Final Year Project Report

PART I: Falsification of Hybrid Systems

PART II: Generation of Valid Counterexamples in Hybrid Systems

Amrita Ray
Roll Number: 2018SMCS001
Registration Number: 201802050101025

July 7, 2020

Supervisor : Dr. Rajarshi Ray

Int.Master's-PhD/Master's Program in Mathematical and Computational Science

School of Mathematical and Computational Sciences

Indian Association for the Cultivation of Sciences
Kolkata-700032

CONTENTS

PART 1: Falsification of Hybrid Systems

Third Semester Report

PART 1: FALSIFICATION OF HYBRID SYSTEMS

I. INTRODUCTION

A hybrid system is a system which exhibits both continuous and discrete dynamics. Hybrid systems are used widely in modelling real-life system where we have instantaneous discrete transitions and continuous evolution of the system over time between these transitions, and finds applications in a wide variety of fields like railways, robotics, and aviation, to mention a few. Transition between modes is triggered when the continuous dynamics, often modelled by differential equations, reaches some threshold value.

Since these models are used in devices which are used in practical scenarios, stringent safety conditions must be ensured during the design phase of the device. By *safety*, it is meant that the system reaches no pre-specified error state. A hybrid system is considered safe if for all initial states no unsafe states are reachable. For safety checking, two notions: *Verification* and *Falsification* are used. Here, we discuss a falsification technique known as *trajectory splicing* in detail.

A. Hybrid Automata

A Hybrid automaton [1] is a mathematical model of a hybrid system. It is described formally as a tuple $(L, X, Inv, Init, Flow, Trans)$ where:

- L : is the set of locations of the hybrid system. This is a finite set, $\{l_1, l_2, \ldots, l_n\}$. These represent the control modes of the hybrid automaton.
- X : is the set of continuous real-valued variables. This represents the continuous dynamics of the automaton. Each location is associated with a set of continuous dynamics.
- Inv : This is a subset of $\mathbb{R}^n$ and the domain of the continuous variables. This constrains the possible values of X, at a particular location. This implies that all the trajectories of a hybrid automata should lie in the envelope.
- $Init$: This is the set of initial states. A tuple belonging to this set would be (l_{init}, C_{init}), where $l_{init} \in L$ and $C_{init} \subseteq Inv$.
- $Flow$: The flow is a real-valued function with models the evolution of the continuous variables, x_i's, with respect to time. Each location is associated with a flow function. The flow function is given as an ODE, $\dot{x} = f(x)$.
- $Trans$: This is the set of discrete transitions or 'jumps' between control modes. A transition, denoted by δ takes a state, $s_i \in S$ to $s_j \in S$, provided some transition conditions known as guard conditions are satisfied.

A state s_i of a hybrid automaton is given by a tuple (l_i, x_i). The set of all such s_i is called the *state space* of the HA. A state transition occurs when x_i lies in set which triggers the transition, called the guard set, denoted by G. For the continuous vectors of the s_i, the transformation is given by $x' = Mx + b$. M(δ) is the transformation matrix. We will consider flows of linear differential equation form, that is,

$$\dot{x} = Ax + u$$

where, M is a $n \times n$ matrix, $u, x, b \in \mathbb{R}^n$ and A $\in \mathbb{R}^{n \times n}$.

The behaviour of a hybrid system is formally described by *runs* of a hybrid automaton. This is a sequence of alternating time intervals and transitions, in which the continuous part of the automaton evolves in a particular location and x is updated to $x = M(x)$, after which at $t = \tau$, x satisfies a guard condition and immediately takes a discrete transition δ.

A run of a hybrid automaton is given by:

$$(l_0, x_0) \xrightarrow{\tau_0} (l_0, y_0) \xrightarrow{\delta_0} (l_1, x_1) \xrightarrow{\tau_1} (l_1, y_1) \ldots$$

$$\xrightarrow{\delta_{n-1}} (l_n, x_n) \xrightarrow{\tau_n} (l_n, y_n)$$

$\forall$ i, the following conditions should be satisfied:

1) $(l_0, x_0) \in Init$
2) $\forall t \in [0, \tau_i)$
3) $flow_{l_i}(x_i, t) \in Inv(l_i)$
4) $flow_{l_i}(x_i, \tau_i) = y_i$
5) $y_i \in G(\delta_i)$
6) $x_{i+1} = M(y_i)$

This can be a finite or an infinite sequence. A trajectory for a location is obtained by applying some input control signal, k, to the state s, for a time duration, τ, which is called the dwell time of the system in that location. During, the duration $[0, \tau)$, the guard conditions are never satisfied, and the trajectory lies entirely in the invariant. At $t = \tau$, the guard conditions are satisfied, and

the system immediately transistions to the next discrete state.

B. Verification and Falsification

1) Verification

Verification problems [1] [6] on hybrid systems are most easily understood when they are formulated as a reachability analysis problem on the state space. There are two ways to look at verification. One way is to say that the all states that are reachable from the initial set of states, do not intersect with the unsafe set of states. If we look at it backwards, safety can be guaranteed by showing that the set of states that reach the unsafe set is unreachable from the initial state set. Verification of a system requires that a set of properties, termed *safety properties*. If this set of properties are satisfied, then a system is termed as safe. One of the challenges of verification is defining this set of properties and proving the completeness of the property set, that is, ensuring coverage. By completeness, the mathematical property of completeness of a set is not implied over the set of properties. Rather completeness has been used here in the colloquial sense, that is, has the system passed verification for all the undesirable situations that can arise in a real system? Coverage, here, again means, has the verification of the system covered for all the possible practical situations? This is difficult to guarantee when it comes to designs of embedded controllers for any real-life system.

Another difficulty of verifying any hybrid system, is that any hybrid system that models a real-life system will have uncountable number of distinct states in the continuous state space. Doing a reachability analysis is impossible in these cases. One of the methods is to produce a conservative over-approximation of the system. A commonly used over-approximation for getting the accurate set of reachability analysis is known as flow-pipe generation. Emptiness of the intersection of flow-pipe with the unsafe states implies that the system is safe.

Using over-approximations is also problematic. In the case the system is unverifiable, that is, it cannot be proved if the unsafe states are reachable, it cannot be said whether the system actually violates the property or the verification tool fails to check safety because of the over-approximations. This is shown in Fig. 1.

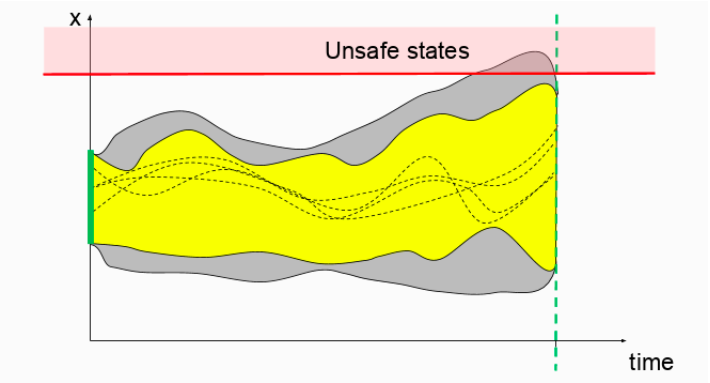
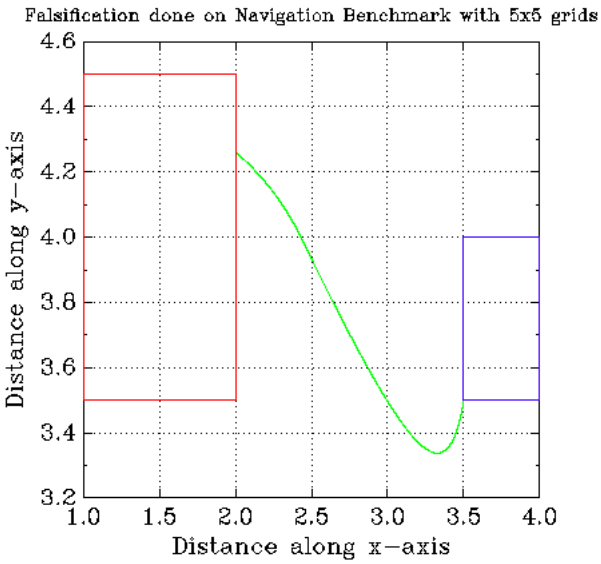


Fig. 1: Here, this is a system that moves a certain distance over time, and we have a set of initial states given by the solid green line. The figure shows a snapshot of the system at time=t, given by the dotted green line. The unsafe state is a certain height. The verification algorithm has computed an over-approximation of the actual reachable set(the yellow area), given the grey envelope. The over-approximated boundary gives an intersection with the unsafe state, though the actual reachable set does not. The system is verifiable, but proved otherwise by the tool, making it a faulty judgement. Figure used from Presentation on Falsification of Hybrid Systems Using Symbolic Reachability and Trajectory Splicing by Dr. Rajarshi Ray [4]

2) Falsification

Falsification [1] [6] is the more practical approach to checking the safety of a system. This is an approach that works by searching for errors rather than prove correctness, that is, it initially assumes that the system is unsafe and tries to prove this assumption. Falsification problems, are search problems and ask the question: *Is it possible to find a trajectory from an initial set of states to an unsafe set of states?*

These trajectories, if and when they exist, are called counterexamples. Obtaining concrete counterexamples not only are essential to safety check a model, they are also important in finding out bugs in the model. Integrating verification techniques with falsification helps in generation of such concrete counterexamples. The following figure shows a falsification check done on a navigation benchmark model. The navigation benchmark [1] models



h

Fig. 2: The blue box, shows the initial set of states of the model, and the red box shows the specified state of unsafe states. The green line shows the violating trajectory which leads the system from the initial state to the unsafe condition. The falsification on this navigation benchmark system passes; the system is now proved to be unsafe. Image generated using XSpeed.

the motion of an object on a plane which has been segmented into 1×1 squares. Each square has its own set of dynamics, which represent the continuous dynamics of this system. The dynamics changes discretely as the object moves from one square to another, so the boundaries of the square are the guard conditions.

In the figure given, Fig. 2, The blue box, shows the initial set of states of the model, and the red box shows the specified state of unsafe states. The green line shows the violating trajectory which leads the system from the initial state to the unsafe condition.

II. PREVIOUS WORK

A. *Using trajectory splicing NLP approach-Zutshi et al*

The paper [6] takes the idea of falsification for model checking, and discusses a technique called *trajectory splicing* for finding concrete trajectories to unsafe states. The approach starts with a candidate sequence of disconnected trajectory segments, where each segment lies within a distinct location of the system. These do not yet form a concrete trajectory as there are gaps between the endpoints of one segment and the beginning of another. The trajectory splicing approach uses local optimisation at each of these gaps, by casting these gaps as objective functions of an optimisation problem to minimise the distance between two segments, until the distance becomes close to zero. If such a minimisation is possible, then candidate segments can be spliced to become a concrete trajectory. If not, then the set of candidates is rejected, and another candidate set of segments is picked. It is to note that the candidate sequence of disconnected segments is picked manually, and supplied to the optimiser routine.

The paper discusses and sets up a general optimisation framework to perform the concrete counterexample search by the trajectory splicing method. Since optimisation for hybrid systems is often non-convex and non-linear, the paper also discusses an approach to partitioning such a problem into independent sub-problems. We discuss the approach of setting up the main optimisation routine in more detail below.

Problem Setup: We have a hybrid automaton A, and $\phi : (l_f, X_f)$, which is the safety property of interest. An abstract counterexample in the form of a sequence of modes and transitions is given:

$$C :< l_0, \delta_0, l_1, \delta_1 \ldots \delta_{n-1}, l_n >$$

where $l_f = l_n$. The goal is now to find a concrete counterexample to this property. The concrete counterexample can be written as:

$$(l_0, x_0) \overset{\tau_0}{\rightarrow} (l_0, y_0) \overset{\delta_0}{\rightarrow} \ldots$$

$$\overset{\delta_{n-1}}{\rightarrow} (l_n, x_n) \overset{\tau_n}{\rightarrow} (l_n, y_n)$$

The search is now performed to obtain a candidate set of trajectory segments, which will yield a *segmented trajectory*.

A segmented trajectory H is a sequence of $N+1$ trajectory segments $< h_1, \ldots, h_n >$ where $h_i : (l_i, x_i) \overset{\tau_i}{\rightarrow} (l_i, y_i)$. The mode sequence is specified by the abstract counterexample, and

1) $(l_0, x_0) \in S$, where S is the state space of the hybrid automaton.
2) $(l_f, y_f) \in S_f$, where S_f is the set of final states of the hybrid automaton
3) $y_i \in G(\delta_i)$ for $i = (0, \ldots, N-1)$

Defining the cost function: Let d be a metric on the continuous states of the hybrid automaton. Then the cost of the segmented trajectory H is:

$$COST(H) = \sum_{j=0}^{N-1} d(M(\delta_j)(y_j), x_{j+1})$$

, where cost of any two consecutive trajectory segments, h_j, and $h_j + 1$:

$$COST(h_i, h_j) = d(M(\delta_j)(y_j), x_{j+1})$$

Since, $d(x, y) \geq 0 \ \forall x, y$, $COST(H) \geq 0$. The goal is to find a segmented trajectory where the cost is minimum, and since the function is lower bounded by 0, this is effectively searching for a segmented trajectory where the cost is 0. So, given a hybrid automaton, a safety property, a sequence of modes and transistions, and SegTraj, a set of segmented trajectories, we want to search for a segmented trajectory H, such that

$$\min COST(H), s.t. S \in SegTraj$$

This leads to the set up of the optimisation problem, which now can be framed as follows:

$$\min_{x_0, \tau_0, \ldots, x_N, \tau_N} \sum_{i=0}^{N-1} COST(s_i, s_{i+1})$$

subject to,

$$COST(s_i, s_{i+1}) = d(y_i, x_{i+1}), \ i = (0, \ldots, N-1)$$
$$\tau_{min} \leq \tau_i \leq \tau_{max}, \ i = (0, \ldots, N)$$
$$(l_0, x_0) \in Init$$
$$(x_i, y_i) \in Inv, \quad i = (0, \ldots, N)$$
$$flow_{l_i}(x_i, \tau_i) = y_i, \quad i = (0, \ldots, N)$$
$$flow_{l_i}(x_i, t) \in Inv(l_i), \quad \forall t \in [0, \tau_i], i = (0, \ldots, N)$$
$$y_i \in G(\delta_i), \quad i = (0, \ldots, N-1)$$

If the cost function is evaluated to 0, then a violating trajectory exists for this particular safety property. Otherwise, the cost function evaluates to a strictly positive value, implying no such concrete trajectory exists, and hence the system is safe.

B. Using constrained NLP approach-Bogomolov et. al

The paper [1] builds on the idea of trajectory splicing to find a counterexample for a safety property. It addresses two problems in the previous approach:

1) The candidate sequence is selected by observing simulations of the system. While this is feasible for relatively small hybrid systems, it becomes impractical for large and complex hybrid systems. Moreover, since simulations are inherently incomplete, it may not possible to observe a violating trajectory even if it exists.
2) The invariant set, in the previous approach, was decided by using information on the bounds of the system model. Since the reachable set is a subset of this invariant set, using bounding information to design the invariant set gives coarser invariant sets. In particular, if the bounding information is partially missing, then the optimisation routine might get trapped in a local minima and cost more iterations to converge.

In the approach proposed in this work , ideas from verification tools to do reachability analysis and compute the reachable set is used to improve the falsification routine. This is done by the following methods:

1) Reachability analysis is used to automatically produce a candidate sequence of trajectory segments. As mentioned previously, in the former approach, the process of picking a candidate set of segmented trajectories was not automated, and hence some knowledge of the system is required to select this sequence. This process automates the selection, and hence no knowledge of the model is required.
2) The constraints that are implications of the reachability analysis are used to prune the search space of the optimiser. This ensures a faster convergence of the routine.
3) After the finding a concrete trajectory, it is put through a validation loop, so that the trajectory strictly lies inside the invariant set.

An algorithmic view of the falsification process is shown below:

Given a hybrid automaton HA, and set of unsafe states, S_B,

1) Call the reachability analyser.
2) Get a candidate sequence, from the reachable set computed. If none exist, terminate.
3) Call the optimizer and input the candidate sequence.
4) If the cost is 0, then a concrete violating trajectory has been obtained. If not, reject this sequence and go back to step 2.
5) If a concrete trajectory is found, call the validation routine, and input the sequence. If validation returns true, then output the sequence and declare the system unsafe.
6) If the validation returns false, call the refinement routine, and input the trajectory. After refinement, go back to step 3.

Symbolic Reachability Analysis: This section briefly discusses the reachability analysis.

A state (l, x) is reachable if there is a run of the hybrid automaton, such that $(l_0, x_0) \in Init$ and $(l_n, x_n) = (l, x)$. A symbolic state s is a tuple (l, C) s.t $l \in L, C \subseteq Inv(l)$, and reachable states can be expressed as a union of symbolic states. Here, S_B denotes the set of unsafe symbolic states, and defines the unsafe states of the hybrid automaton, HA. Then, from this, it can be concluded that, if $\exists s_B \in S_B$ s.t s_B is reachable, the system is declared unsafe.

For the reachability analysis, a BFS is carried out to the depth N. The BFS produces what is called an exploration graph (S, V), where S is the set of symbolic states, and V is the set of edges $(s, s') \in S \times S$ and each edge represents a transition $\delta(s, s')$. A run of a hybrid automaton matches a path $s_0, s_1, \ldots, s_N = (l_0, C_0), \ldots, (l_N, C_N)$ in the exploration graph, if for $i = (0, \ldots, n)$, $x_i \in C_i$ and $\delta_i = \delta(s_i, s_{i+1})$.

The symbolic reachability analysis algorithm is as follows:

1) Initialize a queue of set of states that belong to Init.
2) For each set of states, compute the continuous set of states reachable by time elapse.
3) For each outgoing transition, compute the symbolic set of states reachable in the new location
4) Put back all the obtained sets of states in the queue, unless they are already present.

Abstract and Concrete counterexamples: An abstract counterexample in the exploration graph (S, V) is a sequence

$$s_0, \delta_0, \ldots, \delta_{N-1}, s_N$$

where $s_i \in S$ and $\delta_i = \delta(s_i, s_{i+1}) \in V$, such that $s_0 \subseteq Init$ and $s_N \bigcap S_B \neq \phi$. An abstract counterexample is also concrete if the path has a matching run in the hybrid automaton.

Given this, the falsification task now can be framed as: Given an HA and a set of unsafe symbolic states S_B, a falsification task (HA, S_B) is to find a concretizable counterexample of HA.

Search Space pruning using additional constraints: The search space of the optimisation problem can be restricted with three constraints:

1) From the reachability analysis, we have the constraint $C_i \subseteq Inv(l_i)$, and use it to restrict the search space of the optimisation problem.
2) *Constraints on the starting point x_i:* The starting point must always lie in the reachable set, and hence always belongs to Ω_0 which is the over-approximation of the reachable set over the time interval $[0, \Delta t]$. So, the constraint $x_i^{min} \leq x_i \leq x_i^{max}, \forall i : 0 \leq i \leq n,$, is added, where

   $$x_{i,d}^{min} = \min_{x \in \Omega_0} \{proj_d(x)\}, d \in Z, d \in [1, n]$$

   and

   $$x_{i,d}^{max} = \max_{x \in \Omega_0} \{proj_d(x)\}, d \in Z, d \in [1, n]$$

3) *Constraints on the end points y_i:* If l_i is not the location of the last segment, then the endpoint of the segment must satisfy the guard condition to take the next discrete transition. Additionally, the set of reachable states in l_i is a subset of C_i, and hence, the constraint $y_i \in G(\delta_i) \bigcap C_i$. If l_i is the location of the last segment, then the endpoint of the segment must lie in the unsafe state set, $S_B = (l_B, C_B)$. So, for $i = N$, the constraint on y_i becomes $y_i \in C_B \bigcap C_N$
4) *Constraints on dwell time*: From the reachability analysis, it can be deduced that endpoint y_i of a trajectory must lie in the $G(\delta_i) \bigcap C_i$. To add finer constraints on dwell

times, for the location i, we take the time projection of maximum of these y_i's (if the dwell time is more than this, the segment crosses the guard and hence takes a discrete transition) and the minimum of y_i (If the dwell time of a segment is less than this, the segment never reaches the guard set). This results in constraint $\tau_i^{min} \leq \tau_i \leq \tau_i^{max}, \quad \forall i : 0 \leq i \leq n-1$, is added, where,

$$\tau_i^{min} = \min_{x \in C_i \bigcap G(\delta_i))} \{proj_{d'}(x)\}$$

$$\tau_i^{max} = \max_{x \in C_i \bigcap G(\delta_i))} \{proj_{d'}(x)\}$$

Here, in the notation, it is written $x \in C_i \bigcap G$ and $proj_{d'}(x)$ because, even though, we call the endpoints y_i, these lie on the x-axis. The y-axis gives dwell times.

Setting up the optimisation problem: As can be seen, the cost function is non-linear, so the optimisation over this function is not very simple. One of the ways to perform it relatively efficiently is to turn some of the constraints into soft constraints, and add it to the cost function. Here, the constraints on dwell time, and initial points are kept as hard constraints, and the constraints on the endpoints are added to the cost function as penalty terms. Here, we take dp as the metric to calculate the distance between a point, p and a polyhedron, P and define,

$$dp(p, P) = 0, \quad \forall p \in P$$
$$dp(p, P) > 0 \quad otherwise \tag{1}$$

The minimisation problem now becomes:

$$\min_{x_0, \tau_0, \ldots, x_N, \tau_N} \sum_{i=0}^{N-1} COST(s_i, s_{i+1}) + dp(y_n, C_B)$$

subject to,

$$\tau_i^{min} \leq \tau_i \leq \tau_i^{max}, \ \forall i : 0 \leq i \leq n-1,$$

$$x_0 \in C_0,$$

$$x_i \in Inv(l_i),$$

$$x_i^{min} \leq x_i \leq x_i^{max}, \ \forall i : 0 \leq i \leq n,$$

where,

$$COST(s_i, s_{i+1}) = d(M(\delta_i)(y_i), x_{i+1}) + dp(y_i, G(\delta_i)) + dp(y_i, C_i)$$

and $y_i = flow_{l_i}(x_i, \tau_i)$

As before, a concrete trajectory is found if and only if the cost function evaluates to 0.

The invariant constraint is not present in the cost function, as it would contribute infinite terms to the function, as we must check containment $\forall t$ of the trajectory and this is a continuous function, and hence would have infinite number of points. But to ensure correct trajectories, this constraint has to be satisfied. For this, a validation routine is called.

Trajectory Validation: As the flow function, $y_i = flow_{l_i}(x_i, t)$ is a continuous function, this is a difficult problem to encode. The continuity gives infinte points $y_i(t)$, $\forall t \in [\tau_{min}, \tau_{max}]$ at a location l_i. Numerically, performing a containment check is impossible.

To encode this, the containment check is then done at discrete time point t_k by calculating $dp(y_k(t_k), Inv(l_i))$. A distance of zero implies non-violation, and a positive distance indicates otherwise. In case, a violating point is found, point of violations and the dwell times are noted and added as penalty terms to the cost function so that when the reachability analyser runs again, it avoids these trajectories, as for these points, the cost function never evaluates to 0. The more the number of discrete points taken for this containment check, the better approximation we have for the containment check of the trajectory.

III. CURRENT IMPLEMENTATION

The current implementation was done with the reachability analysis tool XSpeed, with the goal to make optimisation routines perform better.

A. Adding Hard constraints over initial space variables x_i's to trim the search-space of the optimiser

In the current implementation, constraints on the space variable were added in the form of upper and lower bounds on the variables on each dimension to form the over-approximated reachable envelope for each location. This, geometrically translates to having hyper-rectangles as search space for each

location. For a better and tighter envelope, since the goal was to prune the search space, we used the initial polytope, which is a subset of this box, to construct the envelope. The initial polytope for each location was obtained using the reachability analysis.

The experiments were performed on two navigation benchmark models. Model 5 is a navigation benchmark model of 3×3 squares, and Model 8 is a navigation benchmark model with 5×5 squares. The procedure to do this is the following:

```
for i=location.id=1 to location.id=N do
    S → getsymbolicstate(i);
    Polytope P → getInitialPolytope(S);
    for every linear constraint L in P do
        myopt.add.constraint(L);
    end
end
```

Algorithm 1: Procedure for trimming the search-space

Theoretically, this pruning is supposed to make the algorithm run faster, as the search space has been made smaller, but in practice, we found that this takes a longer time as compared to the previous implementation. This might be because, in the implementation level, giving such a large number of constraint to the optimiser slows it down.

In the following tables, HC stands for Hard constraints and BC stands for Box constraints.

In both the models, as observed, that on an average, imposing hard constraints leads to taking more time for generating concrete counterexamples. The difference is not so obvious when generating only the first counterexample, but it becomes quite noticeable when all counterexamples are generated. The counterexamples here were short, and it is expected that while generating long counterexamples, the difference will be even more significant.

TABLE I: Test for Navigation Benchmark on 5×5 grid and 3x3 grid for generating **all CEs**

Model	Times ran	Avg. Time:HC	Avg. Time:BC
3x3 Nav Bench	10	715789 ms	682072 ms
5x5 Nav Bench	10	558284 ms	467739 ms

TABLE II: Test for Navigation Benchmark on 5×5 grid and 3x3 grid for generating **the first CE**

Model No.	Times ran	Avg. Time:HC	Avg. Time:BC
3x3 Nav Bench	10	102348 ms	101990 ms
5x5 Nav Bench	10	106019 ms	101215 ms

B. Breaking the problem to NLP-NLP

We converted the original non-linear optimisation problem for trajectory splicing to an iterative procedure which solves two smaller non-linear minimisation programs, nlp_x and nlp_t in each iteration. For nlp_x, the variables are x_i's, the initial points of the trajectory segments, while the dwell times are taken to be constants and for nlp_t, the variables are the dwell times of the segments, where the initial points are taken to be constants. We set up the nlp_x problem by fixing the dwell time variables, using the τ_{max} and τ_{min} as the guess values. Alternatively, We set up the nlp_t problem by fixing the space variables, using the x_{max} and x_{min} as the guess values.

For the next iteration, the fixed T for the nlp_x are the optimal values obtained by solving the time problem, and the fixed X for the nlp_t are the optimal values from the nlp_x.

The trajectory splicing is then done by iterative solving of nlp_x and nlp_t until the objective function gives a lower value than the tolerance value which is set close to zero, at which point we obtain a spliced trajectory. This procedure and the plot obtained by this method is given below in Algorithm 2 and Fig. 3.

Result: X', T'

construct nlp_x with initial guess value x_{min}, x_{max}

construct nlp_t with initial guess value τ_{min}, τ_{max}

set tolerance-val ;

initialise minf to some very large value;

while *minf* $\geq$ *tolerance-val* **do**

    Solve optimisation problem nlp_t

    with fixed by setting X to X';

    Get the optimal dwell times

    T' and the optimal value minf;

    Solve optimisation problem nlp_x

    by fixing T to T';

    Get optimal x values X' and optimal

    value minf;

end

Algorithm 2: Algorithm for decomposing to NLP-NLP

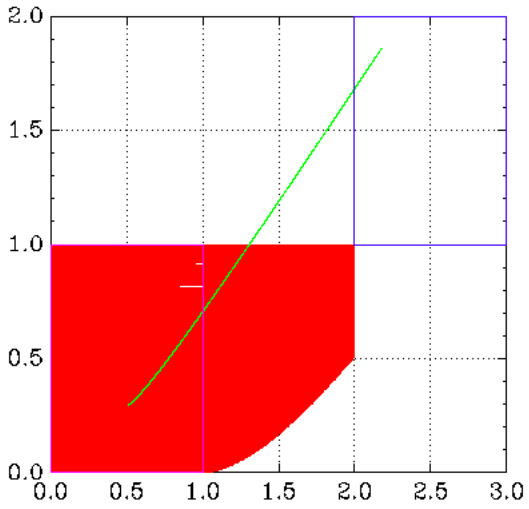


Fig. 3: NLP-NlP decomposition tested on a 3 $\times$ 3 grid navigation benchmark model, generated using XSpeed.
The x-axis gives the distance along x, and the y-axis gives the distance along distance along y-axis. The y-axis is not fully visible due to the choice of the initial region. The pink box shows the set of initial states and the blue box shows the set of unsafe sets, and the green line shows the violating trajectory. Due to lack of a validation routine, the process produces an invalid trajectory. The trajectory is invalid because it is not contained in the invariant set of the model.

IV. MOVING FORWARD

A. *Using explicit constraint conditions on the initial and the end points of trajectories*

As seen above, the optimisation problem uses a mixture of hard and soft constraints. One of the goals, moving forward is to encode all constraints as hard constraints. This means encoding the distance constraints on the endpoints as explicit constraints given to the optimiser, rather than addition to the cost function as penalty terms. The goal is that the objective function captures only the distance between the endpoint of one trajectory segment and the initial point of the subsequent segment. Once encoded, a performance analysis will be done between the current implementation, and the future implementation. The expectation is that such optimisation routine with fully hard constraints will perform better, as optimisers reportedly perform better with explicit constraints.

B. *Moving to an LP problem*

Now that the problem can be decomposed into two sub-problems, the goal now is to make the nlp_t and nlp_x problems into completely linear programming problems by appropriately constructing lp_x and lp_t. This is because a linear programming problem can be solved much faster, and also, linear optimisers are more robust than non-linear optimisers, which will also, expectedly, help the optimisation routine perform much better. For construction of such a linear programming on the space variables, the above goal mentioned, that is encoding all the constraints will help. As we have seen, in the current implementation, the distances between the endpoint y_i and the invariant set, and the endpoint y_i and the intersection set of the guard set and envelope sets were given by a metric which we had called dp. This metric happens to be non-linear real-valued function. Adding explicit constraints will remove these non-linear terms from the cost of function of the optimisation problem over x. Another option is to find a metric that calculates distance of a point from a set that is a linear real-valued function, which would then ensure that the penalty terms for ensuring the

constraints on the endpoints, are also linear.

PART II: Generation of Valid Counterexamples in Hybrid Systems

Fourth Semester Report

PART II: GENERATION OF VALID COUNTEREXAMPLES IN HYBRID SYSTEMS

V. INTRODUCTION

We discuss possible solutions to the problem of checking the validity [1] of trajectories generated in the process of falsification of a property of a given system by some alternate methods. The validity check ensures that the trajectories that have been generated are contained in the feasible region which we designate a system to have. This, in a real life situation, means we check if the solution can even exist. First, we discuss the problem of checking validity in a continuous system and eventually, move on to extending our discussion to hybrid systems.

VI. PRELIMINARIES

A. Hybrid Systems

A hybrid system [1] [3] is a system which exhibits both continuous and discrete dynamics. The mathematical formalisation of hybrid systems are called hybrid automata(on). It is described by a tuple $(L, X, Inv, Init, Flow, Trans)$ where:

- L : is the set of locations of the hybrid system and is either a finite set or countably infinite set, $\{l_1, l_2, \ldots, l_n\}$. A location is a control mode of the hybrid automaton. There is a discrete transition between each location and these are the discrete dynamics of the automaton. The set of all locations is the discrete state space of the automaton.
- X : is the set of continuous real-valued variables. This represents the continuous dynamics of the automaton. Each location is associated with a set of continuous dynamics.
- Inv : This is a subset of $\mathbb{R}^n$ and the domain of the continuous variables. This constrains the possible values of X at a particular location. This implies that all $x \in X$ of a hybrid automata should lie in the envelope.
- $Init$: This is the set of initial states. A tuple belonging to this set would be (l_{init}, C_{init}), where $l_{init} \in L$ and $C_{init} \subseteq Inv$. We will sometimes denote $Init$ as I_0.
- $Flow$: The flow is a real-valued function which models the evolution of the continuous variables, x_i, with respect to time. Each location is associated with a flow function. The flow function is given as an ODE, $\dot{x} = f(x)$.
- $Trans$: This is the set of discrete transitions or 'jumps' between control modes. A transition, denoted by δ, takes a state or location, $l_i \in L$ to $l_j \in L$, provided some transition conditions known as guard conditions are satisfied.
- $Guard$: The set of conditions given 'between' each location to trigger the discrete transition which takes the system from one location to the next. Guard conditions can be a set of conditions or a single conditions. We will denote the set of guard conditions as G.

The continuous dynamics of the hybrid automaton (HA) can be given by either a linear ODE or non-linear ODE. Here, we will consider HA with continuous dynamics of the linear ODE form.

B. Trajectories

We can describe two kinds of trajectories [1]– trajectories of a continuous system and, a further extension of this definition, trajectories of a hybrid system. The trajectories of a continuous system (in the context of HA, this is the trajectory of the automaton within a particular location) can be defined as the solutions to the flow equation (the ODE that gives the dynamics of the location). Mathematically, the trajectory is given by

$$\dot{x} = f(x_0, t)$$

, where $f : \tau \to X$, and X is the set of space variables and τ is the time interval for which the trajectory dwells in that location. τ is termed as the dwell time of the trajectory.

Trajectories of a hybrid system have two sets of functions. A hybrid system with n control modes or locations is considered. One is a set of functions, $\{\delta_1, \delta_2 \ldots \delta_n\}$, where δ_i, $i \in \mathbb{N}$, which handles the discrete transition from location l_i to l_{i+1}, and another is the set of continuous, real-valued functions, which are the flow equations associated to locations. Then the hybrid trajectory can be formally described as a 3-tuple (τ, δ, f), where τ is the set of dwell times of the trajectory, given by $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ for each location l_i, δ is the sequence of discrete transitions and is given by $\delta =$

$\{\delta_0, \delta_1, \ldots, \delta_n\}$ and f is the sequence of flow equations and is given by $f = \{f_1, f_2, \ldots, f_i, \ldots, f_n\}$, where f_i is the continuous dynamics is at the location l_i. Each *run or execution* of a HA gives a hybrid trajectory.

C. Non-linear Optimisation

An optimisation problem [2] is to select decision variables $\eta_1, \eta_2, \ldots, \eta_n$ from a given feasible region determined by a set of constraints, such that the value of a given objective function $f(x_1, x_2, \ldots, x_n)$ is optimal (maximum or minimum). The optimisation problem is called a non-linear programming problem (NLP) if the objective function is non-linear and/or the feasible region is given by non-linear constraints. A general non-linear problem is stated such:

$$Maximize \quad f(x_1, x_2, \ldots, x_n)$$

subject to,

$$g_1(x_1, \ldots, x_n) \leq b_1$$

$$g_2(x_1, \ldots, x_n) \leq b_2$$

$$\vdots \qquad\qquad \vdots$$

$$g_m(x_1, \ldots, x_n) \leq b_m$$

where each of the constraint function g_1 to g_m is given. Using vector notation, as we will use here most of the times, we will denote the vector $x = (x_0, \ldots, x_n)$. Then the problem can be stated in a more concise form:

$$Maximize \quad f(x)$$

subject to,

$$g_i(x) \leq b_i, \quad i = (1, \ldots, m)$$

A linear programming problem can be taken as a special case of the non-linear programming problem.

D. Reachability Analysis

For a hybrid system to be considered safe, the verification of it's safety properties must be done [3]. The verification problem of a safety property is often framed as a reachability analysis problem on the state space of a hybrid system [5]. For forward reachability analysis, safety verification is equivalent to showing that the set of states reachable from the initial states don't intersect the set of unsafe states. For backward reachability analysis, safety verification is done by showing that the set of states that can reach an unsafe state does not intersect the initial set of states. However, for more complex, often non-linear hybrid system, a verification check is often not possible. For checking the safety of such systems, we approach the problem through falsification which asks the question, can a hybrid system generate a trajectory from a safe state to an unsafe state when such trajectories exist? If the answer is yes, then the property is falsified, and becomes unsafe. However, even with taking the falsification approach, reachability analysis can be used to optimise the falsification process [1]. In combining reachability analsis with falsification, the benefits of reachability can be used as follows:

- The reachibility analysis of a hybrid system can be used to produce a set of locations to generate trajectory segments in and form an abstract counterexample, in searching for a concrete counterexample. Doing so allows us to treat the system as a black-box which removes the need to have expert knowledge about the system of concern and fully automates the search for a violating trajectory.
- The constraints generated from the reachability analysis can be used to effectively limit the search space of the nonlinear optimisation problem. This leads to a faster and more stable convergence. These constraints can be used as invariants of the system, as the reachable region is the feasible region of the hybrid system, and does give us the invariant set needed to verify the validity of the trajectory segment.

Figure 4 show the reachable region computed by the reachability analysis tool XSpeed [1], in a 3x3 navigation model. The navigation benchmark [1] models the motion of an object on a plane which

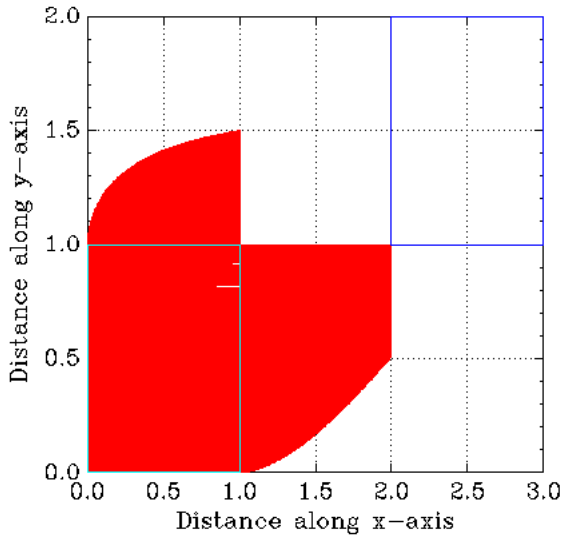


Fig. 4: Figure generated by XSpeed [1]. The yellow box shows the initial region of the system and the blue box shows the unsafe region that we designate the system. The red region shows the entire area reachable by starting from a point in the initial set.

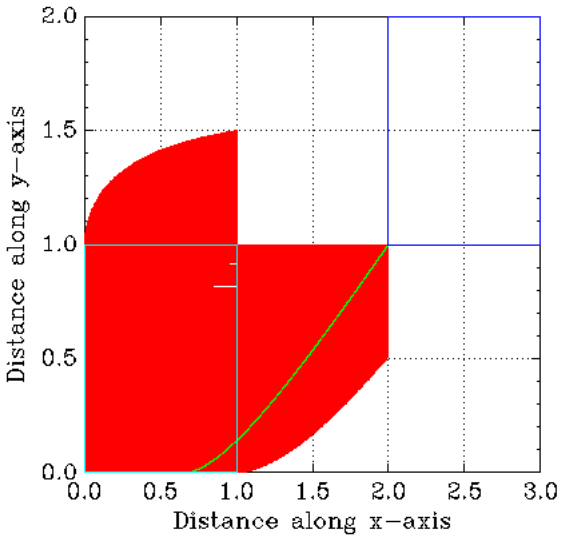


Fig. 5: Figure generated by XSpeed [1]. The yellow box shows the initial region of the system and the blue box shows the unsafe region. The red region shows the entire area reachable by starting from a point in the initial set. This also shows a valid trajectory which falsifies the property of concern here.

has been segmented into 1×1 squares. Each tile of the model is a discrete location and each tile has its own set of dynamics, which represent the continuous dynamics of this system. The dynamics changes discretely as the object moves from one square to another, so the boundaries of the square are the guard conditions. The next figure, Figure 5, shows the same system, however, it also shows a trajectory generated by the falsification approach combined with the reachabilty analysis approach discussed above.

E. Problem Statement

In a hybrid or a continuous system, after generating a trajectory, we would need to check if the trajectory is indeed a valid one, that is, does it represent a feasible path to go from the initial state to the unsafe state, given the constraints on the system. In the next two sections, we will state the validity problem formally for a continuous and a hybrid system.

1) Validity checking in continuous systems

a) Valid Trajectory

A valid trajectory [3] in a continuous system is formally described by

$$\dot{x} = f(x_0, t)$$

where $f(x_0, t) \in Inv$, $\forall t \in [0, \tau]$ and $f : [0, \tau] \rightarrow \mathbb{R}^n$ and $x_0 \in Init$, and $\dot{x}, x_0$ are n-dimensional vectors. Physically, this statement implies that the flow equation is bounded by the invariant at all times during its evolution.

To ensure this computationally is difficult as $[0, \tau]$ is a continuous interval. Analytically, the validity checking is to be done for infinite points, which is not possible computationally. For this, fine time-steps are used which are good enough approximations of a continuous interval.

b) Refinement Points

For a concrete trajectory, the set of all points of the trajectory which lie outside the invariant are called refinement points [1]. Refinement points are all tuples $(t, f(t))$ such that $f(t) \notin Inv$ and $t \in [0, \tau]$. A violating trajectory is where the trajectory has at least one refinement point.

c) Validity Problem

For trajectory validation [1] in a continuous system, the checking for containment has to be done for a continuous interval. To remove this constraint, the containment checking is done at discrete time points t_k, for which the validity checking is done for $f(x_0, t_k)$. A distance of 0 between the invariant and $f(x_0, t_k)$ implies either $f(x_0, t_k)$ is inside the invariant or on the invariant boundary and the positive distance implies a violation. In case a violation is detected, the problem is to restart the search for new initial points in the $Init$ set, to generate a valid trajectory, and to ensure that the optimiser does not pick the points in the $Init$ set for which the refinement point was reported.

2) Validity Checking in Hybrid Systems

a) Valid Trajectory

The definition of a valid trajectory [1] in a hybrid system is an extension of the definition that was given for a continuous system. As mentioned before, a continuous system can be seen as a hybrid system with only one state. Generally, however, a hybrid system has multiple states and each of these states have a set of continuous dynamics associated to it, which may or may not be the same. Each distinct state also may or may not have distinct invariant sets associated. For the definition of valid trajectories here, we will assume the hybrid system has multiple distinct states, and denote it by $L = \{l_1, l_2, \ldots, l_n\}$ which is finite or countably infinite. Each state has distinct continuous dynamics and the set of continuous dynamics over all of the states is given by $F = \{f_1, f_2, \ldots f_n\}$, where f_i corresponds to the continuous dynamics of state l_i. We will assume there is an invariant set associated with each of the states and the set of invariants is given by $Inv = \{Inv_1, Inv_2, \ldots Inv_n\}$. Then, each valid trajectory segment is given by

$$\dot{x_i} = f(x_i, t)$$

, where $f_i(x_i, t) \in Inv_i$, $\forall t \in [0, \tau_i]$ and $f_i : [0, \tau_i] \to \mathbb{R}^n$, $\dot{x_i}, x_i$ are n-dimensional vectors, and $(l_0, x_0) \in Init$. For the system to make a jump from one state to another, we impose the condition,

$$y_i = f_i(x_i, \tau_i)$$

where, $y_i \in G(\delta_i) \cap Inv_i$, if, $i = \{0, 1, \ldots, N-1\}$. For i=N, $y_i \in Inv_i$. Generally, a segmented trajectory like this may not represent a trajectory of the hybrid automaton, since the segments may be disconnected. But if it is possible to create a continuous trajectory from the segments, we obtain a valid trajectory of the hybrid automaton. We will discuss an approach to obtain such a continuous, valid trajectory in the following section.

b) Refinement Point

The definition of refinement points [1] carries over from the definition given in the continuous system. However, now that we are working with trajectory segments, rather than a single trajectory, the refinement points in a hybrid system occur from trajectory segments violating the specific invariant set of the location. Formally, the refinement point for a location, l_i, are tuples, $(i, t, f_i(t))$, such that $f_i(t) \notin Inv_i$ and $t \in [0, \tau_i]$. A violating trajectory segment for a location, l_i, is a segment that has at least one refinement point.

c) Validity Problem in a Hybrid System

For trajectory validation in a hybrid system, we encounter two problems. One is the checking the containment of the trajectory segment at location l_i, over a continuous interval $[0, \tau_i]$, which makes it difficult to encode it [1]. The second problem comes from generating a valid trajectory which represents an actual trajectory of the hybrid system. One of the approaches of generating a full trajectory from segments is splicing trajectory segments over a location sequence. However, both splicing and validation depends on the strategy that the optimiser routines restart the search of splice-able trajectory segment or a valid trajectory segment by picking new initial points from the feasible regions, and generating new trajectory segment to check. This means if splicing and validation are to be done, they cannot be done alternatively. This is because a set of segments that might be splice-able may not give a valid trajectory (the segments may violate the invariants at one

or more places), in which case, the validation routine rejects the set, and generates a new set of trajectory segments, which are valid for their respective invariant sets. However, the new set of trajectories may not be splice-able, in which case the splicing routine will reject it. To use a splicing and validation algorithm, they must be done together, rather than doing each discretely.

VII. Previous Work

A. Multiple Shooting Approach for Falsification in Hybrid Systems–Zutshi et al.

The falsification technique [5] in this paper takes the approach of searching for a complete trajectory over segmented trajectories, which are assumed to have gaps present between them. With every iteration, the goal is to minimise this gap under a threshold value (which would be very close to 0), and henceforth obtain a complete trajectory. Such a falsification technique is a *Multiple Shooting* [4] technique, which have trajectories starting from multiple starting conditions. To explain the technique used, we will go through a few concepts that have been used to construct the technique.

- For two states x and y, we have set of relations $\overset{u,t}{\rightarrow} \subseteq \chi \times \chi$, and it satisfies the properties:
  1) Each state is reachable from itself, for t=0, $\forall u : x \overset{0,u}{\rightarrow} x$.
  2) If $u_1 = u_2$, for each $x \in \chi$, $0 \leq k \leq t$, and, $x \overset{t,u_1}{\rightarrow} y$ then, $x \overset{t,u_2}{\rightarrow} y$.
  3) We assume the system S, has the function Sim_S, which takes $x \in \chi$, $t \geq 0$, and $u \in U$, and computes $y = SIM_S(x, t, u)$
- *Time-bounded Reachability Problem:* Given a System S, an initial set $\chi_0 \subseteq \chi$, a set of unsafe states, $\chi_f \subseteq \chi$, and a time-bound/time-constraint, $T \geq 0$, the *reachability problem* decides if $\exists$ an initial state, $x_0 \in \chi_0$, an unsafe state $y \in \chi_f$, time $t \in [0, T]$, and input signal $u \in U$ such that $x_0 \overset{t,u}{\rightarrow} y$.
- For a segmented trajectory, the cost of a segmented trajectory w.r.t a metric d, is given by adding up the 'gaps' between the adjacent trajectory segments, that is,

$$\sum_{0}^{n} cost(\tau_{i+1}(start.point), \tau_i(end.point))$$

  A zero cost segmented trajectory, therefore, implies an actual system trajectory.
- The state space of the system is broken down into tiles, in a process which is called $\epsilon - tiling$. Formally a tiling $C : \{C_1, \ldots, C_j, \ldots\}$ is a finite or countably infinite family of closed and bounded subsets C_j of χ called *cells*. Let there be a real number $\epsilon > 0$
  1) The union of these cells must cover the entire state space, i.e, $\bigcup_{C_i \in C} C_i = \chi$.
  2) The cell interiors must be pairwise disjoint.
  3) For each cell, $C_i \in C$, $\exists$ a representative state $[x] \in C_i$, that is no further than ϵ away from every state in the cell.
  4) The proposed tiling uses hyper-cubes as tiles (cells).

  All the hybrid states (l, x) in any given cell C_i, have the same discrete location l, and the decimal representation of the continuous states x matches for at least k decimal places where, $k > 0$, and $\epsilon = 10^{-k}$.
- *Abstraction State Graph:* Let $\Delta > 0$ be a fixed time step. The abstract state graph H for this time step is defined as where the vertices are the cells of the $\epsilon - tiling$, and the edges are (C_i, C_j), whenever, $C_i \overset{\Delta}{\rightsquigarrow} C_j$. As Δ increases, so does the complexity of the relation $\overset{\Delta}{\rightsquigarrow}$.

For finding abstract counterexamples, the approach that is used is called *Scatter-and-Simulate*. The approach is to explore the abstract state graph by randomly sampling a finite sub-graph $G(V, E)$. The reason this is called *Scatter-and-Simulate* [5] is because the algorithm uniformly samples a reachable cell to obtain initial state and performs repeated simulations from the obtained states for a fixed state Δ. For each cell, K number of concrete states and input signals are are sampled, and K is called the scatter amount, and K¿0. The size of the sub-graph is restricted by a parameter L, where the number of vertices of the sub-graph is called L. The exploration of the graph is done by a guided A* search. To prioritise the likeliest counterexamples, the notion of cost is associated with the paths in the graph. As each path P is associated with at least one segmented trajectory τ, the cost of the path is equated to the cost of the segmented trajectory which has the minimum cost.

For computing the shortest path, a modification of Dijkstra's algorithm is used. If no such path exists, the scatter-and-simulate step is restarted with increased values for the size of the graph L and the scatter amount K. To refine the CEs obtained from the abstraction from the $\epsilon - tiling$ C , the notion of a refined tiling is used. For this, a $\delta - tiling$ D where $\delta < \epsilon$, and the cells in D subdivide those in C. It is then verified that the abstract counterexamples corresponding to the -tiling continue to be counterexamples in the -tiling.

The properties which are difficult to falsify are tested against this method and the success rate of this method is compared to that of S-Taliro, which is a single-shooting approach, and dReal which is a SMT solver. For S-Taliro, scatter-and-simulate found a falsification for every run of the algorithm, while S-Taliro sometimes failed. For dReal, only the cases where a validation is known to exist are used for comparison. Scatter-and-simulate performed better as compared to dReal on the designed benchmarks.

B. Validation in Falsification of Hybrid System with trajectory splicing approach-Bogomolov et. al

In this work [1], the falsification of a given property is done by attempting to find a concrete counterexample from an abstract counterexample over trajectory segments. The paper also deals with the issue of the trajectory validation, that is, given a set of invariants over the hybrid system and a generated concrete counterexample, does the counterexample lie within the invariant boundary, making the concrete counterexample a feasible counterexample.

Since the flow equation $y_i(t) = flow_{l_i}(x_i, t)$, that is, the continuous function that gives the dynamics of the hybrid system at a particular location l_i, is a continuous function of t, $t \in [\tau_{min}, \tau_{max}]$, this is a difficult problem to encode. To address this, during trajectory validation, discrete time points, t_k such that, $t_k \in [\tau_{min}, \tau_{max}]$ are taken, so that $y_i(t_k)$ can be computed. Then, the validation step is performed by computing the distance between this point and the invariant set. A distance of zero implies that the trajectory point lies inside the invariant and a non-zero distance implies a violation of the invariant cover.

For the case where a trajectory violates the invariant, that is, $\exists$ at least one point for which the distance between the point and the invariant set is positive, the trajectory segment which the point belongs to, and the dwell time of the segment of concern is noted as refinement points [1], and the optimisation problem of searching for candidate trajectory segments is modified to avoid generating invalid trajectory segments by changing either the initial point of the segment or the dwell time of the segment. This is done by adding the term $d(y_i(t_k), Inv)$ to the optimisation problem for splicing the trajectory segments. The algorithm stores the refinement points and restarts the search until a valid trajectory is found.

VIII. Proposed Solutions

A. Proposed solution in continuous systems

We take a system with continuous dynamics, which is analogous to taking a single location in a hybrid system, and doing a validity check for the trajectory segment only for that location. The solution that was proposed for checking validity in a continuous system was the following:

1) We solve a constraint satisfaction problem to generate the initial starting points for our trajectory generation.
2) Check the validity of the trajectory generated using the initial point(s) in step 1 until some fixed τ.
3) If the trajectory is valid, then exit.
4) If not valid, then obtain the first refinement point $(t_{ref}, f(x_0, t_{ref}))$, t_{ref} being the time at which the trajectory exits the invariant envelope.
5) Form an optimisation problem, Opt, to minimise the distance between $(t_{ref}, f(x_0, t_{ref}))$. Opt gives a new initial point for trajectory-generation which ensures there is no violation at t_{ref}.
6) Check the validity of this new trajectory in step 2 and proceed.

$d(x, y)$ is a metric that is used here to calculate distance of the refinement point to the invariant set. The optimisation problem Opt is framed as follows:

$$\min \sum cost(d((t_{ref}, f(x_0, t_{ref})), Inv))$$

subject to the constraints:

$$t_{ref} \in [0, \tau],$$

$$x_0 \in Init,$$

$$t \geq 0$$

$$f(x_0, t) = y$$

$$y \in [y_{min}, y_{max}]$$

The problem tries to minimise the distance between the reported violation point and the envelope. The minimum value that can be obtained is 0 and the optimiser searches an initial point for which a trajectory that is generated will make this distance at the point of violation 0. This allows us to obtain initial points which ensure that the violation point is inside the envelope.

We call this optimisation routine each time that a violating point is reported and use the results of the routine to find a new trajectory to ensure that the reported point gets contained in the invariant.

For the implementation of the algorithm, we keep the following data notations in hand:

1) A list, where we store the refinement points reported from the simulation. Denoted by refine.list.
2) A boolean variable Status. Status is set to true if no refinement points are reported, else status is false.
3) An integer variable Max.refine, which is set to the maximum number of refinements.

τ, is the dwell time of the trajectory in this location. The *Invariant* set, which constraints the system and *Initial* set, from which the initial points of the trajectory is picked, denoted by Init, is given as inputs to the algorithm. The function Simulation.trajectory generates a trajectory and reports violation points during the simulation. The function, Cspsolve takes as argument the *Init* set and outputs a vector, which is the start vector of the trajectory. The function, OptSolve, performs the optimisation problem given above and outputs a vector, X_{ref} which is the optimal starting point of the trajectory for the violation point reported in that iteration.

Input: τ, Invariant, Init, max.refine
Result: Initial point vector to generate Valid trajectory

```
1  X_0=Cspsolve(Init);
2  Create refine.list;
3  refine.list=IsEmpty(True);
4  Traj=Simulation.trajectory(X_0, status, τ,
     Invariant);
5  if violation is reported then
6  |   status=false
7  |   refine.list ← Violation point;
8  |   Set up the optimisation problem Opt;
9  |   X_ref=OptSolve;
10 |   Set a counter for refinements done as
   |    ref.count;
11 |   while (ref.count ≤ max.refine AND
   |    !(status)) do
12 |   |   if (!(refine.list.size)) then
13 |   |   |   NewTraj=Simulation.trajectory(X_ref,
   |   |   |    status, τ, Invariant);
14 |   |   |   Validate NewTraj;
15 |   |   if (!(status)) then
16 |   |   |   refine.list ← Violation point;
17 |   |   |   X_ref=OptSolve;
18 |   |   |   Increment ref.count;
19 |   |   else
20 |   |   |   Output NewTraj is valid and
   |   |   |    exit loop;
21 |   end
22 end
23 else
24 |   Output Traj is valid and exit;
```

Algorithm 3: Procedure for validation of a trajectory in a continuous system

B. Proposed Solution in Hybrid Systems

One of the solutions that is proposed here does away with the splicing routine, and takes a deterministic approach to trajectory generation from trajectory segments and validating the same. For this approach we take a hybrid automaton HA, with multiple, distinct states, where each state has a distinct linear ODE associated to it for the continuous dynamics. Rather than splicing the trajectory segments for a given sequence of locations, $\{l_1, \ldots l_n\}$, to generate a trajectory at location l_i, we use the endpoint of its predecessor

trajectory segment.

To illustrate more clearly, let us take two consecutive locations l_0 and l_1, with continuous functions f_0, f_1 and dwell times τ_0, τ_1, respectively.

The discrete transition is given by δ_0, and $G(\delta_0)$ is the guard set of δ_0. For l_0 we pick x_0 such that $x_0 \in Init$. We simulate the trajectory segment T_0 under f_0 from t=0 to t=τ_0. Let $y_0 = f_0(x_0, \tau_0)$. Since $y_0 \in G(\delta_0)$, we take y_0 as the initial point of the next trajectory segement, T_1, in location 1. So, T_1 is given by $f_1(y_0, t)$, such that, $t \in \{0, \tau_1\}$. This does away with the issue of handling splicing alongside validation.

The solution that is proposed, keeping the above approach, uses the previously discussed validation algorithm for continuous systems. We will denote this algorithm as ValidAlgo for the pseudo-code for the algorithm given below. The algorithm is as follows:

1) We designate an Initial region, $Init$, in l_0, from where x_0 will be picked.
2) Simulate the trajectory segment, T_0 till τ_0, and check the validity of T_0. If valid, then move on to location l_1.
3) If not, use ValidAlgo to obtain a valid T_0. Extract y_0 such that, $y_0 = f_0(x_0, \tau_0)$.
4) Use y_0 as the initial points for T_1. Simulate T_1 till τ_1 and check validity. If valid then move on to the next location.
5) If not valid, obtain refinement point, $(1, t_{ref}, f(y_0, t_{ref}))$. Then form the optimisation problem Opt1, such that, $dist(f(y_0, t_{ref}), Inv_1)$ is minimum. Opt1 can pick new points to simulate the new trajectory segments only from $Init$. Solve Opt1 to obtain such a point, x_0'.
6) Start simulating T_0 from x_0'
7) Repeat Steps 2 to 5. Once a satisfactory initial point has been found, we have generated a continuous and valid trajectory from l_0 through to l_1.
8) For the next subsequent locations, we follow the same steps, that is, for each location, l_i, that reports a refinement point $(i, t_{ref}, f(y_{i-1}, t_{ref}))$, we use the optimisation problem Opt_i to obtain a point $x_0' \in Init$, such that $dist(f(y_{i-1}, t_{ref}), Inv_i)$ is minimum. Then we start simulating from the first location and check validity in all the predecessor locations to l_i. In case of a failure at some location l_k, we go back to the first location, and repeat the procedure for l_k using the Opt_k.

The Optimization problem for a location, l_i is given as follows. The cost function, $Cost_i = dist((t_{ref}, f(y_{i-1}, t_{ref})), Inv_i)$. The optimisation problem is framed as:

$$\min cost(dist((t_{ref}, f(y_{i-1}, t_{ref})), Inv_i))$$

subject to the constraints,

$$t_{ref} \in [0, \tau_i],$$

$$x_0 \in Init,$$

$$t \geq 0$$

$$f_i(y_{i-1}, t) = y_i, \quad \forall i \in [1, N]$$

$$f_0(x_0, t) = y_0$$

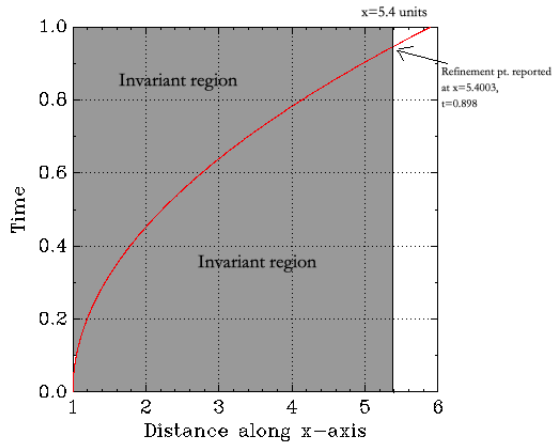
$$y_i \in G(\delta_i)$$

Input: Location sequence, τ set, Invariant Sets, Init, Max.refine(i)
Result: Valid Continuous Trajectory from l_1 to l_3

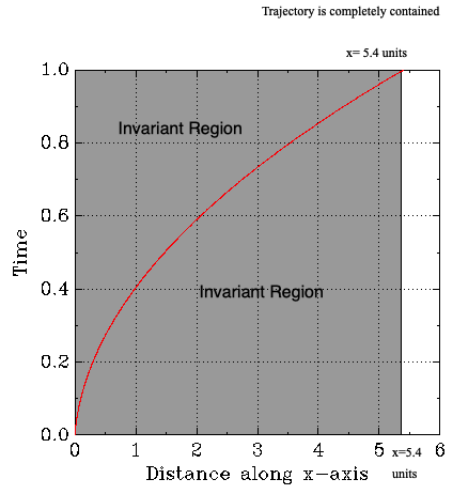
X_0=Cspsolve(Init);
Create refinelist;
refinelist=IsEmpty(True);
Function `Simulate0` (X_0, Inv_0, τ_0)**:**
    Traj1=ValidAlgo(X_0, Inv_0, τ_0) ;
    X_{sol} = Extract.endpoint(Traj1);
    return X_{sol};
;
Function `Simulate1` (X_1, Inv_1)**:**
    Traj1=Simulation.trajectory(X_1, status, τ_1, Inv_1);
    if *violation is reported* **then**
        status=false
        refine.list $\leftarrow Violation\ \ point$;
        Set up the optimisation problem Opt1;
        Set a counter for refinements done as ref.count;
        while *(ref.count* $\leq$ *max.refine AND !(status))* **do**
            if *(!(refine.list.size))* **then**
                NewTraj0=Simulate0(X_{ref}, Inv_0);
                X_{in}= Extract.endpoint(Traj1);
                NewTraj=Simulation.trajectory(X_1, status, τ, Inv_1);;
                if *(!(status))* **then**
                    refine.list $\leftarrow Violation\ \ point$;
                    X_{ref}=Opt1;
                    Increment ref.count;
                else
                    Output NewTraj is valid and exit loop; Empty refinelist;
                    Move to loc2;
        end
    end
    else
        Output Traj is valid and Move to loc2;
;

Algorithm 4: Proposed Algorithm for validation in Hybrid Systems(1x3 grid Specific Form)

Function `Simulate2` (X_2, Inv_2)**:**
    Traj2=Simulation.trajectory(X_2, status, τ_2, Inv_2);
    if *violation is reported* **then**
        status=false
        refine.list $\leftarrow Violation\ \ point$;
        Set up the optimisation problem Opt2;
        Set a counter for refinements done as ref.count;
        while *(ref.count* $\leq$ *max.refine AND !(status))* **do**
            if *(!(refine.list.size))* **then**
                NewTraj0=Simulate0(X_{ref}, Inv_0);
                X_a= Extract.endpoint(NewTraj0);
                NewTraj1=Simulate1(X_a, Inv_0);
                X_b= Extract.endpoint(NewTraj1);
                NewTraj=Simulation.trajectory(X_b, status, τ, Inv_1);;
                if *(!(status))* **then**
                    refine.list $\leftarrow Violation\ \ point$;
                    X_{ref}=Opt2;
                    Increment ref.count;
                else
                    Output NewTraj is valid and exit loop; Empty refinelist;
        end
    end
    else
        Output Traj is valid and exit;
;
Driver Function `Main`**:**
    set.parameters();
    X_0=Cspsolve(Init);
    X_1=Simulate0(X_0, Inv_0);
    Traj2=Simulate1(X_1, Inv_1);
    X_2=Extract.endpoint(Traj2);
    Traj3=Simulate2(X_2, Inv_2); **return** 0;
return



(a) The trajectory, when the refinement point is reported. The Invariant is given by the grey region.



(b) The valid trajectory, after the algorithm is run. The Invariant is given by the grey region.

Fig. 6: **Trajectories, before and after validation**

IX. Results

A. Experimental Models

All experiments were performed using the reachability tool XSpeed. The optimisation problem for generating initial points which would give a valid trajectory was solved using the open-source library *Nlopt*. In the experiments, a non-gradient based optimiser routine, *COBYLA* was used. The parameter, *tolerance*, is fixed beforehand. *Tolerance* measures the accuracy of the containment of a point on the trajectory. A point is considered valid if the distance to the invariant set is less than or equal to the specified tolerance value. In all the experiments done below, it has been set at 10^{-6}. All experiments were performed on a machine with Intel Core i5 CPU, 1.6GHz and 8 GB RAM.

1) Continuous Systems

a) Example 1: Free-Falling Point Object

A 2-dimensional model of free-falling point object (variables s, t) was used to check the above algorithm. The model describes an object being thrown with some pre-set initial velocity, and the evolution of the system is looked at from time 0 unit to time 1 unit. The dynamics of the system is given by:

$$\frac{\mathrm{d}s}{\mathrm{d}t} = u + 9.8t$$

where u is some constant.

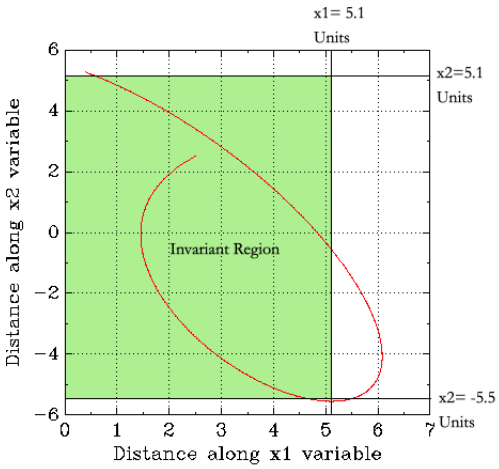
The invariants of this model are set at $t \in [0, 1]$ and $s \in [0, 5.4]$. The initial points of the system are x=1.000 units and t=0.000 sec.

The system, in the initial run, violates the invariant constraint $s \in [0, 5.4]$ at t=0.898 sec, as shown in Figure 4.(a), and we store the point (x=5.40038, t=0.898) as the refinement point. Using this point and OptSolve, we generate a new set of initial points (x=5.36057e-08 , t=0), which, when simulated till t=1.00 sec, produces a trajectory that is reported as valid, as shown in Figure 4(b).

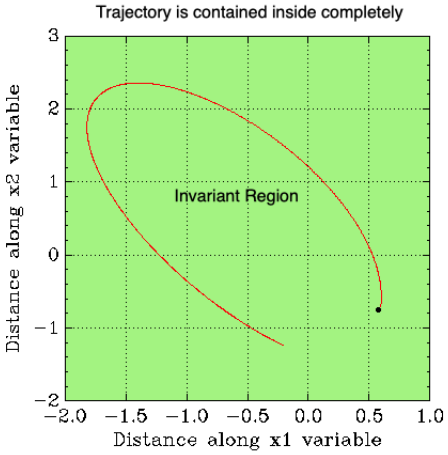
In this example, the number of refinements that were needed to produce a valid trajectory was 1.

b) Example 2: Five-Dimensional System

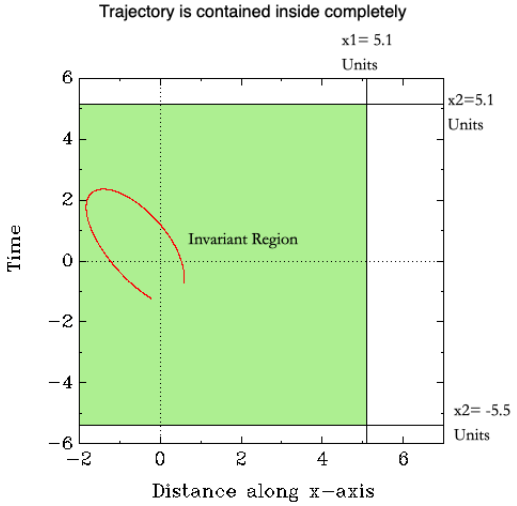
The next model that we consider is a five-dimensional system, where each dimensional variable has its own set of dynamics. The system was given invariant constraints that $x_i \in \{5.1, 5.5\}$, $i = \{1, 2, 3, 4, 5\}$. The initial coordinate of the system shown in the figure was $\{2.5, 2.5, 2.5, 2.5, 2.5\}$ and the system was to evolve from t=0 to t=1 under the given invariant constraints. The initially generated



(a) The trajectory, when the refinement point is reported. The Invariant is given by the green region.



(b) The valid trajectory, after the algorithm is run. The Invariant is given by the green region. The initial point of the validated trajectory is given by the black dot.



(c) Zoomed out view of the Trajectory inside of the Invariant

Fig. 7: **Trajectories, before and after validation**

trajectory violates the invariant constraints, visually, in two of the dimensions, as shown in Figure 5.(a). The first violation is reported at the coordinate {1.6638,1.2306,5.1270,2.6869,-0.09881} at t=0.049 sec. The algorithm requires 6 refinements to arrive at an initial point for which the simulated trajectory is completely inside the invariant region specified. The optimal starting point that generates a valid trajectory is: {0.594281, -0.711054, 1.649401, 0.090512, 2.2137}. Table III, shows the violations and the subsequent refinements in trajectory generation.

An important point to note is that the number of refinements required to produce a valid trajectory

TABLE III: **Refinement Points and the subsequent refined trajectory**

Starting Point of the Trajectory	Violation Coordinates	Violating time
{2.5, 2.5, 2.5, 2.5, 2.5}	{1.66, 1.23, **5.13**, 2.67, -0.10}	0.049
{1,1.678,1.8,1.09,0}	{0.76, -0.29, **5.12**, 1.96, -2.15}	0.09
{1,1.678,1.8,1.09,2.25}	{0.57, -0.38, **5.11**, 1.55, -0.55}	0.156
{0.36, -1.25, 0.89, -0.25, 2.35}	{-1.12, 2.81, **-5.51**, -2.98, 2.77}	0.233
{0.42, -1.35, 1.32, -0.27, 2.38}	{-1.09, 2.92, **-5.50**, -3.15, 2.95}	0.238
{0.43, -1.36, 1.35, -0.26, 2.38}	{-1.10, 2.94, **-5.51**, -3.16, 2.93}	0.240
{0.59, -0.71, 1.65, 0.09, 2.21}	None	None

are also dependent on the first guess value that is provided to the NLOPT routine. While the advantage is that, for a good (lucky) guess value, the valid trajectory can be obtained in as few as one step, there exists starting values for which NLOPT will be unable to converge and even though there might exist valid trajectories, they will never be reported by the algorithm.

2) Hybrid Systems

For the hybrid system, we construct 1x3 grid system. Each grid is association with the linear ODE,

$$\frac{\mathrm{d}s}{\mathrm{d}t} = u + gt$$

where u is some constant, and g is given a different value for each grid. For this example, Grid 1 has g=9.8, Grid 2 has g=4.5, and Grid 3 has g=0.8. The invariants of this model are $s \in [0, 1]$ for Grid 1, $s \in [0.5, 1.5]$ for Grid 2 and $s \in [1, 2]$ for Grid 3. The Init region of the system is $x_0 \in [0, 1]$ and t=0.

The algorithm is unsuccessful in finding a valid trajectory for the hybrid case.

X. Conclusions

While for the continuous dynamics region it is possible to obtain a valid trajectory, the algorithm for the hybrid system is unable to produce a valid trajectory. This is possibly due to an implementation error, as one of the reasons that the algorithm fails is because when the search restarts after failure at one location, the optimiser is unable to avoid the starting point the leads to an invalid trajectory. The future goal is to correct this implementation error and obtain valid trajectories in hybrid systems by this method. Another goal is to use a derivative based optimiser routine if possible, since the non-derivative based optimisers are in general, slower and need more number of steps to converge. This would make the algorithm more efficient.

References

[1] S. Bogomolov, G. Frehse, A. Gurung, D. Li, G. Martius, and R. Ray, *Falsification of hybrid systems using symbolic reachability and trajectory splicing*, in Proceedings of the 22Nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC '19, New York, NY, USA, 2019, ACM, pp. 1–10.

[2] J. B. Orlin, *Nonlinear programming*.

[3] E. Plaku, L. E. Kavraki, and M. Y. Vardi, *Hybrid systems: from verification to falsification by combining motion planning and discrete search*, Formal Methods in System Design, 34 (2008), p. 157–182.

[4] R. Ray, *Falsification of hybrid systems using symbolic reachability and trajectory splicing*, Presentation, 2019.

[5] A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski, *Multiple shooting, cegar-based falsification for hybrid systems*, in Proceedings of the 14th International Conference on Embedded Software, EMSOFT '14, New York, NY, USA, 2014, Association for Computing Machinery.

[6] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, and J. Kapinski, *A trajectory splicing approach to concretizing counterexamples for hybrid systems*, in 52nd IEEE Conference on Decision and Control, Dec 2013, pp. 3918–3925.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Rajarshi Ray, for giving me the opportunity to work under him in this project and for the patient guidance, encouragement and advice he has always provided throughout the year.

I would also like to thank my lab partner, Ritasmita Chattopadhyay, for her help and support.

I am also very thankful to all my classmates for their advice and encouragement.