Extracting Counterexamples Induced by Safety Violation in Linear Hybrid Systems

Manish Goyal\textsuperscript{a}, Parasara Sridhar Duggirala\textsuperscript{a}

\textsuperscript{a}Department of Computer Science, University of North Carolina at Chapel Hill, USA

Abstract

Control design for linear systems typically involves pole placement and computing Lyapunov functions. While these tools are useful for ensuring stability, they are not always helpful in ensuring safety. Control designers can employ model checking as a tool for checking safety. We believe that supplementing the model checker to provide various types of counterexamples for the safety specification would help the control designer in the control development process. In this paper, we describe a technique for obtaining the variety of counterexamples for a safety violation in linear hybrid systems. More specifically, we develop algorithms to extract the longest counterexample — the execution that stays in the unsafe set for the longest contiguous time, deepest counterexample — the execution that ventures the most into the unsafe set in a user specified direction, and the robust counterexample — the unsafe execution from which some bounded perturbation yields a new counterexample. These measures for classifying counterexamples can further assist in quantifying controllers’ performance.

Key words: Safety verification; hybrid systems; counterexample; dynamic programming; linear programming.

Designing a controller for a system is an iterative process. First, the control designer is provided with a system model and specification. The designer uses tools in his repertoire to come up with a controller, check if the system satisfies the required specification and iteratively refines the controller. Stability and safety are two important classes of specifications. While the tools for stability such as performing pole placement and computing Lyapunov functions provide very intuitive information to the designer, similar tools for safety verification do not exist. Employing model checkers for safety specification yields in a counterexample for safety violation (if safety is indeed violated). However, current model checkers do not have the capability to generate a variety of counterexamples that give additional information to control designer. Such lack of information prevents the designer from comparing different possible refinements of an existing controller.

These challenges are exacerbated when the system is a hybrid system and has several modes of operation. For proving stability or convergence properties of hybrid systems, one has to come up with a common Lyapunov function [34,31] or a set of Lyapunov functions [14,32,44]. These artifacts are not immediately useful in comparing the performance properties of two different hybrid controllers. In such circumstances, metrics on counterexamples for safety specification (or performance specification) can be used as a proxy for comparing performance of different controllers. Thus, providing an important counterexample would greatly reduce the burden of the system designer and provide a more detailed insight into the system behavior.

In verification of hybrid systems domain, while a lot of attention was paid for generating counterexamples for hybrid systems with timed and rectangular dynamics, not many approaches have been developed to extracting various counterexamples for hybrid systems with linear dynamics. This is primarily because most of the model checking approaches in affine hybrid system verification focus on computing over-approximation of reachable set and hence establish the safety specification.

Our goal to generate various types of counterexamples stems from the desire to provide intuition to the control system designer during the process of controller synthesis. A controller that is originally stable and safe, can become unsafe if the safety specification is tightened or the operating conditions are changed. To the control designer, not all counterexamples for this safety specifi-
counterexamples for safety specification, it is also ap-

While we present our analysis in the form of generating

extension algorithms leverage the superposition princi-

ics. The reachable set computation and counterexample

ation algorithm for hybrid systems with linear dynam-

of computing simulation-equivalent reachable set [10],

some perturbation in any direction still yields an execu-

resultant in the unsafe set. The designer might also be interested in the counterex-
ample such that some perturbation around this provides

ally, the algorithms presented reuse the artifacts

during the model checking process.

Example 1 Consider the classic case of a regulation
control problem where the control designer wants to make
the error between the observation and the desired value
to be 0. The typical execution profiles after applying
the feedback control would look similar to Figure 1. In such
cases, the control designer is most concerned about the
amount of overshoot that occurred, its duration, and the
robust overshoot profile. For instance, the blue colored
execution has the longest duration of overshoot in inter-
val 1, whereas the red one has the maximum overshoot in
the interval 2. Further, in interval 1, a profile equidistant
from both blue and red would be the most robust because
some perturbation in any direction still yields an execu-

Current verification techniques, al-

though inform the designer whether the overshoot hap-

pened or not, do not provide them with enough support
to quantify and classify multiple overshoot profiles.

In this paper, we present multiple types of counterex-
amples (deepest, longest and robust) that we believe are
important for control designer, and provide algorithms
for generating them when the specification is violated.
In other words, these counterexamples characterize the ex-
tent of violation in terms of metrics - depth, length, and
robustness. This approach builds on our previous work
of computing simulation-equivalent reachable set [10],
which includes the set of states encountered by a simu-
lation algorithm for hybrid systems with linear dynam-
ics. The reachable set computation and counterexample
generation algorithms leverage the superposition prin-

ple and the generalized star representation [23,10]. Ad-
ditionally, the algorithms presented reuse the artifacts
generated during the model checking process.

While we present our analysis in the form of generating
counterexamples for safety specification, it is also ap-

licable to other types of performance specification. For
example, a control designer might want to reduce the
amount of time spent by the system in a region with sub-
optimal performance characteristics. Similarly, the con-

We argue that these counterexamples can also be used
as the proxy for comparing performance of controllers
that are unsafe. We demonstrate this by comparing
the longest and deepest counterexamples for two dif-
ferent adaptive cruise controllers. We also evaluate our
approach on several linear hybrid system benchmarks.

Keeping to the motivation of extracting a variety of
counterexamples, we focus particularly on scenarios
where the safety specification is violated. Our evaluation
suggests that the cost of generating these counterexam-

ples while being less than the safety verification time,
is dependent on the duration of overlap between the
reachable set and the unsafe set.

Related Work: Counterexamples currently play very
important role in the domain of model checking. While
in the beginning, counterexamples were a mere side
effect of model checking, they were regarded as an
important artifact due to their practical relevance. Prin-
marily, they provide intuition to the system designer
about the reason why the system does not satisfy
the specification. More recently, techniques were de-
veloped to uncover deep bugs which would otherwise
take a long time to uncover [12,13]. The introduction
of Counter-Example-Guided-Abstraction-Refinement
(CEGAR) [15,16] changed the role of counterexamples
from a mere feature to an algorithmic tool. In CEGAR,
the counterexample acts as a primary guide to restrict-
ning the space of the possible refinements. In the domain
of automated synthesis, Counterexample Guided Induc-
tive Synthesis (CEGIS) framework [43,42], as the name
suggests, leverages counterexamples from verification
for inductive synthesis.

Generating specific type of counterexamples has been
an active research topic in model checking. In one of the
recent works [29], the authors provide techniques to
generate longest and deepest counterexamples for linear
dynamical systems. In the domain of hybrid systems,
many CEGAR based approaches pursue various notions
of counterexamples [25,19,17,7,6,36,22,39,41,46,26].
Most of them are restricted to the domain of timed and
rectangular hybrid systems. The current state of the art
tools such as SpaceEx [27] and HyLAA [9] spit out the
counterexample that violates the safety specification at
the earliest time and at the latest time respectively.

![Figure 1. Classical case of overshoot in stabilizing controllers.](image-url)
Counterexamples also play an important role in falsification techniques [24,21]. Instead of proving that the specification is satisfied, falsification tools like S-Talio [8] and Breach [20] search for an execution that violates the specification. Given a specification of Cyber-Physical System in Metric Temporal Logic (MTL) [30] or Signal Temporal Logic (STL) [33], falsification techniques employ a variety of techniques [35,40,48,18] for discovering an execution that violates the specification. Unlike the counterexamples given in this paper, the counterexamples returned by falsification techniques need not be the deepest or the longest counterexamples.

The approach presented in this paper bears some resemblance to the CEGIS based approach described in [38,37]. Here, the verification condition that the system satisfies an STL [38] specification is encoded as a mixed-integer linear program (MILP). If the specification is violated, one can investigate the results of MILP to obtain counterexamples. In [28], the authors extend the previous work and provide an intuition/reason for the system failing to satisfy the specification.

The rest of the paper is structured as follows. Preliminary definitions and background details regarding reachable set computation using simulations are stated in Section 1. Section 2, 3 and 4 describe approaches to generate the deepest, longest and robust counterexamples respectively. Application of counterexamples is explained using adaptive cruise controller in Section 5. The evaluation results of counterexample generation on various benchmarks are provided in Section 6. In Section 7, the authors discuss future directions that can be pursued based on the work presented here.

1 Preliminaries

States and vectors are elements in $\mathbb{R}^n$ are denoted as $x$ and $v$. The inner product of two vectors is denoted as $v_1^T v_2$. Given a sequence $seq = s_1, s_2, \ldots$, the $i^{th}$ element in the sequence is denoted as $seq[i]$. In this work, we use the following mathematical notation of a linear hybrid system.

Definition 1 A linear hybrid system $H$ is defined to be a tuple $(Loc, X, Flow, Inv, Trans, Guard)$ where:

$Loc$ is a finite set of locations (also called modes).
$X \subseteq \mathbb{R}^n$ is the state space of the behaviors.
$Flow : Loc \rightarrow \text{AffineDeq}(X)$ assigns an affine differential equation $\dot{x} = A_l x + B_l$ for location $l$ of the hybrid automaton.
$Inv : Loc \rightarrow 2^X$ assigns an invariant set for each location of the hybrid system.
$Trans \subseteq Loc \times Loc$ is the set of discrete transitions.
$Guard : Trans \rightarrow 2^n$ defines the set of states where a discrete transition is enabled.

For a linear hybrid system, the invariants and guards are given as the conjunction of linear constraints.

The initial set of states $\Theta$ is a subset of $Loc \times \mathbb{R}^n$, where second element in the pair is a conjunction of linear constraints. An initial state $q_0$ is a pair $(Loc_0, x_0)$, such that $x_0 \in X$, and $(Loc_0, x_0) \in \Theta$. The unsafe set of states is a subset of state space, $U \subseteq \mathbb{R}^n$.

Definition 2 Given a hybrid system and an initial set of states $\Theta$, an execution of the hybrid system $H$ is a sequence of trajectories and transitions $\xi_0, \xi_1, \xi_2, \ldots$ such that (i) the first state of $\xi_0$ denoted as $q_0$ is in the initial set, i.e., $q_0 = (Loc_0, x_0) \in \Theta$, (ii) each $\xi_i$ is the solution of the differential equation of the corresponding location $Loc_i$, (iii) all the states in the trajectory $\xi_i$ respect the invariant of the location $Loc_i$, and (iv) the state of the trajectory before each transition $a_i$ satisfies $Guard(a_i)$.

The set of states encountered by all executions that conform to the above semantics is called the reachable set. Linear dynamical systems can be considered as hybrid systems with one mode. The closed form expression for their trajectories is given as $\xi(t) = e^{A_l t} \xi(0) + \int_0^t e^{A_l (t-\mu)} B_l x d\mu$ where $A_l$ and $B_l$ define the affine dynamics of the mode $l$. Since this paper deals with finding counterexamples, we focus on counterexamples that can be generated using a specific simulation engine for hybrid systems. More specifically, we use the simulation engine that is described in [10]. This simulation engine also accounts for non-determinism induced due to discrete transitions. The closed form expression of a linear dynamical system execution involves matrix exponential; thus, we are better off using simulation engine that generates simulation as a proxy for an execution. For a unit time (also called the step), the hybrid system simulation starting from state $q_0$ is denoted as $\xi_H(q_0)$.

Definition 3 A sequence $\xi_H(q_0) = q_0, q_1, q_2, \ldots$, where each $q_i = (Loc_i, x_i)$, is a $(q_0)$-simulation of the hybrid system $H$ with initial set $\Theta$ if and only if $q_0 \in \Theta$ and each pair $(q_i, q_{i+1})$ corresponds to either: (i) a continuous trajectory in location $Loc_i$ with $Loc_i = Loc_{i+1}$ such that a trajectory starting from $x_i$ would reach $x_{i+1}$ after exactly unit time with $x_i \in Inv(Loc_i)$, or (ii) a discrete transition from $Loc_i$ to $Loc_{i+1}$ (with $Loc_{i-1} = Loc_i$) where $3a \in Trans$ such that $x_i = x_{i+1}$, $x_i \in Guard(a)$ and $x_{i+1} \in Inv(Loc_{i+1})$. Bounded-time variants of these simulations, with time bound $T$, are called $(q_0, T)$-simulations.

If the pair $(q_i, q_{i+1})$ corresponds to a continuous trajectory, $q_{i+1}$ is called the continuous successor of $q_i$, otherwise $q_{i+1}$ is the discrete successor of $q_i$.

While talking about the continuous or discrete behaviors of simulations, we abuse notation and use $x_i$, the continuous component of the state instead of $q_i$. 


Observations On Simulation Algorithm: We would like to make a few observations regarding the simulation algorithm that we have presented. First, the simulation engine allows the execution to make a discrete transition even when the invariant is violated. That is, if $x_i$ and $x_{i+1}$ are two successive states in the simulation, $x_{i+1}$ can make a discrete transition to the new mode even when $x_{i+1} \not\in \text{Inv}(\text{Loc}_i)$ as long as $x_{i+1} \in \text{Guard}(a)$. This is necessary to handle the common case where a guard is the complement of an invariant, and a sampled simulation jumps over the guard boundary during a single step. If these types of behaviors are not desired, the guard can be explicitly strengthened with the invariant of the originating mode.

If a guard is enabled and the invariant is still true, or if multiple guards are enabled, the simulation engine can make a non-deterministic choice. Consider that a one-dimensional system has two locations $l_1$ and $l_2$ such that $\text{Flow}(l_1) : \dot{x} = 1$, $\text{Inv}(l_1) : x \in [0, 50]$, transition $a = (l_1, l_2)$, and $\text{Guard}(a) : x \geq 45$. The initial set is $\Theta = (l_1, x \in [0, 5])$. After the guard is enabled in $l_1$ i.e., $x \geq 45$, the simulation engine, in a non-deterministic manner, can either take a discrete transition to $l_2$ or continue evolving in $l_1$ as long as its invariant is true. At $x = 50$, the trajectory can no longer continue to stay in $l_1$ as the invariant will be violated. Hence, at $x = 50$, the engine is forced to take the transition to $l_2$.

Second, the simulation engine given in Definition 3 does not check if the invariant is violated for the entire time interval, but only at a discrete time instance. Computationally, it is very hard to give certainty about whether a predicate was satisfied during an entire time interval, and hence we consider this to a valid assumption. Readers familiar with industrial simulation engines can relate this to a feature of not detecting zero crossings.

Third, the discrete jumps are only enabled at time instances that are multiples of the unit time. For discrete transitions that are a result of change in controller input that is driven by software, such an assumption is valid as one can consider the control system providing actuation values at discrete instances of time. This notion might not accurately represent the discrete transitions that are a result of environmental impact such as impulse responses. However, we still argue that such a notion of execution is useful because of two reasons. First, it is impossible (except for some very specific cases) to finitely represent the execution trace when the discrete transition is a result of the environment. The closest we can get to such representation is to consider executions that are defined in Definition 3. Second, by reducing the time step, one can get arbitrarily close to the execution that is a result of impulse response.

Finally, in order to avoid Zeno executions, the simulation engine forces the system should spend at least unit time in each mode.
1.1 Superposition principle, Generalized Stars, and Simulation-equivalent Reachable Set

We now present some of the building blocks in computing the reachable set. First is the superposition principle, second is the generalized star representation that is used for representing the set of reachable states and finally, the reachable set algorithm for a single mode and the simulation-equivalent reachable set that is returned by Algorithm in [10].

Definition 9 Given any initial state \( x_0 \), vectors \( v_1, \ldots, v_m \) where \( v_i \in \mathbb{R}^n \), scalars \( \alpha_1, \ldots, \alpha_m \), the trajectories of linear differential equations in a given location \( l \) always satisfy

\[
\xi(x_0 + \sum_{i=1}^{m} \alpha_i v_i, t) = \xi(x_0, t) + \sum_{i=1}^{m} \alpha_i (\xi(x_0 + v_i, t) - \xi(x_0, t)).
\]

An illustration of the superposition principle for two vectors is shown in Figure 2. We exploit the superposition property of linear systems in order to compute the simulation-equivalent reachable set of states for a linear hybrid system. Before describing the algorithm for computing the reachable set, we introduce the data structure called a generalized star that is used to represent the reachable set of states.

Definition 10 A generalized star (or simply star) \( S \) is a tuple \((c, V, P)\) 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 \( P: \mathbb{R}^n \to \{\top, \bot\} \) is a predicate.

A generalized star \( S \) defines a subset of \( \mathbb{R}^n \) as follows.

\[
[S] = \{ x \mid \exists \bar{\alpha} = [\alpha_1, \ldots, \alpha_m]^T \text{ such that } x = c + \sum_{i=1}^{m} \alpha_i v_i \text{ and } P(\bar{\alpha}) = \top \}
\]

Sometimes we will refer to both \( S \) and \([S]\) as \( S \). Additionally, we refer to the variables in \( \bar{\alpha} \) as basis variables and the variables \( x \) as orthonormal variables. Given a valuation of the basis variables \( \bar{\alpha} \), the corresponding orthonormal variables are denoted as \( x = c + V \times \bar{\alpha} \).

Similar to [10], we consider predicates \( P \) which are conjunctions of linear constraints. This is primarily because linear programming is very efficient when compared to nonlinear arithmetic. We therefore harness the power of these linear programming algorithms to improve the scalability of our approach.

Example 2 Consider a set \( \Theta \subset \mathbb{R}^2 \) given as \( \Theta^1 = \{ (x, y) \mid x \in [4, 6], y \in [4, 6] \} \). The given set \( \Theta \) can be represented as a generalized star in multiple ways. One way of representing the set is given as \( (c, V, P) \) where \( c = (5, 5), V = \{ [0, 1]^T, [1, 0]^T \} \) and \( P = \alpha_1 \leq -1 \land \alpha_2 \leq 1 \). That is, the set \( \Theta \) is represented as a star with center \((5, 5)\) with vectors as the orthonormal vectors in the Cartesian plane and predicate where the components along the basis vectors are restricted by the set \([-1, 1] \times [-1, 1] \).

Reachable Set Computation For Linear Dynamical Systems Using Simulations: We briefly describe the algorithm for computing simulation-equivalent reachable set for a single mode here, this is primarily done to present some crucial observations which will later be used in the algorithms for generating specific counterexamples. Longer explanation and proofs for these observations and algorithms is available in prior work [23,10].

At its crux, the algorithm exploits the superposition principle of linear systems and computes the reachable states using a generalized star representation. For an \( n \)-dimensional system, this algorithm requires at most \( n + 1 \) simulations. Given an initial set \( \Theta \equiv (c, V, P) \) with \( V = \{v_1, v_2, \ldots, v_m\} \) \( (m \leq n) \), the algorithm performs a simulation starting from \( c \) (denoted as \( \xi(c, 0) \)), and \( \forall 1 \leq j \leq m \), performs a simulation from each \( c + v_j \) (denoted as \( \xi(c + v_j, 0) \)). For a given time instance \( i \), the reachable set denoted as \( Reach_i(\Theta) \) is defined as \( \langle c_i, V_i, P \rangle \) where \( c_i = \xi(c, i) \) and \( V_i = \langle v'_1, v'_2, \ldots, v'_m \rangle \) where \( \forall 1 \leq j \leq m, v'_j = \xi(c + v_j, i) - \xi(c, i) \). Notice that the predicate does not change for the reachable set, but only the center and the basis vectors are changed.

An illustration of this reachable set computation is shown in Figure 3. Here, as the system is 2-dimensional, a total number of three simulations are performed - one from center \( c \), and one from each \( c + v_1 \) and \( c + v_2 \). The reachable set after time \( i \) is given as the star with center \( c'_i = \xi(c, i) \), basis vectors \( v'_1 = \xi(c + v_1, i) - \xi(c, i) \) and \( v'_2 = \xi(c + v_2, i) - \xi(c, i) \), and the same predicate \( P \) as given in the initial set.

\footnote{We abuse the notation \( \Theta \) to denote the initial set as well as its star representation.}
Figure 3. Illustration of the reachable set using sample simulations and generalized star representation. Notice that the predicate remains the same over time.

Simulation-Equivalent Reachable Set for Hybrid Systems with Linear Dynamics: The algorithm presented in [23] has been extended in [10] to compute the simulation equivalent reachable set for hybrid systems that accommodates for the invariants in each mode and the guard transitions for discrete mode jumps. This is achieved by introducing a new technique called invariant constraint propagation and dynamic aggregation and de-aggregation. Since our focus in this paper is to generate interesting counterexamples, we apply fully-deaggregated version of the reachable set computation algorithm and all reachable sets are given as stars.

Remark 1 For a discrete transition $a_i$ from mode $i$ to mode $i+1$, a set of constraints $A$ are propagated from a star $S_i \in \text{mode } i$ to $S_{i+1} \in \text{mode } i+1$ via $\text{Guard}(a_i)$ iff

$$A \triangleq S_i \cap \text{Guard}(a_i) \neq \emptyset \text{ and } A \subseteq S_{i+1}$$

As a consequence of propagation, the initial set for mode $i+1$ after the discrete transition $a_i$ is the full intersection of the reachable set $S_i$ with $\text{Guard}(a_i)$.

The reachable set algorithm computeSimEquivReach returns the reachable set in the form of a tree. The root node of the tree is the initial set $\Theta$. Each node in this tree is a generalized star $S_i$ of the form $S_i \triangleq (c, V, P)$ corresponding to the set of states visited at a discrete step $i$. Notice that the predicate in $S_i$ might be different from the predicate of the initial set $\Theta$ so as to accommodate the mode invariants and discrete transitions induced due to hybrid behavior. Each node in reach tree can have at most one continuous successor that corresponds to the evolution for unit time in the same mode, and multiple discrete successors each corresponding to the reachable set after the discrete transition. We denote this tree form of the reachable set as ReachTree.

The construction of ReachTree is illustrated in Figure 4.

Figure 4. Illustration of ReachTree construction. There are 6 modes. During a discrete transition, only predicates satisfying the guard are propagated.

Figure 5. Representation of a ReachTree. Discrete transitions are shown in red and continuous transitions in green. Each node has at most one continuous and as many discrete successors as the number of enabled guards.

The part of the system shown has 6 modes - $A$, $B$, $C$, $D$, $E$, and $F$. Inv $A$, Inv $B$, Inv $C$ are the invariants for modes $A$, $B$ and $C$ respectively. There are 4 nodes corresponding to mode $A$ where $A_{i+1}$ is the continuous successor of $A_i$, $1 \leq i \leq 3$. $A_1$ itself can be the root node or a successor - continuous or discrete - to some another node. A discrete transition $(X \to Y)$ from mode $X$ to mode $Y$ is active when its associated guard $(G_{X \to Y})$ becomes enabled, and constraints $X \cap G_{X \to Y}$ are propagated (Remark 1). Hence, during the transition from $A_2$ to $B_1$, predicates denoting the set $A_2 \cap G_{A \to B}$ are propagated. It means that the initial set $B_1$ is the full intersection of the reachable set $A_2$ and the associated guard $G_{A \to B}$.

As our reachable set construction algorithm explores all possible transitions, a node has as many discrete successors as the number of active discrete transitions, in addition to having at most one continuous successor. This
behavior translates into 3 scenarios: 1) only continuous-, 2) only discrete-, 3) continuous- as well as discrete-successors. For instance, $C_2$ has one continuous and 2 discrete successors as it satisfies the invariant $\text{inv} C$, and it has active transitions to both $E$ and $F$. $C_3$ does not have any discrete successor because there is no active discrete transition from $C_1$. In a similar fashion, $A_4$ has just one successor which is discrete because $A_4$ violates $\text{inv} A$ but $G_{A\rightarrow E}$ is enabled. The $\text{ReachTree}$ constructed in this manner is shown in Figure 5. The dashed transitions denote that there may or may not be a transition.

**Definition 11** Consider an initial set $\Theta$, bound $T$, and the simulation equivalent reachable set as $\text{ReachTree}$. Given a star $S_i \in \text{ReachTree}$, we call a sequence of stars $\sigma = R_1, R_2, \ldots, R_m$ a chain starting from $S_i$ if and only if $R_1 = S_i$ and $\forall 2 \leq j \leq m, R_j$ is (either continuous or discrete) successor of $R_{j-1}$.

**Remark 2** Given a star $S_i \triangleq \langle c_i, V_i, P_i \rangle$ in $\text{ReachTree}$ and its successor (either discrete or continuous) $S_{i+1} \triangleq \langle c_{i+1}, V_{i+1}, P_{i+1} \rangle$, observe that one has to either perform interaction with the invariant or with the guards for obtaining the predicate $P_{i+1}$. Hence $P_{i+1} \subseteq P_i$. Thus, given a valuation of $\alpha$ such that $P_{i+1}(\alpha) = \top$, it is true that all the stars that are the parents of $P_{i+1}$, the valuation of $\alpha$ is contained in the predicate. Additionally, one can use this valuation of basis variables to generate the trace starting from the initial set $\Theta$ to $P_{i+1}$. We call the procedure that generates this execution as $\text{getExecution}(\alpha, S_{i+1}, \text{ReachTree})$.

A side effect of the above observation is that all the trajectories that reach the star $S_{i+1} \triangleq \langle c_{i+1}, V_{i+1}, P_{i+1} \rangle$ would originate from the subset of the initial set $\Theta' \triangleq \langle c_0, V_0, P_{i+1} \rangle$.

**Assumptions:** Similar to the assumptions in our earlier work [10], we assume that ODE solvers give the exact result. While theoretically unsound, such an assumption is adopted due to its practicality. Second, we use floating-point arithmetic in our computations and do not track the errors by floating point arithmetic. A user concerned about the inaccuracy of numerical simulation can either use validated simulations [2] or compute the linear ODE solution as a matrix exponential to an arbitrary degree of precision. The algorithms presented are oblivious to the simulation engine used. We assume the initial set and unsafe region to be convex polytopes. However, generalized star provides flexibility to compute the reachable set even when the initial set is non-convex [23].

## 2 Deepest Counterexample

In this section, we will present the algorithm that would return the deepest counterexample for a safety specification and a direction. We illustrate the way to obtain the deepest counterexample using Figure 6.

![Figure 6. Illustration of the deepest counterexample in the direction of $v$.](image)

Suppose that in the $\text{ReachTree}$ computation, there are three stars $S_1, S_2$, and $S_3$ that overlap with the unsafe set $U$. Given a direction $d$, the procedure to compute the deepest counterexample would be the following. (1) For each of the stars $S_i$, compute the maximum depth $\text{depth}_i$ of star $S_i$ as $\text{max}(d^T x)$ with $x \in (S_i \cap U)$. (2) Select the star $S_j$ with maximum value of $\text{depth}_j$. (3) Extract the corresponding value of basis variables $\bar{\alpha}$ which achieves the maximum depth and extract the corresponding execution. The correctness of the algorithm trivially follows from Definition 6 and the correctness of the simulation-equivalent reachable set. The algorithm is presented formally in Algorithm 1.

**Algorithm 1:** Algorithm that computes the deepest counterexample with respect to a given direction $d$.

The main loop in lines 2-12 iterates through all the stars in the reachable set given as $\text{ReachTree}$ and selects the stars that overlap with the unsafe set $U$. The optimization problem for maximizing the value of the cost function $d^T x$ for the overlap with the unsafe set is generated in line 4, which is then solved in line 5. If the depth computed in line 5 is greater than the current maximum value (lines 6-10), then the maximum value is updated and the value of basis variables corresponding to the optimal solution as well as the current star are stored. In
3 Longest Counterexample

In this section, we will describe the algorithm for obtaining the counterexample that spends the longest contiguous time in the unsafe set. For this purpose, we leverage the generalized star representation and the property of the reachable set that is provided in Remark 2.

We illustrate the problem of finding the longest counterexample through Figure 7. Consider three consecutive stars $S_1$, $S_2$, and $S_3$ in the reachable set having overlap with the unsafe set as shown. If one picks the state $e_1 \in S_1$, then the post states of $e_1$, denoted as $e_2$ and $e_3$, do not lie in the unsafe set. However, if one picks $l_1 \in S_1$, then its post states, $l_2$ and $l_3$, lie in the unsafe set.

The key insight behind the generation of longest counterexample is that one has to select the appropriate state which visits the maximum number of contiguous overlaps between the unsafe set and the reachable set. In this instance, any state $x_1 \in S_1$ such that $x_1 \in S_1 \cap U$, with its successors $x_2, x_3$ such that $x_2 \in S_2 \cap U$ and $x_3 \in S_3 \cap U$ is the appropriate choice.

For finding such a state, we perform constraint propagation (similar to the invariant constraint propagation in [10]). That is, we identify the constraints $C$ on the basis variables $(\bar{\alpha})$ such that $\forall \bar{\alpha}$ such that $C(\bar{\alpha}) = \top$, we have, $x_1 = c_1 + V_1 \times \bar{\alpha} \in S_1 \cap U$, $x_2 = c_2 + V_2 \times \bar{\alpha} \in S_2 \cap U$, and $x_3 = c_3 + V_3 \times \bar{\alpha} \in S_3 \cap U$.

To extract these sets of constraints, we convert the unsafe set $U$ into the center and basis vectors of each of the stars $S_1$, $S_2$, and $S_3$. Thus, $S_i \cap U \triangleq \langle c_i, V_i, P_i, Q_i \rangle$. From Remark 2, we know that the set of states that reach $\langle c_i, V_i, P_i, Q_i \rangle$ originate from $\langle c_0, V_0, P_0, Q_0 \rangle$. Hence, the set of states that would visit all the intersections of the unsafe set should originate from $\langle c_0, V_0, P_1 \land Q_1 \land P_2 \land Q_2 \land P_3 \land Q_3 \rangle$. It follows that if the set of constraints $P_1 \land Q_1 \land P_2 \land Q_2 \land P_3 \land Q_3$ is satisfiable, then the trajectory corresponding to the basis variables that satisfy these constraints visits the unsafe set at all three consecutive time instances.

Building on the above discussion, the algorithm to compute the longest counterexample would iterate as follows. We first consider contiguous sequences of stars $S_1, S_2, \ldots, S_m$ that overlap with the unsafe set $U$. We then compute the set of constraints $C$ such that if $C$ is satisfiable, then there exists a trajectory that stays in the unsafe set for at least $m$ duration. We find the longest sequence of stars such that the corresponding constraint $C$ is satisfiable and provide the counterexample trace. This procedure is formally defined in Algorithm 2.

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{input} : Initial set $\Theta$, the simulation equivalent reachable tree $\text{ReachTree}$ and unsafe set $U$
\State \textbf{output}: Counterexample $ce$ that spends longest contiguous time in $U$
\State $\sigma_1 = -\infty$; $\text{lengthStar} \leftarrow \bot$; $ce \leftarrow \bot$
\For {each star $S_i$ in $\text{ReachTree}$}
\If {$S_i \cap U \neq \emptyset$}
\For {each chain $\sigma$ starting with $S_i$}
\State Transform $U$ into $\langle c_i, V_i, Q_i \rangle$ where $\sigma[i] \triangleq \langle c_i, V_i, P_i \rangle$;
\State $C_{\sigma} \leftarrow \bigwedge_{i=1}^{m} Q_i \land P_i$;
\If {if $C_{\sigma}$ is feasible and $|\sigma| > \text{length}_\infty$}
\State $\text{length}\_{\infty} \leftarrow |\sigma|$;
\State $\text{lengthStar} \leftarrow S_i$;
\EndIf
\EndIf
\EndFor
\EndIf
\EndFor
\EndIf
\EndIf
\end{algorithmic}
\end{algorithm}

\textbf{Algorithm 2:} Algorithm that computes the longest counterexample.

The algorithm proceeds as follows: the main loop (lines 2–14) iterates over all stars that have an overlap with the unsafe set $U$. The inner loop (lines 4–12) enumerates all the contiguous sequences $\sigma$ starting with $S_i$ and computes the set of constraints $C_{\sigma}$ for the sequence. If the constraints are feasible, then the valuation of the basis variables that satisfies these constraints and the star $S_i$ are stored. The length of the longest counterexample is also updated. In line 16, the execution corresponding to the longest counterexample is obtained using the valuation $\bar{\alpha}_{\text{len}}$.

\begin{theorem}
The execution returned by Algorithm 2 returns the longest counterexample.
\end{theorem}

\begin{proof}
We prove this by contradiction. Suppose
that for the given initial set $\Theta \defeq \langle c_0, V_0, P_0 \rangle$, the longest counterexample $\xi = x_0, x_1, \ldots, x_k$ spends duration $m$ in the unsafe set $U$. Consider that the states $x_j, x_{j+1}, \ldots, x_{j+m-1}$ in the execution $\xi$ lie in the unsafe set. Additionally, suppose that the execution returned by Algorithm 2 returns a counterexample of length strictly less than $m$.

From the soundness and completeness result of simulation equivalent reachability [10], we have that $\exists$ stars $S_j, S_{j+1}, \ldots, S_{j+m-1}$ in ReachTree such that $\forall j \leq r \leq j + m - 1, x_r \in S_r$. Therefore, it should be the case that $\forall r, j \leq r \leq j + m - 1, U \cap S_r \neq \emptyset$. Additionally, since the trajectory $\xi$ passes through $U \cap \Sigma_r$, it should be the case that $\xi \in \langle c_0, V_0, P_r \cap Q_i \rangle$ where $\Sigma_r \defeq (c_r, V_r, P_r)$ and $U \defeq (c_r, V_r, Q_r)$. Therefore, the constraint $C_\sigma$ that is computed for the sequence $\sigma = S_j, S_{j+1}, \ldots, S_{j+m-1}$ should be feasible and would be updated as the longest counterexample in lines 7-11. Which is a contradiction.

Analysis and Optimizations: In the ReachTree, a star can have at most one continuous successor and $d$ discrete successors where $d$ is the highest number of discrete transitions from any mode. If we consider the full tree with at least one step executed in each mode, the discrete transitions from any mode. If we consider the full tree with at least one step executed in each mode, the worst case possible number of sequences $\sigma$ of length $m$ would be $O((d + 1)^m)$. Hence, the worst case time for computing the length would be to perform $O(k^2(d + 1)^k)$ linear program optimizations. However, in practice, such worst case bounds are not observed. In almost all of our experiments, the duration for overlap is not the order of $k$, each star has at most one active transition, and the number of sequences to be handled is at most one or two sequences of the maximum length.

One of the optimizations that can be performed for eliminating certain counterexamples is to conduct something similar to a binary search. That is, given a sequence $S_j, S_{j+1}, \ldots, S_{j+k-1}$ starting from star $S_j$ that overlaps with $U$, we can check if $S_{j+\lfloor \frac{k}{2} \rfloor}$ overlaps with $U$. If there is no overlap, we can assert that the length of the longest unsafe sequence is less than $k/2$. However, this is a heuristic which may help in saving run time in some cases but not all.

4 Robust Counterexample

In this section, we will present the algorithm to obtain the robust counterexample. Recall that a counterexample starting from $x_r$ is said to be $\delta$-robust if and only if for all states $x \in B_\delta(x_r)$, there exists an unsafe execution starting from $x$. Informally, if we perturb the execution starting from $x$, by less than $\delta$, it remains unsafe. For obtaining this counterexample, we leverage the convexity property of reachable set.

For an unsafe star, the ideal robust counterexample is the center of the maximum ball inscribed inside the intersection of the star with the unsafe set. Since computing the maximum ball inscribed in a convex polytope is computationally hard [47,5], we, therefore, compute a proxy as some internal state of the polytope. In our case, this is the centroid of extreme points in each orthonormal direction. We illustrate the approach using Figure 8 where $x_{\text{ideal}}$ is the center of the maximum ball inscribed and $x_r$ is its proxy. The generalization to the case of multiple stars intersecting with the unsafe set for the given sequence is trivial.

Consider a star $S_1$ overlapping with the unsafe set. After obtaining the set $S_1 \cap U$, we find extreme points by optimizing (maximizing and minimizing) the cost function in each direction $x$ and $y$. Suppose these points are $x_{\text{low}}, x_{\text{high}}, y_{\text{low}}$ and $y_{\text{high}}$, respectively. Then the robust unsafe state is the centroid of these points.

$$x_r = (x_{\text{low}} + x_{\text{high}} + y_{\text{low}} + y_{\text{high}})/4$$

Remark 4 For each point $x$ in a convex set $X$, there exists $m \geq n+1$ points $x_1, \ldots, x_m \in X$ such that the point $x \in X$ is represented as their convex combination. That is, $\exists$ scalars $\beta_1, \ldots, \beta_m \geq 0$ with $\sum_{i=1}^m \beta_i = 1$ such that

$$x = \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_m x_m.$$
or a state not on the edge. In first case, \( \delta = 0 \), otherwise \( \delta \) is the euclidean distance from \( x_r \) to its nearest vertex, which is positive.

We use the longest contiguous sequence of unsafe stars from Section 3 to find the robust counterexample.

**Algorithm 3**: Algorithm that computes the robust counterexample such that a small perturbation yields a new counterexample.

In Algorithm 3, \( \sigma \) is the chain starting from \( \text{lengthStar} \) and has the length of the longest counterexample. \( C_\sigma \) represents the intersection of unsafe set \( U \) with stars in \( \sigma \). In main loop (lines 7-13), we formulate optimization problems to find the centroid \( \text{ce}_d \) in each orthonormal direction \( d \). In line 14, the robust counterexample \( \text{ce} \) is obtained as the centroid of all \( \text{ce}_d \) computed in the main loop. The user can provide additional directions for finding extreme points which, in turn, may result into a different robust counterexample.

**Runtime Analysis**: Since the robust counterexample is obtained with respect to the longest unsafe sequence, the worst case complexity is proportional to computing the longest counterexample, that is \( O(k^2 \cdot 2^k) \) as explained in Section 3. The heuristic approach based on conducting binary search applies here as well.

5 Analysis of Adaptive Cruise Controllers Using Counterexamples

In an adaptive cruise control system, the cars operate in autonomous manner. The leading car moves at a constant velocity; the following car slows down or speeds up automatically by sensing its velocity and the distance from the leading car. A control designer focuses on developing feedback controller for stabilizing this system. But a stable controller may not be safe for all initial states, where safety is defined as some minimum distance between these two vehicles or reasonable speed of the follower. As stated earlier, the objective is to evaluate the performance of controllers which violate the safety specification.

We provide an illustration of multiple adaptive cruise
Figure 11. Illustration of controllers’ performance on adaptive cruise control. $s$ is the distance between two vehicles and $t$ is the time. The unsafe specification is $0 \leq s \leq 2$ and the desirable condition is $27 \leq s \leq 30$. Although the system with controller II gets more close to the leading car, it tries to stabilize faster once it is at the desirable distance.

Consider the execution profiles after applying 3 different stable controllers are given. Since all 3 controllers are unsafe as shown, these executions can be used in evaluating their performance. For instance, controller 2 execution ventures the most in the unsafe region in the direction of vehicles’ movement. Although controller 1 execution is not the farthest in the unsafe region but it stays there for the longest time interval. Similarly, controller 3 execution is the most robust among all.

For simulation purpose, we pick adaptive cruise controller provided in [45]. We are unaware of the rationale behind the specific controller presented in [45]. However, given such a black-box scenario, our approach can be used to compare two controllers based on the safety specification. Consider the leading car is moving with a constant speed $v_f$, the follower’s velocity is $v$, its acceleration is $a$, and the distance between two vehicles is $s$. The differential equations for the automatic cruise control system used by the follower are as follows:

$$\begin{align*}
\dot{s} &= (v_f - v) \\
\dot{v} &= a \\
\dot{a} &= g_1 a + g_2 (v - v_f) + g_3 (s - (v + 10))
\end{align*}$$

Here, $g_1$, $g_2$ and $g_3$ are gain variables. The original system has $g_1 = -3$, $g_2 = -3$ and $g_3 = 1$. By changing the values of gain variables, a new controllers can be obtained. We pick $g_1 = -1$ to obtain a different controller for our experiments. The stable equilibrium of the system is $a = 0$, $v = v_f$, and $s = v_f + 10$. The designer can use standard tools like SOSTOOLS [3] to find Lyapunov functions for proving stability of these controllers. The original goal of adaptive cruise control is to always keep the follower at a safe distance from the leader. Because not every stable controller is essentially safe, conducting a quantitative analysis of such controllers would be of...
interest to the designer.

Given the initial set as \( s \in [2, 5] \), \( v \in [18, 22] \), \( v_f = 20 \), and \( a \in [-1, 1] \), the reachable sets computed by HyLAA for above mentioned two adaptive cruise controllers (ACC) are shown in Figure 10. Although both systems eventually stabilize to \( v = v_f = 20 \) or \( s = v_f + 10 \leq 30 \), they are unsafe with respect to the specification \( 0 \leq s \leq 2 \). Notice that the true safety specification is \( s \geq 0 \), but, during the design phase, one would want to work with specification that is conservative. As shown in Figure 10, the longest counterexample after applying controller I is of length 8 whereas its counterpart obtained from controller II has length 7. This means that controller II helps the system to recover faster from the unsafe region.

As an important side effect, our approach can also measure the extent to which a specification is satisfied. For instance, although \( 0 \leq s \leq 2 \) is certainly unsafe, the specification \( 2 \leq s \leq 5 \) is undesirable as it can possibly render the system unsafe if the follower speeds up or the leader slows down. The longest undesirable execution obtained from controller I is of length 13 while controller II gives the longest undesirable execution to be of length 2. Notice that the controller II slows down the system to a speed \( v \) also bad, therefore, the condition \( v \leq 0.5 \) is certainly unsafe, the longest undesirable execution obtained from controller I is of length 13 whereas its counterpart obtained from controller II is of length 7. This means that controller II helps the system to recover faster from the unsafe region.

Building on above discussion, one might change the specification \( 2 \leq s \leq 5 \) to be desirable \( (27 \leq s \leq 30) \) because the system is required to be eventually stable i.e., \( s = 30 \). We plot distance \( s \) against time \( t \) in Figure 11. The longest desirable execution obtained from controller I is of length 13 while controller II gives the longest undesirable execution to be of length 11. This re-emphasize that controller II makes the follower to get to the safe distance quicker as compared to controller I (Refer Figure 10).

To highlight that specifications over two different system variables may semantically differ, Figure 12 shows multiple specifications defined over \( v \). As the given system stabilizes when \( v = v_f = 20 \), the specification \( 19 \leq v \leq 20 \) is regarded as desirable and \( v > 20 \) as unsafe. Having the follower slowed down beyond a reasonable speed is also bad, therefore, the condition \( v \leq 15 \) is considered undesirable. The lengths of longest desirable executions indicate that the system with controller II obtains the desirable speed faster than that with controller I. However, looking at the deepest undesirable executions reveals that controller II slows down the system to a speed 10.145 while controller I helps maintaining it above 13.

This exercise underlines the need for a software tool that can assist the designer in not only evaluating different controllers but also understanding their merits when the specification changes. The analysis will enable them to take action(s) to improve respective controllers.

<table>
<thead>
<tr>
<th>Model</th>
<th>Initial Set</th>
<th>Unsafe Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball</td>
<td>( x \in [1.05, 0.95] )</td>
<td>( [0.2, 0.5] )</td>
</tr>
<tr>
<td>String</td>
<td>( u \in [-0.15, 0.15] )</td>
<td>( [5.6, 5.7] )</td>
</tr>
<tr>
<td>Two Osc</td>
<td>( s \in [1.5, 2.5] )</td>
<td>( [0.5, 1] )</td>
</tr>
<tr>
<td>Tanks</td>
<td>( y \in [1.1] )</td>
<td>( [-0.2, 0.1] )</td>
</tr>
<tr>
<td>Filtered</td>
<td>( s \in [0.2, 0.3] )</td>
<td>( y \in [-0.1, 0.1] )</td>
</tr>
<tr>
<td>Oscillator</td>
<td>( x_2 = 0.0 )</td>
<td>( y_1 = 0.0 )</td>
</tr>
</tbody>
</table>

\( v_{L_m} \in [0.4] \)

\( v_c \geq 2.5 \)

\( v_c \geq 2.2 \)

\( v_c \geq 2.0 \)

\( u = 0.0 \), \( t = 0.0 \)

6 Evaluation on Hybrid Systems Benchmarks

The proposed algorithms have been implemented in a Python based verification tool named HyLAA; although, some of the computational libraries used may be written in other languages. Simulations for reachable sets are performed using scipys odeint function, which can handle stiff and non-stiff differential equations using the FORTRAN library odepack’s lsoda solver. Linear programming is performed using the GLPK library, and matrix operations are performed using numpy. The measurements were performed on a system running Ubuntu 16.04 with an 3.00GHz Intel Xeon E3-1505M CPU with 8 cores and 32 GB RAM.
with step size 0.01 sec. The simulation for Filtered oscillator is carried out for maximum 100 time steps with step size 0.02 sec, and for Forward converter with step size $1 \times 10^{-6}$. The values of input variables in Two tanks benchmark are fixed to 0 which belongs to the actual interval $[0.1, 0.1]$. whereas in Forward converter, the input $(V_{in})$ is fixed to 100 from the interval $[98, 102]$. Most of these benchmarks are originally safe. Since our objective is to highlight counterexamples, we choose unsafe set in a manner that the reachable set intersects with the unsafe set at multiple time instances. We further adjust the size of unsafe set and observe that the intersection window of reachable set with the unsafe set differs proportionally. The initial set and unsafe set are given in Table 1.

For each benchmark, we increase the unsafe region size such that the number of stars intersecting with the unsafe set also increases. This, in turn, may lead to longer counterexamples. The increase in the number of unsafe stars translates directly into the counterexample generation time because every new star adds to the analysis time. The longest counterexample generation can be slower than the overall verification (Refer to III row in the Table 2). This happens because the combined number of constraints to be solved can become fairly large as explained in the algorithm in Section 3.

It is interesting to note that the length of counterexample is not necessarily same as the actual intersection duration of reachable set with the unsafe set. This is the direct consequence of our approach: if a system of constraints during certain time interval is not feasible, we prune the list and again check for its feasibility until we find a solution. In Figure 13, the duration of longest counterexample is different from the actual overlap duration. However, their duration is same in Figure 14.

Another observation is that the variations in the unsafe set size as well as depth direction can provide different counterexamples (Table 3). The time taken for generating deepest counterexample is much less compared to that of the longest one. The reason being we need to scan through the list of unsafe star only once to find the star with maximum depth.

The reader interested in evaluation results for regular linear dynamical systems can refer to [29].

7 Conclusions and Future Work

In this paper, we provided approaches for generating various counterexamples based on metrics such as length, depth and robustness. Our approach relies on a simulation based reachable set computation method for linear hybrid systems. Linear constraints based star representation significantly simplifies our counterexample generation mechanism. We also observe that the variations in unsafe set size and optimizing direction may generate different counterexamples. The proposed work finds its merit in the development of template based techniques for the refinement of initial and unsafe sets. We demonstrated the applicability of these approaches for comparing the performance of two adaptive cruise controllers. Additionally, we evaluated them on several hybrid systems benchmarks and presented our observations on their scalability and performance.

As the next step, we are interested in exploring a counterexample guided controller synthesis framework that leverages these various counterexamples. The counterexample guided inductive synthesis (CEGIS) approach requires to first find a stable feedback controller. Then verification is performed to either prove safety or alternatively find a counterexample. This process is repeated until a valid controller is obtained. The measures such as distance, duration or robustness can be used in determining the validity and merit of a controller during synthesis. We hope that such CEGIS approach would be useful for synthesizing a controller with both safety and stability specification.
<table>
<thead>
<tr>
<th>Model</th>
<th>Dims, Unsafe Set</th>
<th>Longest Counterexample</th>
<th>Actual Inter. Duration</th>
<th>LCE Duration</th>
<th>Verification Time (sec)</th>
<th>LCE Gen Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball</td>
<td>2, 2 (ext, freefall)</td>
<td>(ext, freefall)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>SU [-0.9507 -0.15]</td>
<td>[18 20][21 23]</td>
<td>[18 20][21 23]</td>
<td>0.25</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU [-1.0191 -0.15]</td>
<td>[12 20][21 29]</td>
<td>[13 20][21 29]</td>
<td>0.33</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LU [-0.9618 -0.15]</td>
<td>[7 20][21 37]</td>
<td>[7 20][21 37]</td>
<td>0.38</td>
<td>0.22</td>
<td></td>
</tr>
<tr>
<td>Two</td>
<td>2, 4 (loc3, loc1)</td>
<td>(loc3, loc1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tanks</td>
<td>SU [1.763 1.1]</td>
<td>[21 26][27 40]</td>
<td>[24 26][27 40]</td>
<td>15.24</td>
<td>0.40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU [2.407 1.077]</td>
<td>[16 28][33 78]</td>
<td>[-][34 77]</td>
<td>17.78</td>
<td>5.25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LU [2.497 1.1]</td>
<td>[7 30][31 81]</td>
<td>[15 30][31 81]</td>
<td>20.55</td>
<td>11.46</td>
<td></td>
</tr>
<tr>
<td>Filtered</td>
<td>6, 4 (loc3, loc4)</td>
<td>(loc3, loc4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oscillator</td>
<td>SU [0.2 0.092 0...]</td>
<td>[1 23][50 55]</td>
<td>[1 23]</td>
<td>7.07</td>
<td>2.14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU [0.2 0.0895 0...]</td>
<td>[1 34][44 54]</td>
<td>[1 34]</td>
<td>7.98</td>
<td>5.41</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LU [0.2 0.099 0...]</td>
<td>[1 49][52 66]</td>
<td>[1 49]</td>
<td>8.20</td>
<td>11.09</td>
<td></td>
</tr>
<tr>
<td>Forward</td>
<td>5, 5 (loc1, loc2, loc5)</td>
<td>(loc1, loc2, loc5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Converter</td>
<td>SU [0 0.399 0.223 0 0]</td>
<td>[8 11][12 16][17 18]</td>
<td>[8 11][12 16][17 18]</td>
<td>7.40</td>
<td>0.39</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU [0 0.4 0.2928 0 0]</td>
<td>[6 11][12 16][17 22]</td>
<td>[7 11][12 16][17 22]</td>
<td>7.79</td>
<td>0.83</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LU [0 0.4 0.355 0 0]</td>
<td>[5 11][12 16][17 25]</td>
<td>[6 11][12 16][17 25]</td>
<td>8.84</td>
<td>1.32</td>
<td></td>
</tr>
</tbody>
</table>

Table 2

**Longest Counterexample.** Dims is the no. of dimensions (system variables), Modes is the number of system locations, SU, MU, LU are variations of the unsafe set - Small, Medium and Large, as shown in Table 1. Longest Counterexample is a point in the initial set, simulation from which stays for the longest contiguous time in the unsafe set. \( x_i \) represents all the variables whose values are not explicitly given. Actual Inter. Duration is the mode-wise ordered sequence of discrete time step intervals when reachable set intersects with the unsafe set. LCE Duration is the interval for the longest counterexample. Verification Time is the time Hylaa takes for verification, LCE Gen Time is the time it takes to generate the longest counterexample.

**References**


Table 3

Deepest and Robust Counterexamples. The rows for each benchmark correspond to the size-variant (SU, MU and LU) of the unsafe set shown in Table 1. Direction is the direction in which the depth of the counterexample is obtained. For instance, in a 2-dimensional system \((x, v)\), the direction \(x_2 = 1\) represents a vector \([0, 1] \in \mathbb{R}^2\). DCE Gen Time is the time Hylaa takes to generate the deepest counterexample and RCE Gen Time is the time taken for generating the robust counterpart. As we first obtain the LCE predicates to compute the robust counterexample, RCE Gen Time is inclusive of LCE Gen Time from Table 2. Also, varying the unsafe set size may yield different deepest and robust counterexamples.

<table>
<thead>
<tr>
<th>Model</th>
<th>Deepest Counterexample</th>
<th>Direction</th>
<th>Depth</th>
<th>Verification Time (sec)</th>
<th>DCE Gen Time (sec)</th>
<th>Robust Counterexample</th>
<th>RCE Gen Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ball String</td>
<td>([-1.05 \ 0.0691])</td>
<td>(x_2 = 1)</td>
<td>6.0</td>
<td>0.25</td>
<td>0.00</td>
<td>([-0.956, 0.0])</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>([-1.045 -0.15])</td>
<td>(x_2 = 1)</td>
<td>7.0</td>
<td>0.33</td>
<td>0.00</td>
<td>([-1.019, -0.146])</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>([-1.035 -0.15])</td>
<td>(x_1 = 1)</td>
<td>0.8</td>
<td>0.38</td>
<td>0.01</td>
<td>([-0.956, 0.0])</td>
<td>0.24</td>
</tr>
<tr>
<td>Two Tanks</td>
<td>([1.8995 \ 1.0646])</td>
<td>(x_2 = 1)</td>
<td>0.1</td>
<td>15.24</td>
<td>0.02</td>
<td>([1.677, 1.016])</td>
<td>0.40</td>
</tr>
<tr>
<td></td>
<td>([2.406 \ 1.0282])</td>
<td>(x_2 = 1)</td>
<td>0.3</td>
<td>17.78</td>
<td>0.10</td>
<td>([1.731, 1.003])</td>
<td>5.27</td>
</tr>
<tr>
<td></td>
<td>([2.225 \ 1])</td>
<td>(x_1 = 1)</td>
<td>1.9</td>
<td>20.55</td>
<td>0.12</td>
<td>([2.326, 1.002])</td>
<td>11.50</td>
</tr>
<tr>
<td>Filtered</td>
<td>([0.3 \ 0.0987 0...])</td>
<td>(x_6 = 1)</td>
<td>0.566</td>
<td>7.07</td>
<td>0.08</td>
<td>([0.258 \ 0.0867 \ 0...])</td>
<td>2.17</td>
</tr>
<tr>
<td>Oscillator</td>
<td>([0.3 \ 0.0987 0...])</td>
<td>(x_6 = 1)</td>
<td>0.566</td>
<td>7.98</td>
<td>0.21</td>
<td>([0.258 \ 0.0852 \ 0...])</td>
<td>5.44</td>
</tr>
<tr>
<td></td>
<td>([0.3 \ 0.0987 0...])</td>
<td>(x_3 = 1)</td>
<td>0.6187</td>
<td>8.20</td>
<td>0.22</td>
<td>([0.258 \ 0.0826 \ 0...])</td>
<td>11.12</td>
</tr>
<tr>
<td>Forward</td>
<td>([0.4 \ 0.4 \ 0 \ 0])</td>
<td>(x_3 = 1)</td>
<td>2.9056</td>
<td>7.40</td>
<td>0.01</td>
<td>([0.2 \ 0.399 \ 0.231 \ 0\ 0])</td>
<td>0.40</td>
</tr>
<tr>
<td>Converter</td>
<td>([0.0 \ 0.4 \ 0 \ 0 \ 0])</td>
<td>(x_2 = 1)</td>
<td>0.3003</td>
<td>7.79</td>
<td>0.02</td>
<td>([0.2 \ 0.396 \ 0.346 \ 0\ 0])</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>([0.0 \ 0.4 \ 0 \ 0 \ 0])</td>
<td>(x_3 = 1)</td>
<td>2.9056</td>
<td>8.84</td>
<td>0.02</td>
<td>([0.2 \ 0.397 \ 0.378 \ 0\ 0])</td>
<td>1.36</td>
</tr>
</tbody>
</table>


