Directed Test Generation for Hybrid Systems*

Sudhi Proch
ECE, University of Florida, USA
sproch@ufl.edu

Prabhat Mishra
CISE, University of Florida, USA
prabhat@cise.ufl.edu

Validation of hybrid systems is complex due to interactions of both continuous and discrete dynamics. Simulation is the most widely used form of system validation using a combination of random and constrained-random tests. Directed tests are promising since orders-of-magnitude less number of directed tests can achieve the same coverage goal compared to random tests. While directed test generation is well studied for digital designs, it is still in its infancy for hybrid systems. In this paper, we propose a method for automatically generating directed tests for hybrid systems. The test generation scheme is based on the Rapidly Exploring Random Tree (RRT) algorithm. In contrast to existing methods of using RRT for validation that tries to reach targets (functional scenarios) from the initial state, we propose to employ reverse RRT that starts from a target and tries to reach the initial state. This enables us to generate an accurate testcase for both functional scenarios and interesting corner cases. Our test generation algorithm is upto 33 *times faster (average* 10 *times) compared to state-of-the-art forward RRT techniques.*

I. Introduction

Validation of systems with both continuous and discrete dynamics (hybrid systems) is complex. Automated techniques for efficient functional validation of such systems become necessary to keep pace with increasing validation complexity. Simulation using a set of test vectors is an integral part of validating hybrid systems. Test vector generation can be classified into three broad categories: random, constrained-random and directed. Random test generators, due to their capability to generate a wide variety of functional scenarios, are efficient in verification of unknown errors. Constrained random test generation is an attempt to steer a generic random test generator towards generating test vectors that are likely to activate a set of important functional scenarios. Depending on the nature of a functional scenario, constraint generation can be complex. Moreover, due to probabilistic nature of these constraints, the generated tests may not activate the target functional scenarios.

Let us take an example of a bouncing ball system which is described in detail in Section V-A. As shown in Fig. 1, the ball can be launched with different velocities and from a starting point of different height, resulting in different bounce trajectories.

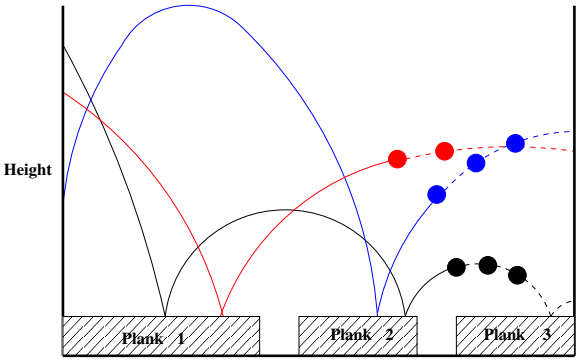

Fig. 1. Different trajectories for a bouncing ball system

The ball bounces on impact with different planks on the ground. The gaps between planks (holes) must be avoided during travel. An interesting functional scenario here could be that of a trajectory where ball only hits planks 2 and 3 before reaching the other side. One can imagine many more such functional scenarios of increasing complexity that might be of interest. A random test generator in this case would generate numerous random bounce trajectories before discovering the desirable functionality. A directed test generator on the other hand would generate exactly one testcase to satisfy one functional scenario. Clearly, the number of directed tests required to reach a coverage goal would be several orders of magnitude less than using random test vectors.

While directed test generation is a well studied problem for digital designs [5], in case of hybrid systems it is mostly performed by human intervention that is cumbersome and error prone. In fact, it may not be

*This work was partially supported by National Science Foundation (NSF) grants CNS-0746261 and CCF-1218629.

possible to develop a directed test manually if the design is complex and/or it involves complex interactions.

In this paper, we present an automated method for generating directed testcases that uses a search technique based on Rapidly Exploring Random Tree (RRT). RRT algorithm is widely used in robotics path planning domain. Existing RRT based methods search in the forward direction. Forward searches are not effective in situations when the initial region is large but the target region is small. In contrast, backward search will be promising to reach the initial region starting from a small target region. The primary contribution of this paper is development of a RRT based directed testcase generation method that employs tree exploration in reverse direction. In order to enable reverse RRT algorithm, we also transform the system model. Experimental results using bouncing ball and thermostat demonstrate the effectiveness of our automated generation of directed tests.

The rest of the paper is organized as follows. Section II presents the system model and outlines the problem statement. Section III presents related approaches. Section IV describes our directed test generation technique using reverse RRT. Section V presents two case studies. Finally, Section VI concludes the paper.

II. Problem Formulation

We first define the system model for hybrid systems. Next, we formally define the problem of directed test generation.

A. System Model

The model M is defined by the tuple $(X, Q, U, f, \mathrm{Inv}, I_{init}, E, G, R)$ where.

- X : finite set of bounded continuous variables $\subset \mathbb{R}^n$
- Q : finite set of discrete states $\{q_1, q_2, q_3 .... q_n\} \subset \mathbb{Z}$
- U : finite set of bounded control inputs $\subset \mathbb{R}^m$
- $f = \{f_q \mid q \in Q\}$ such that f_q is a vector field that defines the time derivative of the continuous variables $x \in X$ given by $\dot{x} = f_q(x, u)$. We assume $u \in U$ to be piecewise continuous. We also assume all f_q to be Lipschitz continuous [11] and integrable in reverse time.
- Inv : assigns to each q an invariant set.
- I_{init} : is the initial region of interest for the system.
- $E \subseteq Q \times Q$: is a set of edges denoting the discrete transition of the system.
- G : is a set of guard conditions assigned to each edge $e = (q, q') \in E$. A transition is taken when the guard condition is satisfied.
- $R : (x, q) \rightarrow (x', q')$ is a reset map defined for each edge $e = (q, q') \in E$. It defines how x changes when that transition is taken.

State of the system is defined by (x, q) with the initial state being $(x_{init}, q_{init}) \in I_{init}$. We assume that reset map R can be inverted such that there is a corresponding relation R' for each edge $e = (q, q') \in E$ which is inverse of R. R' defines transition from x' to x such that normal system operation will cause transformation $(x, q) \rightarrow (x', q')$ with reset map R.

B. Problem Statement

Given a model $M = (X, Q, U, f, \mathrm{Inv}, I_{init}, E, G, R)$ for hybrid system and a specific functional scenario S, the goal is to come up with one possible set of control inputs $(u_1, u_2, \ldots, u_n)$ and a start location $I_0 \in I_{init}$ such that the set of inputs applied over a finite time interval $\{0 \rightarrow T\}$ guide the system from I_0 to a state where functional scenario S is satisfied. A testcase for functional scenario S would then consist of the set $(I_0, u_1, u2, \ldots, u_n)$.

For systems without any control inputs the testcase consists only of the start location I_0 such that functional scenario S is activated if we let the system evolve unforced from I_0 for some finite time interval T.

We assume that the system start region I_{init} and the functional scenario S can be specified using inequality operators $\{\leq, \geq\}$ and logical operators $(\wedge, \vee)$ over the system variables X. A testcase is considered valid only if it originates from I_{init}.

III. Related Work

We can classify related work in two main categories. One of the approaches address testcase generation problem by doing safety validation through reachability analysis of the system model. This approach is used by tools like HYTECH [8], CheckMate [12], d/dt [2], PHAVer [6], SpaceEx [7]. These tools suffer from the system scalability issues. In addition, they also employ over approximations of reachable sets to contain state space explosion that can lead to false positives.

The other class of tools are based on random search techniques. RRT (Rapidly Exploring Random Tree) [10] is one of the popular algorithms in this class which is used by many researchers [3], [4], [1]. Both [4] and [1] use RRT algorithm that starts from a point in the initial region and grows in the forward direction towards a target goal region. In [4], a coverage based criteria is used to bias the goal region sampling. In [1], the results of a learning phase, based on stable system states, are used for goal sample biasing. As described in Section

IV and V, forward RRT algorithms, even with their goal biasing heuristics, become inefficient while searching for a specific system state as they rely on random search in the whole state space. In [3], a bidirectional RRT tree (forward and backward) growth algorithm is used but their focus is hybrid system analysis and not testcase generation.

Closest related work to our technique is [9]. In contrast with [9], we use a reverse RRT growth algorithm that starts from the functional scenario region and grows towards the initial region. As demonstrated in Section V-C, our reverse RRT based test generation performs significantly better than the forward RRT approach of [9].

IV. Directed Test Generation

Our directed test generation method is shown in Algorithm 1.

Algorithm 1: Directed test generation

Input: i) The design model, M
ii) A functional scenario S in the form $a_n \leq X_n \leq b_n$
Output: i) An initial region $I_0 \in I_{init}$
ii) Path from I_0 to S as part of tree structure Γ

```
for i is from 1 to Max_iter do
    I_0 = {}, Γ = {};
    Add edge UDF(S) to Γ;
    for j is from 1 to Max_nodes do
        if I_init ∉ Γ then
            X_rand = Goal_Sample(P(x, μ_init, α));
            X_near = Nearest_Node(X_rand, Γ);
            X_sim = {∫_0^t f(x_near, UDF(u))};
            X_new = Nearest_Node(X_rand, X_sim);
            if X_new ∉ Γ then
                add X_new to Γ;
            end
            else
                (α(σ)) = Dynamic_Bias_Adjust(κ);
            end
        end
        else
            I_0 = I_init ∈ Γ;
            break;
        end
    end
    if I_init ∈ Γ then
        break;
    end
end
return I_0, Γ;
```

Our algorithm constructs a tree structure (Γ) consisting of nodes and edges. Each node stores the system state

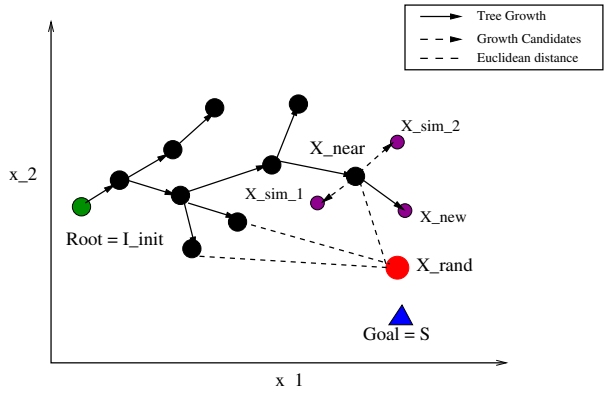

Fig. 2. Tree growth in forward RRT [9]

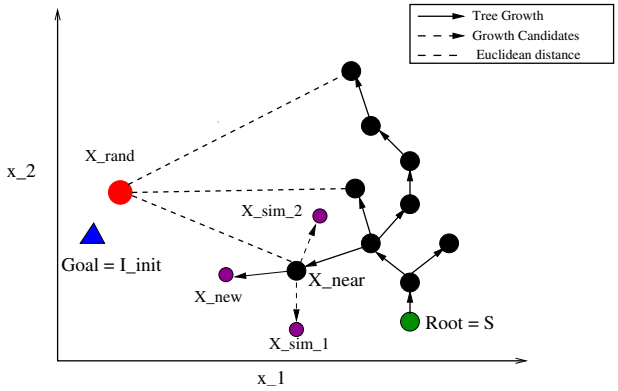

Fig. 3. Tree growth in reverse RRT (our approach)

and edges denote the control input values that lead one system state to the other. Fig. 2 and Fig. 3 show search tree growth in forward (existing approaches) and reverse RRT (our approach), respectively. In contrast to forward RRT algorithm (Fig. 2), that sets a specific initial system state as the root of the tree (start point), our algorithm (Fig. 3) sets the root of the tree in the system state where target functional scenario (denoted by S) is satisfied. The algorithm then tries to grow the tree towards a region of interest. This region of interest in our algorithm is the initial region I_{init}. At each step of the iteration a new edge X_{new} is added to the tree starting from a nearest node X_{near} with respect to a random goal region point X_{rand}. X_{rand} is selected as per a distribution function $P(x, \mu_{init}, \alpha)$ whose aim is to bias the tree growth towards the region of interest. System is simulated for a small time starting from the state in X_{near} by variation of control inputs U, resulting in a set (X_{sim}) of intermediate nodes. X_{new} is then selected from X_{sim} by means of a metric function that grows the tree towards X_{rand}. UDF as mentioned in Algorithm 1 refers to the uniform random distribution function. It should be noted that when our algorithm

grows the RRT in reverse, at every step it is trying to explore one out of many possible conditions that can make present state reachable by variation of control inputs. A hybrid system that is composed of discrete transitions and Lipschitz continuous function is expected to be completely reversible in both forward and reverse directions. Thus our test generation algorithm remains unaffected by the fact that there could be multiple ways of reaching a functional scenario from an initial region, as we need to explore only one of the many ways in which such a path can be traversed.

The remainder of this section describes in detail the important steps of our algorithm: goal sampling distribution (X_{rand}), dynamic bias adjustment of goal sampling and nearest node selection (X_{near}).

A. Goal Sample Distribution

Our goal sample generation scheme attempts to bias the tree growth so that it quickly reaches the initial region (I_{init}) of the system. We use bounded normal distribution function for goal generation. The spread of the distribution function in this case can be dynamically adjusted to result in relatively more uniform distribution if required as explained in subsection IV-B. The goal distribution function P for each system variable x in this case is given by

$$P = \begin{cases} N(x, \mu, \alpha) + C_{norm}, & b_l \leq x \leq b_r \\ 0 & otherwise \end{cases}$$

$N(x, \mu, \alpha)$ is the normal distribution function with mean μ and standard deviation given by function $\alpha(\sigma)$. $\alpha(\sigma)$ is an adaptive function that controls the standard deviation σ of the spread from a range of user provided values. We take the mean of a variable specified by inequalities $b_l \leq x \leq b_r$ as $\mu_x = \frac{b_l + b_r}{2}$

C_{norm} is a normalization factor that is added to make the area under the distribution become unity for the bounded interval. Fig. 4 shows the plot of $P(x, \mu, \alpha)$ for different standard deviation values with a particular mean in a bounded region of a variable.

B. Dynamic Bias Adjustment

During tree growth phase (lines $10 - 15$ in Algorithm 1), a routine keeps track of instances when generated goal fails to extend the tree. Referring to Fig. 3, tree growth is unsuccessful when selected node X_{new} is already contained in the tree. Dynamic bias adjustment helps in this case to steer the tree towards the direction where it keeps growing.

As described in subsection IV-A and depicted in Fig. 4, goal sample generation for testcases is controlled

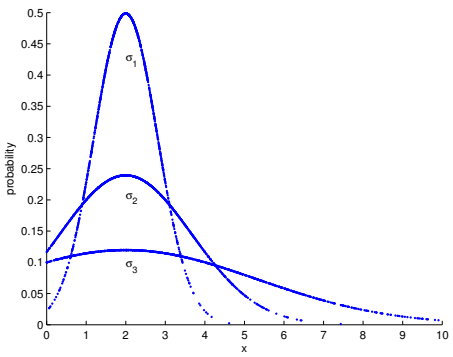

Fig. 4. Probability distribution curves for different standard deviations in a bounded region $[0, 10]$ with mean 2

by the bounded normal distribution function $P(x, \mu, \alpha)$. The dynamic bias is adjusted by the standard deviation function $\alpha(\sigma)$ so that it becomes more uniform in case of undesirable tree growth direction. The scheme of dynamically adjusting the spread of goal samples is a reasonable compromise between faster convergence and avoiding local minima.

C. Nearest Node Selection

Function *Nearest_Node()*, shown in Algorithm 1, is implemented by using a metric function λ. To enable guided tree growth, this function provides a measure of distance for a tree node with respect to the goal region (I_{init}). We use Euclidean norm as our metric function. The Euclidean norm for distance between two n dimensional vectors $\vec{x}$ and $\vec{y}$ is given by

$$\lambda(\vec{x}, \vec{y}) = \left\{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ...(x_n - y_n)^2\right\}^{\frac{1}{2}}$$

For situations, when the generated testcase does not meet the requirements of initial region and hence is not a valid testcase, we explore the functional scenario region for a uniformly random different start location.

V. Case Studies

We demonstrate the applicability of our approach in this section through case studies of two different hybrid systems, namely bouncing ball and thermostat.

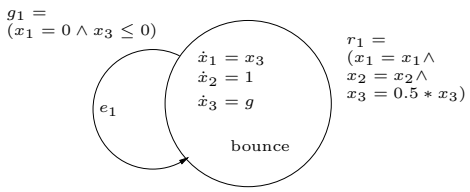

Fig. 5. Model of bouncing ball system

A. Bouncing Ball

The system shown in Fig. 5 models a bouncing ball system similar to the one shown in Fig. 1. The state variables (x_1, x_2, x_3) denote the vertical position, horizontal position and the velocity of the ball. The system has a single discrete state labeled "bounce". As it is an unforced system, U is an empty set. Dynamics of the system variables are shown in Fig. 5 as equations for $\dot{x}_1, \dot{x}_2$, and $\dot{x}_3$. Edge e_1 has guard condition $g_1 = (x_1 = 0 \wedge x_3 < 0)$. It indicates that the transition happens when the ball hits the floor $(x_1 = 0)$ with a negative velocity. Reset map r_1 indicates that the velocity of the ball reverses and reduces to half the previous value on transition e_1.

Consider generating a testcase for this system when ball hits some specific region (plank section) on the floor with certain velocity so that the ball bounces towards a specific section on the opposite wall. We state this scenario as:
"*Starting from a height range of* $[0, 1.35]$ *and velocity range of* $[-2.2, 1.1]$ *generate a testcase to hit floor section* $[3.9, 4.1]$ *such that the velocity of the ball while hitting this region is in the range* $[-0.4, -0.2]$".

Functional scenario region (S) becomes our start region and can be stated with the following inequalities: $S = 0 \leq x_1 \leq 0 \wedge 3.9 \leq x_2 \leq 4.1 \wedge -0.4 \leq x_3 \leq -0.2 \wedge q = 1$
Initial region (I_{init}), towards which we grow the tree, is defined as: $I_{init} = 0 \leq x_1 \leq 1.35 \wedge 0 \leq x_2 \leq 0 \wedge -2.2 \leq x_3 \leq 1.1 \wedge q = 1$.

The testcase generated by our Algorithm 1 is shown highlighted in Fig. 6. The algorithm identifies point $[x_1, x_2, x_3, q] = [1.0313, \approx 0, 0.3465, 1]$ within I_{init} that will satisfy the scenario. Average memory usage and testcase generation time are shown in Table I for multiple attempts to generate this testcase. Fig. 6 also highlights the randomly explored state space used for goal selection during testcase generation.

It should be noted that since this is a system with no control inputs (unforced), the tree grows in a predictable manner depending on the start conditions of the system. Although this example is simple for exploration using RRT algorithm, it demonstrates the applicability of our algorithm to such a system. A relatively more complex, controlled system, is described in next section.

B. Thermostat

Consider an example of a thermostat system with model as shown in Fig. 7.

It has two discrete states namely "on" and "off". System variables are $\{x_1, x_2\}$ where x_1 is the system

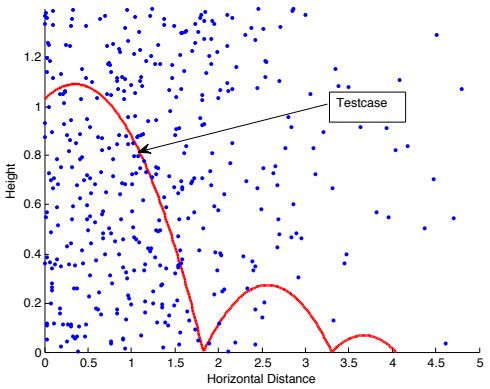

Fig. 6. Testcase produced for bouncing ball system

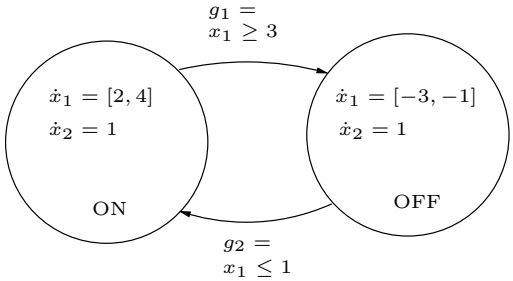

Fig. 7. Model of thermostat system

temperature and x_2 is the total amount of time system is in operation. $U = \{u_{on}, u_{off}\}$ where $u_{on} = [2, 4]$ and $u_{off} = [-3, -1]$ are the heating and cooling rate intervals when the system is in "on" $(q = 1)$ and "off" $(q = 2)$ state, respectively. System transitions from "on" state to "off" state whenever system temperature becomes equal to upper bound of 3. Similarly transition from "off" to "on" state happens when system temperature equals a lower bound 1. None of the variables change values when transition is taken.

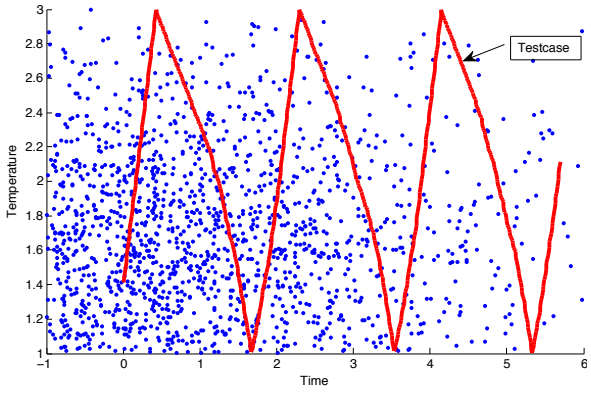

Fig. 8. Testcase produced for thermostat system

Let us consider the test generation for a scenario when

TABLE I
TEST GENERATION TIME AND MEMORY USAGE COMPARISON ('$\star$' IMPLIES FORWARD RRT FAILED)

System	Test	Test Generation time(sec)			Memory Used (MByte)		
		Fwd. RRT [9]	Rev. RRT (our method)	Improvement (times)	Fwd. RRT [9]	Rev. RRT (our method)	Improvement (%)
bouncing ball	1	926.66	57.23	16.19	512.08	509.78	0.45
	2	152.54	47.25	3.23	510.32	509.81	0.1
	3	> 1 Hr*	52.73	$-^\star$	> 1 Hr*	510.05	$-^\star$
	4	123.31	55.53	2.22	509.89	509.71	0.03
	5	> 1 Hr*	48.59	$-^\star$	> 1 Hr*	509.84	$-^\star$
	Avg.	**400.84**	**53.34**	**7.52**	**510.76**	**509.77**	**0.2**
thermostat	1	587.33	17.84	32.92	516.75	509.81	1.34
	2	> 1 Hr*	53.31	$-^\star$	> 1 Hr*	510.91	$-^\star$
	3	> 1 Hr*	27.29	$-^\star$	> 1 Hr*	510.11	$-^\star$
	4	4317.83	428.02	10.09	541.28	517.25	4.44
	5	363.75	56.73	6.41	515.62	510.98	0.9
	Avg.	**1756.3**	**167.53**	**10.48**	**524.55**	**512.68**	**2.26**

"*Start system temperature is in the range* $[1, 2]$ *and temperature reaches* 2.11 *in the on state after a run time of* 5.7 *time units*".

Functional scenario region (S) and initial region (I_{init}) are set in this case with the following inequalities: $S = 2.11 \leq x_1 \leq 2.11 \wedge 5.7 \leq x_2 \leq 5.7 \wedge q = 1$
$I_{init} = 1 \leq x_1 \leq 2 \wedge 0 \leq x_2 \leq 0 \wedge q = 1 \vee 2$

Our algorithm generates a point $[1.41,\ 0, 1]$ within I_{init} as the testcase start point in this case. It also returns the tree structure from which heating/cooling rates and corresponding times can be extracted. The extracted testcase is shown highlighted in Fig. 8. Area of the state space explored for generating the goal points is also highlighted in Fig. 8. Average memory usage and testcase generation time are listed in Table I for multiple attempts to generate this testcase.

C. Comparison

We compare the test generation time and memory usage of our algorithm with our implementation of existing forward RRT algorithm [9]. We generated five testcases for each of the systems presented in the case studies. Results are listed in Table I. For results of Table I, each algorithm was allowed to run for a maximum time of approx. 1 hour on a linux 2.4 GHz machine. It is clearly visible from the results that our algorithm is upto 33 times faster (10 times on average) and requires less memory (upto 4%, 2% on average) compared to forward RRT. It must be highlighted that in many cases, the forward RRT algorithm did not find the required testcase even after running for the maximum allowed time. These cases, highlighted with '$\star$' in Table I, are not considered for improvement computations. In other cases, forward RRT generated an approximate testcase which failed to activate the functional scenario in reality. These results are expected because our algorithm starts from the functional scenario region and grows towards a wide initial region. A wide initial region helps quick convergence of random search for reverse RRT algorithm because finding any point in the initial region during random search would result in a valid testcase that leads to target functional scenario. Forward RRT algorithm on the other hand tries to randomly search the target functional scenario in the complete system space and takes time to converge, often settling for an approximately close point.

VI. CONCLUSION

In this paper, we introduced a directed testcase generation method for hybrid systems. The technique is based on RRT algorithm. Instead of using a forward growth RRT, we have used a reverse RRT algorithm that starts from the region of desired functional scenario and grows towards the initial region. We have combined this technique with an adaptive bias adjustment method that helps in efficient convergence of random search. We demonstrated the applicability of this technique on models of varying complexity. As our algorithm starts from the region of target functional scenario, it converges quickly when growing towards a wide initial region. Corner case scenarios are generally of this nature where designers are interested in a specific final system state starting from a range of initial conditions. In comparison with the forward RRT, our algorithm is upto 33 times faster and requires less memory (upto 4%).

REFERENCES

[1] S. Ahmadyan et al. Goal-oriented stimulus generation for analog circuits. *DAC* 2012.

[2] E. Asarin et al. The d/dt tool for verification of hybrid system 2002.

[3] M. Branicky et al. Rrts for nonlinear, discrete, and hybrid planning and control. *IEEE Conf. on Decision and Control* 2003.

[4] Thao Dang and Tarik Nahhal. Coverage-guided test generation for continuous and hybrid systems. *Form. Methods Syst. Des.* 2009.

[5] M. Chen et al. System-level validation: High-level modeling and directed test generation techniques, Springer 2012.

[6] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech 2005.

[7] G. Frehse et al. Spaceex: Scalable verification of hybrid systems. CAV 2011.

[8] T. Henzinger et al. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer* 1997.

[9] Jongwoo Kim and Joel M. Esposito. Adaptive sample bias for rapidly-exploring random trees with applications to test generation. *American Control Conference* 2005.

[10] Steven M. Lavalle and James J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *Algorithmic and Computational Robotics: New Directions* 2000.

[11] John Lygeros. Lecture notes on hybrid systems. Technical report 2004.

[12] B. Izaias Silva and Bruce H Krogh. Formal verification of hybrid systems using checkmate: a case study. *Proc. American Control Conference* 2000.