HyLAA: A Tool for Computing Simulation-Equivalent Reachability for Linear Systems

Stanley Bak
Air Force Research Laboratory
Aerospace Systems Directorate
stanleybak@gmail.com

Parasara Sridhar Duggirala
Department of Computer Science and Engineering
University of Connecticut
psd@uconn.edu

ABSTRACT

Simulations are a practical method of increasing the confidence that a system design is correct. This paper presents techniques which aim to determine all the states that can be reached using a particular hybrid automaton simulation algorithm, a property we call *simulation-equivalent* reachability. Although this is a slightly weaker property than traditional reachability, its computation can be efficient and accurate.

We present HyLAA, the first tool for simulation-equivalent reachability for hybrid automata with affine dynamics. HyLAA's analysis is exact; upon completion, the tool provides a concrete simulation trace to an unsafe state if and only if the hybrid automaton simulation engine could produce such a trace. In the backend, the tool implements an efficient algorithm for continuous post that exploits the superposition principle of linear systems, requiring only $n + 1$ simulations per mode for an n-dimensional linear system. This technique is capable of analyzing a replicated helicopter system with over 1000 state variables in less than 20 minutes. The tool also contains several novel performance enhancements, such as invariant constraint elimination, warm-start linear programming, and trace-guided set deaggregation.

1. INTRODUCTION

Cyber-physical systems (CPS) that involve interaction between software and the physical world can naturally be modeled using the hybrid automaton formalism. These models allow a mix of discrete and continuous behaviors. Often, the continuous evolution is defined with differential equations that are linear (or affine). Such differential equations represent commonly observed physical systems such as autonomous vehicles, hardware circuits, biological systems, etc. As these CPS are deployed in safety-critical scenarios, it is important to ensure that these systems satisfy the *safety* specification. Further, if a given system design does not satisfy the safety specification, it is useful for debugging to provide the system designer with a concrete counterexample which violates the specification.

Due to increased complexity of CPS, a model-based design framework is increasingly being adopted by industries to do design and development. In this approach, the CPS is modeled in a framework such as Simulink/Stateflow or Modelica and is tested under varying scenarios by using numerical simulations. These simulations, even when they have numerical errors, are regarded as very close approximations of the real behaviors and used for system design and debugging. For CPS, the space of uncertainties is often uncountable, and therefore, one cannot usually conclude that the system satisfies the safety specification from a finite number of sample simulations.

In this paper, we introduce a tool called **HyLAA** (**Hy**brid **L**inear **A**utomata **A**nalyzer) that performs simulation-based verification for hybrid automata with linear ODEs. HyLAA implements an algorithm that computes the reachable set of states of an n-dimensional linear system using only $n+1$ simulations [10]. HyLAA's goal is to perform *simulation-equivalent* reachability for bounded time. That is, it computes the set of states that would be encountered by a hybrid automaton simulation algorithm for all possible nondeterministic choices in the initial state and the discrete transitions. HyLAA declares a system to be safe if and only if all the simulations are safe; it declares a system to be unsafe if and only if there exists a counterexample simulation trace that violates the safety property, and it produces such a trace.

For efficiency, the tool makes two assumptions. First, numerical computations are considered exact and errors due to floating point computations are ignored. Second, the tool assumes that the underlying ODE simulation engine provides exact simulations for the dynamics. We believe that these assumptions are reasonable, given the crucial role floating-point simulations already play in system analysis and design. In practice, floating point errors are usually small, and to reduce/eliminate these errors further, one can use simulation engines with arbitrary precision [1, 12, 5]. Analysis of HyLAA is different from other simulation based verification tools such as C2E2 [9], Breach [8], or Strong [7] in two aspects. First, these tools aim to prove safety irrespective of the semantics of the simulation engine used. Second, in worst case, the number of simulations would be exponential in the number of dimensions.

To clarify, we emphasize the differences between a *hybrid automaton simulation algorithm* and an *ODE simulation engine*. A simulation of hybrid automaton records an execution of the hybrid automaton starting from an initial state while taking into account the different modes, invariants and discrete transitions(the specific algorithm is presented later). To do this, HyLAA makes use of an ODE simulation engine, which

DISTRIBUTION A. Approved for public release; Distribution unlimited. (Approval AFRL PA #88ABW-2016-2897, 30 SEPT 2016)
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author. Copyright is held by the owner/author(s).
HSCC'17, April 18-20, 2017, Pittsburgh, PA, USA
© 2017 ACM. ISBN 978-1-4503-4590-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3049797.3049808

can only simulate the behavior of the system according to a given differential equation. In the context of MATLAB, a hybrid automaton simulation algorithm would be similar to simulation engine underlying Simulink / Stateflow (although our semantics are slightly different, see the next section), whereas an ODE simulation engine refers to a standard ODE solver such as `ode45`.

2. PRELIMINARIES

We consider affine hybrid automata defined as follows.

DEFINITION 1. *An affine hybrid automaton is defined as a tuple $\langle Loc, X, Flow, Inv, Trans, Guard\rangle$ where*

Loc is a finite set of locations (also called modes).

$X \subseteq \mathbb{R}^n$ is the state space of the behaviors.

$Flow : Loc \rightarrow AffineDeq(X)$ assigns an affine differential equation $\dot{x} = A_l x + B_l$ for location l of the hybrid automaton.

$Inv : Loc \rightarrow 2^{\mathbb{R}^n}$ assigns an invariant set for each location of the hybrid automaton.

$Trans \subseteq Loc \times Loc$ is the set of discrete transitions.

$Guard : Trans \rightarrow 2^{\mathbb{R}^n}$ defines the set of states where a discrete transition is enabled.

For the hybrid automata we consider, the invariants and guards are given as conjunction of linear constraints.

A reachability problem combines an affine hybrid automaton with an *initial set of states* Q, which is a finite set of elements in $Loc \times 2^{\mathbb{R}^n}$, where second element in the pair is given as conjunction of linear constraints. An *initial state* q_0 is a pair (Loc_0, x_0), where an element exists in the initial set of states with the given location Loc_0, and the point x_0 satisfies both the corresponding linear constraints and the invariant of Loc_0. Unsafe states are indicated by having an explicit set of error modes, $U \subseteq Loc$.

Given an initial state $q_0 = (Loc_0, x_0)$, an execution of the hybrid automaton $\sigma(x_0) = \tau_0 a_1 \tau_1 a_2 \ldots$ is a sequence of trajectories and actions such that each τ_i is the solution of the affine differential equation for the location Loc_i and respects its invariant, (τ_0 starts from Loc_0), the state before a discrete transition a_i should satisfy the $Guard(a_i)$ and the state after the discrete transition satisfies the invariant of the successor mode Loc_{i+1}. We abuse notation and denote the state of the system following a trajectory after time t as $\tau_i(q_0, t)$ where q_0 is the initial state of the trajectory.

The closed form expression for the trajectories is given using the state transformation matrix $\Phi_i : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^{n\times n}$ where $\tau_i(x_0, t) = \Phi_i(t, 0)x_0 + \int_0^t \Phi(t, s)B_i(s)ds$. For linear time invariant systems (LTIs), the state transformation matrix $\Phi_i(t_2, t_1) = e^{A_i(t_2-t_1)}$. For efficiently analyzing these models, numerical simulations from an ODE simulation engine are routinely used. We now define a trace produced by a hybrid automaton simulation algorithm.

DEFINITION 2. *Given a hybrid automaton H and an initial set of states Q, a sequence $\rho_H(q_0, h) = q_0, q_1, q_2, \ldots$, where each $q_i = (Loc_i, x_i)$, is called a (q_0, h)-simulation of H if and only if $q_0 \in Q$ and each pair (q_i, q_{i+1}) corresponds to either (i) a continuous trajectory in location $Loc_i = Loc_{i+1}$ such that a trajectory starting from x_i would reach x_{i+1} after h time units with $x_i \in Inv(Loc_i)$, or (ii) a discrete transition from Loc_i to Loc_{i+1} where $a = (Loc_i, Loc_{i+1})$ and $a \in Trans$ such that $x_i = x_{i+1}$, $x_i \in Guard(a)$ and $x_{i+1} \in Inv(Loc_{i+1})$. We drop the subscript when it is clear from context. Bounded-time variants of these simulations are called (q_0, h, T)-simulations.*

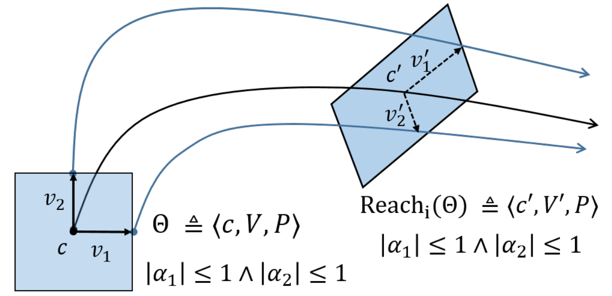


Figure 1: Due to superposition, $\tau(c + v_1 + v_2, t) = c' + v_1' + v_2'$, where $c' = \tau(c, t)$, $v_1' = \tau(c + v_1, t) - \tau(c, t)$ and $v_2' = \tau(c + v_2, t) - \tau(c, t)$. In the star representation, only the center and the basis vectors change as time elapses. The star's predicate, $|\alpha_1| \leq 1 \wedge |\alpha_2| \leq 1$ remains the same.

For these simulations, h is called the *step size* and T is called *time bound*. Pairs of states corresponding to (i) are said to arise from a *continuous-post step*, and pairs from condition (ii) come from a *discrete-post step*.

Notice that hybrid automaton simulation traces which conform to Definition 2 do not check if the invariant is violated for the entire time interval, but only at multiples of the step size h. Also, the discrete transitions are only enabled at time instances that are multiples of h and nondeterminism is allowed in discrete transitions if more than one guard is satisfied, or the invariant remains true. One subtlety is that discrete-post pairs only have the invariant checked in the first state. This has the effect of allowing discrete transitions from states where the invariant is false (i.e., guards are checked before invariants). This is necessary to handle the common case of systems where a guard is the negation of a location's invariant, and a step of the simulation may jump over the boundary. If such behaviors are not desired, the guards can be explicitly strengthened to include the negation of the invariant.

For readers familiar with the simulation engines in standard tools like Simulink / Stateflow or Modelica, the described trajectories do not perform any special *zero-crossing detection* and the transitions are not necessarily *urgent*. In this paper, we attempt to determine if an affine hybrid automaton is safe with respect all bounded-time simulations that conform to the conditions in Definition 2.

DEFINITION 3. *A given simulation $\rho_H(q_0, h)$ is said to be safe with respect to an unsafe set of modes U if and only if $\forall (Loc_i, x_i) \in \rho_H(q_0, h)$, $Loc_i \notin U$. Safety for time-bounded simulations are defined similarly.*

DEFINITION 4. *A hybrid automaton H with an initial set of states Q, time bound T, and unsafe set of modes U is said to be safe if all simulations starting from Q for bounded time T are safe.*

Solutions to linear ODEs (in our case trajectories τ_i in each mode) satisfy a superposition property, illustrated in Figure 1. Given any initial state x_0, vectors $v_1, \ldots, v_m$ where $v_i \in \mathbb{R}^n$, and scalars $\alpha_1, \ldots, \alpha_m$,

$$\tau_i(x_0 + \Sigma_{i=1}^m \alpha_i v_i, t) = \tau_i(x_0, t) + \Sigma_{i=1}^m \alpha_i(\tau_i(x_0 + v_i, t) - \tau_i(x_0, t)).$$

Before describing the algorithm for computing the reachable set, we finally introduce a data structure called a *generalized star*, which is used to represent the reachable set of states in HyLAA.

DEFINITION 5. *A* generalized star Θ *is a tuple* $\langle c, V, P\rangle$ *where* $c \in \mathbb{R}^n$ *is called the* center, $V = \{v_1, v_2, \ldots, v_m\}$ *is a set of* m *(*$\leq n$*) vectors in* $\mathbb{R}^n$ *called the* basis vectors, *and* $P : \mathbb{R}^n \rightarrow \{\top, \bot\}$ *is a predicate. A generalized star* Θ *defines a subset of* $\mathbb{R}^n$ *as follows.*

$$[\![\Theta]\!] = \{x \mid \exists \bar{\alpha} = [\alpha_1, \ldots, \alpha_m]^T \text{ such that } x = c + \Sigma_{i=1}^{n} \alpha_i v_i \text{and } P(\bar{\alpha}) = \top\}$$

Sometimes we will refer to both Θ *and* $[\![\Theta]\!]$ *as* Θ. *In this paper, we consider predicates* P *that are conjunctions of linear constraints.*

3. REACHABILITY ALGORITHM

In this section, we present an algorithm for computing the simulation-equivalent reachable set of states for linear hybrid systems. The description is divided into two parts: (1) computing the set of states for the linear dynamics without considering the invariants and discrete transitions, and (2) accommodating the invariants in each location and accounting for discrete transitions between locations.

3.1 Continuous Dynamics

States reachable under continuous evolution are computed by exploiting the superposition principle and using the *generalized star* representation. For an n-dimensional linear system, this technique requires at most $n+1$ simulations. Let the initial set of states Q be given as a generalized star $\langle c, V, P\rangle$ where $V = \{v_1, v_2, \ldots, v_m\}$. The ODE simulation engine for continuous dynamics, with initial state x_0, step size h, and number of steps k, denoted $\rho(x_0, h, k)$, returns a sequence $x_0, x_1, \ldots, x_k$ such that $x_i = \tau(x_0, i \cdot h)$. We denote x_i as $\rho(x_0, h, k)[i]$. The reachable set at time instances $i \cdot h$ is computed by Algorithm 1 as a generalized star.

Algorithm 1: Algorithm that computes the simulation-equivalent reachable set at time instances $i \cdot h$ from $n+1$ simulations.

```
  input  : Initial Set: Θ ≜ ⟨c, V, P⟩, time step: h, steps: k
  output: Reach(Θ) = Reach_0(Θ), ..., Reach_k(Θ)
1 for each i from 0 to k do
2     c' ← ρ(c, h, k)[i];
3     for each v_j ∈ V do
4         x'_j ← ρ(c + v_j, h, k)[i];
5         v'_j ← x'_j − c';
6     V' ← {v'_1, ..., v'_m};
7     Reach_i(Θ) ← ⟨c', V', P⟩;
8     Append Reach_i(Θ) to Reach(Θ);
9 return Reach(Θ);
```

The algorithm in line 2 computes the state of trajectory starting from the initial state c at time $i \cdot h$ as c'. The loop in lines 3 to 5 computes x'_j, the state of the trajectory starting from $c+v_j$ at time $i \cdot h$. The reachable set at time $i \cdot h$ is given as as generalized star $\langle c', V', P\rangle$, where $V' = \{v'_1, \ldots, v'_n\}$ with $v'_j = x'_j - c'$. The correctness of Algorithm 1 follows from the superposition principle and has been previously established [10]. An illustration of the reachable set computation as described in the algorithm is presented in Figure 1.

With this algorithm, extracting concrete trajectories which reach a given star is straightforward. This process involves expressing the desired point as a vector sum of star's center and scalar multiples along each of the basis vectors. The scalar multiples, for an ordered basis set, is called a *basis point*. The desired trajectory is the sequence of points where the state at discrete time step is obtained by the vector sum of the corresponding center and the multiplication of basis point with the corresponding basis vector matrix.

3.2 Hybrid Dynamics

In this section, we use the described continuous-post algorithm while accounting for invariants and discrete transitions to perform simulation-equivalent reachability. The pseudo-code is given in Algorithm 2.

Algorithm 2: Algorithm that computes the simulation-equivalent reachable set for hybrid automata.

```
   input  : Initial set of states: Q, Time step: h
   output: Simulation-equivalent reachable set
 1 ReachSet ← ∅; cur_state ← ∅; waiting ← ∅;
 2 for q ∈ Q do
 3     push(waiting, q);
 4 while ¬empty(waiting) ∨ cur_state ≠ ∅ do
 5     if cur_state = ∅ then
 6         /* discrete-post step */
 7         ⟨Θ, l⟩ ← pop(waiting);
 8         Θ ← Θ ∩ Inv(l);
 9         if Θ ≠ ∅ then
10             for q ∈ discreteTransitions(Θ, l) do
11                 push(waiting, q);
12             cur_state ← ⟨Θ, l⟩;
13             ReachSet ← cur_state ∪ ReachSet;
14     else
15         /* continuous-post step */
16         ⟨Θ, l⟩ ← cur_state;
17         Θ ← Alg1(Θ, h, 1);
18         for q ∈ discreteTransitions(Θ, l) do
19             push(waiting, q);
20         Θ ← Θ ∩ Inv(l);
21         if Θ ≠ ∅ then
22             cur_state ← ⟨Θ, l⟩;
23             ReachSet ← cur_state ∪ ReachSet;
24         else
25             cur_state ← ∅;
26 return ReachSet;
```

For each given iteration of the outermost loop, the algorithm performs one discrete transition or computes the reachable set according to continuous evolution by one step. First, each of the elements of the initial set of states Q, which are pairs of stars and locations, are pushed onto the waiting list in line 3.

Initially, the reachable set for discrete transitions is calculated in lines 7 to 13 (called discrete post). In a discrete-post step, the set of states that violate the invariant are pruned in

line 8, followed by checking for guard successors in line 10. Pruning the set of states that violate the invariant first ensures two conditions as required in Definition 2: (i) all initial states satisfy the invariant of initial location, and (ii) after each discrete-post step, the invariant of the destination mode is satisfied. If the set of states that satisfy the invariant is non-empty, in line 12 the star is assigned to cur_state and then it gets added to the final reachable set.

After cur_state is assigned, the algorithm will compute the reachable set with the continuous dynamics (called continuous post). In line 17, the earlier continuous algorithm is called for a single step h. In the continuous-post operation, guard checking (line 18) is performed *before* invariant trimming (line 20). This is needed to maintain the condition of valid simulations in Definition 2 that discrete-post steps are possible even if the destination state does not satisfy the invariant of the current mode (this was to be able simulate the common case of guards being the complements of a state's invariant). Finally, if the reachable set for one step satisfies the invariant, line 22 updates cur_state and adds it to the final reachable set. Otherwise, cur_state is discarded, and the next state can be removed from the waiting list.

As a side effect, this algorithm always keeps track of the reachable set that satisfies the invariant (line 20). That is, if the invariant is violated by a given state in the current reachable set, its future continuous trajectory is not part of the reachable set. Without this operation, some of the states which have previously violated the invariant could re-enter the invariant region, and then appear to be reachable. A simulation trajectory as defined in Definition 2, however, could not contain such states. Some reachable set computation tools postpone the pruning of the reachable set until all continuous-post steps complete (the invariant becomes completely false or the time bound is reached), which can lead to this type of error. A demonstration of this is provided in Section 4.2.

The HyLAA implementation has several enhancements to the described algorithm. In HyLAA, time is tracked by maintaining the cumulative number of continuous-post operations performed on each star, and another condition for stopping a continuous-post step is if the number of steps performed exceeds the desired time bound. Additionally, the guard check in the discrete-post logic is optional, depending on if the user wants to support urgent transitions, where no time elapses. It may be desirable to disable these since they can lead to infinite loops where time does not pass (Zeno behavior). Further, star aggregation may need to be performed in order to prevent large numbers of states from being added to the waiting list. However, aggregated stars may be overapproximative and not correspond to real simulations, so deaggregation may be necessary to find concrete counter-example traces. This approach will be discussed more in Section 4.4.

In terms of efficiency and data structures, the predicates in stars are conjunctions of linear constraints. Checking for intersections with guards and invariants (which are linear constraints), can be done by performing a linear optimization in the normal direction of the linear constraint's hyperplane, subject to the constraints that the predicate of the star is true. If an intersection exists, pruning states that violate invariant would require adding an extra constraint to the star. Checking if a star is empty can also be done by checking if any point satisfies the star's linear constraints (the objective can be 0). Thus all the main operations needed for Algorithm 2 can be exactly and efficiently performed.

4. HYLAA TOOL IMPLEMENTATION

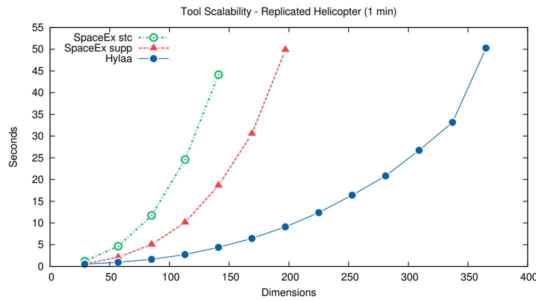
The HyLAA tool implements the theory presented in Section 3. The tool is developed in Python, although many of the core computation components are libraries written in other languages. Simulations are performed using `scipy`, which uses the FORTRAN library `odepack`'s `lsoda` solver. This solver supports a wide variety of differential equations and allows the user to set the simulation accuracy. Linear programming is performed using the `glpk` library, and matrix operations are done using `numpy`. Visualization for the reachable set is provided using the libraries in `matplotlib`. HyLAA supports a live animation mode which shows the generalized stars during the course of the computation, a step-by-step mode when the user presses a button before each continuous or discrete-post operation, and a video export mode that uses `ffmpeg` to output the visualization to a file format such as `.mp4`. The model input file is Python code instantiating HyLAA-specific objects. To ease model development, we have created a printer using the HyST model conversion tool [2]. This allows input models to be created in the SpaceEx [11] format using the SpaceEx model editor, and then exported and used by HyLAA.

In this section, we describe three novel features of HyLAA namely, invariant constraint propagation, warm-start optimization, and trace based aggregation and present some experimental evaluation in comparison to other tools. Owing to space limitations, all the three features have not been fully described and proofs of correctness are not presented. The experimental evaluations have been limited to instances that highlight the new features in HyLAA and are not extensive. The algorithms behind each of the features and the corresponding correctness proofs are presented in other work [4].

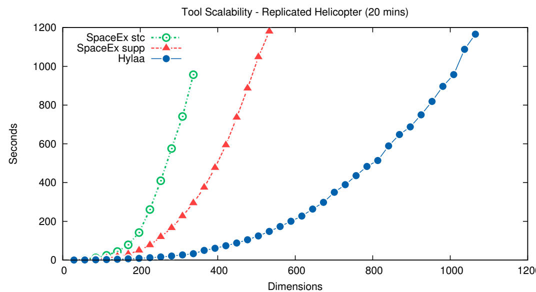
4.1 Continuous-Post Scalability

We first evaluate the scalability of HyLAA, which uses the $n+1$ simulations approach described in Section 3.1. In order to do this, we use the helicopter/controller benchmark provided on the SpaceEx website[1]. We use the same parameters as the `x8_over_time_large` variant of the benchmark, which consists of a 28-dimensional helicopter plus a time dimension, a nondeterministic set of initial states with a 30 second target time and a 0.1 second step size. As this system contains no discrete switches, it serves as an evaluation of the efficiency of the continuous-post operation. We compared HyLAA with `supp` and `stc` scenarios in SpaceEx [11] using the default accuracy settings. We also attempted to use the linear reachability mode from Flow* [6], but did not succeed in finding a suitable set of parameters when using the initial states in the `x8_over_time_large` system (although Flow* did succeed with a smaller initial set of states). In order to show scalability, we replicated the helicopter several times within the same model. This was repeated until the computation did not finish within the 20 minute timeout. The results are shown in Figure 2. HyLAA's approach generally performs better, and is able to complete a 365 dimensional system (13 helicopters) in under a minute, and a 1065 dimensional system (38 helicopters) in under 20 minutes. The execution of the reachability tools was scripted using the `hypy` library [3], along with a model transformation pass for replicating the helicopter system written in HyST [2]. The measurements were performed using a 2.30GHz Intel i5-

[1] http://spaceex.imag.fr/news/helicopter-example-posted-39

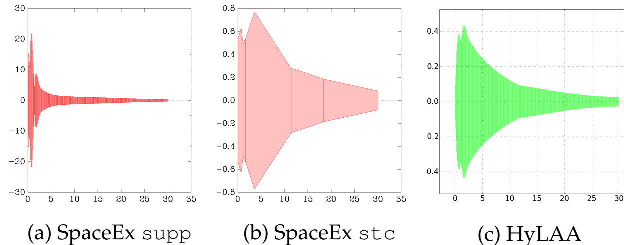


(a) 1 min limit



(b) 20 min limit

Figure 2: Scalability of reachability computation on the replicated helicopter system with a one minute and 20 minute time limit. The $n+1$ simulation continuous-post operation in HyLAA is generally faster than the `supp` and `stc` methods implemented in SpaceEx.

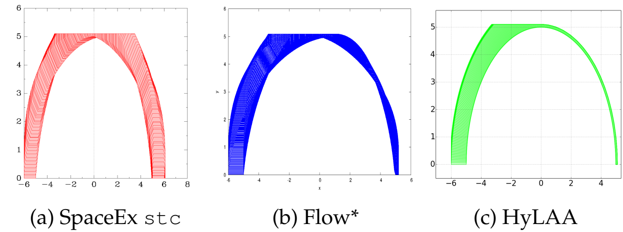


(a) SpaceEx `supp` (b) SpaceEx `stc` (c) HyLAA

Figure 3: Plots of x_8 over time for a two-helicopter system show that the default settings used for measuring SpaceEx produce less accurate reachability plots than HyLAA. Note the y-axis scale with the `supp` method is much larger.

5300U CPU with 16 GB ram running Ubuntu 14.04 x64.

Note that the guarantees provided by SpaceEx and HyLAA are different. SpaceEx computes reachable set that includes states encountered by trajectories at all time instances whereas HyLAA computes reachable set at discrete instances of time. However, HyLAA can produce concrete counterexample traces. We would also like to highlight that there is an inherent challenge in comparing reachability tools, since the choice of parameters often provides a trade off between accuracy and runtime. Under the measured default parameters, Figure 3 compares output plots for the 2-helicopter case for x_8 over time, which shows that HyLAA's result appears more accurate. While the `supp` scenario seems to reach states where $x_8 > 20$, HyLAA's output always remains bounded in $[-0.45, 0.45]$. Furthermore, HyLAA's accuracy does not change for higher dimensional variants. Of course, we could increase SpaceEx's accuracy through parameter selection, but this would serve only to increase the computation times measured in Figure 2, and it would be difficult to justify one choice of parameters over another.



(a) SpaceEx `stc` (b) Flow* (c) HyLAA

Figure 4: Invariant trimming is performed when $y > 5.1$.

4.2 Invariant Constraint Elimination

During a continuous-post operation, it is possible that the generalized star being tracked has a partial intersection with the invariant. HyLAA eliminates states where the invariant is violated by adding an additional linear constraint into the predicate of the generalized star. This constraint is propagated forward in time and is added to predicates of all the following generalized stars. Therefore, successive time steps with a partial invariant intersection result in more constraints being added to the star. For this reason, HyLAA checks if each newly-added constraint is strictly stronger than the previous one, and drops the previous one if so. Without this optimization, the number of constraints added would be equal to the number of steps where a partial intersection takes place, which gets larger as the step size is decreased.

An example demonstrating this is the harmonic oscillator system, with $\dot{x} = y$ and $\dot{y} = -x$. Trajectories of this system rotate clockwise around the origin. The initial states are $x \in [-6, -5]$ and $y \in [0, 1]$, and the single mode's invariant is taken as $0 \leq y \leq 5.1$. In this system, trajectories from most of the initial states actually violate the invariant, and should be pruned from the reachable set.

Reachability plots for this system in SpaceEx `stc`, Flow* and HyLAA are shown in Figure 4. Notice that SpaceEx does not perform such pruning during the continuous-post operation (instead postponing it until afterwards), resulting in large error in the computed set of states. Flow*'s result appears correct, with the seeming overapproximation being an artifact of the octagon plotting mode. HyLAA's generalized star adds several constraints to account for the invariant and hence the final star in the reachable set has 79 constraints. With invariant constraint elimination, this number is reduced to 25.

4.3 Warm-Start Linear Programming

At each step during a continuous-post operation, the generalized star representation of reachable set must handle possible discrete transitions by checking if a guard condition can be satisfied. Moreover, additional constraints might be added to the star to prune the states that violate invariant. If plotting is enabled, the generalized star set has to be projected accurately on a 2-d plane for visualization. A simple box overapproximation can be rendered by finding maximum and minimum values of each axis variables. To obtain a more accurate plot, HyLAA chooses a number equidistant angles (for example, 64), maximizes the cost function along these directions, and renders the projection on 2d plane. Since these operations are performed by solving linear programing (LP) problems, the optimization of the LP engine is essential to HyLAA's performance.

HyLAA uses the GNU Linear Programming Kit (`glpk`) to solve LPs, which is an optimized ANSI C implementa-

tion that gets called from HyLAA's Python code. Internally, `glpk` uses a two-phase Simplex method to solve LPs. The first phase comes up with a feasible solution and the second phase applies the simplex heuristics to drive the feasible solution to the optimal solution. To improve the performance of solving multiple linear programs, HyLAA uses a warm-start LP optimization. Here, HyLAA stores the solution of the previous LP, and uses it as an initial guess for subsequent LPs. This allows `glpk` to skip the first phase of LP, and in many instances the second phase as well. For example, when plotting, a vertex of the star is the maximal point for many plotting directions. In these cases, the LP engine would be able to detect this and terminate immediately with zero additional Simplex iterations.

This warm-start optimization is helpful during guard (and invariant) intersections as well. For guard intersections, HyLAA finds the vertex of the star that is closest to the guard. After a single time step, the basis vectors in the star often only change slightly, and thus the same vertex of the star is often remains the closest one to the guard. The previous LP solution, in this case, would immediately lead to the new LP solution and improves the efficiency of HyLAA.

4.4 Trace-Guided Deaggregation

A well-known issue with flow-pipe construction reachability analysis is that, upon encountering a guard intersection, a single set of states might yield to multiple states in the successor location after a discrete transition. This can be observed in the invariant trimming example in Figure 4. If there was a guard with condition $y \geq 5.1$, then the guard would be enabled at multiple time steps, creating several successor stars in the next mode. As the time duration for which the guard is enabled remains constant, using a smaller step size might increase the number of successor stars. In the worst case, every star set has multiple successor stars during a discrete transition, leading to an exponential increase in the number of stars to be tracked after a few discrete transitions.

One common solution to this problem is to aggregate states in the same mode prior to each continuous-post operation. While this prevents the exponential problem described above, it might lead to a different set of potential problems. First, the aggregation is usually not exact. For generalized stars where the predicate is a set of linear constraints, for example, the exact union of two stars cannot generally be expressed using a conjunction of linear constraints. To see this, notice that a set of linear constraints defines a convex set, whereas the union of two stars can be non-convex, or even disjoint. Since the aggregation is not exact, it is no longer the case that if an aggregated star reaches an unsafe state then a concrete trace exists to the unsafe state. Second, in general, full aggregation based on convex representations (such as support functions in SpaceEx or parallelotopes in Flow*) cannot bound error. To see this, imagine the single mode whose reachable set was shown in Figure 4 contains a guard with a *true* condition. Since the guard is always enabled, all of the states in the plot would be aggregated. Therefore, perfect convex aggregation would include all of the states in the middle of the semi-circle, that are not part of the reachable set.

Thus, it is undesirable not to do aggregation due to exponential blowup in tracked states, but it is also undesirable to do aggregation due to unbounded error with convex overapproximations. HyLAA includes a compromise between these two, where aggregation is performed aggressively, but, upon reaching a guard, a concrete trace needs to be generated which reaches the guard. If such a trace cannot be generated, it means one of the stars along the path from the initial states to the current star includes an aggregated star which contributes to the overapproximation. This aggregated star is identified by backtracking, split into two smaller aggregated stars with less overapproximation, and the reachable set computation is resumed with the new aggregated stars. In this way, a guard is taken if and only if a concrete trace exists to the guard. Since the set of unsafe states is defined as subset of modes in the hybrid automaton, such a state will only be reached if there exists a concrete path. To the authors knowledge, this is the first approach to offer such aggregation and deaggregation strategies. A video demonstration of the deaggregation method is available on HyLAA website [2].

5. CONCLUSION

In this paper, we have presented a tool called HyLAA for performing simulation-equivalent reachability of affine hybrid automata. HyLAA can efficiently compute all the states reached by a specific hybrid automaton simulation algorithm. We have demonstrated the efficiency of the continuous-post operation on HyLAA by computing reachable set of a 1065 dimensional system in under 20 minutes, and described several improvements to handle the discrete transitions.

HyLAA is still in its early phases and many further enhancements are being considered. We plan to add support for resets on guards, as well as support for nondeterministic inputs within the differential equations. We wish to add support for hierarchical models and parallelize the simulation engine to improve scalability.

6. REFERENCES

[1] *Computer Assisted Proofs in Dynamic Groups (CAPD).* http://capd.ii.uj.edu.pl/index.php.

[2] S. Bak, S. Bogomolov, and T. T. Johnson. HyST: A source transformation and translation tool for hybrid automaton models. In *18th International Conference on Hybrid Systems: Computation and Control*, Seattle, Washington, Apr. 2015. ACM.

[3] S. Bak, S. Bogomolov, and C. Schilling. High-level hybrid systems analysis with hypy. In *ARCHâĂŹ16: Proc. of the 3rd Workshop on Applied Verification for Continuous and Hybrid Systems*, 2016.

[4] S. Bak and P. S. Duggirala. Rigorous simulation-based analysis of linear hybrid systems. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017.

[5] O. Bouissou and M. Martel. Grklib: a guaranteed runge kutta library. In *IMACS*, 2006.

[6] X. Chen, E. Abraham, and S. Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. *2013 IEEE 34th Real-Time Systems Symposium*, 0:183–192, 2012.

[7] Y. Deng, A. Rajhans, and A. A. Julius. STRONG: a trajectory-based verification toolbox for hybrid systems. In *Quantitative Evaluation of Systems*, pages 165–168. Springer, 2013.

[8] A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, pages 167–170. Springer, 2010.

[9] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok. C2E2: a verification tool for stateflow models. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 68–82. Springer, 2015.

[10] P. S. Duggirala and M. Viswanathan. Parsimonious, simulation based verification of linear systems. In *International Conference on Computer Aided Verification*, pages 477–494. Springer, 2016.

[11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[12] N. Nedialkov. VNODE-LP: Validated solutions for initial value problem for ODEs. Technical report, McMaster University, 2006.

[2] http://stanleybak.com/hylaa/#hscc2017