Embedding Hybrid Automata in Model-Based Design for Cyber-Physical Systems

Stanley Bak[1], Omar Ali Beg[2], Sergiy Bogomolov[3],
Taylor T. Johnson[2], Luan Viet Nguyen[2], and Christian Schilling[4]

[1] Air Force Research Laboratory, USA,
[2] University of Texas at Arlington, USA
[3] IST Austria, Austria
[4] University of Freiburg, Germany

Abstract. Hybrid automata are an important formalism for modeling dynamical systems exhibiting mixed discrete-continuous behavior such as control systems and are amenable to formal verification. However, hybrid automata lack expressiveness compared to integrated model-based design (MBD) frameworks such as the MathWorks' Simulink/Stateflow (SLSF). In this paper, we propose a technique for correct-by-construction compositional design of cyber-physical systems (CPS) by embedding hybrid automata into SLSF models. Hybrid automata are first verified using hybrid automata verification tools, and then automatically translated to embed the hybrid automata into SLSF models such that the properties verified are transferred and maintained in the translated SLSF model. The resultant SLSF model can then be used for automatic code generation and deployment to hardware, resulting in a correct-by-construction implementation. We show the effectiveness of our approach on a range of CPS case studies, and validate the overall correct-by-construction methodology—from formal verification through implementation in hardware controlling an actual plant—using a closed-loop buck converter.

1 Introduction

In this paper, we present the theory and associated implementation of the translation of hybrid automaton models (used for verification) to the MathWorks Simulink/Stateflow (SLSF) models (used for design refinement, simulation, implementation, and code generation for target embedded hardware). Our approach is particularly of interest if the design process is structured in a bottom-up fashion. In other words, we assume that the individual system components are first modeled in detail, such as modeling a control algorithm as a hybrid automaton and verifying properties (typically safety) for it. These components are then linked together to form the whole system under consideration within SLSF. This leads to overall system models consisting of heterogeneous components where a number of components are modeled as hybrid automata, but the entire system may be too complex to formally model and verify. In the last

DISTRIBUTION A. Approved for public release; Distribution unlimited. (Approval AFRL PA #88ABW-2015-2402)

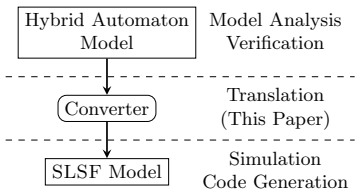


Fig. 1: High-level overview of the model-based design process enabled by this work. Verification using the hybrid automaton is first performed in a hybrid systems model checker, then we automatically generate a trajectory-equivalent SLSF diagram. The diagram can then be embedded into a more complex system, possibly with other, unverified, components (because they are too large to verify, exist for legacy reasons, etc.), and can then be used for code generation and implementation in actual systems.

decade, a number of powerful design and analysis tools for hybrid automata such as SpaceEx [7, 8, 16], Flow* [12], and dReach [17] have emerged. Those tools are particularly useful for the formal analysis and verification of hybrid automata. In the proposed approach, a designer can ensure the correctness of individual components before using our conversion process and linking the tools together in SLSF (see Fig. 1).

We introduce a technique to automatically convert hybrid automata into *trajectory-equivalent* SLSF diagrams. By trajectory-equivalent, we mean that as more simulations of the SLSF diagram are run, the non-determinism in the original hybrid automaton will be exhaustively explored. One technical challenge is that hybrid automata and SLSF differ in semantics: a hybrid automaton is typically defined with *may*-semantics with respect to the discrete transitions, whereas SLSF employs *must*-semantics. In other words, a transition in SLSF is taken as soon as the transition guard is enabled, whereas the hybrid automaton still has the freedom to stay in the current location as long as the location invariant has not been violated. Our approach incorporates additional randomization steps into the resulting SLSF diagram. In this way, in every run, the diagram produces a different trace that still reflects the hybrid automaton semantics. After running more and more simulations, we get a better and better approximation of the reachable state space of the original hybrid automaton.

Related Work. Significant research has been done on the translation of SLSF diagrams into other analysis tools, such as hybrid systems model checkers [1, 3, 6, 10, 11, 23, 28, 29, 31, 33]. Agrawal et al. [1] suggest an algorithm to translate SLSF diagrams into the equivalent HSIF [10, 11, 28, 29] models. The Compositional Interchange Format (CIF) provides a common input language focused on model compositionality for networks of hybrid automata [2]. Alur et al. translated SLSF to linear hybrid automata for applying symbolic analysis to improve test coverage of SLSF [3]. In a different setting, Schrammel et al. [31] consider the translation problem for complex SLSF diagrams where involved treatment of zero-crossings is needed. Manamcheri et al. [23] have developed the tool HyLink to translate a restricted class of SLSF to hybrid automata. The application of the above techniques is restricted by the fact that no complete semantics of SLSF is provided (in spite of recent progress [6, 9, 18, 19, 23, 30]).

In contrast to all these existing works, we consider the converse direction, i.e., to translate a given hybrid automaton into an SLSF diagram. In this setting, we benefit from clear and unambiguous hybrid automata semantics and may for-

mally verify properties of the hybrid automata prior to translating them to SLSF diagrams. Pajic et al. [20, 25–27] consider a similar problem of converting timed automata encoded in UPPAAL [21] to SLSF diagrams. However, in their translation, they consider only runs of UPPAAL models that obey the *must*-semantics. In our work, beyond considering the much more expressive framework of hybrid automata (as timed automata are a subclass of hybrid automata), we provide a translation handling the non-determinism by producing trajectory-equivalent SLSF diagrams. Operational semantics of (purely discrete) Stateflow have been developed [19], and alternative formalizations of discrete semantics have been investigated using, e.g., translation from Stateflow-to-C [30]. In contrast to these prior works, we focus on continuous-time Stateflow diagrams.

Contributions. This paper has four primary contributions.

(a) This is the first work, as far as we are aware, to provide a translation scheme from hybrid automata to SLSF diagrams, which is useful as part of a model-based design (MBD) process. *(b)* In order to overcome the difference in semantics between the modeling frameworks, we introduce the notion of trajectory-equivalence, and show how the conversion preserves trajectory-equivalence with respect to several sources of non-determinism in hybrid automata. *(c)* We provide an implementation of the trajectory-equivalent translation scheme as a part of the HyST model translation framework [4], which enables completely automatic translation of existing hybrid automaton models. *(d)* We show the applicability of our contributions in several case studies where hybrid automata are automatically translated to SLSF for simulation, use in larger SLSF diagrams, and deployment to actual hardware. For one case study—a closed-loop buck converter—the entire correct-by-construction MBD process is illustrated, from verification through implementation in hardware. This includes formal verification of the hybrid automaton in SpaceEx, translation to SLSF, code generation for the controller in SLSF, then subsequent compilation, and finally execution in embedded hardware controlling the physical plant.

Paper Organization. The remainder of the paper is organized as follows. After introducing the necessary background in Sec. 2, we present our trajectory-equivalent translation scheme in Sec. 3, followed by a discussion in Sec. 4. In Sec. 5, we evaluate our approach on several case studies, and conclude in Sec. 6.

2 Preliminaries

In this section, we introduce the preliminaries that are needed for this work. In Sec. 2.1, we define a hybrid automaton model and discuss its semantics. This is followed by a discussion of SLSF in Sec. 2.2.

2.1 Hybrid Automata

A hybrid automaton is formally defined as follows.

Definition 1 (Hybrid Automaton). *A* hybrid automaton *is a tuple* $\mathcal{H} = (Loc, Var, Init, Flow, Trans, Inv)$ *with: (a) the finite set of locations* Loc*, (b) the*

set of continuous variables $Var = \{x_1, \ldots, x_n\}$ *from* $\mathbb{R}^n$*, (c) the initial condition, given by* $Init(\ell) \subseteq \mathbb{R}^n$ *for each location* ℓ*, (d) the flow, a deterministic function* $Flow(\ell)$ *from the variables to their derivatives for each location* ℓ*, (e) the discrete transition relation Trans, where every transition is a tuple* $(\ell, g, \upsilon, \ell')$ *with: (i) the source location* ℓ *and the target location* ℓ'*, (ii) the guard, given by a constraint* g*, (iii) the update, given by a mapping* υ*, and (f) the invariant* $Inv(\ell) \subseteq \mathbb{R}^n$ *for each location* ℓ*.*

The semantics of a hybrid automaton $\mathcal{H}$ are defined as follows. A *state* of $\mathcal{H}$ is a pair $(\ell, \mathbf{x})$ that consists of a location $\ell \in Loc$ and a point $\mathbf{x} \in \mathbb{R}^n$. Formally, $\mathbf{x}$ is a valuation of the continuous variables in Var. For the following definitions, let $T = [0, \Delta]$ be an interval for some $\Delta \geq 0$. A *trajectory* of $\mathcal{H}$ from state $s = (\ell, \mathbf{x})$ to state $s' = (\ell', \mathbf{x}')$ is defined by a pair $\rho = (L, \mathbf{X})$, where $L : T \to Loc$ and $\mathbf{X} : T \to \mathbb{R}^n$ are functions that define for each time point in T the location and values of the continuous variables, respectively. We use the following terminology for a given trajectory ρ. A sequence of time points where location switches happen in ρ is denoted by $(\tau_i)_{i=0\ldots k} \in T^{k+1}$. In this case, we define the *length* of ρ as $|\tau| = k$. Trajectories $\rho = (L, \mathbf{X})$, and the corresponding sequence $(\tau_i)_{i=0\ldots k}$, must satisfy the following conditions: *(a)* $\tau_0 = 0$, $\tau_i < \tau_{i+1}$, and $\tau_k = \Delta$ – the sequence of switching points increases, starts with 0 and ends with Δ, *(b)* $L(0) = \ell$, $\mathbf{X}(0) = \mathbf{x}$, $L(\Delta) = \ell'$, $\mathbf{X}(\Delta) = \mathbf{x}'$ – the trajectory starts in $s = (\ell, \mathbf{x})$ and ends in $s' = (\ell', \mathbf{x}')$, *(c)* $\forall\, i\ \forall\, t \in [\tau_i, \tau_{i+1}) : L(t) = L(\tau_i)$ – the location is not changed during the continuous evolution, *(d)* $\forall\, i\ \forall\, t \in [\tau_i, \tau_{i+1}) :\ (\mathbf{X}(t), \dot{\mathbf{X}}(t)) \in Flow(L(\tau_i))$ holds and thus the continuous evolution is consistent with the differential equations of the corresponding location, *(e)* $\forall\, i\ \forall\, t \in [\tau_i, \tau_{i+1}) :\ \mathbf{X}(t) \in Inv(L(\tau_i))$ – the continuous evolution is consistent with the corresponding invariants, and *(f)* $\forall\, i < k\ \exists\, (L(\tau_i), g, \upsilon, L(\tau_{i+1})) \in Trans : \mathbf{X}_{end}(i) \in g \wedge \mathbf{X}(\tau_{i+1}) = \upsilon(\mathbf{X}_{end}(i)) \wedge \mathbf{X}_{end}(i) = \lim_{\tau \to \tau_{i+1}^-} \mathbf{X}(\tau)$ – every continuous transition is followed by a discrete one; $\mathbf{X}_{end}(i)$ defines the values of continuous variables right before the discrete transition at the time moment τ_{i+1}. A state s' is *reachable* from state s if there exists a trajectory from s to s'.

In the following, we primarily refer to *symbolic states*. A symbolic state $s = (\ell, \mathcal{R})$ is defined as a pair, where $\ell \in Loc$ and $\mathcal{R}$ is a convex and bounded set consisting of points $\mathbf{x} \in \mathbb{R}^n$. The continuous part $\mathcal{R}$ of a symbolic state is also called *region*. The symbolic state space of $\mathcal{H}$ is called the *region space*. The initial set of states $\mathcal{S}_{init}$ of $\mathcal{H}$ is defined as $\bigcup_\ell (\ell, Init(\ell))$. The reachable state space $\mathrm{Reach}(\mathcal{H})$ of $\mathcal{H}$ is defined as the set of symbolic states that are reachable from an initial state in $\mathcal{S}_{init}$, where the definition of reachability is extended accordingly for symbolic states. We refer to the set of all the trajectories of $\mathcal{H}$ starting in $\mathcal{S}_{init}$ by $\mathrm{Traj}(\mathcal{H})$.

2.2 Continuous-Time Stateflow Diagrams

Simulink is a graphical modeling language for control systems, plants, and software. Stateflow is a state-based graphical modeling language integrated within

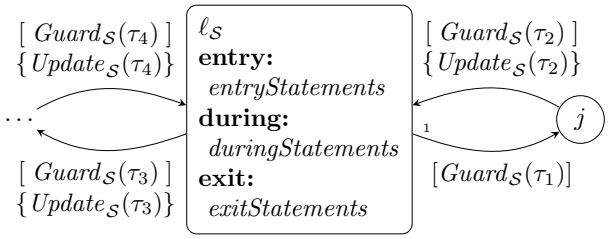


Fig. 2: Snippet of a general continuous-time Stateflow diagram with a state $\ell_{\mathcal{S}}$, a junction j, and four transitions $\tau_1 - \tau_4$.

Simulink. Continuous-time Stateflow diagrams provide methods for modeling hybrid systems that consist of continuous and discrete states and behaviors. In this section, we describe a *restricted subclass of continuous-time Stateflow diagrams* to which we translate a hybrid automaton. In particular, we focus only on continuous-time Stateflow state transition diagrams and we do not consider models with hierarchical states.

Roughly, a Stateflow state transition diagram may be thought of as an extended state machine with variables of various types. In addition to states, Stateflow diagrams may have junctions that are instantaneous. A transition between states may occur at each simulation time step, whereas multiple junction transitions may occur in a single simulation time step.

A continuous-time Stateflow diagram (see Fig. 2) is roughly analogous to a hybrid automaton, but their behavior differs in several ways. In particular, Stateflow diagrams are deterministic, have urgent transitions with priorities, and generally events to process during simulation like enabled transitions are determined by zero-crossing detection algorithms.

We define a Stateflow diagram more formally now.

Definition 2 (Stateflow diagram). *A Stateflow diagram is a tuple* $\mathcal{S} = (Loc_{\mathcal{S}}, Junc_{\mathcal{S}}, Var_{\mathcal{S}}, Trans_{\mathcal{S}}, Actions_{\mathcal{S}})$. *Here, (a)* $Loc_{\mathcal{S}}$ *is a finite set of states (We also use the term* locations *instead.), (b) the junctions* $Junc_{\mathcal{S}}$ *are like locations, but all of which may be evaluated in a single simulation event step (i.e., they are instantaneous "states"), (c)* $Var_{\mathcal{S}}$ *is a finite set of variables of various types, (d) the* $Actions_{\mathcal{S}}(\ell_{\mathcal{S}})$ *for each location* $\ell_{\mathcal{S}}$ *are actions described by Matlab or C statements that are performed at different event times subdivided into entry, during, and exit actions (also, we note that these statements are evaluated sequentially, while hybrid automaton actions are executed concurrently), (e) the discrete transition relation* $Trans_{\mathcal{S}}$ *where every transition* $\tau \in Trans_{\mathcal{S}}$ *is formally defined as a tuple* $(\ell_{\mathcal{S}}, Guard_{\mathcal{S}}, Update_{\mathcal{S}}, TP_{\mathcal{S}}, \ell'_{\mathcal{S}})$*: (i) the source location or junction* $\ell_{\mathcal{S}} \in Loc_{\mathcal{S}} \cup Junc_{\mathcal{S}}$ *and the target location or junction* $\ell'_{\mathcal{S}} \in Loc_{\mathcal{S}} \cup Junc_{\mathcal{S}}$*, (ii) the guard, given by a constraint* $Guard_{\mathcal{S}}$*, must be satisfied for a transition to be taken, (iii) the update, given by a mapping* $Update_{\mathcal{S}}$*, modifies state variables, and (iv) the priority, given by* $TP_{\mathcal{S}}$*, is a natural number between* 1 *and* $od(\ell_{\mathcal{S}})$*—the outdegree of (number of transitions leaving) the state or junction* $\ell_{\mathcal{S}}$*—that indicates the order in which transitions are taken if more than one is enabled.*

A simulation of an SLSF diagram produces a simulation trajectory, which is closely related to a trajectory of a hybrid automaton.

Definition 3 (Simulation trajectory). *For an initial state x_0, a time bound $\mathcal{T}_{max}$, error bound $\varepsilon > 0$, and time step $\tau > 0$, a* simulation trajectory *(of length k) is a sequence $((R_i, t_i))_{i=1\ldots k}$, where $R_0 = \{x_0\}, t_0 = 0, R_i \subseteq \mathbb{R}^n, t_i \in \mathbb{R}^{\geq 0}$, and (a) $\forall\, i : 0 \leq t_{i+1} - t_i \leq \tau, t_k = \mathcal{T}_{max}$, (b) $\forall\, i\ \forall\, t \in [t_i, t_{i+1}]$: the simulation state after time t is in R_i, and (c) $\forall\, i : dia(R_i) \leq \varepsilon$.*

By $\mathrm{Trac}(\mathcal{S})$ *we denote the set of all simulation trajectories of an* SLSF *diagram $\mathcal{S}$. Note that in practice, any simulation trajectory is finite-length, although we purposely avoid a finite-length assumption in the definition of simulation trajectories, so that we may relate possibly infinite trajectories of a hybrid automaton with similiar possibly infinite simulation trajectories.*

3 Translating a Hybrid Automaton to a Continuous-Time Stateflow Diagram

In this section, we describe our main contribution, namely how to translate from a hybrid automaton to an equivalent SLSF diagram. As already outlined in Sec. 1, one main difference is the absence of *non-determinism* in SLSF diagrams.

In the hybrid automaton formalism described above, there are several sources of non-determinism.

1. *Transition.* If there is more than one outgoing transition in a location, any of them can be taken as long as the guard is enabled and the target location's invariant is satisfied after applying the transition update.
2. *Dwell time.* The amount of time that a hybrid automaton remains in a location is only determined by the invariant and the transition guards – it is forced to leave the location *only* by the invariant. It is not sufficient for the guard to be enabled at *some* point in time, as the automaton can still choose to remain in the location until the invariant becomes false.
3. *Initial state.* A hybrid automaton is allowed to start in a whole region, which may be an uncountable number of possible initial states.
4. *Updates.* Updates (as parts of transitions) may be non-deterministic. This gives a (possibly uncountable) number of successor states after a discrete transition.
5. *Flow.* There is one more possible source for non-determinism, namely having uncertainties in the flow definition. However, we do not consider this case in this paper.

To compare simulation trajectories of an SLSF diagram with trajectories of a hybrid automaton, we introduce the concept of correspondence.

Definition 4 (Correspondence). *A trajectory of a hybrid automaton and a simulation trajectory in the constructed* SLSF *diagram* correspond *to each other if the sequence of discrete locations and transitions encountered is the same, and the difference between continuous points of the trajectory and the simulation trajectory in each location can be bounded by an arbitrarily small constant.*

The primary goal of our construction is to ensure that the set of simulation trajectories for the SLSF diagram can be *trajectory-equivalent* to the original hybrid automaton.

Definition 5 (Trajectory-Equivalence). *An* SLSF *diagram is* trajectory-equivalent *to a hybrid automaton $\mathcal{H}$ if, for every trajectory of the hybrid automaton, there exists a corresponding simulation trajectory of the* SLSF *diagram, and conversely, for every simulation trajectory of the* SLSF *diagram has a corresponding trajectory in the hybrid automaton.*

We achieve this by replacing non-determinism in the hybrid automaton by (uniformly distributed) random number generation in the SLSF diagram. In this way, by executing multiple SLSF simulations we can approximate the reachable states of the original hybrid automaton.

There may be additional numerical issues with SLSF that are outside the scope of this work. For example, the integration of the differential equations in SLSF may not be exact, which may cause differences in observed behavior. In practice, simulations are fairly accurate, and can be made arbitrarily more accurate by reducing the simulation time step.

Translation Overview. In our converter, we currently support initial regions and non-deterministic updates to hyper-rectangles, as well as deterministic updates which can be arbitrary functions. When non-deterministic assignments or initial states are used, they must be completely inside the invariant of the target location, which can be statically checked. The choice of the initial continuous state and the non-determinism possible during updates, therefore, can be done by randomly choosing one point from the set of all points available.

In the rest of this section, we focus on the harder problem of non-determinism from the transitions and dwell time. We first give an overview of the translation scheme. Here it is helpful to regard the trajectory of a hybrid automaton as a sequence of jumps, and after each jump, the automaton chooses the next transition and dwell time. The crucial difference in our conversion is that the choices might be infeasible, i.e., violating the invariant. To account for this, we incorporate a backtracking mechanism, where the current state of all variables is stored when entering a new location. Note that *time* is an entity which is implicitly present in all hybrid automaton models and we can always add a (fresh) time variable t with flow $\dot{t} = 1$. This allows for a general translation scheme without further knowledge about the hybrid automaton under consideration.

We translate a hybrid automaton location ℓ into a corresponding *location cluster* $\hat{\ell}$, comprising of a number of SLSF states, junctions, and transitions. The clusters are then connected by the same transitions as in the original hybrid automaton. A simulation trajectory of the resulting SLSF diagram then visits those clusters. Inside a cluster, the execution consists of three *phases*, depicted in Fig. 3.

Three phases in a location cluster. In the first phase, we *randomly* choose a transition *out* from the transitions currently available. In the second phase, we choose a time threshold $\mathcal{T}$. In the final phase, we incorporate the original continuous dynamics of the location ℓ.

In the translated model, the transition tries to be taken by checking the original guard condition, but only after dwelling in $\hat{\ell}$ for at least until time

Fig. 3: High-level location cluster translation pattern consisting of three phases. The location cluster $\hat{\ell}$ denotes a group of SLSF states and junctions which reflects the behavior of the hybrid automaton in the location ℓ.

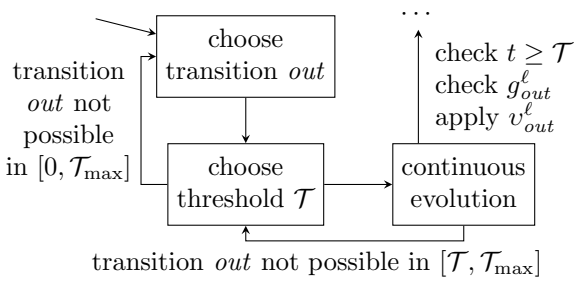
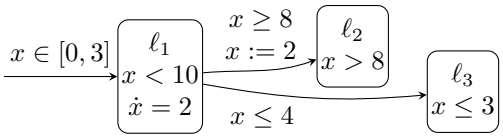


Fig. 4: Snippet of an example hybrid automaton with three locations $\ell_1 - \ell_3$.



moment $\mathcal{T}$. If the transition *out* cannot be taken – possibly due to an invariant violation – in the time frame $[\mathcal{T}, \mathcal{T}_{\max}]$, where $\mathcal{T}_{\max}$ is the maximum simulation time, we *backtrack*[1] and return to the second phase, and select a new time threshold $\mathcal{T}$ which is strictly less than the previously-chosen threshold. To ensure termination, we bound the number of times backtracking may occur before trying $\mathcal{T} = 0$. If the chosen transition can still not be taken, we can conclude that it cannot be taken at all, and go back to the first phase, this time trying another transition.

In the following, we illustrate the process using an example simulation.

Example. We consider an execution in some location cluster for a simple location ℓ_1 with one continuous variable x and two outgoing transitions, as depicted in Fig. 4. For simplicity, assume that the location is entered at time $t = 0$ in state $x = 0$ and the total simulation time is $\mathcal{T}_{\max} = 20$.

First we store the current continuous state $(t, x) = (0, 0)$. Next, in phase 1, we choose a transition, say, the one to ℓ_2. Then, in phase 2, we choose a random minimum dwell time in the range $[0, 20]$, say $\mathcal{T} = 3$. The simulation proceeds in phase 3 until an event occurs. In this case, events are either violating the location invariant $x < 10$ or enabling the guard condition of the selected transition $t \geq 3 \wedge x \geq 8$. The guard condition is enabled first, at state $(t, x) = (4, 8)$. This transition cannot be taken, however, as the target invariant would be violated after applying the update $x := 2$.[2] The simulation continues until the next event, when the state $(t, x) = (5, 10)$ is reached and a violation of the invariant is detected. That is why the simulation goes back to phase 2, backtracking to the saved state $(t, x) = (0, 0)$. At this point, it was checked that for all $\mathcal{T} \geq 3$, the transition cannot be taken. In phase 2, a new value for $\mathcal{T}$ is chosen from the restricted interval $[0, 3)$, and the simulation is run again in phase 3. After

[1] We note that our notion of backtracking is different from the one that occurs with multiple junctions in SLSF. In particular, we require allowing some dwell time to elapse in states, whereas junctions are instantaneous.

[2] In fact, it is impossible to take this transition at all.

reaching the same conclusion and after further backtracking, a finite threshold of attempts is reached, and $\mathcal{T} = 0$ is forced. Even with $\mathcal{T} = 0$ there will be a violation of the invariant before the transition can be taken. Then, we will conclude that the selected transition can never be taken when starting in the state $(t, x) = (0, 0)$. Thus we can safely ignore this transition, go back to phase 1 and choose the transition leading to ℓ_3, where the process repeats.

Other Practical Issues. In practice there are some additional concerns since the Stateflow simulator is not infinitely precise. If a guard is only enabled at one (singular) point in time, this cannot be detected by the zero-crossing mechanisms used by SLSF, and the transition is usually missed. In order to not exclude certain behaviors systematically, we consider an ε-relaxation of each guard constraint.[3] The simulation time step can then be chosen small enough such that, based on the value of ε and the Lipschitz constant of the dynamics, no transitions will be missed.

Although this may permit more behaviors than the original hybrid automaton, it critically prevents transitions from being missed, which is necessary for trajectory-equivalence. The extra behaviors introduced from this necessary step can be reduced by considering smaller values of ε, which will require a smaller simulation time step. Reducing the time step, however, will be at the cost of additional simulation runtime.

4 Translation Correctness and Discussion

Non-determinism. By replacing non-determinism with random number generation, some behaviors of the original hybrid automaton might be obscured. For instance, a non-deterministic die can roll a six forever, while the probability of this behavior for a random die approaches zero as more rolls are taken. We always deal with finite executions in a simulation, and thus end up with a finite number of choices, so there is still a nonzero chance that the 'right' random values will be chosen.[4]

Generalizations. Although we consider a large class of hybrid automata, further generalizations are possible. For example, the initial sets and non-deterministic resets in our framework were hyper-rectangles, whereas in general the initial state could be in a non-convex set, and the reset might be an arbitrary function which maps from a single state to a non-convex set. To handle such systems, we need a way to sample in the non-convex destination sets, which may be possible in certain situations, but is difficult in general. One possibility would be to require the user to give this sampling function.

Another generalization possible is to consider non-deterministic dynamics. More general hybrid automata may include differential inclusions or other non-deterministic ways for the continuous states to evolve. This could be handled by

[3] For instance, a guard constraint of the form $x = c \wedge y \leq x$ becomes $c - \varepsilon \leq x \leq c + \varepsilon \wedge y \leq x - \varepsilon$.

[4] We assume that the hybrid automaton is Zeno-free.

adding ranged inputs to the system, and at each time step choosing a random value in the range for each input. However, as the time steps become smaller, the random inputs will approximate the main value in their ranges, which in practice results in poor simulation coverage. An alternative is to choose a time step where the inputs will vary, such that a trade-off is possible between the amount of coverage possible, and the effect of this tendency towards the mean. Other simulation methods, perhaps based on state exploration mechanisms such as rapidly-exploring random trees (RRTs) [22] may also be possible.

Trajectory-Equivalence. The conversion process described above maintains the defined notion of trajectory-equivalence. For this, we consider an idealized conversion, where there are no numerical errors in the simulation, the value of ε is zero, and the SLSF diagram encodes the intended semantics of the described transformation process.

Theorem 1. *If $\mathcal{H}$ is a Zeno-free hybrid automaton and $\mathcal{S}$ is the* SLSF *diagram encoded using our transformation process, then $\mathcal{S}$ is trajectory-equivalent to $\mathcal{H}$.*

Proof. We first show the forward direction, i.e., given an arbitrary trajectory of the hybrid automaton, there exists a set of random decisions in the constructed SLSF diagram that produce a corresponding simulation (trajectory).

Recall that correspondence requires that the encountered locations can be the same, and that the deviation in continuous states can be bounded by an arbitrarily small constant.

For the ordering of locations, notice that the random choice of an outgoing transition in phase 1 of the construction can pick the corresponding transition from the trajectory. Since the minimum dwell time is chosen randomly, it can be picked to be arbitrarily close to the dwell time in the hybrid automaton trajectory. In this way, as long as the continuous evolution in the simulation remains close to the hybrid automaton trajectory's continuous evolution, every transition will be explored.

The second part of correspondence requires that the deviation in the continuous states is bounded. We show that this bound can be chosen to be arbitrarily small across both every continuous evolution and after every discrete transition. During a continuous evolution, if the start state in a location in the simulation is chosen close to the start state in the corresponding location in the hybrid automaton trajectory, its deviation will also be bounded as a function of the Lipschitz constant (see Proposition 1 in [14]). Thus, for a single bounded continuous evolution and every nonzero final state deviation desired, there is a corresponding nonzero initial state deviation that will achieve the desired closeness.

During initial state selection, since we consider hyper-rectangles, the set of states is bounded. Randomly choosing states, we will in finite time pick one arbitrarily close to any trajectory's start state in the hybrid automaton.

Finally, for updates, the dwell time of a simulation can be made arbitrarily close to a hybrid automaton trajectory, and since the state can be made arbitrarily close, a deterministic update function (under assumptions of Lipschitz continuity) can also result in a state arbitrarily close to the trajectory. For

nondeterministic updates, the argument is similar to the initial state selection, and thus the continuous states of the simulation remain arbitrarily close to the hybrid automaton trajectory.

The sequence of discrete transitions between the trajectory and simulation match. Since each trajectory is a finite sequence of discrete transitions (due to Zeno-free behavior) and continuous evolutions (each of which can have arbitrarily small error between the trajectory and a possible simulation), the accumulated error for the whole trajectory can also be made arbitrarily small. Thus, the constructed SLSF diagram has simulations which correspond to any arbitrary hybrid automaton trajectory.

The reverse direction in the proof attempts to show that any arbitrary simulation has a corresponding hybrid automaton trajectory. Again, we proceed by decomposing this into showing that the sequence of locations is the same, and that the deviation in the continuous state is bounded.

Since we assumed an idealized relaxation where ε is zero, every transition in the simulation exactly matches the guard conditions in the hybrid automaton, and thus the hybrid automaton can match the simulation. Every update in the constructed SLSF diagram is also copied from the automaton, so that the automaton's trajectory can match the random choices made by a simulation.

For continuous trajectories, the simulation will choose some dwell time where the invariant remains satisfied until a guard becomes true. The hybrid automaton can also pick the same dwell time, and its invariant will also remain true until the same guard condition is reached. Thus, the hybrid automaton can pick a trajectory which corresponds to the simulation.

Since every trajectory of the hybrid automaton corresponds to a simulation trajectory of the SLSF diagram, and every simulation trajectory corresponds to a trajectory, the two models are trajectory-equivalent. □

This proof required three assumptions, mentioned before the theorem. First, we assumed the simulations were exactly accurate. Although real simulations will always have some error, this can be reduced to arbitrarily small values by reducing the time step used in the simulation. Similarly, for the second assumption we can consider smaller and smaller values of ε, although in degenerate cases this might permit extra transitions in the simulation. For example, a degenerate guard like $x < 5 \wedge x > 5$ will always be false, but any positive ε-relaxation will have a possible transition when $5 - \varepsilon < x < 5 + \varepsilon$. The third assumption is that the SLSF diagram correctly encodes the described transformation process. This means that correctness is subject to possible implementation bugs in our conversion implementation in HyST, as well as the semantics of Stateflow. In addition to the trajectory-equivalence theorem, we provide empirical justification for the correctness of the implementation of our translation scheme, through extensive case studies as presented in the next section.

5 Evaluation and Experimental Results

To evaluate the translation methodology presented in this paper, we implemented a prototype translator that uses the HyST intermediate representation

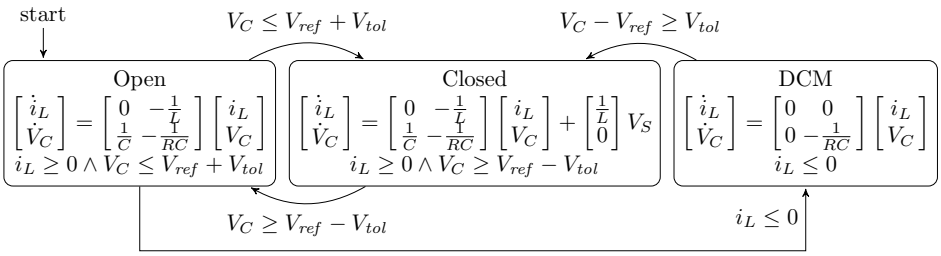


Fig. 5: Hybrid automaton model of the buck converter plant with hysteresis controller.

for source-to-source transformation of hybrid automata [4], and the SLSF API within Matlab (tested with versions 2014a through 2016a). The input to the translator is a hybrid automaton $\mathcal{H}$ in the SpaceEx XML format. Networks of hybrid automata are first composed within HyST to yield a single hybrid automaton representing the network. Once parsed in the tool, an object representing the syntactic structure of $\mathcal{H}$ is traversed, and then the tool applies the sequence of translation steps described in Sec. 3. In the simulator, we varied the seeds of the uniform pseudo-random number generator `rng` in Matlab. We evaluated the prototype tool using several examples. For this we first computed the reachable states of the models in SpaceEx or Flow*, then performed the translation and simulations in SLSF. The tool and examples are available for download.[5]

5.1 Buck Converter with Hysteresis Controller

A buck converter is a DC-to-DC switched-mode power supply that takes a DC input source voltage and lowers ("bucks") it to a smaller DC output voltage [24]. A standard model of the converter has three modes, where: the switch is closed and the voltage source is connected, the switch is open and the voltage source is disconnected, and based on the possible dynamics of the converter, a third mode, known as the discontinuous conduction mode (DCM), where the current is not allowed to go below zero (which is physically unrealizable, but may occur without a third mode). Interested readers may find detailed derivations of models in power electronics textbooks [32]. A hybrid automaton model of the closed-loop buck converter (plant and controller) appears in Fig. 5.

A standard closed-loop controller for the buck converter is a hysteresis controller, which changes the mode of the buck converter plant based on the measured output voltage. Its operation depends upon opening and closing of the MOSFET switch. Intuitively, it operates like a thermostat, i.e., the switch is toggled so that the source voltage is connected to the circuit if the output voltage is too low, and it is toggled in case if the output voltage is too high to disconnect the voltage source. We note that by Kirchhoff's voltage law (KVL),

[5] The tool and examples described in this paper are available here: `http://swt.informatik.uni-freiburg.de/tool/spaceex/ha2slsf`
More details on the HyST framework are here: `http://verivital.uta.edu/hyst/`

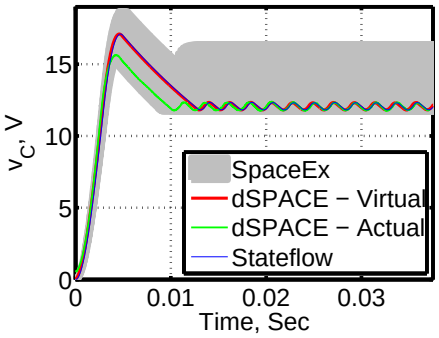


Fig. 6: Reachable states of the hybrid automaton computed with SpaceEx, verifying the voltage-regulation property, along with HiL simulation results of the translated SLSF diagram on the DS1103 ("virtual plant"), and control of the physical plant with the translated SLSF diagram ("actual plant"). Our results validate the high-level vision of correct-by-construction control implementation from Fig. 1.

$V_C = V_{out}$ [32]. In part to avoid switching too frequently, a hysteresis band is typically used so switches occur when $V_{out} \geq V_{ref} + V_{tol}$ or $V_{out} \leq V_{ref} - V_{tol}$. This creates a voltage ripple on the output voltage that should be within a given range V_{rip} of the desired reference output voltage V_{ref}. Together, these define a safety property for the buck converter: $\phi(t) \triangleq t \geq t_s \Rightarrow V_{out}(t) = V_{ref} \pm V_{rip}$, which projected onto the phase space is $\phi \triangleq V_{ref} - V_{rip} \leq V_{out} \leq V_{ref} + V_{rip}$. SpaceEx is used to verify ϕ by computing the reachable states Reach($\mathcal{H}$) (to a fixed-point) from a startup state where the initial states $\mathcal{S}_{init}$ are $i_L = 0$ and $V_C = 0$. For every time $t \geq t_s$ after a startup trajectory of duration t_s, if $V_{ref} - V_{rip} \leq V_{out}(t) \leq V_{ref} + V_{rip}$, then the converter satisfies the specification ϕ. The reachable states of the closed-loop buck converter hybrid automaton are computed with SpaceEx, and as shown in Fig. 6, it meets the specification for a sufficient choice of V_{rip}. For unbounded-time verification, we compute the reachable states to a fixed-point in SpaceEx without any timing variables.

A hardware setup consisting of a buck converter plant and a dSpace DS1103 is used to perform the experiments with the physical plant. The DS1103 contains a Power PC processor and a DSP board and is used for implementation of the hybrid automata in both hardware-in-the-loop (HiL) simulations with a "virtual plant" (the plant model simulated on the DS1103 hardware) and the actual buck converter plant.

The hysteresis controller is implemented on the DS1103, such that C code is generated from the translated SLSF diagram in Matlab onto the DS1103. A discrete fixed-step solver with a time step of 20 microseconds is used for the code generation process and also for the DS1103's sampling and control periods, which is sufficiently small to ensure ε is sufficiently small, as discussed in Sec. 3 and Sec. 4. The embedded controller generates suitably spaced rectangular pulses to operate the MOSFET switch of the prototype buck converter. For the experiments with the actual plant, the input signals fed to the controller (specifically the V_C voltage) are replaced from the simulation model with the measurement of the actual plant, and the output signals (the desired mode, open or closed) are fed to the actual plant instead of the simulation model. The experimental results are recorded and a comparison to SLSF simulations is shown in Fig. 6. The experimental and simulation traces are contained in the SpaceEx reach sets, which further validates the translation correctness and that the safety property

is maintained in the implementation. Note that in the hardware experiments, the controller has essentially been determinized, as the non-determinism in the hybrid automaton model was to model plant inaccuracies.

6 Conclusion

In this paper, we presented a trajectory-equivalent transformation of a hybrid automaton into a continuous-time SLSF diagram, and described its implementation in a prototype software tool. For non-deterministic models, our approach adds auxiliary randomization for various sources of non-determinism to mimic the semantics of hybrid automata. We have empirically validated our approach on a number of challenging benchmarks.

For the future, it will be interesting to further refine and extend our approach by, e.g., considering the translation of *networks* of hybrid automata—directly without first composing them—into SLSF diagrams and exploring further sources of non-determinism such as non-deterministic flows. Another gainful direction would be to make the distribution over all possible executions uniform. A possible way is to have a preliminary run to detect *all* time intervals when a transition can be taken. Then, after backtracking exactly once, the dwell time can be chosen uniformly over all those intervals. Our approach, in contrast, is based on the assumption that in the majority of cases no backtracking is needed. In a further step, a focus on rare events in the line of [13] could be added.

References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. ENTCS 109, 43–56 (2004)
2. Agut, D.N., van Beek, D., Rooda, J.: Syntax and semantics of the compositional interchange format for hybrid systems. The Journal of Logic and Algebraic Programming 82(1), 1 – 52 (2013)
3. Alur, R., Kanade, A., Ramesh, S., Shashidhar, K.C.: Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In: Proceedings of the 8th ACM International Conference on Embedded Software. pp. 89–98. ACM, New York, NY, USA (2008)
4. Bak, S., Bogomolov, S., Johnson, T.T.: HyST: A source transformation and translation tool for hybrid automaton models. In: Proc. of the 18th Intl. Conf. on Hybrid Systems: Computation and Control (HSCC). ACM (2015)
5. Bak, S., Johnson, T.T.: Periodically-scheduled controller analysis using hybrid systems reachability and continuization. In: Real-Time and Embedded Technology and Applications Symposium, IEEE. IEEE Computer Society (2015)
6. Balasubramanian, D., Păsăreanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.: Polyglot: Modeling and analysis for multiple statechart formalisms. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 45–55. ISSTA '11, ACM, New York, NY, USA (2011)
7. Bogomolov, S., Donzé, A., Frehse, G., Grosu, R., Johnson, T.T., Ladan, H., Podelski, A., Wehrle, M.: Guided search for hybrid systems based on coarse-grained space abstractions. International Journal on Software Tools for Technology Transfer pp. 1–19 (2015)
8. Bogomolov, S., Frehse, G., Greitschus, M., Grosu, R., Pasareanu, C.S., Podelski, A., Strump, T.: Assume-guarantee abstraction refinement meets hybrid systems. In: Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. pp. 116–131. LNCS, Springer (2014)
9. Bouissou, O., Chapoutot, A.: An operational semantics for Simulink's simulation engine. In: Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems. pp. 129–138. LCTES '12, ACM, New York, NY, USA (2012)

10. Carloni, L., di Benedetto, M.D., Pinto, A., Sangiovanni-Vincentelli, A.: Modeling techniques, programming languages, design toolsets and interchange formats for hybrid systems. Tech. rep. (2004)
11. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and tools for hybrid systems design. Foundations and Trends in Electronic Design Automation 1 (2006)
12. Chen, X., Abraham, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification, LNCS, vol. 8044, pp. 258–263. Springer Berlin Heidelberg (2013)
13. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Bultan, T., Hsiung, P.A. (eds.) Automated Technology for Verification and Analysis, LNCS, vol. 6996, pp. 1–12. Springer (2011)
14. Duggirala, P.S., Mitra, S., Viswanathan, M.: Verification of annotated models from executions. In: Proceedings of the Eleventh ACM International Conference on Embedded Software. EMSOFT '13, IEEE Press, Piscataway, NJ, USA (2013)
15. Fisher, M.E.: A semiclosed-loop algorithm for the control of blood glucose levels in diabetics. Biomedical Engineering, IEEE Transactions on 38(1), 57–61 (1991)
16. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Computer Aided Verification. pp. 379–395 (2011)
17. Gao, S., Avigad, J., Clarke, E.M.: δ-complete decision procedures for satisfiability over the reals. In: Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings. pp. 286–300 (2012)
18. Hamon, G.: A denotational semantics for Stateflow. In: Proceedings of the 5th ACM International Conference on Embedded Software. pp. 164–172. EMSOFT '05, ACM, New York, NY, USA (2005)
19. Hamon, G., Rushby, J.: An operational semantics for Stateflow. International Journal on Software Tools for Technology Transfer 9(5-6), 447–456 (2007)
20. Jiang, Z., Pajic, M., Alur, R., Mangharam, R.: Closed-loop verification of medical devices with model abstraction and refinement. International Journal on Software Tools for Technology Transfer 16(2), 191–213 (2014)
21. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
22. Lavalle, S.M., Kuffner, J.J., Jr.: Rapidly-exploring random trees: Progress and prospects. In: Algorithmic and Computational Robotics: New Directions. pp. 293–308 (2000)
23. Manamcheri, K., Mitra, S., Bak, S., Caccamo, M.: A step towards verification and synthesis from Simulink/Stateflow models. In: Proc. of the 14th Intl. Conf. on Hybrid Systems: Computation and Control (HSCC). pp. 317–318. ACM (2011)
24. Nguyen, L.V., Johnson, T.T.: Benchmark: DC-to-DC switched-mode power converters (buck converters, boost converters, and buck-boost converters). In: Applied Verification for Continuous and Hybrid Systems Workshop (ARCH 2014). Berlin, Germany (Apr 2014)
25. Pajic, M., Mangharam, R., Sokolsky, O., Arney, D., Goldman, J., Lee, I.: Model-driven safety analysis of closed-loop medical systems. IEEE Transactions on Industrial Informatics 10(1), 3–16 (2014)
26. Pajic, M., Jiang, Z., Lee, I., Sokolsky, O., Mangharam, R.: From verification to implementation: A model translation tool and a pacemaker case study. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th. pp. 173–184. IEEE (2012)
27. Pajic, M., Jiang, Z., Lee, I., Sokolsky, O., Mangharam, R.: Safety-critical medical device development using the UPP2SF model translation tool. ACM Trans. Embed. Comput. Syst. 13(4s), 127:1–127:26 (Apr 2014)
28. Pinto, A., Carloni, L., Passerone, R., Sangiovanni-Vincentelli, A.: Interchange format for hybrid systems: Abstract semantics. In: Hespanha, J.P., Tiwari, A. (eds.) Hybrid Systems: Computation and Control, vol. 3927, pp. 491–506. Springer Berlin Heidelberg (2006)
29. Pinto, A., Sangiovanni-Vincentelli, A.L., Carloni, L.P., Passerone, R.: Interchange formats for hybrid systems: Review and proposal. In: Morari, M., Thiele, L. (eds.) Hybrid Systems: Computation and Control, LNCS, vol. 3414, pp. 526–541. Springer (2005)
30. Sampath, P., Rajeev, A.C., Ramesh, S.: Translation validation for Stateflow to C. In: Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference. pp. 23:1–23:6. DAC '14, ACM, New York, NY, USA (2014)
31. Schrammel, P., Jeannet, B.: From hybrid data-flow languages to hybrid automata: A complete translation. In: Proceedings of the 15th ACM International Conference on Hybrid Systems: Computation and Control. pp. 167–176. ACM, New York, NY, USA (2012)
32. Severns, R.P., Bloom, G.: Modern DC-to-DC Switchmode Power Converter Circuits. Van Nostrand Reinhold Company, New York, New York (1985)
33. Tiwari, A., Shankar, N., Rushby, J.: Invisible formal methods for embedded control systems. Proceedings of the IEEE 91(1), 29–39 (Jan 2003)

A Appendix

A.1 Additional Conversion Details

We provide a detailed description of our translation. It iteratively converts every location ℓ of a hybrid automaton and its outgoing transitions into an SLSF diagram in the following way (see Fig. 7). We first describe the data structures we use in our construction. The list *outList* stores the ordering in which the outgoing transitions of the location ℓ are considered in the simulation. The variable *out* keeps track of the currently chosen outgoing transition. The variable $\mathcal{T}_v$ stores the first time moment when the location invariant is violated. $\mathcal{T}_{\max}$ keeps the maximum simulation time, i.e., the simulation is stopped as soon as this bound has been reached. The variable $\mathcal{T}$ stores the time threshold after which the outgoing transition can be taken. The variable R keeps the maximum number of backtrackings we want to allow, whereas r stores the current number of backtrackings in the location cluster $\hat{\ell}$. Finally, t stores the current simulation time.

We continue with the description of every individual state in our construction. The current simulation time and the hybrid automaton state is stored in the (SLSF) state ℓ_{in}. Furthermore, the algorithm *randomly* chooses the ordering in which the outgoing transitions are considered. In this way, we handle the non-determinism due to multiple simultaneously enabled transition guards. Finally, the variable $\mathcal{T}_v$ is initialized to $\mathcal{T}_{\max}$ as we do not have any information about the invariant violation at that moment.

The state ℓ_{choose} covers two kinds of non-determinism. It takes care of the situation when the intersection of the invariant and the transition guard is non-singular, i.e., when a switch to the next location can happen not only at a particular time moment, but within a *time interval*. Note that if the continuous dynamics are non-monotonic, there can be multiple *disjoint* time intervals where the guard is enabled. We resolve such situations by generating a *random* time threshold $\mathcal{T}$ in the state ℓ_{choose} and allowing the discrete transition only from the time moment $\mathcal{T}$ onward, i.e., we add a constraint of the form $t \geq \mathcal{T}$ as a part of the transition guard for every outgoing transition from the location ℓ. Thus, we disable the must-semantics up until time moment $\mathcal{T}$. As we randomly *vary* the threshold $\mathcal{T}$ in every simulation, we incorporate the switching time moments due to the original may-semantics of a hybrid automaton.

Note that we also use the state ℓ_{choose} for *backtracking* purposes. We observe that an unfortunate choice of the outgoing transition *out* and the time threshold $\mathcal{T}$ can lead to the simulation getting stuck, as the transition guard of *out* is not enabled in the time frame $[\mathcal{T}, \mathcal{T}_{\max}]$ and thus the transition cannot be taken. In such cases, we return to the state ℓ_{choose} to select a further time threshold $\mathcal{T}$. For this purpose, we restore the simulation time t and the state of the hybrid automaton from the moment we entered $\hat{\ell}$. Afterward, we can choose the next time threshold from the interval $[t, \mathcal{T}]$. Here, we observe that in general before reaching the time threshold, the invariant can be violated. Thus, we actually select a new threshold from the interval $[t, \min(\mathcal{T}, \mathcal{T}_v)]$. In this way, we end

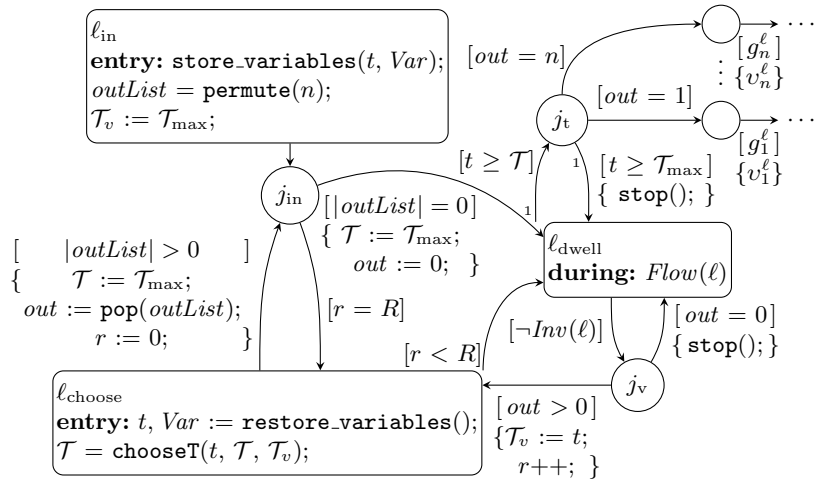
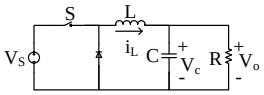


Fig. 7: General location cluster of some location ℓ with n outgoing transitions. `(re-)store_variables` stores and restores the current simulation state (including the time variable t) from when entering the cluster, respectively. `permute`(n) returns a permuted list *outList* with all integers from 1 to n. `pop`(*outList*) removes and returns the first element from *outList*. `chooseT` chooses a new time threshold $\mathcal{T}$. A subscript "1" indicates that a transition has the highest priority among all the outgoing transitions from a state/junction.

up with a sequence of monotonically decreasing thresholds. Still, as it is not guaranteed that the chosen threshold is eventually equal to 0, we add a further termination criterion by bounding the number of backtracking by some user-defined constant $R > 0$. The last time before exceeding this limit, we try out the weakest threshold $\mathcal{T} = 0$ to ensure that we have covered all cases. If the transition cannot be taken at all, we either proceed with a further outgoing transition (junction j_{in}) or, if none is left, the simulation is stopped and reports an actual deadlock in the model.

The continuous evolution corresponding to the location ℓ is modeled by the state ℓ_{dwell}. We can leave this state due to two conditions. First, the invariant can be violated. Then we store the time moment when the violation has happened in the variable $\mathcal{T}_v$ and move to the state ℓ_{choose} (junction j_{v}). Note that if we have already considered all the outgoing transitions of ℓ, we will stop the simulation, since a deadlock has been found. In the other case, the time threshold $\mathcal{T}$ can be reached. We move to the successor location of ℓ if the guard of the chosen transition *out* is enabled (junction j_{t}). Furthermore, here we also check whether the maximum simulation time $\mathcal{T}_{\max}$ has been reached, in which case we stop the simulation.



(a)

Component / Parameter Name	Symbol	Value
Source Input Voltage	V_S	24 V
Desired Output (Reference) Voltage	V_{ref}	12 V
Actual Output Voltage	$V_C = V_{out}$	12 V $\pm$ V_{rip}
Hysteresis Band Tolerance	V_{tol}	0.1 V
Voltage Ripple Tolerance	V_{rip}	0.6 V
Load Resistance	R	10 Ω
Capacitor	C	2.2 mF
Inductor	L	2.65 mH

(b)

Fig. 8: (a) Buck converter circuit—a DC input V_S is decreased to a lower DC output $V_C = V_o = V_{out}$. (b) Buck converter parameter values and variations.

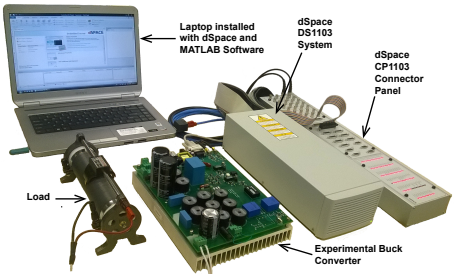


Fig. 9: The buck converter plant controlled with a dSPACE DS1103 system. Our results controlling the actual plant with the translated controller validate the high-level vision of correct-by-construction control implementation from Fig. 1.

A.2 Case Study: Buck Converter Additional Details

The buck converter circuit appears in Fig. 8(a). Parameter values used for the case study appear in Figure 8(b).

A hybrid automaton model of the buck converter plant and a hysteresis controller as a network of hybrid automata appears in Fig. 10, where θ is a synchronization label and δ is a discrete control signal, and a bisimilar hybrid automaton model after flattening (composing) the network was shown earlier in Fig. 5. The composed model from Fig. 5 is used for verification, translation, and code generation purposes as discussed earlier, while the network model is conceptually simpler and illustrates the decomposition between the physical plant hardware and the controller. The physical hardware used in the evaluation appears in Fig. 9.

Fig. 12 shows the reachable states in the phase space, and illustrates that the SLSF simulations are contained in the reachable states computed with SpaceEx and gives empirical evidence for the correctness of the translation.

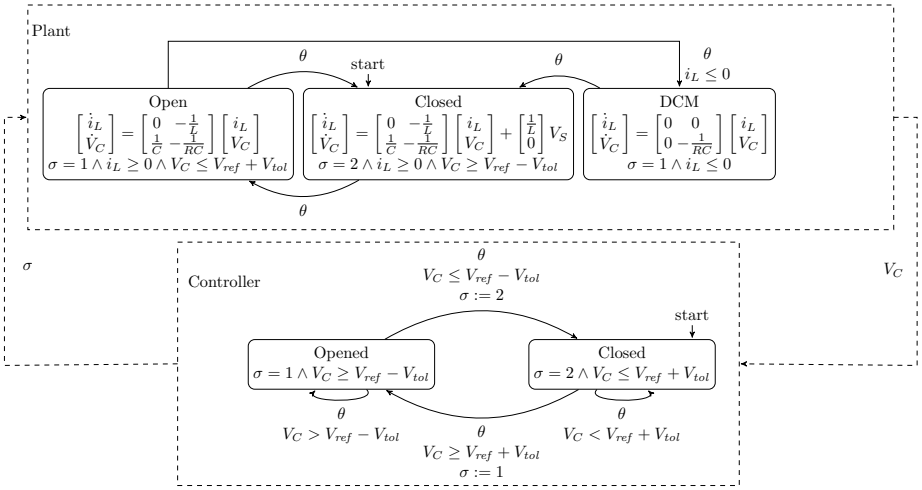


Fig. 10: Hybrid automaton models of the buck converter plant with hysteresis controller.

A.3 Case Study: Yaw Damper Controller for 747 Aircraft

A yaw damper is a multiple-input multiple-output (MIMO) system which uses the aileron and rudder in order to reduce oscillations in the yaw and roll angle of an aircraft. In this section, we use the proposed method to analyze the control design of a yaw damper for a 747 aircraft, taken from the Control Systems Toolbox case studies in Matlab.

In particular, we analyze the final designed controller, which includes a washout filter capable of eliminating oscillations, but maintaining the spiral mode. The spiral mode is a desired control characteristic in yaw damper systems, where an impulse input from the aileron will result in a bank angle which does not immediately decrease to zero.

The model for the system is given at Mach 0.8 at 40,000 ft using standard linear time-invariant dynamics, $\dot{x} = Ax + Bu$. There are four physical variables in the system $x = (x_1, x_2, x_3, x_4)^T$, which are sideslip angle (x_1), yaw rate (x_2), roll rate (x_3), and bank angle (x_4), represented by the column vector x. The two inputs $u = (u_1, u_2)^T$, are the rudder (u_1) and aileron (u_2). The outputs are the yaw rate and bank angle.

The specific values for A and B are:

$$A = \begin{bmatrix} -0.0558 & -.9968 & 0.0802 & 0.0415 \\ 0.598 & -0.115 & -0.0318 & 0 \\ -3.05 & 0.388 & -0.4650 & 0 \\ 0 & 0.0805 & 1 & 0 \end{bmatrix}, B = \begin{bmatrix} .00729 & 0 \\ -0.475 & 0.00775 \\ 0.153 & 0.143 \\ 0 & 0 \end{bmatrix}$$

This physical system is put into a feedback loop with a washout filter, which has a single variable w and dynamics $\dot{w} = x_2 - 0.2 \cdot w$. The filter variable is

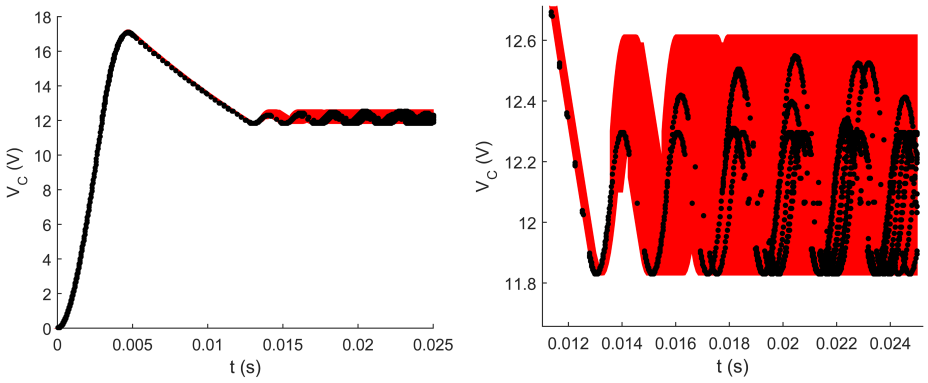


Fig. 11: Left: Buck converter V_C versus time, with SpaceEx reach set for the hybrid automatom model in red, and black points from 10 simulation traces of the translated SLSF diagram. Right: Detailed and zoomed view illustrating multiple simulation trajectories.

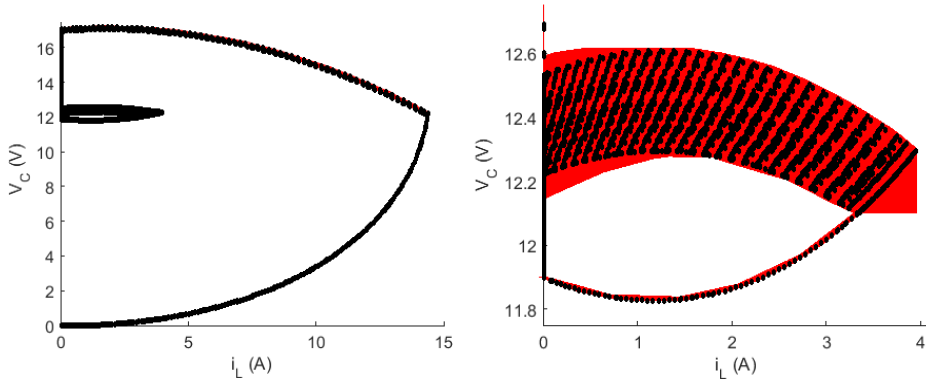


Fig. 12: Left: Buck converter V_C versus i_L (phase space), with SpaceEx reach set in red, and black points from 100 simulation traces. Right: Detailed and zoomed view illustrating multiple simulation trajectories.

combined with the yaw to produce an effect on the rudder input. In particular, the washout filter adds to u_1 the value $2.34 \cdot (x_2 - 0.2 \cdot w)$.

We consider analysis of a system model which has the guarantees given by a real-time scheduler, which periodically executes the washout filter and sets the output values. Between controller executions we take the output of the washout filter to be constant (zero-order hold). The control task is guaranteed to execute every period using a common scheduler like Rate Monotonic (RM) or Earliest Deadline First (EDF). There is non-determinism in the exact time the controller runs, however, due to the offset of the execution of the control task within each period. Since the control logic is simple, we take the control task to be nonpreemptive and short, so that the model will sample the physical system and update the filter output at a single point in time, but that point in time may

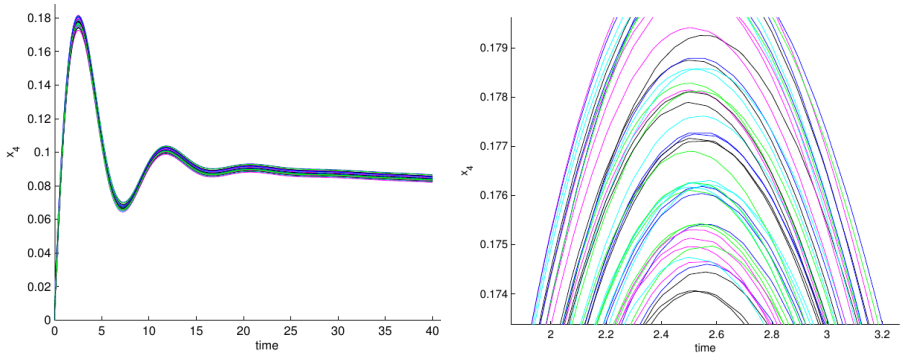


Fig. 13: 50 simulations of the yaw damper system. Left: The spiral mode is confirmed. Right: Non-determinism in controller execution time causes simulated trajectories to cross.

vary within each period. Furthermore, we look at the system response due to an impulse input from the aileron from a range of start conditions. We take the initial bank angle to be between 0 and 0.1.

This system was modeled in SpaceEx, and reachability analysis was attempted in both SpaceEx and Flow*. Due to the large number of discrete switches, however, neither tool is able to directly compute reachability (the computed reach sets grow exponentially).

Instead, we investigate the system using our conversion to SLSF and randomized execution. Since the main source of non-determinism in this model is the discrete switches, we can investigate simulations of the system where they occur at varying offsets from the start of each period.

The simulations showed the expected response of the system when using a controller period of $T = 0.1$. The response of the system is shown in Fig. 13. Here, the impulse response from the aileron to the bank angle is plotted, which does not immediately converge (spiral mode), and does not contain excessive oscillations. Thus, using the technique proposed in this paper we are able to analyze a system which cannot be directly analyzed using reachability tools.

This system can be analyzed formally, however this requires a non-trivial model transformation using the technique of continuization, as well as using a smaller control period. Continuization converts the periodically-actuated model into a continuous one with bounded noise, where the bound is based on the controller period and maximum rate of change of the output signal [5]. The same model can be used as the basis for the conversion using continuization, as well as the conversion to SLSF for simulation and further Matlab-based analysis and code generation. In this way, the conversion to SLSF is one part of a larger toolflow, where models are first created in SpaceEx, possibly converted for formal analysis using HyST, and then can be directly imported into SLSF after the conversion described in this paper for simulation and controller synthesis, as well as embedding in a larger CPS model.

A.4 Case Study: Glycemic Control in Diabetics

Glycemic control is an approach to control the blood glucose levels in insulin dependent diabetes mellitus patients. There are several different mathematical models of glycemic control used to design insulin infusion devices that help diabetic patients control their blood glucose levels [15]. Here we investigate a nonlinear hybrid system of the glycemic control in diabetic patients such that all dynamics are defined by polynomials. The mathematical model is described by the following ODEs:

$$\dot{G} = -0.01G - X(G + G_B) + g(t) \tag{1}$$

$$\dot{X} = -0.025X + 0.000013I \tag{2}$$

$$\dot{I} = -0.093(I + I_B) + u(t)/12 \tag{3}$$

In Equation 1 and Equation 3, G and I are the plasma glucose concentration and the plasma insulin concentration above their basal value G_B and I_B, which are equal to 4.5 and 15, respectively. The variable X shown in Equation 2 is the insulin concentration in an interstitial chamber. Moreover, $g(t)$ and $u(t)$ are the influx of glucose and the insulin control input, presented in Equation 4 and Equation 5, respectively.

$$g(t) = \begin{cases} t/60 & \text{if } t \leq 30 \\ (120 - t)/180 & \text{if } 30 < t \leq 120 \\ 0 & \text{if } t > 120 \end{cases} \tag{4}$$

$$u(t) = \begin{cases} 25/3 & \text{if } G(t) \leq 4 \\ 25/3(G(t) - 3) & \text{if } 4 < G(t) \leq 8 \\ 125/3 & \text{if } G(t) > 8 \end{cases} \tag{5}$$

The glycemic control was first modeled in SpaceEx and then translated to Flow* by using the HyST model converter. This model is nonlinear, non-deterministic, and includes 4 variables, 9 locations and 18 discrete transitions in total. The simulations of the glycemic control model translated to SLSF are shown in Fig. 14. We simulated the translated model with 100 different randomized executions. All simulation traces of G are contained in the reach set computed by Flow*, which validates the translation.

A.5 Case Study: Fischer Mutual Exclusion

Fischer mutual exclusion is a timed distributed algorithm that ensures a mutual exclusion safety property, namely that at most one process in a network of N processes may enter a critical section simultaneously. An automaton for Fischer appears in Fig. 15. Fischer involves two real timing parameters, A and

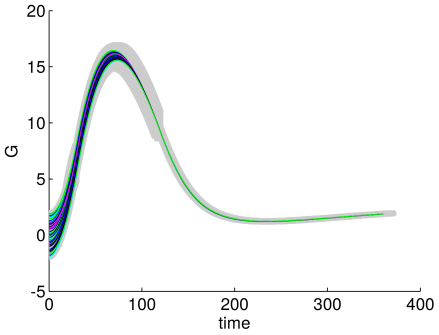


Fig. 14: 100 simulations of the glycemic control model with simulations and reach set computed by Flow* (gray) for variable G.

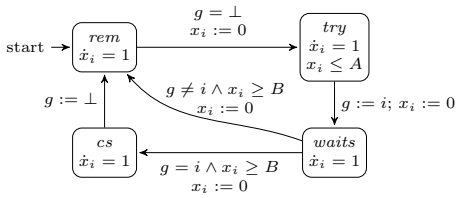


Fig. 15: Fischer's mutual exclusion algorithm for a process with identifier $i \in \{1, \ldots, N\}$. Here, g is a global variable of type $\{\bot, 1, \ldots, N\}$, x_i is a local variable of type $\mathbb{R}$, and both A and B are constants of type $\mathbb{R}$.

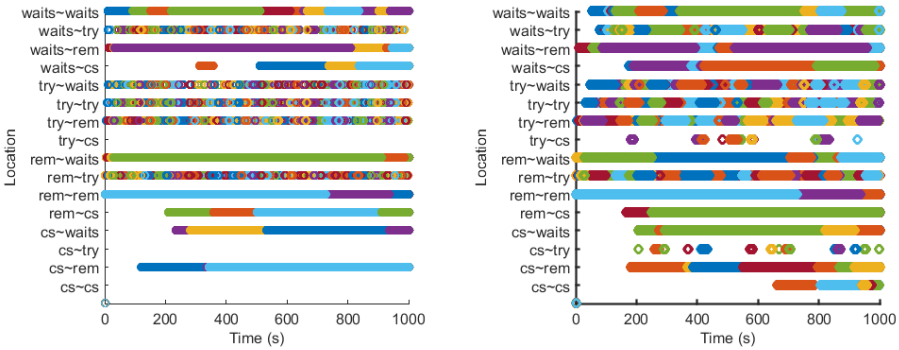


Fig. 16: Locations reached for 1000 SLSF simulations of Fischer, where different colors indicate different trajectories. Left: safe case ($A < B$). Right: unsafe case ($A > B$).

B, and mutual exclusion is ensured iff $A < B$. Let $Loc \triangleq \{rem, try, waits, cs\}$. We translated a network of two automata ($N = 2$) from SpaceEx to SLSF. In one instance, we ensured $A < B$ picking $A = 5$ and $B = 70$, so mutual exclusion was maintained, which we verified in SpaceEx using the PHAVer scenario. In the other instance, we ensured $A > B$ by picking $A = 75$ and $B = 70$, and mutual exclusion was not maintained. Consequently, we could not verify this instance using SpaceEx's PHAVer scenario since a location $cs \sim cs$ was reachable, corresponding to the case where both processes are in the critical section. We conducted $K = 1000$ simulations with maximum time $T = 1000s$ of the translated SLSF model in each case. In Fig. 16 we show respectively the property

satisfaction and violation through the automatic translation from SpaceEx to SLSF by plotting the corresponding locations versus time, where different colors correspond to different simulations. In the safe case ($A < B$), the locations reached via simulations all maintained the mutual exclusion property and were $Loc^2 \setminus \{cs \sim cs, try \sim cs, cs \sim try\}$. In the unsafe case ($A > B$), the locations reached via simulation included every location (e.g., all 16 locations of the permutations of Loc^N for $N = 2$) and violated the mutual exclusion property. These results give further empirical evidence for the correctness of the translation procedure.

A.6 Additional Case Studies

Table 1 summarizes the different types of benchmarks that were all successfully translated and checked for trajectory-equivalence. The experiments are performed on Intel I5 2.4GHz processor with 8GB RAM. All of benchmarks are available in the supplementary files.

Table 1: Overview of the benchmark problems successfully translated to SLSF by using the method in this paper. Column Type presents different classes of dynamics, where LC, NLC, LH, and NLH are abbreviations for linear continuous, nonlinear continuous, linear hybrid, and nonlinear hybrid, respectively. Columns $|Var|$, $|Loc|$, and $|Trans|$ show the number of variables, locations, and transitions, respectively, while t_c and t_s show respectively the time our tool required to translate the model, and the time to simulate the translated SLSF diagram twice.

No.	Name	Type	$\|Var\|$	$\|Loc\|$	$\|Trans\|$	t_c	t_s
1	biology_1	NLC	7	1	0	8.894	20.912
2	biology_2	NLC	9	1	0	7.892	12.939
3	bouncing_ball	LC	2	1	1	8.149	11.960
4	brusselator	NLC	2	1	0	7.428	10.650
5	buckling_column	NLC	2	1	0	7.738	11.056
6	coupledVanderPol	NLC	4	1	0	8.202	11.746
7	E5	NLC	5	1	0	8.230	36.635
8	fischer_N2_flat_safe	LH	6	16	82	20.158	54.145
9	fischer_N2_flat_unsafe	LH	6	16	82	19.287	59.627
10	glycemic_control_1	NLH	5	3	4	8.319	15.385
11	glycemic_control_2	NLH	5	3	4	8.301	15.567
12	glycemic_control_poly1	NLH	4	9	18	10.528	23.938
13	glycemic_control_poly2	NLH	4	6	10	9.237	19.341

Table 1: (continued)

No.	Name	Type	$\lvert Var \rvert$	$\lvert Loc \rvert$	$\lvert Trans \rvert$	t_c	t_s
14	helicopter	LC	28	1	0	10.096	14.897
15	Hires	NLC	9	1	0	7.912	9.001
16	jet_engine	NLC	2	1	0	7.667	11.816
17	lac_operon	NLC	2	1	0	7.586	13.257
18	lorentz	NLC	3	1	0	7.739	11.253
19	lotka_volterra	NLC	2	1	0	7.740	11.025
20	circuits_n2	NLH	3	3	2	9.39	13.895
21	circuits_n4	NLH	5	3	2	8.506	14.202
22	circuits_n6	NLH	7	3	2	8.585	15.113
23	circuits_n8	NLH	9	3	2	8.624	15.386
24	circuits_n10	NLH	11	3	2	8.752	15.813
25	circuits_n12	NLH	13	3	2	9.604	19.837
26	OREGO	NLC	4	1	0	9.157	11.111
27	randgen	LH	3	3	6	9.056	15.112
28	Rober	NLC	4	1	0	8.266	16.999
29	roessler	NLC	3	1	0	9.144	12.771
30	small_circuit	NLC	5	1	0	10.265	13.660
31	spiking_neuron	NLH	2	2	2	8.703	13.559
32	spring_pendulum	NC	4	1	0	9.861	6.251
33	vanderpol	NLC	2	1	0	8.119	12.226