Compositional Design of Simulink/Stateflow Models with Hybrid Automata

Sergiy Bogomolov[1], Taylor T. Johnson[2], Luan Viet Nguyen[2], and Christian Schilling[1]

[1] Albert-Ludwigs-Universität Freiburg, Germany
[2] University of Texas at Arlington, USA

Abstract. Hybrid automata are an important formalism for modeling dynamical systems exhibiting mixed discrete-continuous behavior such as control systems. A number of powerful tools to model and formally analyze hybrid automata have recently emerged. However, hybrid automata still lack expressiveness compared to integrated model-based design (MBD) frameworks such as MathWorks Simulink. In this paper, we propose a correct-by-construction framework for the *compositional* system design of Simulink/Stateflow (SLSF) models. In particular, our technique enables an automatic *embedding* of the hybrid automata design and formal analysis into the SLSF MBD process. For this purpose, we provide a translation of a hybrid automaton model to a semantically equivalent SLSF model. Here, the issues that arise due to the non-deterministic behavior of hybrid automata require special attention. We observe that SLSF assumes *must*-semantics, i.e., a discrete transition fires as soon as its guard is enabled, whereas a hybrid automaton typically works in the *may*-setting. In our approach, we introduce a number of randomization steps to account for these discrepancies. We implement the translation method in a prototype software tool and illustrate our approach on a closed-loop control system for a DC-to-DC power converter used as a sub-component in larger SLSF models of cyber-physical systems (CPS).

1 Introduction

In this paper, we present a technique to *automate* the *compositional* design of Simulink models. 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. These components are then linked together to form the whole system under consideration. Here, we assume that the system model consists of heterogeneous components and a number of components are modeled as hybrid automata. In the last decade, a number of powerful design and analysis tools for hybrid automata such as SpaceEx [8] and dReach [9] have emerged. Those tools are particularly useful for the *formal* analysis and verification of hybrid automata. Using this approach, a designer can ensure the *correctness* of every individual component before linking them together.

In this work, we introduce a technique to automatically convert hybrid automata built in SpaceEx into *semantically equivalent* MathWorks Simulink/Stateflow (SLSF) diagrams. We note that hybrid automata and Simulink differ in their semantics. In particular, the handling of non-determinism requires special attention: We observe that a hybrid automaton is typically defined with *may*-semantics wrt. the discrete transitions, whereas Simulink employs *must*-semantics. In other words, a transition in SLSF fires 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 is not violated. Our approach essentially incorporates additional *randomization* steps into the resulting SLSF diagram. In this way, in every run, the diagram produces a different trace which, however, still reflects the hybrid automaton semantics. Therefore, after running multiple simulations, we get an approximation of the reachable state space of the original hybrid automaton.

Overall, the resulting SLSF diagram can *mimic* the non-deterministic behavior induced by the analyzed hybrid automaton. Therefore, our model translator can be used as part of the model-based design (MBD) process as shown in Fig. 1. We highlight that these semantics differences and non-determinism may not lead only to incomplete results using deterministic SLSF diagrams, but may also lead to unsound simulation results—in the sense that the resulting simulation trace is not contained in the reachable states of the hybrid automaton. We provide an example where a naive (non-semantics preserving) translation from a hybrid automaton to an SLSF diagram yields unsound results.

Contributions. The primary contributions of this paper include: *(a)* developing the first (to the best of our knowledge) semantics-preserving translation procedure from non-deterministic hybrid automata models to SLSF diagrams that enables a correct-by-construction model-based design (MBD) process, *(b)* the implementation of this translation procedure in a software tool, and *(c)* the evaluation of the translation procedure on several case studies, including a power/energy cyber-physical system (CPS).

Related Work. There is a large body of research on the translation of SLSF models into other tools [1, 3, 4, 6, 7, 16, 23, 24, 26, 28] and commercial tools like Esterel's SLSF-to-Lustre. Agrawal et al. [1] suggest an algorithm to translate SLSF models into the equivalent HSIF [6, 7, 23, 24] models. The Compositional Interchange Format (CIF) provides a common input language focused on model compositionality [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. [26] consider the translation problem for complex SLSF diagrams where involved treatment of zero-crossings is needed. Manamcheri et al. [16] 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 [4, 5, 10, 11, 16, 25]).

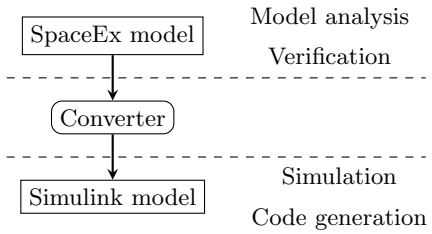

Fig. 1: High-level overview of the model-based design process. Verification using the hybrid automaton model is performed in SpaceEx, then with the semantics-preserving translator presented in this paper, we generate an SLSF diagram that satisfies the same properties as the hybrid automaton. The diagram may be integrated into more complex systems with possibly less formal components (because they are too large to verify, exist for legacy reasons, etc.), and may then be used in code generation processes. The resulting implementations thus have core functionality that is correct-by-construction, under the assumption that the code generation process also preserves semantics (as ensured, e.g., in the work by Sampath et al. [25]).

In this work, we go another way by considering a model conversion in a reverse direction, i.e., by converting a given hybrid automaton into an SLSF model. In this setting, we benefit from the clear and unambiguous definition of hybrid automata semantics. Pajic et al. [13, 20–22] consider a similar problem of converting timed automata encoded in the Uppaal [15] semantics into SLSF models. 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, we provide a *semantically preserving* translation of SpaceEx models, which properly handles the non-determinism of hybrid automata. While operational semantics of (purely discrete) Stateflow have been developed [11], and alternative formalization of discrete semantics have been investigated using, e.g., translation from Stateflow-to-C [25]. In contrast to these prior works, we focus on continuous-time Stateflow diagrams.

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

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 Simulink/Stateflow 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* $\mathcal{H} = (Loc, Var, Init, Flow, Trans, Inv)$ *defining*

- *the finite set of locations* Loc,
- *the set of continuous variables* $Var = \{x_1, \ldots, x_n\}$ *from* $\mathbb{R}^n$,
- *the initial condition, given by* $Init(\ell) \subseteq \mathbb{R}^n$ *for each location* ℓ,
- *the* $Flow(\ell)$ *for each location* ℓ, *a relation over the variables and their derivatives. We assume* $Flow(\ell)$ *to be of the form* $\dot{x}(t) = Ax(t)+b$, *where* $x(t) \in \mathbb{R}^n$, A *is a real-valued* $n \times n$*-matrix and* b *is a real-valued* n*-dimensional vector,*
- *the discrete transition relation* $Trans$, *where every transition is formally defined as a tuple* $(\ell, g, \upsilon, \ell')$*:*
  - *the source location* ℓ *and the target location* ℓ',
  - *the guard, given by a linear constraint* g,
  - *the update, given by an affine mapping* υ, *and*
- *the invariant* $Inv(\ell) \subseteq \mathbb{R}^n$ *for each location* ℓ.

The semantics of a hybrid automaton $\mathcal{H}$ is defined as follows. A *state* of $\mathcal{H}$ is a tuple $(\ell, \mathbf{x})$, which consists of a location $\ell \in Loc$ and a point $\mathbf{x} \in \mathbb{R}^n$. More 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 tuple $\rho = (L, \mathbf{X})$, 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. Furthermore, we will 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 $(\tau_i)_{i=0\ldots k}$) have to satisfy the following conditions:

- $\tau_0 = 0$, $\tau_i < \tau_{i+1}$, and $\tau_k = \Delta$ – the sequence of switching points increases, starts with 0 and ends with Δ
- $L(0) = \ell$, $\mathbf{X}(0) = \mathbf{x}$, $L(\Delta) = \ell'$, $\mathbf{X}(\Delta) = \mathbf{x}'$ – the trajectory starts in $s = (\ell, \mathbf{x})$ and ends in $s' = (\ell', \mathbf{x}')$
- $\forall\, i\ \forall\, t \in [\tau_i, \tau_{i+1}) : L(t) = L(\tau_i)$ – the location is not changed during the continuous evolution
- $\forall\, i\ \forall\, t \in [\tau_i, \tau_{i+1}) : \quad (\mathbf{X}(t), \dot{\mathbf{X}}(t)) \in Flow(L(\tau_i))$, i.e., $\dot{\mathbf{X}}(t) = A\mathbf{X}(t) + b$ holds and thus the continuous evolution is consistent with the differential equations of the corresponding location
- $\forall\, i\ \forall\, t \in [\tau_i, \tau_{i+1}) : \mathbf{X}(t) \in Inv(L(\tau_i))$ – the continuous evolution is consistent with the corresponding invariants
- $\forall\, i < k\ \exists\, (L(\tau_i), g, \upsilon, L(\tau_{i+1})) \in Trans : \mathbf{X}_{end}(i) \in g \wedge \mathbf{X}(\tau_{i+1}) = \upsilon(\mathbf{X}_{end}(i)) \wedge \mathbf{X}_{end}(i) = \lim_{\tau \to \tau_{i+1}^-} \mathbf{X}(\tau)$ – 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 mostly refer to *symbolic states*. A symbolic state $s = (\ell, \mathcal{R})$ is defined as a tuple, 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 $\bigcup_\ell(\ell, \mathit{Init}(\ell))$. The reachable state space 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 Traj($\mathcal{H}$).

We assume there is a given set of symbolic bad states $\mathcal{S}_{bad}$ that violate a given property. As mentioned in Sec. 1, we assume that the system model is designed in a bottom-up fashion. Therefore, we are interested in ensuring that every individual hybrid automaton is correct, i.e., avoids the bad states. In other words, we check with a hybrid model checker such as SpaceEx whether there exists a sequence of symbolic states which contains a trajectory from $\mathcal{S}_{init}$ to a symbolic *error state*, where a symbolic error state s_e has the property that there is a symbolic bad state in $\mathcal{S}_{bad}$ that agrees with s_e on the discrete part, and that has a non-empty intersection with s_e on the continuous part.

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 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 is roughly analogous to a hybrid automaton, but their semantics 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.

Definition 2 (Stateflow diagram). *Formally, a Stateflow diagram is a tuple* $\mathcal{S} = \langle Loc_\mathcal{S}, Junc_\mathcal{S}, Var_\mathcal{S}, Trans_\mathcal{S}, Actions_\mathcal{S} \rangle$.[3] *Here,*

- $Loc_\mathcal{S}$ *is a finite set of locations (states),*
- *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"),*

[3] In this paper, we do not consider compositions of automata or the hierarchical state machine-like modeling capabilities of Stateflow.

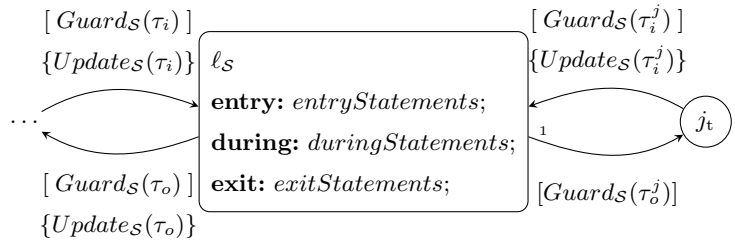

Fig. 2: General continuous-time Stateflow diagram and its various components.

- *$Var_{\mathcal{S}}$ is a finite set of variables of various types,*
- *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,*[4] *Also, we note that these statements are evaluated sequentially, while hybrid automaton actions are executed concurrently.*
- *the discrete transition relation $Trans_{\mathcal{S}}$ where every transition $\tau \in Trans_{\mathcal{S}}$ is formally defined as a tuple $(\ell_{\mathcal{S}}, Guard_{\mathcal{S}}, Update_{\mathcal{S}}, TP_{\mathcal{S}}, \ell'_{\mathcal{S}})$:*
  - *the source location or junction $\ell_{\mathcal{S}} \in Loc_{\mathcal{S}} \cup Junc_{\mathcal{S}}$ and the target location or junction $\ell'_{\mathcal{S}} \in Loc_{\mathcal{S}} \cup Junc_{\mathcal{S}}$,*
  - *the guard, given by a constraint $Guard_{\mathcal{S}}$, must be satisfied for a transition to be taken,*
  - *the update, given by a mapping $Update_{\mathcal{S}}$, modifies state variables, and*
  - *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 trace, which is closely related to a trajectory from a hybrid automaton.

Definition 3 (Simulation trace). *For an initial state x_0, a time bound $\mathcal{T}_{max}$, error bound $\varepsilon > 0$, and time step $\tau > 0$, a* simulation trace *(of length k) is a finite 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*

- $\forall\, i : 0 \leq t_{i+1} - t_i \leq \tau, t_k = \mathcal{T}_{max}$,
- $\forall\, i\ \forall\, t \in [t_i, t_{i+1}]$: *the simulation state after time t is in R_i, and*
- $\forall\, i : dia(R_i) \leq \varepsilon$.

By $\mathrm{Trac}(\mathcal{S})$ *we denote the set of all simulation traces of an* SLSF *diagram $\mathcal{S}$.*

3 Translation of a Hybrid Automaton to SLSF

In this section, we describe our main contribution, a translation scheme from a hybrid automaton to an SLSF diagram. As already outlined in Sec. 1, one main

[4] Our tool by default assumes Matlab statements.

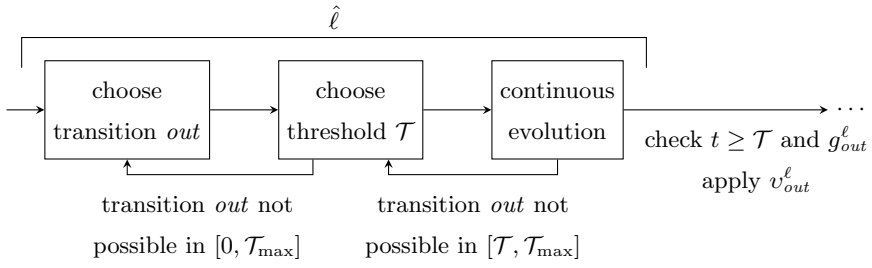

Fig. 3: High-level location pattern translation scheme. The location cluster $\hat{\ell}$ denotes a group of SLSF states and junctions which reflects the behavior of the hybrid automaton in the location ℓ.

difference between the hybrid automaton semantics and the Simulink semantics is the absence of *non-determinism* in the latter. In this work, we consider two sources of non-determinism. First, a number of transition guards might be enabled at the same moment. Moreover, a hybrid automaton is allowed to stay in a particular location as long as the invariant is valid. Therefore, a hybrid automaton is *only* forced to leave the location when the invariant becomes invalid. Note that our primary goal is to ensure that the SLSF diagram can essentially *mimic* the behavior of the original hybrid automaton. We achieve this by *incorporating* non-determinism in the form of *uniformly distributed* random number generation. In this way, by executing multiple Simulink simulations we can *approximate* the reachable state space of the original hybrid automaton.

Translation Overview We first give a high-level overview of the translation scheme (see Fig. 3). The execution of the resulting SLSF diagram consists of three *phases* for every location ℓ of a hybrid automaton. In the first phase, we *randomly* choose a transition *out* from the transitions currently available. In the next phase, we choose a time threshold $\mathcal{T}$. In the final phase, we incorporate the original continuous dynamics of the location ℓ. Furthermore, the outgoing transitions from the third phase essentially correspond to the outgoing transitions of the location ℓ. In addition, we enforce that the transition under consideration must correspond to the transition *out* chosen before, and that taking it can happen only after dwelling in ℓ for at least until time moment $\mathcal{T}$. If the transition *out* cannot be taken in the time frame $[\mathcal{T}, \mathcal{T}_{\max}]$, where $\mathcal{T}_{\max}$ is the maximum simulation time, we *backtrack* and select a new time threshold $\mathcal{T}$. Finally, if the chosen transition cannot be taken at all (i.e., for $\mathcal{T} = 0$), we try the next one. We note that our notion of backtracking is different than the one that occurs with multiple junctions in SLSF. In particular, in our framework, we require allowing some dwell time to elapse in states, whereas junctions are instantaneous.

Translation Steps. Now 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. 4). We refer to all the SLSF states and junctions corresponding to the location ℓ of a hybrid

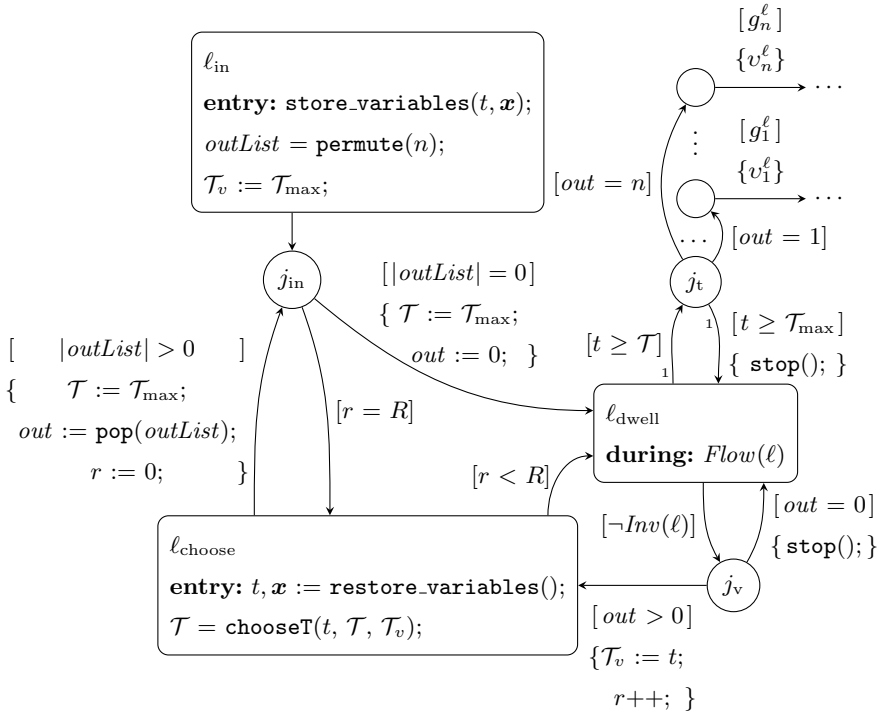

Fig. 4: General location cluster of some location ℓ with n outgoing transitions. `(re-)store_variables` stores and restores the current simulation state (including the time variable t) from when entering the cluster, respectively. `permute`(n) returns a permuted list *outList* with all integers from 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.

automaton as a *location cluster* $\hat{\ell}$. In order to reflect the non-determinism induced by a hybrid automaton, we add a number of auxiliary SLSF states and junctions. 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 larger than one point, 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. Intuitively, 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 *randomly* choose the next time threshold from the interval $[t, \mathcal{T}]$. Here, we observe that the invariant violation can in general happen before the time threshold has been reached. Thus, we actually select a new threshold from the interval $[t, \min(\mathcal{T}, \mathcal{T}_v)]$. In this way, we end 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 backtrackings 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 and stop the simulation.

4 Translation Discussion and Correctness

To evaluate our approach from a theoretical point of view, we assume a perfect simulator and infinite precision in both the state variables and the sampling time, i.e., the simulator can choose infinitesimally small sampling steps. Additionally, we suppose the pseudo-random uniform random number generator is able to mimic total non-determinism.

As the SLSF diagrams resulting from the translation rely on backtracking, it is important to guarantee termination of every simulation.

Proposition 1. *The execution of one simulation terminates if the respective input hybrid automaton is zeno-free.*

The simulation traces we produce under-approximate the reachable state space of the original hybrid automaton. We formally state this result in the following proposition.

Proposition 2. *Let $\mathcal{H}$ be a hybrid automaton and $\mathcal{S}$ be the* SLSF *diagram obtained by our transformation. It holds that* $\mathrm{Trac}(\mathcal{S}) \subseteq \mathrm{Traj}(\mathcal{H})$.

We observe that a hybrid automaton can generally exhibit an *uncountable* number of trajectories. Under this assumption, we can conclude that the above non-strict inclusion actually reduces to a strict one as we can get at most a *countable* number of simulation traces.

Finally, we note that our translation scheme does not depend on the specific form of the continuous dynamics used in the considered hybrid automaton. Therefore, although we only consider affine hybrid automata which can be handled by SpaceEx in this work, our approach is also applicable to systems described with non-linear differential equations and in general to all systems which are supported by Simulink.

5 Evaluation and Experimental Results

To evaluate the translation methodology presented in this paper, we implemented a prototype translator in Matlab 2014. The input to the translator is a hybrid automaton $\mathcal{H}$ in the SpaceEx XML format.[5] Once parsed in the tool, an object representing the syntactic structure of $\mathcal{H}$ is walked, and then the tool applies the sequence of translation steps from Fig. 4. 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, then performed the translation and simulations to illustrate the semantics preservation. The tool and examples are available for download.[6]

[5] Networks of hybrid automata may first be composed within SpaceEx to yield a single hybrid automaton representing the network.

[6] `http://swt.informatik.uni-freiburg.de/tool/spaceex/ha2slsf`

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 [12, 14, 17]. 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 [27]. A hybrid automaton model of the plant and a hysteresis 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 [12]. Intuitively, it operates similarly to a thermostat, where the switch is toggled so the source voltage is connected if the output voltage is too low, and it is toggled so that it is disconnected if the output voltage is too high. We note that by Kirchhoff's voltage law (KVL), $V_C = V_{out}$ [27]. 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 specification of the buck converter:

$$\phi(t) \triangleq t \geq t_s \Rightarrow V_{out}(t) = V_{ref} \pm V_{rip}, \text{ which projected onto the phase space is}$$
$$\phi \triangleq V_{ref} - V_{rip} \leq V_{out} \leq V_{ref} + V_{rip}.$$

The bad states $\mathcal{S}_{bad}$ are defined as those where $\neg\phi$ is satisfied. SpaceEx may be used to verify ϕ after t_s time by computing the reachable states Reach($\mathcal{H}$) from a startup state where the initial states $\mathcal{S}_{init}$ have $i_L = 0$ 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. We compute the reachable states of the buck converter automaton with a hysteresis controller using SpaceEx (shown in Fig. 6) and see it meets the specification since for $t \geq t_s$, we have Reach($\mathcal{H}$) $\subseteq \overline{\mathcal{S}_{bad}}$ [12, 14, 17].[7]

Instantiation of the Translated Buck Converter in Larger SLSF Models We utilize the translated buck converter automaton in the context of system design for a larger system. Buck converters are common subcomponents of more complex systems [17]. For example, a popular architecture for DC-to-AC conversion of renewable energy sources like photovoltaics is known as a multilevel inverter, which effectively connects N DC voltage sources at appropriate times to approximate the AC sinusoid of the electric grid. However, renewable sources like photovoltaics have time-varying voltage sources (due to a variety of factors like

[7] It is sufficient to either (a) compute the infinite-horizon reach states or (b) apply a stability argument to yield infinite-horizon verification, see [12, 14].

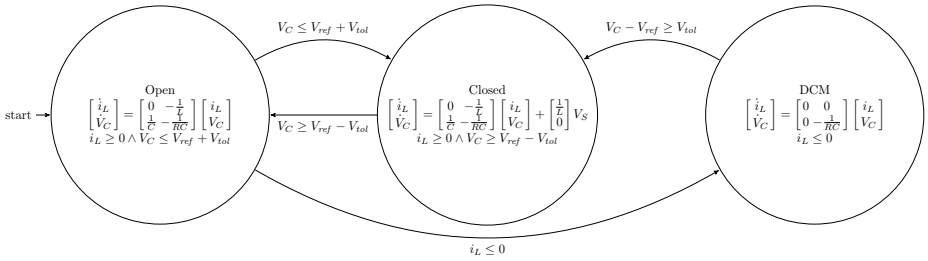

Fig. 5: Hybrid automaton model of the buck converter plant with hysteresis controller.

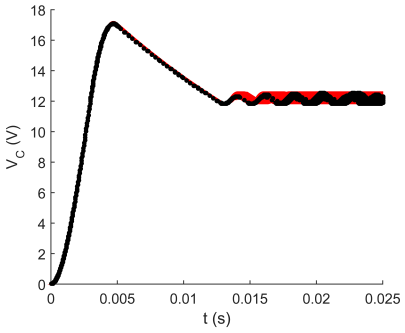

Fig. 6: Buck converter V_C versus time, with SpaceEx reach set in red, and black points from 10 simulation traces from the translated SLSF diagram using the translation method presented in this paper.

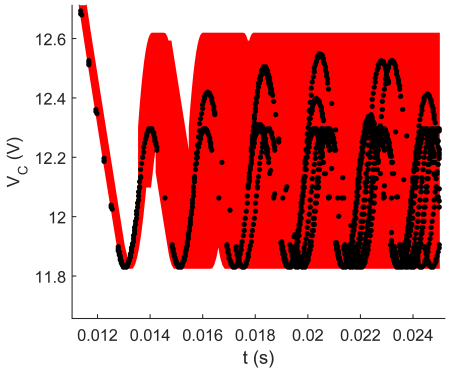

Fig. 7: Zoomed version of Fig. 6 illustrating SpaceEx reach set in red, and black points from 10 simulation traces from the translated SLSF diagram.

shading and weather), so regulation of the photovoltaics is frequently done prior to abstracting the renewable source as an ideal DC voltage source [19]. Common numbers of voltage sources and buck converters are in the tens and more, so with two state variables each, it becomes infeasible beyond small choices of N to utilize tools like SpaceEx for verification as the overall state space consists of tens of real dimensions ($2N$) and $O(2^N)$ discrete states.

5.2 Random Generator Case Study

As a second case study, we consider translation of an example with increased discrete complexity, in terms of both a larger number of locations and more transitions. We utilize the prototype HyRG tool to randomly generate hybrid automata with piecewise affine dynamics and non-determinism [18]. Fig. 10 shows the reachable states computed with SpaceEx for a random hybrid automaton example with 20 locations and 2 real, continuous variables, where the SLSF simulation traces are plotted on top from an initial condition approximately at the point $(x_1, x_2) = (2, 25)$. Fig. 11 shows the same example zoomed to the region

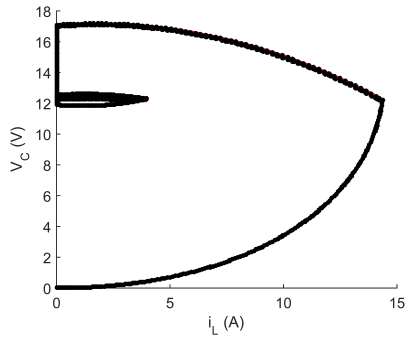

Fig. 8: Buck converter V_C versus i_L (phase space), illustrating SpaceEx reach set in red, and points from 100 simulation traces from the translated SLSF diagram.

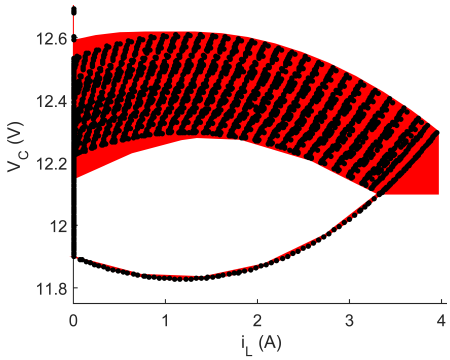

Fig. 9: Zoomed version of Fig. 8 illustrating SpaceEx reach set in red, and points from 100 simulation traces from the translated SLSF diagram.

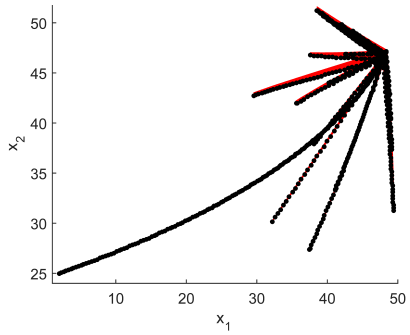

Fig. 10: Random generator x_1 versus x_2 (phase space), illustrating SpaceEx reach set in red, and black points from 50 simulation traces from the translated SLSF diagram.

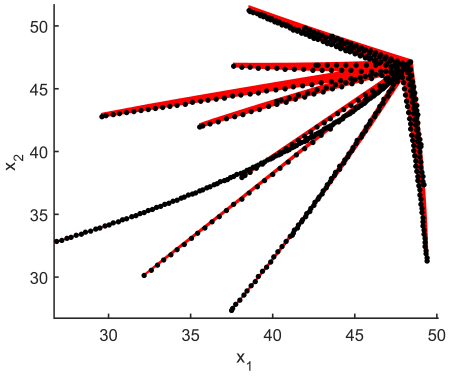

Fig. 11: Zoomed version of Fig. 10, illustrating SpaceEx reach set in red, and black points from 50 simulation traces from the translated SLSF diagram.

of the state space where several discrete transitions are enabled, represented in the figures by the polytopes from approximately the point $(x_1, x_2) = (50, 47.5)$. Specifically, 8 transitions are all simultaneously enabled, only one of which would ever be taken in SLSF without preserving the may semantics. However, using the semantics preserving translation scheme described in this paper, traces following each of these 8 possible trajectories are found.

6 Conclusion

In this paper, we have presented a semantics-preserving transformation of a hybrid automaton into a continuous-time SLSF model, and described its implementation in a prototype software tool. Our approach is based on adding

auxiliary sources of non-determinism to mimic the may-semantics of hybrid automata. 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 into the Simulink models and exploring further sources of non-determinism such as non-deterministic flows and updates. Additionally, using the recent efforts in defining semantics of various subclasses of SLSF models [4, 5, 10, 11, 16, 25], it would be interesting to attempt to formally prove correctness of our translation scheme, perhaps building on [5].

References

1. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. Electronic Notes in Theoretical Computer Science 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. EMSOFT '08, ACM, New York, NY, USA (2008)
4. 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)
5. 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)
6. 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)
7. 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)
8. Frehse, G., Le Guernic, C., Donz, 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)
9. 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)
10. 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)
11. Hamon, G., Rushby, J.: An operational semantics for Stateflow. International Journal on Software Tools for Technology Transfer 9(5-6), 447–456 (2007)
12. Hossain, S., Dhople, S., Johnson, T.T.: Reachability analysis of closed-loop switching power converters. In: Power and Energy Conference at Illinois (PECI). pp. 130–134 (2013)

13. 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)
14. Johnson, T.T., Hong, Z., Kapoor, A.: Design verification methods for switching power converters. In: Power and Energy Conference at Illinois (PECI), 2012 IEEE. pp. 1–6 (Feb 2012)
15. 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)
16. 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)
17. 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)
18. Nguyen, L.V., Schilling, C., Bogomolov, S., Johnson, T.T.: HyRG: A random generation tool for piecewise affine hybrid automata. In: NASA Formal Methods. Springer Berlin / Heidelberg (2015 (Under Review))
19. Nguyen, L.V., Tran, H.D., Johnson, T.T.: Virtual prototyping the distributed control of a fault-tolerant modular multilevel inverter for photovoltaics. IEEE Transactions on Energy Conversion (2015), to Appear
20. Pajic, M., Mangharam, R., Sokolsky, O., Arney, D., Goldman, J., Lee, I.: Model-driven safety analysis of closed-loop medical systems. Industrial Informatics, IEEE Transactions on 10(1), 3–16 (2014)
21. 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)
22. 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)
23. 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, Lecture Notes in Computer Science, vol. 3927, pp. 491–506. Springer Berlin Heidelberg (2006)
24. 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, Lecture Notes in Computer Science, vol. 3414, pp. 526–541. Springer Berlin Heidelberg (2005)
25. 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)
26. 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. HSCC '12, ACM, New York, NY, USA (2012)
27. Severns, R.P., Bloom, G.: Modern DC-to-DC Switchmode Power Converter Circuits. Van Nostrand Reinhold Company, New York, New York (1985)
28. 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: Proof Sketches of Propositions

We discuss the propositions made in Sec. 4.

Proposition 1. *The execution of one simulation terminates if the respective input hybrid automaton is zeno-free.*

A simulation trace comprises of a sequence of location clusters. By construction, it is easy to see that the time the simulator stays in some location cluster is finite. On the other hand, by assuming non-zenoness, we ensure time progress in the simulation. Since the total time is bounded by some maximum simulation time $\mathcal{T}_{\max}$, the claim follows. Note that a simulation can also stop earlier in the case of a deadlock.

Proposition 2. *Let $\mathcal{H}$ be a hybrid automaton and $\mathcal{S}$ be the* SLSF *diagram obtained by our transformation. It holds that* $\mathrm{Trac}(\mathcal{S}) \subseteq \mathrm{Traj}(\mathcal{H})$.

This proposition cannot be formally proven, as no formally defined semantics of the continuous-time SLSF is provided. We observe that a hybrid automaton can generally exhibit an *uncountable* number of trajectories. Under this assumption, we can conclude that the above non-strict inclusion actually reduces to a strict one as we can get at most a *countable* number of simulation traces.

We need to show that for each $\mathcal{S}$-trajectory ρ_s there exists some corresponding $\mathcal{H}$-trajectory ρ_h. For this purpose, we partition ρ_s into *stages*, where a stage is a visit of a location cluster, i.e., the time from the moment when a location cluster is entered until it is either left (possibly to be re-entered immediately in case of a self-loop) or the simulation is stopped. When the simulator leaves a location cluster $\hat{\ell}$, i.e., moves to the next stage, this semantically reflects a location switch of the hybrid automaton. We use a bottom-up constructive argument by an induction over the number of stages in ρ_s. Without loss of generality, we only consider stages which do not feature any backtrackings, i.e., the simulator was always able to extend the trajectory based on the chosen values of the transition *out* and time threshold $\mathcal{T}$. In the base case, there is only one stage with no transition taken. Here, we can simply assume the trajectories to be equal, because the SLSF diagram $\mathcal{S}$ uses the original invariant and flow of the hybrid automaton $\mathcal{H}$. For the induction step, we consider an $(n+1)$-stage trajectory. We look at the first stage, i.e., the time interval up to the point $\mathcal{T}$ when the initial location cluster is left. As $\mathcal{S}$ uses the original guard and update, there exists some trajectory in $\mathcal{H}$ for which the location switch can happen at time $\mathcal{T}$. The simulation state coincides with the hybrid automaton state both before and after moving to the second stage. As two stages are independent of each other, we can apply the hypothesis to the remaining n stages. Therefore, we define ρ_h as the sequence of sub-trajectories corresponding to the individual stages. The length of the trajectory ρ_h is equal to the number of stages of ρ_s.

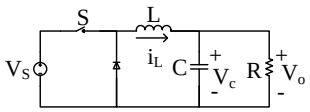

Fig. 12: Buck converter circuit—a DC input V_S is decreased to a lower DC output $V_C = V_o = V_{out}$.

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}	0.2 V
Voltage Ripple Tolerance	V_{rip}	0.6 V
Load Resistance	R	10Ω
Capacitor	C	2.2 mF
Inductor	L	2.65 uH

Table 1: Example buck converter parameter values and variations.

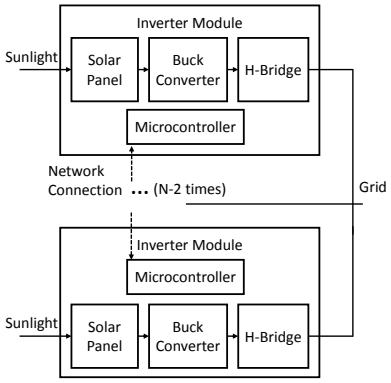

Fig. 13: Solar array architecture using a multilevel inverter for DC-to-AC conversion and consisting of N buck converters.

B Appendix: Additional Details on Buck Converter Case Study

The buck converter circuit appears in Fig. 12. Parameter values used for the case study appear in Table 1. Such converters are commonly instantiated in larger models, such as the multilevel inverter used to interface photovoltaics to the electric grid in Fig. 13 [19].

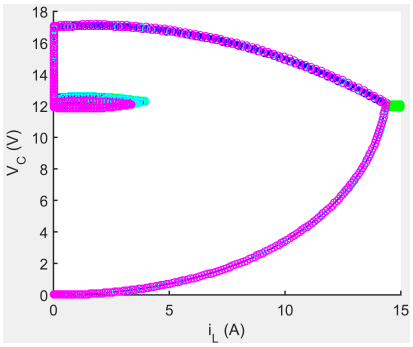

Fig. 14: Buck converter reachable states (red) computed in SpaceEx, where the simulations for the non-semantics preserving translation converge to a different limit cycle than the actual model (green scatter plot area around $i_L = 15$ and $V_C = 12$). The semantics preserving simulations are contained in the reach set computed with SpaceEx.

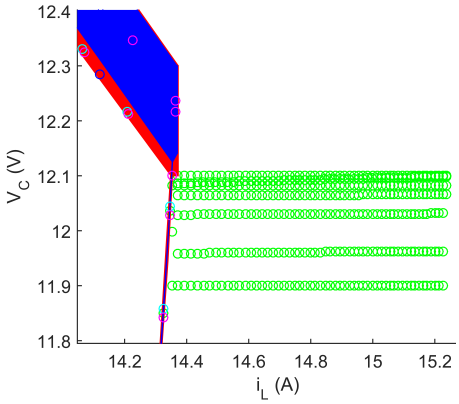

Fig. 15: Buck converter zoomed plot of Fig. 14 of the reach set (red and blue sets) computed with SpaceEx, simulation traces with semantics preservation (non-green scatter plot), and simulation traces without semantics preservation (green scatter plot). Here, the non-semantics preserving simulation traces converge to a different limit cycle than the actual model. The semantics preserving simulation traces are contained in the reach set computed with SpaceEx.

C Appendix: Additional Experimental Results

In addition to the semantics-preserving translation described in this paper, for simple usability reasons, we also implemented an option where the translator does not explicitly preserve semantics. Sometimes this is sufficient for the translation process to maintain behavioral equivalence, but it does not always work in numerous scenarios like non-deterministic ones considered in this paper. Additionally, this translation may even yield drastically different behaviors as seen in Fig. 14 and Fig. 15, which show the buck converter example without semantics preservation. In Fig. 15, we can see that the translated model without semantics preservation even converges to a different limit cycle than the correct one and leaves the reachable states computed by SpaceEx.