapsto Y_C$ for a given input $u : \tau \mapsto Y_C$ by numerically integrating the state's derivatives. The numerical integration task is performed by either a fixed-step solver or a variable-step solver. Assuming the use of a fixed-step solver with step size $\delta$, the semantics of the continuous dynamical system $(X_C, U_C, Y_C, I, f, h)$ over an interval $\tau = [t_i, t_f]$ is the collection of tuples $(x, y, u)$ satisfying
*(a)* initial condition: $x(t_i) \in I$,
*(b)* continuous evolution: for all $t \in [t_i, t_f]$, $x(t + \delta) = x(t) + \delta f(x(t), u(t))$, and
*(c)* output evolution: for all $t \in [t_i, t_f]$, $y(t) = h(x(t), u(t))$.

In case of a variable-step solver, the semantics of the continuous dynamical system $(X_C, U_C, Y_C, I, f, h)$ over an interval $\tau = [t_i, t_f]$ is the collection of tuples $(x, y, u)$ satisfying
*(a)* initial condition: $x(t_i) \in I$,
*(b)* continuous evolution: for all $t \in [t_i, t_f]$, $x(t + \delta) = x(t) + \delta f(x(t), u(t))$ for some $0 < \delta < \epsilon$, and
*(c)* output evolution: for all $t \in [t_i, t_f]$, $y(t) = h(x(t), u(t))$.
Here $\epsilon$ is some bound on the sample time the variable-step solver uses.

In subsequent sections, we show that even though analysis of continuous dynamical systems with respect to their original semantics might be infeasible, simple tools can be developed that perform analysis with respect to these alternate simulation semantics. When combined with the various techniques that have been developed for analyzing discrete systems, this gives us an approach for developing tools for hybrid systems. The discretized systems we consider are not theoretically sound abstractions of the original systems. However, for a very large class of systems and for most practical systems, they do provide good insight on the behavior of the actual systems. We do not attempt to make the precise connection between the two models in this paper.

## IV. HYBRID SYSTEMS

Hybrid systems involve the interaction of discrete and continuous dynamics. In the Matlab modeling environment, discrete behavior in the plant and controller is usually specified using Stateflow charts and the continuous behavior using Simulink blocks. Systems modeled using Simulink and Stateflow differ from traditional hybrid dynamical systems in that all Simulink models are deterministic (with respect to the simulation semantics). A generic Simulink/Stateflow model that contains only one Stateflow chart is shown in Figure 2. For purposes

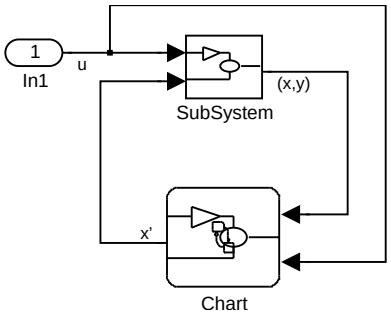


Fig. 2

A GENERIC SIMULINK/STATEFLOW MODEL

of simulation, the Matlab tool considers the Stateflow chart as just another direct feedthrough[6] Simulink block. Stateflow charts in a Simulink model inherit their sample rate from the Simulink blocks, unless their sample rate has been explicitly specified. Stateflow charts could also be triggered by signals from Simulink blocks. An unusual feature of Matlab is that a direct feedthrough loop through a Stateflow chart is treated as an algebraic loop (constraint). For discussion in this paper, we assume that no such loops are present.

Given an interval $\tau = [t_i, t_f]$, a Matlab simulation run is defined over a particular *trajectory* $\{[t'_0, t_1], [t'_1, t_2], \ldots, [t'_{n-1}, t_n]\}$, where $t'_0 = t_i$, $t_n = t_f$, and $t_i = t'_i < t_{i+1}$. The choice of the $t_k$'s where the Stateflow block executes and makes discrete jumps to the state is governed by the sample rate of the Stateflow chart. The semantics in each interval $[t'_k, t_{k+1}]$ is given as described in Section III-A. The state at time $t'_i$ is given using the Stateflow semantics applied on the initial state at time instance $t_i$.

Note that the semantics of a hybrid dynamical system is defined as a collection of all tuples $(\tau, x, y, u)$ such that $\tau$ is *any* trajectory and $x : \tau \mapsto X_D \times X_C$, $y : \tau \mapsto Y_D \times Y_C$, and $u : \tau \mapsto U_D \times U_C$ are mappings that satisfy the conditions (a)–(c) of Section III-A for all intervals $[t'_k, t_{k+1}] \in \tau$ and the condition for discrete evolution at every $t_i, i = 1, \ldots, n-1$.

*Example 2:* As a simple example of a hybrid system, consider a thermostat that controls the temperature $x$ of a room. The thermostat senses the temperature and turns a heater on and off if the threshold values $x_{min}$ and $x_{max}$ are reached, where

[6]Blocks whose current outputs depend on the current inputs are called direct feedthrough blocks.

$0 < x_{min} < x_{max}$ and $x_{min}, x_{max} \in \mathbb{R}^+$. When the heater is off, the temperature of the room decreases and when the heater is turned on, the temperature increases according to the following dynamics:

$$\begin{array}{lcll} off & : & \dot{x} & = & -Kx \\ on & : & \dot{x} & = & -K(x-h) \end{array}$$

Here, the parameter $K \in \mathbb{R}^+$ is the room constant and the parameter $h > x_{min} + x_{max}$ is a real-valued constant that depends on the power of the heater.

The discrete logic to switch between these two modes is given by the following two guarded transitions:

$$\begin{array}{l} state = off \ \wedge \ x \leq x_{min} \longrightarrow state = on \\ state = on \ \wedge \ x \geq x_{max} \longrightarrow state = off \end{array}$$

Let us say that the initial condition is given by $x \geq x_{min} \ \wedge \ x \leq x_{max}$ and the heater is off.

This hybrid model of the thermostat can be represented in Simulink and Stateflow as shown in the figures below. The complete system contains a Stateflow chart that keeps track of the mode of the system and Simulink blocks that describe the continuous dynamics. See Figure 3.

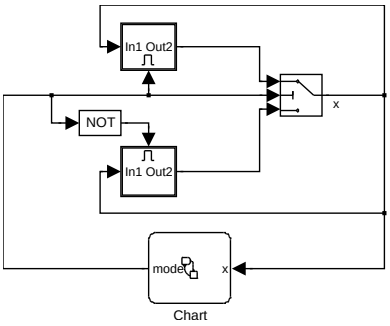


Fig. 3

SIMULINK MODEL OF THE THERMOSTAT EXAMPLE

The Stateflow chart contains two states corresponding to whether the heater is "on" or "off" and transitions between these two modes and is shown in Figure 4.

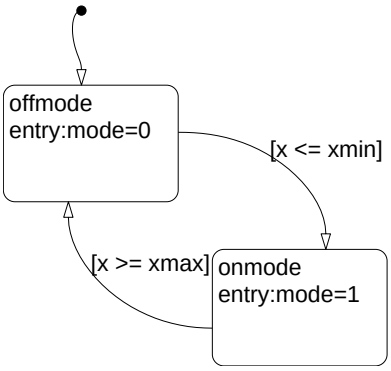


Fig. 4

STATEFLOW COMPONENT IN THE THERMOSTAT MODEL

The Stateflow chart outputs the mode based on which the Simulink component chooses a particular dynamics. The two

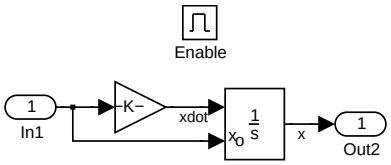


Fig. 5

SIMULINK MODEL OF THE HEATER "OFF" MODE

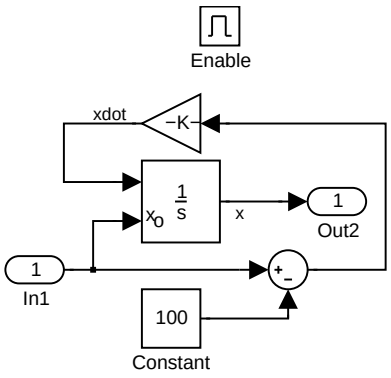


Fig. 6

SIMULINK MODEL OF THE HEATER "ON" MODE

triggered Simulink subsystem blocks describe the dynamics of the temperature in each of the two modes and are shown in Figures 5 and 6.

Note that we have used some hypothetical values for the constants $h$, $K$, $x_{min}$, and $x_{max}$ in the Simulink model. See Example 6 in Section V-E for symbolic analysis of the more general example with symbolic parameter values.

## V. Analysis for Hybrid Systems

We describe new techniques for formal verification of hybrid models. We showed that hybrid system models in Simulink/Stateflow can be translated into transition systems. These transition systems are represented in SAL and techniques for their analysis are implemented as tools over SAL. This makes the tools more generally applicable and independent of the Simulink/Stateflow modeling language.

### A. An Overview of SAL

Symbolic Analysis Laboratory (SAL) is a framework for combining different tools to perform symbolic analysis of discrete systems. At the core of SAL is a language for specifying transition systems in a compositional way. This language serves as the target for translators that extract the transition system description from various domain-specific modeling languages. The intermediate language also serves as a description from which different analysis tools can be driven. Existing analysis tools are interfaced with SAL by translating the intermediate language to the input format for the tools and translating the output of these tools back to the SAL intermediate language.

A transition system *module* in SAL consists of a *state type*, an *initialization condition* on this state type, and a binary *transition relation* on the state type. The state type is defined by

four pairwise disjoint sets of *input*, *output*, *global*, and *local* variables. The initialization and transition relation are specified using guarded assignments over a strongly typed expression language. A *guarded assignment* consists of a guard and a list of assignments. A *guard* is a boolean expression in the current local, global, and output variables and current and next input variables. An *assignment* is an equality between a left-hand-side next variable (i.e., value of the variable in the next discrete time step) and a right-hand-side expression in current and next variables. New modules can also be defined using synchronous and asynchronous composition of existing modules using a renaming facility to avoid name clashes.

The semantics of a SAL module $M$ is given in terms of a Kripke structure $(Q, E, L)$, where $Q$ is the set of states (a state is a valuation of the input, output, local, and global variables), $E \subset Q \times Q$ is a binary relation on the state space $Q$ induced by the guarded transitions, and $L$ is a labeling function that maps each edge in $E$ to the name of the guarded transition that induces that edge. Properties of a SAL module can be stated in any temporal logic, and the interpretation is the usual one [33].

### B. Simulink/Stateflow Models and Hybrid Systems in SAL

Following the discussion in Sections II and III-A, we can transform certain Simulink/Stateflow models into SAL. To see this, note that each pushdown automata is translated directly into a SAL basemodule. Furthermore, the differential equations arising from the Simulink component are converted into difference equations as shown in Section III-A. Note that we have a choice of using the discretization based on either the fixed step simulation semantics or the variable step simulation semantics. Using the same ideas, we can also represent suitable discretized models of hybrid dynamical systems in SAL. We show the benefit of doing this via some illustrations in later sections.

*Example 3:* We can describe the leader control continuous dynamical system outlined in Example 1 in SAL by using the variable step simulation semantics for the differential equations as follows:

```
AHS1 :  CONTEXT =
BEGIN
  leader :  MODULE =
  BEGIN
    INPUT a1 :  REAL, δ :  REAL
    LOCAL gap, v1, v0 :  REAL
    INITIALIZATION
        gap ≥ 2 AND v1 ≥ v0
    TRANSITION
        [
        0 < δ ∧ δ ≤ 1 ⟶
              v1' = v1 + δ * a1;
              v0' = v0 + δ * (a1 + v1 - v0);
              gap' = gap + δ * (v1 - v0) ]
  END;
END
```

Note that SAL allows us to state the initial condition symbolically. It also explicitly specifies the acceleration $a1$ of the leading car to be an input variable so that analysis tools can suitably deal with it. See Example 4.

In the above description, $\delta$ is constrained to be between 0 and 1. In different instances, the upper bound (which is 1 in this case) may need to be different. In case of a fixed time step semantics, $\delta$ is replaced by a constant.

### C. Symbolic Simulation

The Matlab Simulink tool provides extensive simulation facility. Simulation refers to traversing one trajectory of the system behavior from the possible infinite. To run a simulation, the designer must (a) specify initial conditions by giving values of all state variables, (b) choose a particular input function (in case the system has inputs coming from the environment), (c) give some default values to all parameters used in the modeling of the system, and optionally (d) choose a solver and/or a sample time for certain blocks. The simulation tool then computes the system behavior under these specific choices.

Even after doing several simulations with different choices for (a)–(d) above, the designer cannot be sure that the system works correctly in *all* possible scenarios. For instance, in the leader control system of Example 1, simulation would show the behavior of the system under a particular profile of the acceleration of the car in front. But, safety requires that there be no crash under *every* possible acceleration maneuver of the leading car. Similarly, running simulation on the thermostat model of Example 2 will show that the thermostat works as desired for the particular values of parameters $h$, $K$, $x_{min}$, and $x_{max}$ that were chosen for the simulation.

*Symbolic simulation* refers to performing simulation on sets of states represented symbolically. Thus, symbolic simulation differs from regular simulation in two respects. First, it simultaneously traverses a bunch of trajectories instead of a single trajectory through the state space. Second, a set of states is represented symbolically rather than explicitly. This allows representation of a potentially infinite number of states and simulation of a potentially infinite number of trajectories in one symbolic simulation.

We use the language of first-order logic to symbolically represent sets of states. We recall that a state is a valuation of all the state and output variables. A set of states can be specified using a first-order formula over the state and output variables. A crucial step in performing symbolic simulation is the computation of the set of all states that are reachable from the current set of states (represented as a first-order formula). If $\phi(x, y)$ is a first-order formula that represents the current set of states, and

$$\psi(x, y, u, u') \longrightarrow \bigwedge_i (x_i' = e_i(x, y, u, u'))$$

is a guarded transition with guard $\psi(x, y, u)$ and assignments $x_i' = e_i(x, y, u, u')$,[7] then the set of states reached after taking this transition is given by

$$\exists(\bar{x}, \bar{y}, \bar{u}, \bar{u}') : \quad [\phi(\bar{x}, \bar{y}, \bar{u}, \bar{u}') \ \wedge \ \psi(\bar{x}, \bar{y}) \wedge \bigwedge_i (x_i = e_i(\bar{x}, \bar{y}, \bar{u}, \bar{u}')) \ \wedge \ \bigwedge_j (x_j = \bar{x_j})].$$

Note that the $x_j$'s in the above formula are all the state and output variables that are left unchanged by the guarded transition.

[7] The $x_i$'s are state and output variables and the $e_i$'s are expressions that can contain variables $x, y, u$, and $u'$. The expression $e_i$ evaluates to a value of the same type as $x_i$.

Note also that the input variables are existentially quantified, which means no assumption is being made on them. However, if the input is known to satisfy certain constraints, then these can be incorporated in this framework as well.

The existential quantifier in the expression above must be eliminated to ensure that the formulas do not get arbitrarily large very soon, and we discuss this in Section V-D.

Other approaches to performing simulation that are not based on the use of a quantifier elimination procedure have been discussed in the literature as well. A particular case of symbolic simulation is the idea of using intervals to represent sets of states. The polygonal state space can then be simulated using particular numerical methods. There is a need to do an over-approximation whenever the state set is not representable by a polygon [12].

### D. Quantifier Elimination

The cylindrical algebraic decomposition (CAD) algorithm [13, 21] decides the full first-order theory (equality and the greater-than relation included) of ordered real closed fields. Given a set of polynomials over $n$ variables, the CAD procedure decomposes the real $n$-dimensional space into a finite set of regions where each polynomial's evaluation is sign-invariant. The quantifier elimination procedure for real closed fields is obtained as a side effect of the CAD decomposition. Over the last 25 years, the CAD algorithm has been improved and made more efficient [29, 27, 19]. One such efficient implementation is available via the tool QEPCAD [20], which is built over a symbolic algebra library called SACLIB [11].

The tool QEPCAD can be used to perform quantifier elimination over the first-order theory of real closed fields and, consequently, it can be used as a decision procedure for the same theory. As seen above, quantifier elimination is a crucial step in symbolic simulation and reachability algorithms. Note that the QEPCAD tool cannot handle variables that are not of type real, and hence it can be used only on formulas in which all the nonreal variables can be eliminated by suitable preprocessing.

*Example 4:* Following up on Examples 1 and 3, we now show a symbolic simulation step for the system described in Example 3:

$$
\begin{aligned}
\phi_0 \;&:\; gap \geq 2 \;\wedge\; v1 \geq v0 \\
\phi_1 \;&:\; \exists(\bar{gap}, \bar{v1}, \bar{v0}, \bar{a1}, \delta) : \bar{gap} \geq 2 \;\wedge\; \bar{v1} \geq v0 \;\wedge \\
&\qquad v1 = \bar{v1} + \delta\bar{a1} \;\wedge \\
&\qquad v0 = \bar{v0} + \delta(\bar{a1} + \bar{v1} - \bar{v0}) \;\wedge \\
&\qquad gap = \bar{gap} + \delta(\bar{v1} - \bar{v0}) \;\wedge \\
&\qquad 0 < \delta \leq 1 \\
\phi_1' \;&:\; (gap > 2 \;\wedge\; 9gap + v0 - v1 - 18 \leq 0) \;\vee \\
&\qquad (v0 \leq v1 \;\wedge\; 9gap + v0 - v1 - 18 \geq 0)
\end{aligned}
$$

We have shown only one simulation step in the example above because we can *prove* that $gap \geq 0$ always using the results from this one symbolic propagation step. See Example 5.

The quantifier elimination problem has a high time and space complexity. Consequently, techniques for simplification are required before the quantifier elimination tool can be used. In particular, we perform the following two simplifications:

*Solving for quantified variable* : Certain quantified variables can be easily eliminated by solving for them. For example, given the equality $x + y = z + 5$, one can solve for $x$ to obtain $x = z+5-y$. Thus, a quantified formula $\exists x : x+y = z+5 \;\wedge\; \phi(x)$ is equivalent to the formula $\phi(x/z+5-y)$, where $x/z+5-y$ denotes that we replace all occurrences of $x$ in $\phi$ by the expression $z + 5 - y$.

*Logical simplification* : We can use logical equivalences to reduce the size of the formula that is given to the quantifier elimination tool. One of the tautologies that is very useful is $(\exists x : \phi(x) \;\wedge\; \psi) \leftrightarrow (\exists x : \phi(x)) \;\wedge\; \psi$, if $x$ does not occur in $\psi$. This allows us to move parts of the formula that do not contain the quantified variable outside the scope of the quantifier, thus reducing the size of the quantified formula in the process.

Finally, the quantifier elimination procedure is quite sensitive to the ordering of quantified variables. Logically equivalent quantified formulas $\exists x \exists y : \phi(x, y)$ and $\exists y \exists x : \phi(x, y)$ may take drastically different time and space resources for computation.

### E. Invariant Generation and Checking

Symbolic simulation can be used to compute the reachability region as well. In the $i$-th simulation step, the symbolic simulation procedure yields the set of states that are reached in exactly $i$ transitions. Thus, in order to compute the reachable state set, one must collect the set of all states that are reachable in $i$-steps for $i = 0, 1, 2, \ldots$. Each successive iteration would then yield successive approximations of the reachable state set. The exact reachable state space is obtained only in the condition that this process terminates. In case of termination, the set of reachable states is obtained as a formula, which by definition is also the strongest invariant for the given transition system.

*Example 5:* In Example 4 we showed a symbolic simulation step for the the leader control system. Assuming the same notation and same formulas $\phi_i$'s from before, successive approximations $\psi_i$'s of the reachability set would be

$$
\begin{aligned}
\psi_0 \;&=\; \phi_0 \;=\; gap \geq 2 \;\wedge\; v1 \geq v0 \\
\psi_1 \;&=\; \psi_0 \;\vee\; \phi_1 \\
\psi_1' \;&=\; \psi_0 \;\vee\; (gap > 2 \;\wedge\; 9gap + v0 - v1 - 18 \leq 0) \;\vee \\
&\qquad (v0 \leq v1 \;\wedge\; 9gap + v0 - v1 - 18 \geq 0)
\end{aligned}
$$

The last formula $\psi_1'$ is *logically equivalent* to the formula $gap \geq 2 \;\wedge\; v1 \geq v0$. This logical equivalence can also be shown using the quantifier elimination decision procedure that is used in the symbolic simulation steps. This establishes that the formula $\psi_0$ is an invariant of the system. The invariant $\psi_0$ implies that $gap > 0$, and this establishes that the rear car never crashes onto the car in front under the given leader control law.

The method outlined above for generating an invariant assertion by computing the exact reachable region using forward symbolic propagation is, in general, not sufficient in many cases. In some of these other cases, a combination of approaches based on forward and backward propagation with suitable narrowing and widening might be required. See [35] for the details. For an example of some of these ideas, see also Example 6.

However, the technology outlined above is *sufficient* for invariant *checking*. To check if a formula $\phi$ is an inductive invariant, we test (i) if the formula describing the initial states implies

the formula $\phi$ and (ii) if the result of symbolic propagation starting from the formula $\phi$ (logically) implies the formula $\phi$. Both of these tests can be done using a quantifier elimination procedure. In fact, Examples 4 and 5 can also be seen as checking that the formula given as the initial condition is an inductive invariant.

Simple invariants on the values of variables can also be specified using *types*. Richer type system allows specification of more complex relations between the values of different variables. Invariant checking can be used to perform *typechecking* on such rich type systems. The designer can easily annotate his Simulink/Stateflow model by such type information using additional Simulink blocks.

To illustrate that the symbolic propagation method can in fact *generate* invariants, we consider the thermostat example.

*Example 6:* The thermostat hybrid system discussed in Example 2 can be expressed using guarded transitions. Let us assume that the variables $x, x_{min}, x_{max}$, and $h$ are state (local) variables declared to be reals. The variable $\delta$ is declared to be a real input variable and is constrained to be between 0 and $1/K$.

$$
\begin{array}{ll}
state = off \ \wedge \ x \leq x_{min} \longrightarrow & state' = on \\
state = on \ \wedge \ x \geq x_{max} \longrightarrow & state' = off \\
state = on \ \wedge \ x < x_{max} \ \wedge \ \delta > 0 \ \wedge \ K\delta \leq 1 \longrightarrow & \\
\qquad x' = x + \delta(-K)(x - h) & \\
state = off \ \wedge \ x > x_{min} \ \wedge \ \delta > 0 \ \wedge \ K\delta \leq 1 \longrightarrow & \\
\qquad x' = x + \delta(-Kx) &
\end{array}
$$

The parameters $x_{min}$, $x_{max}$, and $h$ could be any real numbers that satisfy the condition $0 < x_{min} < x_{max} < h$ (this is part of the specification of the problem). We do not explicitly mention this conjunct in the expressions below, but it is implicitly assumed in the computation.

Starting with an initial state in which we assume nothing on the value of $x$ and $state$ variables, symbolic simulation gives the following:

$$
\begin{array}{lll}
\phi_0 & : & true \\
\phi_1 & : & (\exists(\bar{state}) : \bar{state} = off \ \wedge \ x \leq x_{min} \ \wedge \ state = on) \\
 & \vee & (\exists(\bar{state}) : \bar{state} = on \ \wedge \ x \geq x_{max} \ \wedge \ state = off) \\
 & \vee & (\exists(\bar{x}, \delta) : state = on \ \wedge \ \bar{x} < x_{max} \ \wedge \\
 & & \quad 0 < \delta \leq 1/K \ \wedge \ x = \bar{x} - K\delta(\bar{x} - h)) \\
 & \vee & (\exists(\bar{x}, \delta) : state = off \ \wedge \ \bar{x} > x_{min} \ \wedge \\
 & & \quad 0 < \delta \leq 1/K \ \wedge \ x = \bar{x} - K\delta\bar{x}) \\
\phi_1 & : & (state = on \ \wedge \ x \leq x_{min}) \ \vee \\
 & & (state = off \ \wedge \ x \geq x_{max}) \ \vee \\
 & & (state = on \ \wedge \ x < h) \ \vee \\
 & & (state = off \ \wedge \ x > 0) \\
\phi_1 & : & (state = on \ \wedge \ x < h) \ \vee \ (state = off \ \wedge \ x > 0)
\end{array}
$$

We do not show the rest of the computation here, but it can be checked that we get the same formula after the second symbolic simulation step as well. Thus, the set of states represented by $\phi_1$ is an invariant of the system.

Note that we can get a stronger invariant if we make a stronger assumption on the parameter $\delta$. As $\delta$ is constrained to be in a smaller neighborhood of $0^+$, the upper and lower bound on $x$ in the invariant gets closer to $x_{max}$ and $x_{min}$, so that in the limit, the invariant is $(state = on \ \wedge \ x \leq x_{max}) \ \vee \ (state = off \ \wedge \ x \geq x_{min})$.

We emphasize here that in this computation, no assumption was made on (i) the values for the parameters $h, x_{min}$, and $x_{max}$, or (ii) the initial state of the system.

### F. Abstraction of the Continuous Component

An abstraction of a system is any system that exhibits all the behaviors (trajectories) of the original system, possibly more. Abstract systems are usually smaller and are obtained by suitable generalization or pruning of information from the original system. Abstraction is essential for analyzing systems containing a large number of state variables. Since fairly efficient model checking tools are available for searching through a large, but finite, (discrete) state space, one of the challenges in building analysis tools for hybrid systems is to come up with suitable abstractions for the continuous components that are refined enough to suffice for proving the properties of interest.

In the most simple form, one could use the most coarse abstraction of the continuous component of a hybrid model. The most coarse and trivial abstraction of any system is the system that accepts all behaviors. This corresponds to having no information about the continuous subsystem. This means that we execute the discrete transitions in a completely nondeterministic environment. The resulting discrete transition system can be model checked, but such an analysis is unlikely to give any useful information about the model.

More refined abstractions can be constructed using invariants that are established using the techniques of Section V-E.

In general, construction of an abstraction for a transition system involves
(i) defining the abstract states: In case of predicate abstraction, certain predicates over the original local, input, and output variables are mapped onto boolean variables (losing information in the process); and
(ii) mapping the transitions to abstract transitions: In case of predicate abstraction, this amounts to mapping the guard and assignments of the original system into abstract guards and transitions over the new boolean variables. This requires certain theorem proving capabilities.

Invariants are useful in both of the above steps. The atomic formulas that appear in an invariant can be used as predicates that are mapped to new boolean variables. Furthermore, the invariants also help to discharge some of the proof obligations that arise in step (ii) of the above process.

*Example 7:* Controllers for hybrid systems are often designed under a multiobjective setting [28, 36]. Most often, the requirements are that of safety (i.e., all the system trajectories satisfy certain constraints) and efficiency (i.e., optimizing certain other parameters). We showed in Example 6 that the switching law between the two modes of operation of the thermostat control is safe. Now if there is an additional controller hierarchically above the simple mode switching one, which, for example, controls the power of the heater (value of the parameter $h$) to optimize power consumption, we could verify that by using a suitable abstraction of the original system model. The abstraction

could be a simple one based on the inductive invariant property we established for the base system in Example 6.

The combination of building abstractions using information from the invariants generated by symbolic propagation and model checking the resulting system is a powerful tool chain for scaling the techniques proposed in this paper to larger and more complex embedded control systems.

## VI. Conclusion

Much work has been done in the design and analysis of hybrid systems [14,7,4,8,17]. This paper is an attempt to develop tools and techniques for performing formal analysis on models that are developed in Simulink and Stateflow. This MathWorks software is one of the leading tool suites available to engineers for designing hybrid and embedded control systems. But the techniques developed in this paper are not specific to this particular input formalism. We have implemented our analysis tools over SAL, which is an intermediate format for representing transition systems, quite independent of any modeling language.

We have presented a wide range of formal technologies for hybrid control systems starting from completely automated and invisible techniques like static analysis of simple program properties and symbolic simulation, through extended typechecking, to abstraction and invariant generation. These analysis tools can be embedded into design languages like Stateflow/Simulink to provide greater assurance and quick error detection.

Stateflow design language is based on the concept of hierarchical automata from Statecharts [15], but the semantics of Stateflow diagrams is different from the semantics of Statecharts in several ways. There have been efforts at providing semantics to Statecharts [16]. Hierarchical automata were used for this purpose in [30], and a set of several different semantics for Statecharts was given in [37].

Quantifier elimination tools have been used in the hybrid system world in a variety of contexts. Formulas and expressions over the first-order theory of real closed fields arise naturally when linear and non-linear control systems are described. Many problems in control theory can be reduced to finding solutions of systems of polynomial equations, disequations, and inequalities [22]. Quantifier elimination is also used in obtaining decidability results for reachability in safety-critical embedded systems and hybrid systems [26]. Many applications, especially in mechanical engineering and in numerical analysis, lead to formulas with trignometric functions involved [32]. In fact, CAD-based quantifier elimination procedures have been used to solve problems regarding stationarity, stability, and reachability of control system designs [23]. Requiem [31] is a tool for performing exact reachability state set computation for linear systems specified using nilpotent matrices. It uses the quantifier elimination procedure implemented inside Mathematica. The computation of the reach set for parametric inhomogenous linear differential systems is done using implicitization and quantifier elimination in [6].

A *finite* decomposition of the real space $\mathbb{R}^n$ into open sets and points such that each partition element preserves a first-order formula over reals is crucial not only for getting a decision procedure for the first-order theory, but also for obtaining finite abstractions of certain hybrid systems [5]. In fact, a model-theoretic structure over the reals in which every (first-order) definable subset of $\mathbb{R}^n$ is a *finite* union of points and open intervals is called a *o-minimal* structure. It is shown in [5] that hybrid systems that are definable over some o-minimal structure admit finite abstractions. The class of o-minimal structures over the reals includes structures with richer signatures as well.

The transition system we associate with a hybrid system in this paper is different from the usual one. In particular, the transition system that corresponds to the continuous flow in hybrid systems, as defined in in Section III-A, is an approximation and is different from the standard definition, as in [5].

## References


[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(3):3–34, 1995. 1

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 1

[3] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001. 4

[4] R. Alur, T. Henzinger, and E. D. Sontag (eds.). *Hybrid Systems III*. Springer-Verlag, Berlin, 1996. volume 1066 of *Lecture Notes in Computer Science*. 10

[5] Rajeev Alur, Tom Henzinger, Gerardo Lafferriere, and George J. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(2):971–984, July 2000. 10

[6] H. Anai and V. Weispfenning. Reach set computations using real quantifier elimination. In M. D. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control HSCC 2001*, volume 2034 of *LNCS*, pages 63–76. Springer-Verlag, 2001. 10

[7] P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (eds.). *Hybrid Systems II*. Springer-Verlag, Berlin, 1995. volume 999 of *Lecture Notes in Computer Science*. 10

[8] P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (eds.). *Hybrid Systems IV*. Springer-Verlag, Berlin, 1997. volume 1273 of *Lecture Notes in Computer Science*. 10

[9] Gérard Berry, Hubert Comon, and Alain Finkel, editors.

*Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.
1

[10] A. Bouajjani, J. Esparza, and O. Maler.
Reachability analysis of pushdown automata: Application to model-checking.
In A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR 97: 8th International Conference on Concurrency Theory*, volume 1243 of *LNCS*, pages 135–150. Springer-Verlag, 1997.
4

[11] B. Buchberger, G. E. Collins, M. J. Encarnacion, H. Hong, J. R. Johnson, W. Krandick, R. Loos, A. M. Mandache, A. Neubacher, and H. Vielhaber.
SACLIB 1.1 user's guide.
In *RISC-Linz Report Series, Tech Report No 93-19*. Kurt Gödel Institute, 1993.
`www.eecis.udel.edu/~saclib/`.
8

[12] Alongkrit Chutinam and Bruce H. Krogh.
Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations.
In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, volume 1569 of *LNCS*, pages 76–90. Springer-Verlag, 1999.
8

[13] G. E. Collins.
Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition.
In *Proc. Second GI Conf. Automata Theory and Formal Languages*, pages 134–183, 1975.
Vol. 33 of *Lecture Notes in Comp. Sci., Springer, Berlin*.
8

[14] R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel (eds.).
*Hybrid Systems*.
Springer-Verlag, Berlin, 1993.
volume 736 of *Lecture Notes in Computer Science*.
10

[15] D. Harel.
Statecharts: A visual formalism for complex systems.
*Sci. Comput. Program.*, 8:231–274, 1987.
10

[16] D. Harel and A. Naamad.
The statemate semantics of statecharts.
*ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
10

[17] T. Henzinger and S. Sastry (eds.).
*Hybrid Systems: Computation and Control*.
Springer-Verlag, Berlin, 1998.
volume 1386 of *Lecture Notes in Computer Science*.
10

[18] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya.
What's decidable about hybrid automata?
*Journal of Computer and System Sciences*, 57:94–124, 1998.
1

[19] H. Hong.
An improvement of the projection operator in cylindrical algebraic decomposition.
In *Proc. ISAAC 90*, pages 261–264, 1990.
8

[20] H. Hong.
Quantifier elimination in elementary algebra and geometry by partial cylindrical algebraic decomposition version 13.
In *The world wide web*, 1995.
`http://www.gwdg.de/~cais/systeme/-saclib, www.eecis.udel.edu/~saclib/`.
8

[21] Mats Jirstrand.
Cylindrical algebraic decomposition - an introduction.
Technical Report LiTH-ISY-R-1807, Dept of EE. Linköping University, S-581 83 Linköping, Sweden, Dec 1995.
Available by anonumous ftp at ftp.control.ee.liu.se.
8

[22] Mats Jirstrand.
Algebraic methods for modeling and design in control.
Licentiate thesis LIU-TEK-LIC-1996:05 Linkping Studies in Science and Technology. Thesis No 540, Department of Electrical Engineering, Li, 1996.
10

[23] Mats Jirstrand.
Nonlinear control system design by quantifier elimination.
*Journal of Symbolic Computation*, 24(2):137–152, Aug 1997.
10

[24] Warren A. Hunt Jr. and Steven D. Johnson, editors.
*Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*. Springer, 2000.
1

[25] G. Lafferriere, G. J. Pappas, and S. Sastry.
O-minimal hybrid systems.
*Mathematics of Control, Signals, and Systems*, 13(1):1–21, 2000.
1

[26] Gerardo Lafferriere, George J. Pappas, and Sergio Yovine.
Symbolic reachability computations for families of linear vector fields.
*J. Symbolic Computation*, 2001.
To appear.
10

[27] D. Lazard.
*An improved projection for cylindrical algebraic decomposition*.
Technical Report, Informatique, Universite Paris IV, F-75252 Paris Cedex 05, France, 1990.
8

[28] John Lygeros, Claire Tomlin, and Shankar Sastry.
Controllers for reachability specifications for hybrid systems.

*Automatica, Special Issue on Hybrid Systems*, 35(3), March 1999.
9

[29] S. McCallum.
An improved projection operator for cylindrical algebraic decomposition of three dimensional space.
*J. Symbolic Computation*, 5:141–161, 1988.
8

[30] E. Mikk, Y. Lakhnech, and M. Siegel.
Hierarchical automata as model for statecharts.
In *Asian Computing Science Conference (ASIAN'97)*, volume 1345 of *LNCS*. Springer-Verlag, 1997.
10

[31] P. Mishra and G. J. Pappas.
Reachability using quantifier elimination.
In *University of Pennsylvania hybrid systems group webpage*, 2001.
`www.seas.upenn.edu/hybrid/requiem.html`.
10

[32] Petru Pau and Josef Schicho.
Quantifier elimination for trignometric polynomials by cylindrical algebraic trignometric decomposition.
www.risc.uni-linz.ac.at/people/ppau, Research Institute for Symbolic Computation, Johannes Kepler University, A-4040 Linz, Austria, 1999.
10

[33] A. Pnueli.
The temporal logic of programs.
In *Proceedings of the 18th annual symposium on foundations of computer science*, pages 46–57. IEEE computer society press, 1977.
7

[34] A. Puri and P. Varaiya.
Driving safely in smart cars.
In *Proceedings of the 1995 American Control Conference*, 1995.
4

[35] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar.
A technique for invariant generation.
In Tiziana Margaria and Wang Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 113–127, Genova, Italy, April 2001. Springer-Verlag.
8

[36] Claire Tomlin, John Lygeros, and Shankar Sastry.
Synthesizing controllers for nonlinear hybrid systems.
In S. Sastry and T. Henzinger, editors, *Hybrid Systems: Computation and Control*, volume 1386 of *LNCS*, pages 360–373. Springer-Verlag, 1998.
9

[37] M. von der Beek.
A comparison of stateflow variants.
In L. de Roever and J. Vytopil, editors, *Formal techniques in real-time and fault tolerant systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.
10