BUILDING DRYVR : A VERIFICATION AND CONTROLLER SYNTHESIS ENGINE FOR CYBER-PHYSICAL SYSTEMS AND SAFETY-CRITICAL AUTONOMOUS VEHICLE FEATURES

BY

BOLUN QI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Associate Professor Sayan Mitra

ABSTRACT

To test safety of autonomous vehicles, large corporations have raced to log millions of miles of test driving on public roads. While this can improve confidence in such systems, testing alone cannot establish of absence of failure scenarios. In fact, it has also been reported that the amount of data required to guarantee a probability of 10^{-9} fatality per hour of driving would require 10^9 hours of driving [1] [2], which is roughly in the order of thirty billion miles. Formal verification can give guarantees about absence of failures and potentially reduce the amount of testing needed significantly.

Simulation based verification is a promising approach to provide formal safety guarantees to Cyber-Physical Systems (CPS). However, existing verification tools rely on the explicit mathematical models of the system. Detailed mathematical models are often not available or are too complex for formal verification tools. To address this issue, the DryVR approach for verification is presented in [3]. DryVR views a cyber-physical system as a combination of a white-box transition graph and a black-box simulator. This alleviates the need for complete mathematical models, but at the same time exploits models when they are available. A verification algorithm for directed acyclic time-dependent transition graph is also presented in [3].

In this thesis, we present the detailed construction of the DryVR tool with several new functionalities, which includes: (a) verification on state-dependent cyclic transition graph with guard and reset functions; (b) controller synthesis that searches transition graph for given reach-avoid specification; (c) interface that allows user to connect DryVR with arbitrary black-box simulators, and (d) integration with Jupyter Notebook [4]. We also present a case study for autonomous vehicle system in this thesis, and DryVR comes with verification and controller synthesis examples to illustrate its capabilities. The evaluation of included examples is presented in later chapter shows that both verification and controller synthesis are promising starting point for DryVR to become a comprehensive verification and synthesis toolbox for practical CPS.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my adviser Professor Sayan Mitra of the ECE Department at UIUC. He has helped me in many aspects of research and life. He guided me to the verification area and motivated me to work hard on the road of computer science. This work would be impossible without his help.

I would also like to thank Chuchu Fan for providing tremendous support to my graduate life. She helped me choosing and solving research problems and guided me to explore the unknown. Also Chuchu designed and implemented the algorithm for discrepancy function and reach tube calculation, and DryVR is built based on her algorithm.

Thanks to Professor Mahesh Viswanathan, Professor Parasara Sridhar Duggirala, Matthew Potok, Suket Karanwat on the work of C2E2 verification tool. Also I would like to appreciate Minghao Jiang and Rongzhou Li for their work on DryVR verification tool. Thanks also to my lab mates Ritwika Ghosh, Nicole Chan, Hussein Sibale and Yixiao Lin that I have spent the last two years together. And thanks to Carol Wisniewski for the help she gives at CSL.

Thanks to my parents for their unconditional mental and financial supporting of my education in United States. Also Thanks to all my roommates, gym mates and game mates who helped me to ease my stress.

Finally, I would like to thank my girlfriend Xiaoyu Wang, for bringing a lot of happiness to my life.

This research was supported in part by a research grant from NSF (CSR 1422798)

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 MOTIVATION

A small failure in a system can sometimes lead to catastrophic consequences. For example, the recent crash of Uber self-driving vehicle [5] has led to public fears of autonomous cars. A pedestrian crossing a road was unfortunately killed in the incidence, and Uber had to stop road tests which may significantly slow down the evolution of self-driving technology. Another example is the recent recall on Tesla vehicles [6]. Tesla has issued a recall on 123,000 of its Model S sedans due to the risk of power steering. This is about half of the vehicles that Tesla has ever built, and will cost Tesla nearly 10 million dollars.

Control systems in modern vehicles are examples of a more general class of systems called *cyber-physical systems (CPS)*. Cyber-physical systems are systems that integrated computation, networking, and physical processes. Physical processes are couped with software and sensing using networks, and the software is used to control the physical environment. The autonomous driving system and the power steering system are examples of cyber-physical systems because the system interacts with physical environments and controlled by software. Other examples of CPS include smart grid, medical monitoring, robotic systems, and avionics.

Power steering, anti-lock brakes, adaptive cruise control, lane-following are all examples of CPS, and a fully autonomous vehicle (AV) will have many such subsystems. To check the reliability of CPS, testing is performed to find bugs in the system. While testing can help identify bugs and defects, it cannot prove that the system is bug-free. To ensure the absence of bugs in the system, one possible approach is to use formal verification. Formal verification checks all possible behaviors of the system and proves the absence of bugs. Therefore, we are motivated to build a formal verification tool for CPS and safety-critical control systems in autonomous vehicles.

1.2 BACKGROUND

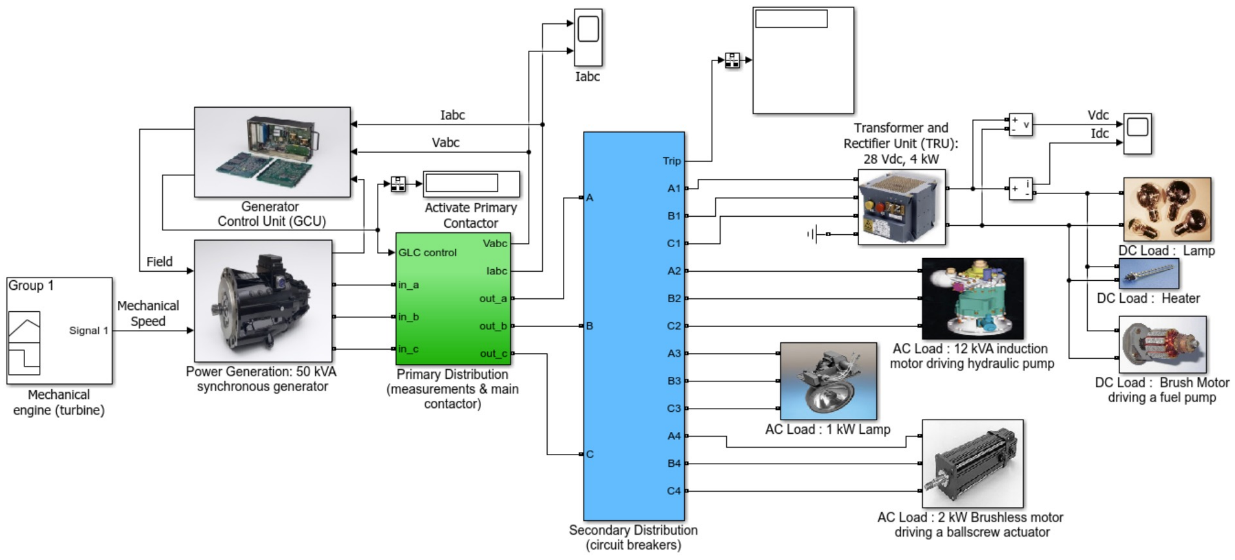
Verifying CPS is challenging since the system involves both software and physical environments. The software evolves in discrete steps, and these steps bring influences to the evolution of the physical processes. However, the state of physical environments evolves continuously with time, which is usually modeled using ordinary differential equations (ODEs). The combination of discrete and continuous steps makes formal verification challenging. CPS with bounded uncertainty in the initial set is even more challenging. This is because there is an uncountably infinite number of states in the initial set, which can lead to infinite numbers of behaviors of the system. Verifying such system would require checking the system whether all of these infinite numbers of executions satisfy the given safety specification or not. Searching for an execution that violates the safety specification is usually infeasible due to a large number of executions.

1.2.1 Verification with explicit models

For most verification tools, the system model and specification must be expressed in a mathematical framework. For example, timed automata [7] are used to model real-time systems. The software is modeled as a finite state machine, and real-value clock variables are used to model timers and stopwatches. The transitions of the finite state machine are defined by guards and reset functions, where guard functions compare clocks values and integers to decide to enable or disable transitions. The framework to model timed automata was introduced in [7]. Rectangular hybrid automata (RHA) is an extension to timed automata with skewed clocks, where the rate of evolution of a clock variable is an interval (ex. $\dot{x} \in [a, b]$). Both timed automaton and RHA are a sub-class of hybrid automata. Hybrid automata are designed to handle the more involved behavior of continuous variables. Continuous variables of hybrid automata are ODEs that can be either linear or non-linear. Hybrid automata have been used to model and analyze a variety of embedded systems including air traffic control systems, vehicle control systems and robots. Popular frameworks for modeling CPS as hybrid automata are Hybrid Automata [8] and Hybrid Input/Output Automata (HIOA) [9].

Many verification tools have been developed over last two decades require the system to be described in one of the above formalisms. UPPAAL [10] and Kronos [11] verify timed automata. HyTech [12] and Passel [13] [14] are verification tools for rectangular hybrid automata (RHA). SpaceEx [15] and Phaver [16] are designed to handle hybrid systems with linear ODEs. More general non-linear hybrid systems are handled by Flow* [17], d/dt [18], Ariadne [19], dReach [20] and C2E2 [21]. All these tools are limited to verify CPS with explicit mathematical models.

Figure 1.1: Aircraft electrical power generation and distribution systems.

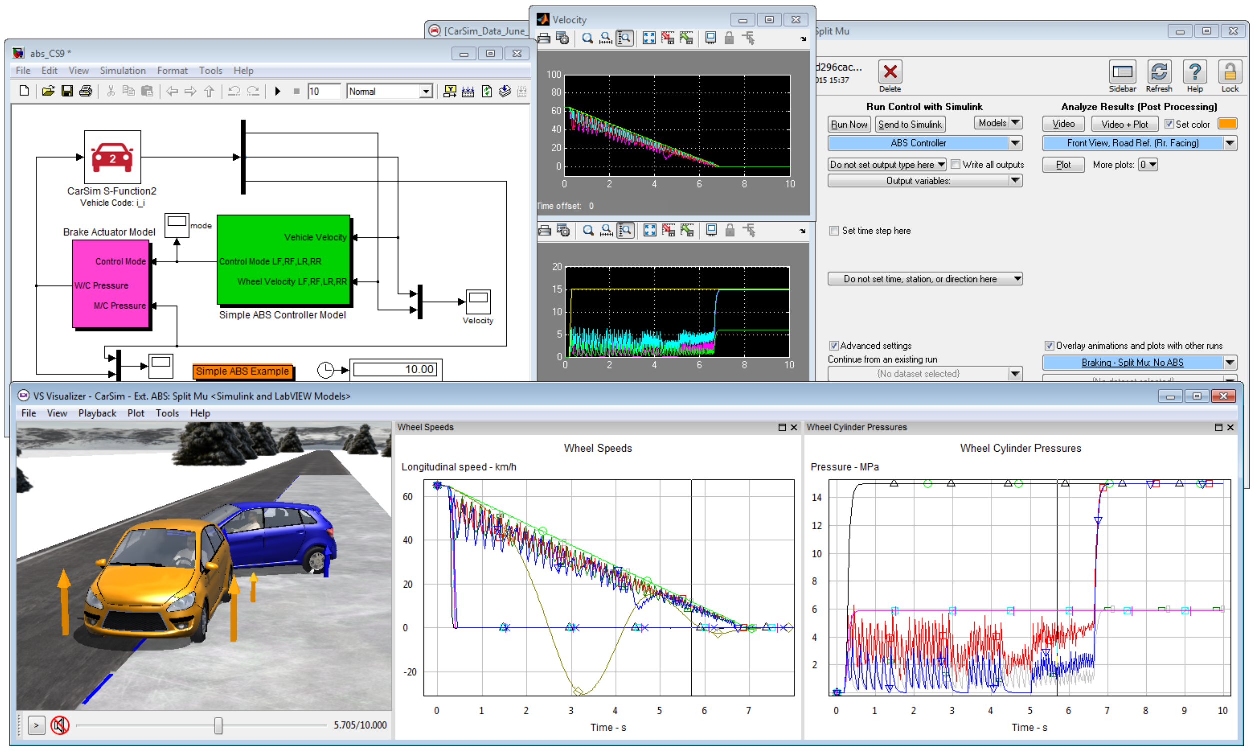

Iabc
Iabc
Vabc
Generator Control Unit (GCU)
Activate Primary Contactor
Field
Group 1
Signal 1
Mechanical Speed
Mechanical engine (turbine)
Power Generation: 50 kVA synchronous generator
GLC control
in_a
in_b
in_c
Vabc
Iabc
out_a
out_b
out_c
Primary Distribution (measurements & main contactor)
Trip
A
B
C
A1
B1
C1
A2
B2
C2
A3
B3
C3
A4
B4
C4
Secondary Distribution (circuit breakers)
Transformer and Rectifier Unit (TRU): 28 Vdc, 4 kW
Vdc
Idc
DC Load : Lamp
DC Load : Heater
DC Load : Brush Motor driving a fuel pump
AC Load : 12 kVA induction motor driving hydraulic pump
AC Load : 1 kW Lamp
AC Load : 2 kW Brushless motor driving a ballscrew actuator

1.2.2 Real world systems

For real-world cyber-physical systems, nice mathematical models describing the transitions and trajectories are often not available or are outside the reach of existing formal analysis tools. Real word control systems are composed of a mix of simulation code, look-up tables, differential equations, and diagrams. Extracting clean mathematical models from these descriptions is time-consuming and sometimes impractical. As an example, an aircraft electrical power generation and distribution system [22] modeled using Simulink [23] is shown in Figure 1.1. The system is composed of multiple components, and each component is composed of multiple sub-components, and each sub-component is composed of look-up tables, simulation code, etc. Another modeling environment is CarSim [24], which is an industrial scale vehicle simulator used by many vehicle manufacturers. The interface of CarSim is shown in Figure 1.2. Each vehicle model has hundreds of components and thousands of parameters. Again, extracting mathematical models for such a vehicle is impractical.

To address this issue, the DryVR verification approach [3] is to verify hybrid systems without complete knowledge of system dynamics. DryVR views a CPS as a combination of a white-box transition graph for mode switch information and a black-box simulator for generating simulation traces. The paper [3] presents an algorithm to verify systems that are described by this combination of white-box (transition graph) and black-box (simulator). It relieves users from extracting mathematical models from complicated

Figure 1.2: CarSim: mechanical simulation.


abs_CS9 *
CarSim S-Function2
Vehicle Code: i_i
Brake Actuator Model
Control Mode
W/C Pressure
M/C Pressure
Vehicle Velocity
Control Mode LF,RF,LR,RR
Wheel Velocity LF,RF,LR,RR
Simple ABS Controller Model
Velocity
Simple ABS Example
Velocity
Run Control with Simulink
Analyze Results (Post Processing)
ABS Controller
Front View, Road Ref. (Rr. Facing)
Braking - Split Mu: No ABS
VS Visualizer - CarSim - Ext. ABS: Split Mu <Simulink and LabVIEW Models>
Wheel Speeds
Longitudinal speed - km/h
Time - s
Wheel Cylinder Pressures
Pressure - MPa
Time - s

systems. However, the previous version of DryVR [3] only supported transition graphs that are directed and acyclic with timing-based transitions. This is a serious limitation for modeling state dependent transitions and models with loops across discrete modes.

1.3 RELATED WORK

1.3.1 Verification

There are two groups of techniques used to verify CPS, which are algorithmic and proof-theoretic. PVS [25] is a theorem prover for high order logic. It has been shown in [26] [27] [28] that certain classes of hybrid models can be formalized in its language, and then the PVS prover can be used to check invariants and simulation properties. While this is a very general method, it is hard to automate as there is limited support for automatic reasoning about real arithmetic in PVS. KeYmaera [29] is an automated and interactive theorem prover. It supports first-order dynamic logical for hybrid program [30] [31], which is a program notation for hybrid systems. KeYmaera also supports hybrid systems with non-linear differential equations, nonlinear discrete jumps, differential inequalities and nondeterministic discrete or continuous inputs. KeYmaera implements automatic proof strategies that decompose the hybrid system specification symbolically, and the level of automation is up to 100% depending on the problem. Case studies show that KeYmaera can be used to verify train control systems [32], vehicle control systems [33] and air traffic control systems [34] [35].

Algorithmic verification tools perform automatic verification using model checking. These algorithms take as input a hybrid automata and specifications and return whether the specification is satisfied or not. These verifications tools can be divided into four categories based on the type of automata they are designed to handle.

Timed automata UPPAAL [10] and Kronos [11] verify timed automata based on an algorithm that is introduced in [36]. The paper [36] establishes that verifying an invariant property of timed automata is PSPACE-complete.

Rectangular hybrid automata HyTech [12] and Passel [13] [14] are verification tools for rectangular hybrid automata (RHA). The algorithm of Hytech is established in [37], where it establishes that verifying invariant properties for RHA is undecidable. Passel attempts to automatically prove safety properties of networks of arbitrarily many interacting copies of a template hybrid automaton with rectangular dynamics. It uses a combination of invariant synthesis and inductive invariant proving.

Linear hybrid automata SpaceEx [15] and Phaver [16] are designed to handle hybrid systems with linear ODEs. Both Phaver and SpaceEx verify systems by computing numerical over-approximations. Phaver uses convex polyhedron to represent the reachable set, and the number of vertices in a convex polyhedron increases exponentially with respect to the dimensions of the system. Therefore Phaver does not work well with high dimensional systems. SpaceEx, the current state of art verification tool for linear hybrid systems, presents an efficient and scalable reachability algorithm. SpaceEx uses support functions [38] as its data structure for representing the reachable states, where support function has increased the scope of linear systems that can be verified to hundreds of state variables.

Non-linear hybrid automata Non-linear hybrid systems are handled by Flow* [17], d/dt [18], Ariadne [19], dReach [20] and C2E2 [21]. Flow* [17] and d/dt [18] use symbolic models for computing the reachable states to infer the safety of the system. Ariadne [19] uses numerical analysis to compute the reachable states in an approximate way. dReach [20] encodes the dynamic of the system as a formula and the safety verification problem as an SMT instance. dReach solves the SMT instance using its solver to verify the safety of the system. C2E2, the current state of art verification tool for both linear and non-linear hybrid systems, implements the simulation-based verification approach described in [21]. It uses discrepancy function to calculate the over-approximate reachable states of the system and gives soundness and relative completeness guarantees.

However, all tools are limited to verify CPS with explicit mathematical models. d/dt [18] and HyTech [12] are based on decidability results. SpaceEx [15], Flow* [17] and Ariadne [19] produce over-approximate reachable states based on symbolic computation. dReach [20] uses approximated decision procedures for fragments of first-order logic. C2E2 [21] is a simulation-based verification tool but still, rely on the ODEs of the system to compute the reachable states. Unlike these verification tools, DryVR is able to verify CPS without the need for explicit mathematical models.

1.3.2 Controller synthesis

One of the new functionalities in this version of DryVR is controller synthesis. Controller synthesis in DryVR finds the white-box transition graph for a given black-box simulator and a reach-avoid specification. Its algorithm is based on rapidly-exploring random trees (RRT) [39]. RRT is an algorithm designed to efficiently search high-dimensional spaces by randomly building a space-filling tree. That is, a rapidly-exploring random tree is iteratively expanded by applying different control inputs that drive the system toward randomly picked points. The RRT algorithm has been successfully applied to solve planning problems for dynamical systems. The paper [40] presented the first randomized approach to kinodynamic planning using RRT. The paper [41] proposed a randomized path planning architecture for dynamical systems in the presence of both fixed and moving obstacles. The paper [42] presented an efficient and general algorithm for nonlinear feedback control synthesis in nonlinear underactuated systems like bipedal walking based on RRT. The paper [43] used RRT to perform motion planning of aerial robot with dynamic constraints. The paper [44] introduced an algorithm to connect exploring trees using trajectory optimization methods. This algorithm efficiently reaches the exact goal of motion planning rather than halting when reaches a tolerance distance to the goal.

Variations of RRT algorithm are also introduced and applied to motion planning problems. The paper [45] presented Dynamic-Domain RRT algorithm, which significantly boosts the performance of the RRT algorithm by controlling the sample space. The paper [46] introduced RRT-connect algorithm for path planning, which is an efficient algorithm designed specifically for path planning problems that involve no differential constraints. The paper [47] presented RRT* algorithm, which claimed to achieve convergence towards the optimal solution thus ensuring asymptotic optimality along with probabilistic completeness. However, it may take infinite time to find the optimized path and the converging rate is low. The paper [48] introduced RRT*-Smart, which is designed to accelerate the converging rate of RRT* algorithm.

1.4 THESIS CONTRIBUTION AND OVERVIEW

This thesis is focused on building DryVR 2.0 with several new functionalities. This includes (a) extended verification functionality to handle state-dependent cyclic transition graph with guards and reset functions; (b) controller synthesis that searches the white-box transition graph for a given black-box simulator and a reach-avoid specification; (c) python interface that allows users to connect DryVR with arbitrary black-box simulators written in any programming languages; (d) integration with Jupyter Notebook [4].

This version of DryVR is designed to handle state dependent transitions and reset on direct cyclic transition graphs. Lots of new features are implemented to accommodate this more general transition graph. This includes procedures for computing the reachable states intersecting with guards, transforming a set of states under reset, and a new dynamic refinement strategy for finding counter-examples quickly. With the new implementation, now DryVR can handle more general hybrid models, for example, complicated traffic scenarios where the vehicles control strategies heavily rely on the relative distance with other vehicles while the complex dynamics of the vehicles are wrapped in a black-box.

The second new functionality is the controller synthesis, which automatically finds the white-box transition graph for a given black-box simulator and a reach-avoid specification. That is, given the inputs as the black-box simulator, the initial set, unsafe regions, target region, and a time bound T, DryVR controller synthesis returns a transition graph which defines a sequence of mode switches such that all executions of the resulting system reach the target within the time bound T, while maintaining a safe distance from the unsafe regions. The controller synthesis in DryVR makes randomized algorithms for planning available to systems with black-box models without complete model information. This function also frees-up users from tediously creating transition graphs manually.

This version of DryVR also comes with a python interface (function) that allows users to connect DryVR with arbitrary black-box simulators built with any programming languages, tools, and frameworks, where the old version is restricted to Python and Simulink models. Furthermore, this version of DryVR can be used in Jupyter Notebook [4]. The Jupyter Notebook serves as the user interface for DryVR, where users can visualize the verification process and interact with the reachtube or the counter-example that generated after the verification.

To demonstrate the applicability of the tool presented, we present a case study of autonomous vehicle examples and the tool comes with 26 verification examples and 6 controller synthesis examples and the evaluation of the examples is a part of the thesis. These examples demonstrate the capability of the tool. The evaluation gives promising results, which shows that DryVR can be applied to more realistic CPS.

The rest of the thesis is organized as follows. In Chapter 2, we provide background knowledges that will be used in the thesis. In Chapter 3, we give an overview of verification in DryVR and show the experiment results. In Chapter 4, we give an overview of controller synthesis in DryVR. In Chapter 5, we show the procedure of building the interface that connects the DryVR and black-box simulators and the procedure of connecting DryVR with Jupyter notebook [4]. And in Chapter 6, we conclude our work and explore the possible future directions.

CHAPTER 2: PRELIMINARIES

Suppose the system has a set of modes $\mathcal{L}$ and n continuous variables. A white-box transition graph G defines mode switches, where the transition graph is a directed graph. Its vertices contain mode information, and its edges represent the allowed switches. Switch conditions (guards) and reset functions are on the edge label. The black-box simulator is viewed as a set of trajectories $\mathcal{TL}$ in $\mathbb{R}^n$ with respect to each mode in $\mathcal{L}$. The dynamic of the system is a black-box to DryVR, but DryVR has access to a simulator that can generate sampled data points on an individual trajectory for a given initial state and mode.

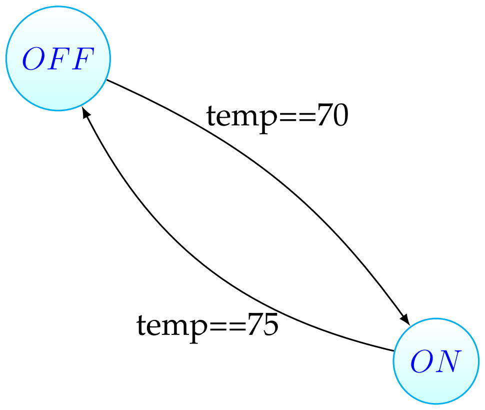
2.1 TRANSITION GRAPHS

A transition graph defines the mode switches over the finite set of modes of the system. A transition graph is a labeled directed graph $G = \langle \mathcal{L}, \mathcal{V}, \mathcal{E}, vlab, elab \rangle$ where (a) $\mathcal{L}$ is a finite set of names for the discrete modes of the system; (b) $\mathcal{V}$ is the set of vertices of the transition graph; (c) $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges; (d) $vlab : \mathcal{V} \rightarrow \mathcal{L}$ is the labeling function that labels vertices with their crossroading modes; (e) $elab$ is the labeling function the labels each edge with its guard and reset functions.

This version of DryVR supports both directed acyclic transition graphs (DAG) and directed cyclic transition graphs. For DAG, there is a nonempty subset $\mathcal{V}_{init} \subseteq \mathcal{V}$ of vertices with no incoming edges, so DryVR uses topological sort to find initial location of G, $\mathcal{L}_{init}$ = $\{\ell | \exists v \in \mathcal{V}_{init}, vlab(v) = \ell\}$. For cyclic transition graphs, users have to specify the initial vertex $\mathcal{V}_{init}$. The number of mode switches for DAG in reachability analysis is limited by the number of vertices $\mathcal{V}$ in the transition graph, but the number of mode switches for cyclic graphs is bound by the time horizon T. Ideally, it should also be bounded by the number of mode switches along any path, but this is currently not implemented. Therefore, the tool may not terminate if the model has Zeno behavior.

Consider a thermostat control system with a single continuous variable $temp$ that models the room temperature with two modes **ON** and **OFF**. The system turns on the thermostat when the room temperature drops below 70 Fahrenheit, and it turns off the thermostat when the temperature is above 75 Fahrenheit. The transition graph of the system is shown in Figure 2.1. The labels in edges are the guard functions that specify when a transition is enabled. The transition occurs when the guard is satisfied.

Figure 2.1: Transition graph for thermostat control system.



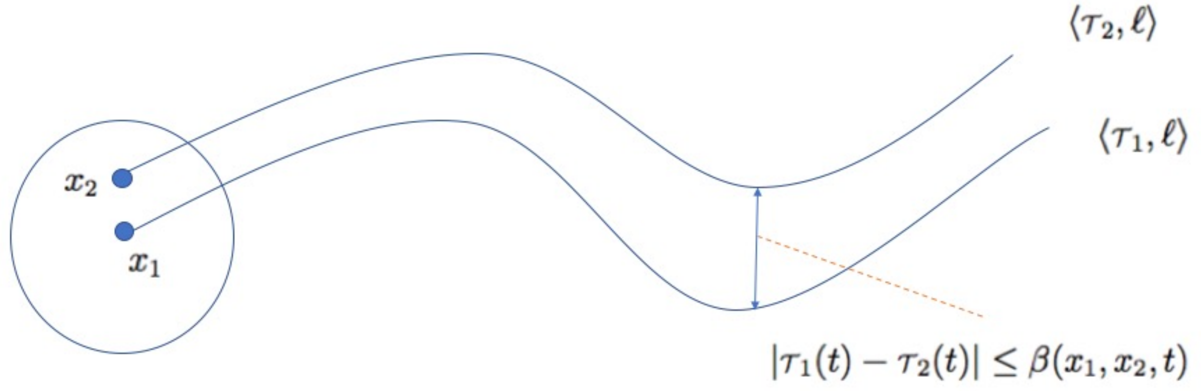
2.2 TRAJECTORIES

The continuous evolution of the variables is modeled by *trajectories*. Let *n* be the number of continuous variables in the hybrid system. A *trajectory* for a *n*-dimensional system can be expressed as a continuous function of the form $\tau : [0, T] \to \mathbb{R}^n$, where $T \geq 0$. $\tau.dom$ represents the domain of the τ, which is the time interval $[0, T]$ of the *trajectory*. $\tau.fstate$ represents the initial state of the *trajectory*, which can also denoted as $\tau(0)$. $\tau.lstate$ represents the last state of the *trajectory*, which can be denoted as $\tau(T)$ and $\tau.ltime = T$. $\mathcal{L}$ is a finite set of names for the discrete modes of the hybrid system, and each *trajectory* is labeled by a mode from $\mathcal{L}$. A labeled *trajectory* is a tuple $\langle \tau, \ell \rangle$ where $\ell \in \mathcal{L}$.

2.3 BLACK-BOX SIMULATOR

DryVR is designed to handle complicated control systems where the dynamics of systems are treated as black-boxes. DryVR uses a black-box simulator to generate a set of labeled trajectories $\mathcal{TL}$ in $\mathbb{R}^n$ for given initial states, modes and a sequence of time points from t_0 to t_k. The black-box simulator returns a sequence of states $sim(x_0, \ell, t_1)$, ..., $sim(x_0, \ell, t_k)$ such that there exists $\langle \tau, \ell \rangle \in \mathcal{TL}$ with $\tau.fstate = x_0$ and $sim(x_0, \ell, t_i) = \tau(t_i)$ for $i \in 1, ..., k$.

Figure 2.2: Discrepancy function.


$\langle \tau_2, \ell \rangle$
$\langle \tau_1, \ell \rangle$
x_2
x_1
$|\tau_1(t) - \tau_2(t)| \leq \beta(x_1, x_2, t)$

2.4 DISCREPANCY FUNCTION

The sensitivity of trajectories is formalized by the notion of a discrepancy function β [49]. A discrepancy function bounds the distance between two neighboring trajectories as a function of the distance between initial states and time. The discrepancy function is shown in figure 2.2. Given a time domain T and the trajectories $\langle \tau_1, \ell \rangle$ and $\langle \tau_2, \ell \rangle$ starting from initial state x_1 and x_2 respectively, the discrepancy function is a uniformly continuous function β of x_1, x_2, and their time domain T. A continuous function $\beta : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is the discrepancy function if for any $t \in T$, the function β bounds the distance between two trajectories as follows:

$$|\tau_1(t) - \tau_2(t)| \leq \beta(x_1, x_2, t)$$

The function also converges to 0 when the distance between x_1 and x_2 converges to 0 as follows:

$$lim_{|x_1 - x_2| \to 0} \beta(x_1, x_2, t) = 0$$

The discrepancy function can be used to compute over-approximated reachable states.

2.5 HYBRID SYSTEMS

An n-dimension hybrid system $\mathcal{H}$ described as a combination of a black-box simulator and a white-box transition graph. $\mathcal{H}$ is a tuple $\mathcal{H} = \langle \mathcal{L}, \Theta, G, \mathcal{TL} \rangle$, where (a) $\mathcal{L}$ is a finite set of names for the discrete modes; (b) $\Theta \subseteq \mathbb{R}^n$ is a compact set of initial states; (c) $G = \langle \mathcal{L}, \mathcal{V}, \mathcal{E}, elab \rangle$ is a transition graph; (d) $\mathcal{TL}$ is a set of deterministic trajectories as generated by the black-box simulator.

A *state* of $\mathcal{H}$ is a point in $\mathbb{R}^n \times \mathcal{L}$, where the set of initial states denoted as $\Theta \times \mathcal{L}_{init}$. Given an initial state $(\tau(0), l_{\mathsf{init}})$, an *execution* of the system is the sequence of trajectories $\mathsf{exec}(\tau(0), l_{\mathsf{init}}) = \langle \tau_{l_1}(t), l_1 \rangle \cdots \langle \tau_{l_k}(t), l_k \rangle$ such that (1) $l_1 \in \mathcal{L}_{init}$, $\tau_{l_1}(0) = \tau(0)$, (2) $l_1, \cdots, l_k$ follows the $\mathcal{E}$ in transition graph, (3) for each consecutive trajectory, $\tau_i.lstate = \tau_{i+1}.fstate$, and (4) for each $i > 1$, there is an edge $e \in \mathcal{E} : v_{i-1} \rightarrow v_i$ with the edge label $elab$, where v_{i-1} is the mode l_{i-1} and v_i is the mode l_i. $\tau_{l_{i-1}}$ stops at a state where the guard on $elab$ met, and τ_{l_i} starts with a state defined by the corresponding reset function of $elab$.

Finally, the set of all executions of the system is denoted as $Execs_{\mathcal{H}}$. A *state* $\langle x, \ell \rangle$ is reachable if there exists an execution that reaches this state at some time t. We denote all reachable states of the hybrid system $\mathcal{H}$ as $Reach_{\mathcal{H}}$. In the next chapter, we will discuss the verification approach to hybrid systems that are composed of white-box transition graphs and black-box simulators. That is, given an unsafe set U, how to decide whether $Reach_{\mathcal{H}} \bigcap \mathcal{U} = \varnothing$.

CHAPTER 3: VERIFICATION IN DRYVR

3.1 INPUT AND OUTPUT

Verification in DryVR takes a hybrid system $\mathcal{H}$ that is composed of a white-box transition graph G and a black-box simulator. In addition to $\mathcal{H}$, we also supply a time bound T and a safety specification (also known as unsafe set $\mathcal{U} \subseteq \mathbb{R}^n \times \mathcal{L}$). The task for DryVR is to decide whether $Reach_{\mathcal{H}} \bigcap \mathcal{U} = \varnothing$. DryVR either returns "SAFE" and an over-approximation of the reach set, or it returns "UNSAFE" and a counter-example execution. It is also possible for DryVR to return "UNKNOWN" when the refinement reaches the limit that is specified by users.

The verification input for DryVR is a JSON file. The format is listed in Figure 3.1, which includes the following parameters:

- **vertex** $\mathcal{V}$ (list of string) is a list of vertices with their corresponding mode name.
- **edge** $\mathcal{E}$ (list of tuple of int) is a list of edges that connect vertices in the transition graph, a edge is defined as a tuple (i,j) where it connects v_i and v_j.
- **variables** (list of string) is a list of the names for continuous variables in the system.
- **guards** (list of string) is a list of guard functions corresponding to each edge.
- **resets** (list of string) is a list of reset functions corresponding to each edge.
- **initialSet** Θ (list of list of float) consists of two arrays that specify the initial set for the hybrid system. The first array is the lower bound and the second array is the upper bound of variables in the system.
- **initialVertex** v_{init} (int) is an integer that specifies the initial vertex. This field is optional if the transition graph is DAG;
- **unsafeSet** $\mathcal{U}$ (string) is an unsafe set. It can be either specified per mode or all modes.
- **timeHorizon** T (float) is a time bound for the verification.
- **directory** (string) is the file path for the black-box simulator.

Figure 3.1: DryVR verification input format

```
{
  "vertex": [transition graph verteices labels (modes)],
  "edge": [transition graph edges],
  "variables": [the name of variables in the system],
  "guards": [guard functions],
  "resets": [reset functions],
  "initialSet": [two arrays defining the lower and upper bound of
     ↪  each variable],
  "initialVertex": initial vertex to start verification,
  "unsafeSet": @[mode name]:[unsafe region],
  "timeHorizon": time bound for the verification,
  "directory": directory of the folder which contains the
     ↪ simulator for black-box system,
  "bloatingMethod": bloating method, which can be either "PW" or
     ↪ "GLOBAL",
  "kvalue": k-value that used by piecewise bloating function
}
```

- **bloatingMethod** (string) specifies the type of discrepancy function used to calculate the reachable states. It can be either global discrepancy "GLOBAL" or piece-wise discrepancy "PW".

- **kvalue** (list of float) is a list of parameters used in piece-wise discrepancy function, which is specified per variable.

An example of the input file is shown in Figure 3.2, where the corresponding transition graph is shown in Figure 2.1. Kvalue is not specified in this input file since the bloating-Method is set to the global discrepancy function.

3.2 REACHTUBE COMPUTATION

Reachtube computation is a subproblem for formal verification. Finding a reachtube for the set of trajectories $\mathcal{TL}$ in a given mode is a difficult problem. Previous simulation-based verification tools with $\mathcal{TL}$ generated by white-box models approximate reachtubes using sensitivity analysis of ODEs. Since DryVR does not have access to dynamics of the system, it uses probabilistic method for estimating sensitivity from the black-box simulator.

Figure 3.2: Input file for thermostat controller

```
{
  "vertex": ["On","Off"],
  "edge": [[0,1],[1,0]],
  "variables": ["temp"],
  "guards": ["temp==70", "temp==75"],
  "resets": ["", ""],
  "initialSet": [[75.0],[76.0]],
  "initialVertex": "On",
  "unsafeSet": "@On:temp>91@Off:temp>91",
  "timeHorizon": 3.5,
  "directory": "examples/Thermostats",
  "bloatingMethod": "GLOBAL",
}
```

The sensitivity of trajectories is formalized by the notion of discrepancy functions. Since DryVR does not have the ODEs for the system, it uses a probabilistic algorithm to find the discrepancy function with only simulation traces. The algorithm is based on PAC learning linear separators [50] and is explained in [3]. The discrepancy function is used to bloat simulation trajectories such that bloated tubes over-approximate the reachable states from the initial set Θ. The bloated reachtube is represented using a sequence of hyper-rectangles. The probabilistic algorithm has a probabilistic guarantee that for any $\epsilon, \delta \in \mathbb{R}_+$, if the number of simulations is more than $[\frac{1}{\epsilon} \ln \frac{1}{\delta}]$, the learned discrepancy function has an error smaller than ϵ, with the probability greater than $1 - \delta$.

We have done several experiments with the algorithm for dozens of models with complex trajectories, where we draw random states in initial sets to simulate and learn a discrepancy function, and draw another random set of 1000 initial states to perform simulations to validate the learned discrepancy function. The experiment results show that 10 to 20 simulation traces are adequate for computing discrepancy functions. DryVR uses 10 simulations by default, and users can tune the number of simulations to get better results.

3.3 VERIFICATION ALGORITHM

We present an algorithm to solve the verification problem in this section. We introduce an algorithm $Verify$ (Algorithm 1) which takes as an input a hybrid system $\mathcal{H} = \langle \mathcal{L}, \Theta, G, \mathcal{TL} \rangle$, an unsafe set U, a time bound T and an initial vertex v_{init} (if the transition graph is not a DAG). DryVR returns "SAFE" with a reachtube, "UNSAFE" with a counter-example or "UNKNOWN" if the refinement reaches the refinement threshold.

$clacReachTube(l, \Theta, T, \mathcal{TL})$ computes the reachtube for a given mode l, an initial set Θ and a time bound T. It uses the black-box simulator to generate the simulation traces and learn the discrepancy function to calculate the reachtube. This function returns the reachtube RT, which is a sequence of hyper-rectangles.

$checkGuard(RT, g)$ checks the guard g over the reachtube RT. The guard can be specified as any linear or nonlinear predicate on the state variables or time. DryVR implements guard checking using Z3 SMT solver [51]. Therefore, the guard function must be expressions that are recognizable by the Z3 solver. This function handles guard checking with the following steps: (1) for each hyper-rectangle in the reachtube, Z3 checks if the hyper-rectangle intersects with the guard; (2) the function marks rectangles that are intersected with the guard, and then combine all marked hyper-rectangles to a single hyper-rectangle; (3) all rectangles in the reachtube that cross over the guard are abandoned. This is because all guards are treated as *urgent* in DryVR. The function returns a set of states rc and the truncated reachtube. If no hyper-rectangle intersects with the guard, the function returns $rc = \varnothing$ and the original reachtube.

$resetSet(rc, r)$ transforms a set of states rc under the reset function r. Reset functions are linear functions that can be either deterministic or non-deterministic. A deterministic reset is expressed as $x' = Ax + b$ on $elab$. Non-deterministic resets are expressed as intervals defining the lower and upper bounds. $x'[i] \in$ [lower_bound[i], upper_bound[i]], is the linear expression for i^{th} variable that needs a reset. DryVR uses Sympy [52] to map rc according to the reset function. The mapped set is collected and returned as the initial set for the next vertex.

$checkReachTube(RT, U)$ checks the safety of the reachtube with a given unsafe set. For each hyper-rectangle in the reachtube, Z3 checks if the hyper-rectangle is intersected with or fully contained in the unsafe set. If any hyper-rectangle in the reachtube is fully contained in the unsafe set, the function returns "UNSAFE", and otherwise if there exist hyper-rectangles that have non-empty intersections with the unsafe set, the function returns "UNKNOWN". If none of these conditions hold, the function returns "SAFE".

$refine(\Theta)$ refines the a given initial set. The refinement function splits the initial set into two smaller initial sets. The dimension it splits is the dimension with largest δ, where δ_i denotes the width of variable i in the initial set.

The $Verify$ Algorithm 3.1 proceeds as follows:

- Push the Θ to an initial set list $Inits$ (Line 1).
- Check the size of the initial set list and return "UNKNOWN" if the size is larger than user-defined refinement threshold $thres$ (Line 2).
- Pop the last initial set Θ from $Inits$ (Line 5).
- Get the mode label using the $vlab$ function for the given vertex v_{init} (Line 6).
- Calculate the reachtube RT with given parameters (Line 7).
- Assign an empty list to the next successors list $SUCC$, The element in $SUCC$ is a tuple $\langle next\theta, nextv, nextT\rangle$, where $next\theta$ is the initial set, $nextv$ is the vertex, and nextT is the remaining time bound (Line 8).
- Assign a null value to the longest reachtube LRT (Line 9).
- For each successor vertex that the current vertex connects, get the guard and reset functions using the edge labeling function $elab$. Calculate the next initial set $next\theta$ for next vertex and pushing the tuple $\langle next\theta, nextv, nextT\rangle$ to $SUCC$ list if the $next\theta$ is not null. Finally, overwrite the LRT if the current RT is longer than the LRT (Line 10).
- Calculate the reachtube and assign it to LRT if LRT is null (It happens when the current vertex has no outgoing edges) (Line 18).
- Check the safety of LRT with the given unsafe set. Returning "UNSAFE" if the checking result is "UNSAFE". Refine the current initial set is the checking result is "UNKNOWN" (Line 20).

- If the checking result is "SAFE", for every tuple $\langle next\theta, nextv, nextT\rangle$ in the $SUCC$, recursively call the $Verify$ function. If any of them returns "UNSAFE", then the function returns "UNSAFE". If any of them returns "UNKNOWN", mark the flag $safety$ to "UNKNOWN". Finally, returning "SAFE" if flag $safety$ is still "SAFE", otherwise refining the current initial set (Line 27).

Suppose the refinement threshold $thres$ is set to ∞, the $Verify$ Algorithm 3.1 guarantees to be sound and complete if the learned discrepancy function for each mode is correct. The algorithm requires the $clacReachTube$ function returns an over-approximation of all reachable states of the system from the initial set, which leads to the soundness of the algorithm since the discrepancy function used to calculate the reachtube satisfies this property. The system is safe if Algorithm 3.1 outputs "SAFE", where the reachable states do not intersect with the unsafe set. If it outputs "UNSAFE", then there exist at least one unsafe trajectory that starts from the initial set Θ. The algorithm also requires that as the size of the initial set gets smaller, the value of the discrepancy function will become smaller, which guarantees the algorithm will terminate. The discrepancy function also satisfies this property and leads to the relative completeness result. Algorithm 3.1 always terminates and output "SAFE" if the system is robustly safe, or outputs "UNSAFE" if there is a trajectory starting from initial set that is robustly unsafe. Combining this algorithm with the probabilistic correctness of the learning discrepancy algorithm, we can conclude this algorithm provides a probabilistic guarantee.

The complete verification process proceeds as follows: (a) randomly draw *SIMUTESTNUM* number of points in the initial set, where *SIMUTESTNUM* is a user value; (b) DryVR runs a hybrid simulation for each randomly picked point and checks the safety for the generated trajectory; (c) if hybrid simulation results are safe, DryVR uses $Verify$ algorithm to check the safety of the entire initial set.

3.4 AUTONOMOUS VEHICLE BENCHMARKS

We have created several benchmarks for autonomous driving control systems, which include common scenarios on the road such as lane merging. The hybrid system of each scenario is constructed using a model with several vehicles. The decision of the vehicles are determined by the mode in the transition graph, and the detailed dynamics of each vehicle comes from a black-box simulator.

Algorithm 3.1 *Verify*($v_{init}, \Theta, G, \mathcal{TL}, \mathcal{U}, \mathcal{T}$)

```
1:  $Inits \leftarrow \{\Theta\}$
2:  while $len(Inits) > 0$ do
3:      if $len(Inits) > thres$ then
4:          return UNKOWN
5:      $\Theta \leftarrow Inits.pop()$
6:      $l \leftarrow vlab(v_{init})$
7:      $RT \leftarrow clacReachTube(l, \Theta, T, \mathcal{TL})$
8:      $SUCC \leftarrow \{\}$
9:      $LRT \leftarrow \varnothing$
10:     for $nextv \in G.succ(v_{init})$ do
11:         $g, r \leftarrow elab(G.edges(v_{init}, nextv))$
12:         $rc, RT \leftarrow checkGuard(RT, g)$
13:         $next\theta \leftarrow resetSet(rc, r)$
14:         if $next\theta$ is not $\varnothing$ then
15:             $SUCC \leftarrow SUCC \cup \langle next\theta, nextv, \mathcal{T} - RT.ltime \rangle$
16:         if $LRT.ltime < RT.ltime$ then
17:             $LRT \leftarrow RT$
18:     if $LRT$ is $\varnothing$ then
19:         $LRT \leftarrow clacReachTube(l, \Theta, \mathcal{T}, \mathcal{TL})$
20:     $safety \leftarrow checkReachTube(LRT, U)$
21:     if $safety$ is UNSAFE then
22:         return UNSAFE
23:     else if $safety$ is UNKNOWN then
24:         $\theta_1, \theta_2 \leftarrow refine(\Theta)$
25:         $Inits \leftarrow Inits \cup \theta_1, \theta_2$
26:         continue
27:     else
28:         for $next\theta, nextv, nextT \in SUCC$ do
29:             $ret \leftarrow Verify(nextv, next\theta, G, \mathcal{TL}, \mathcal{U}, nextT)$
30:             if $ret$ is UNSAFE then
31:                 return UNSAFE
32:             else if $safety$ is UNKNOWN then
33:                 $safety \leftarrow$ UNKNOWN
34:         if $safety$ is UNKNOWN then
35:             $\theta_1, \theta_2 \leftarrow refine(\Theta)$
36:             $Inits \leftarrow Inits \cup \theta_1, \theta_2$
37:             continue
38:         return SAFE
```

Each vehicle is composed of several continuous variables, such as x,y-coordinates of the vehicle and its velocity, heading and steering angle. Each vehicle can be controlled by two input signals: throttle and steering speed. We define different modes to control the vehicle by choosing appropriate values for the throttle and the steering speed value for each mode. The modes include (a) **cruise**: moving forward with constant speed; (b) **speedup**: moving forward while accelerating constantly; (c) **brake**: moving forward while decelerating constantly; (d) **turnleft**: performing left lane switch; (e) **turnright**: performing right lane switch.

For each scenario, we analyze the x,y position of the vehicle (sx, sy) and the velocity of the vehicle in x and y direction (vx, vy). The following scenarios are constructed by defining appropriate sets of initial states and transition graphs labeled by modes of multiple vehicles. The safety requirement for these scenarios is that each vehicle needs to keep a safe distance between other vehicles. The scenarios include (a) **AutoPassing**: Vehicle A starts behind vehicle B in the right lane. And vehicle A performs a series of mode switches to overtake vehicle B. (b) **Merge**: Vehicle A in the left lane is behind vehicles B and C in the right lane. Vehicle A tries to merge in between vehicles B and C. If vehicle B starts to accelerate when vehicle A performs a lane merge, then vehicle A chooses to merge behind vehicle B. The transition graphs for **Merge** and **AutoPassing** scenarios are shown in Figure 3.3, 3.4 where the mode of each vehicle is indicated by the vertex label and the transition condition (guards) and reset functions are indicated by the edge label.

Figure 3.5 shows the plots of the reachtube computed for the **AutoPassing** scenario. Vehicle B starts in front of vehicle A and stays in cruise mode, and vehicle A performs a series of mode switches to overtake vehicle B based on its distance to vehicle B and time. Vehicle A is initially at $sx_a = 0, sy_a \in [-1.0, -0.9]$, and vehicle B is initially at $sx_b = 0, sy_b \in [-22.0, -21.9]$. Both vehicles have the same initial velocity $sy_a = sy_b = -1.0$. The figure shows the vehicles' x and y positions. Vehicle A switches to the left lane and passes vehicle B and then switches back to right lane while B remains in the right lane at a constant speed. The unsafe set $|sx_a - sx_b| < 2\&|sy_a - sy_b| < 2$ is proved to be disjoint from the reachtube.

Table 3.1 summarizes the verification results obtained using DryVR. The initial position is specified in the table for each vehicle and all vehicles have the same initial velocity $sy = -1.0$. **TH** in the table is the time horizon. **Refine** is the number of the refinement in verification, where "-" means the system is unsafe from hybrid simulations. For all

Figure 3.3: Transition graph for **Merge**.

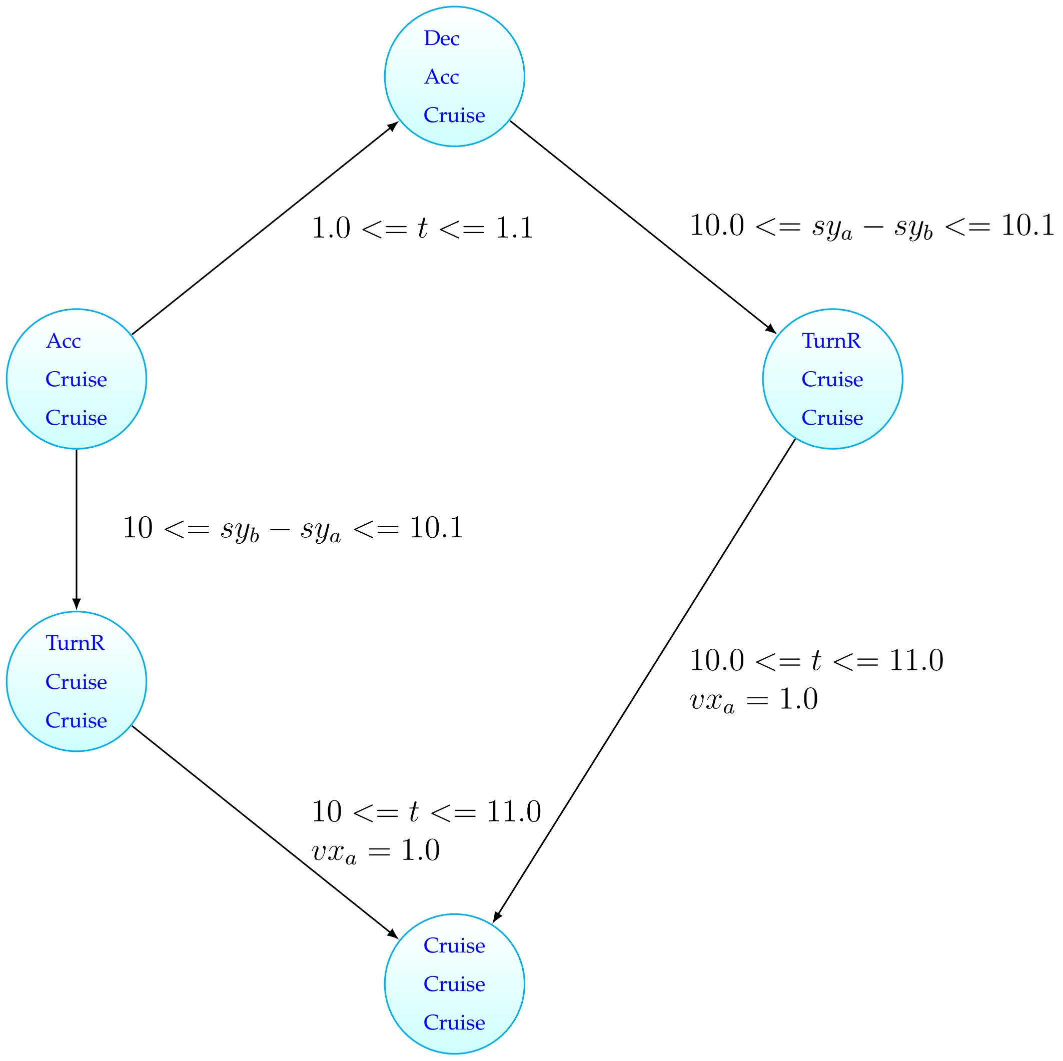


Figure 3.4: Transition graph for **AutoPassing**.

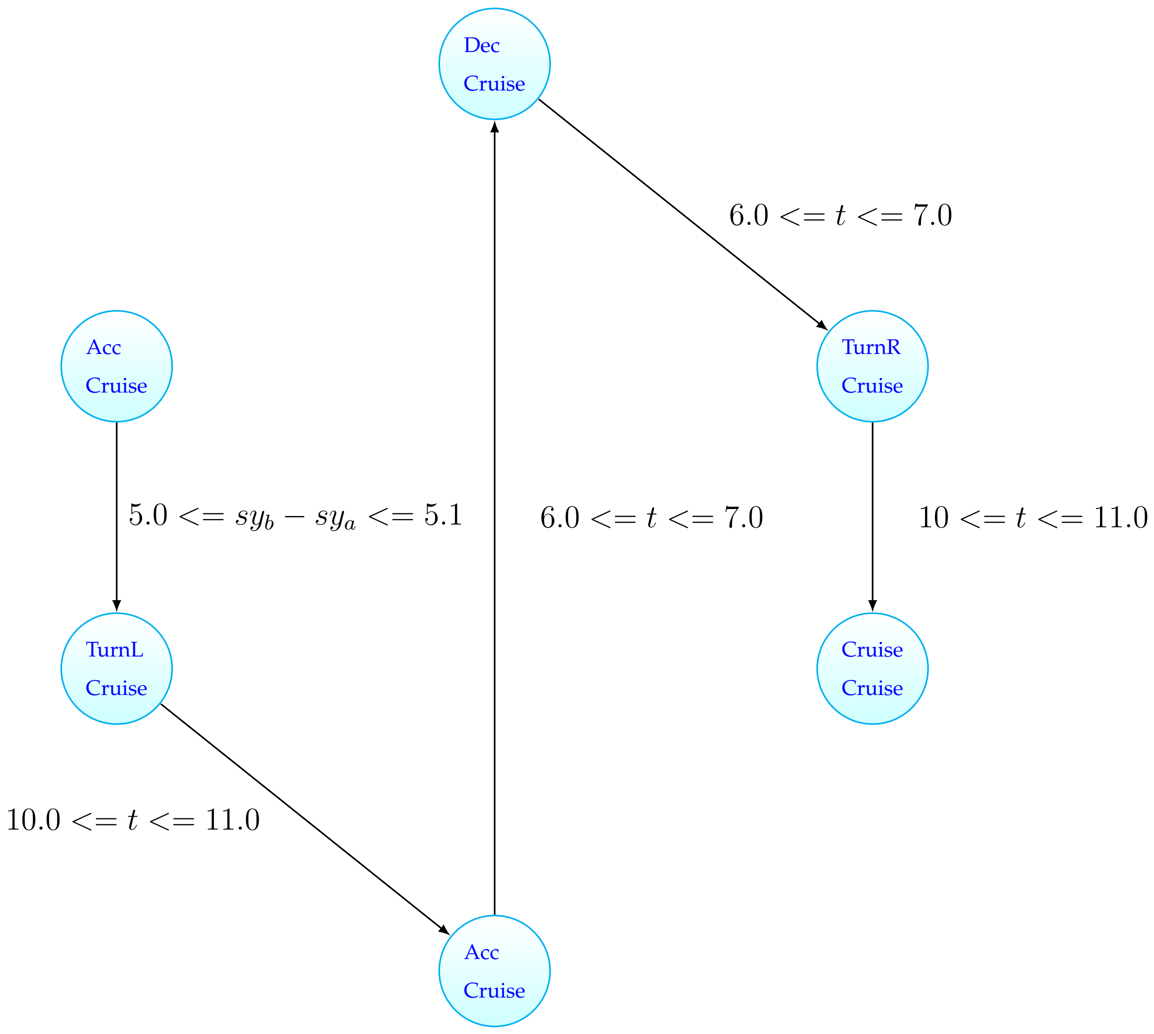
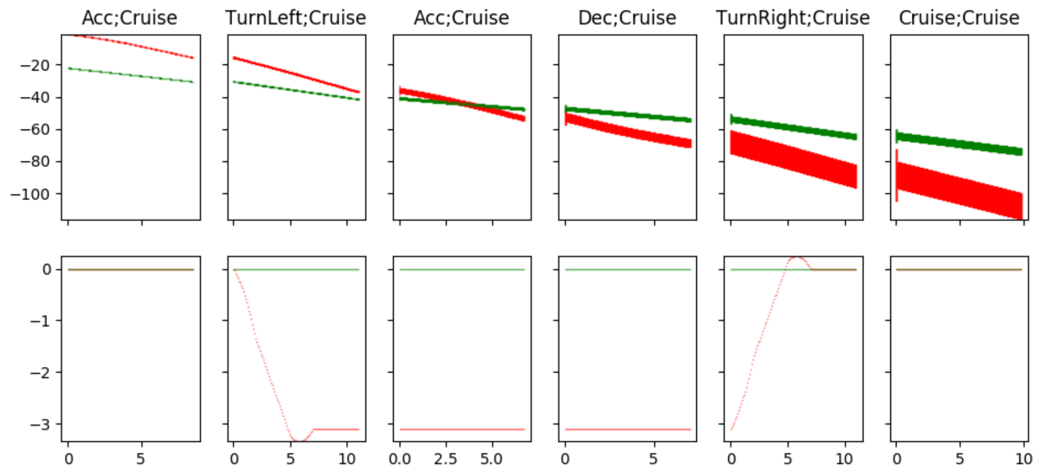


Figure 3.5: **AutoPassing** scenario verification result. The mode label is shown at the top. Red tube is Vehicle A and green tube is Vehicle B. Top:sy_a, sy_b. Bottom:sx_a, sx_b.



examples, the unsafe set is $|sx_a - sx_b| < 2\&|sy_a - sy_b| < 2$ and DryVR is able to verify the safety of the hybrid system and gives counter-example if the system is not safe.

3.5 EXPERIMENTS ON VERIFICATION

Other than autonomous vehicle benchmarks, we tested DryVR with more than two dozen systems. These systems include (a) mixed-signal circuit models with hundreds of nonlinear terms and both time and state-dependent transitions [53]; (b) high dimensional linear systems that derived from fields such as civil engineering and robotics [54]; (c) a set of 2 7 dimensional benchmarks [55].

Table 3.2 summarizes the verification performance using DryVR and Flow* [17]. These experiments were performed on a laptop with Intel Core i7-6600U CPU and 16 GB RAM. The comparison may not be fair since Flow* gives a hard guarantee while DryVR gives a hard guarantee under the assumption that the discrepancy function is correct. The assumption is empirically observed to hold. The **Simu** field indicates the running time for one hybrid simulation and total time indicates the running time for the verification. For most 2-7 dimensional benchmark examples, DryVR outperforms Flow* in terms of running time. Flow* is not able to verify circuits examples with complicated dynamics in our tests, and it reports exceptions during verification. For high dimensional linear systems, Flow* failed to get a result within 3 hours. DryVR is straightforward to use. It's

Table 3.1: Verification result.

Scenario	TH	Initial Set	Refine	Safe	running time
AutoPassing	50	$sy_a \in [-1.0, -0.9]$ $sy_b \in [-22.0, -21.9]$ $sx_a = sx_b = 0$	13	yes	72.69s
AutoPassing	50	$sy_a \in [-1.0, -0.9]$ $sy_b \in [-5.0, -4.0]$ $sx_a = sx_b = 0$	-	no	0.51s
Merge	30	$sy_a \in [0.5, 1.0]$ $sy_b \in [-1.0, -0.5]$ $sy_c \in [-32.0, -31.0]$ $sx_a = -3$ $sx_b = sx_c = 0$	34	yes	117.27s
Merge	30	$sy_a \in [0.5, 1.0]$ $sy_b \in [-1.0, -0.5]$ $sy_c \in [-20.0, -19.0]$ $sx_a = -3$ $sx_b = sx_c = 0$	1	no	8.09s

easy to code-up new examples and get verification results once users have the Python function that allows DryVR to execute the black-box simulator. The Python simulation function that connects to the black-box simulator is discussed in Chapter 5.

3.6 CONCLUSION

In this chapter, we have presented the algorithm for verifying a hybrid system composed of a white-box transition graph and a black-box simulator. We also proved its theoretical guarantees namely soundness and relative completeness. We then presented a case study of autonomous vehicles. The case study demonstrates that DryVR is capable of verifying realistic CPS without the need for explicit mathematical models. We conclude this chapter with the performance analysis of standard benchmark examples.

Table 3.2: Verification performance table.

Model	Dim	Simu (s)	Total Time(s)	Flow*(s)
Biological model I	7	0.01	0.04	66.4
Biological model II	7	0.01	0.04	223.4
Coupled Vanderpol	4	0.03	0.14	1038.3
Spring pendulum	4	0.05	0.16	1377.5
Roessler	3	0.02	0.36	17.1
Lorentz system	3	0.34	1.07	316.7
Lac operon	2	0.47	171.35	44.2
Lotka-Volterra	2	0.02	0.10	3.9
Buckling column	2	0.04	0.43	26.4
Jet engine	2	0.07	12.1s	6.8
Brusselator	2	0.10	3.02	5.2
Vanderpol	2	0.05	2.92	6.4
Vehicle platoon 3	9	0.32	4.28	21.08
Uniform nor sigmoid	3	120.91	1314.22	Exception
Uniform inverter loop	2	10.94	278.56	Exception
Uniform inverter sigmoid	2	24.87	246.76	Exception
Uniform nor ramp	3	173.77	1765.55	Exception
Uniform or ramp	4	176.70	1778.87	Exception
Uniform or sigmoid	4	168.75	2186.00	Exception
Clamped beam	348	540.80	5717.63	Time out
Building model	48	3.28	20.24	Time out
Partial differential equation	20	12.05	41.21	Time out
FOM	20	12.18	40.90	Time out
Motor control system	8	5.22	17.89	Time out
International space station	25	79.99	243.60	Time out

CHAPTER 4: CONTROLLER SYNTHESIS IN DRYVR

4.1 CONTROLLER SYNTHESIS EXAMPLE

A 5x5 arena is shown in Figure 4.1, where a yellow region represents the initial set Θ, a green region represents the target set $\mathcal{T}$, and red regions and area outside of arena are the unsafe set U. A black-box robot model with eight modes "UP", "DOWN", "LEFT", "RIGHT", "UPLEFT", "UPRIGHT", "DOWNLEFT", "DOWNRIGHT" is placed in the initial set. Each mode indicates a direction that the robot can try to move in. Starting from the initial set, the robot should reach the target set within 10 seconds and avoid colliding with the unsafe set. The robot should stay in a mode for at least 1 second before it transit to another mode. The task for DryVR is to find a transition graph that meets such specification.

4.2 INPUT AND OUTPUT

For controller synthesis, the input to DryVR is (1) a list of available modes $\mathcal{L}$; (2) a black-box simulator $\mathcal{TL}$; (3) the initial set Θ; (4) the unsafe regions U; (5) the target set $\mathcal{T}$; (6) a time bound T; (7) a minimum dwell time t_{min} The input (1),(2) and (3) forms a hybrid system $\mathcal{H}$ without a transition graph G. The task for DryVR is to find a transition graph G such that for the resulting hybrid system $\mathcal{H} = \langle \mathcal{L}, \Theta, G, \mathcal{TL} \rangle$, all executions starting from the initial set, will reach the target set within the bounded time $\mathcal{T}$, and will not intersect with any unsafe regions U. The resulting transition graph will stay in one vertex for a least t_{min} second before it transit to another vertex. DryVR either returns a computed time-dependent transition graph G if it finds one, or "FAIL" otherwise.

The json input format is shown in Figure 4.2, which includes the following parameters:

- **modes** $\mathcal{L}$ (list of string) is a list of available modes for the black-box simulator.
- **variables** (list of string) is a list of the names for continuous variables in the system.
- **initialSet** Θ (list of list of float) consists of two arrays that specify the initial set for the hybrid system. The first array is the lower bound and the second array is the upper bound of variables in the system.
- **goalSet** $\mathcal{T}$ (string) is a Z3 [51] expression that specifies the target set.

Figure 4.1: Controller synthesis of black-box robot. A yellow region represents the initial set Θ, a green region represents the target set $\mathcal{T}$, and red regions and area outside of arena are the unsafe set U. The blue tube is the reachtube

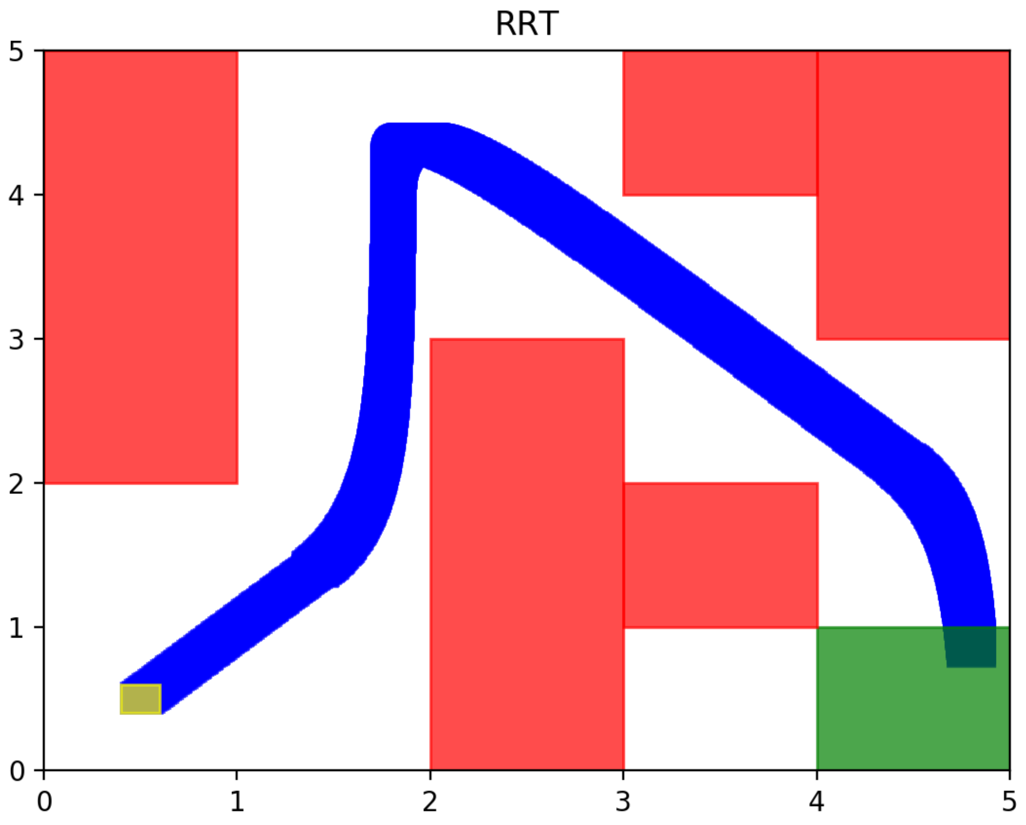


Figure 4.2: DryVR controller synthesis input format.

```
{
  "modes": [available modes],
  "variables": [the name of variables in the system],
  "initialSet": [two arrays defining the lower and upper bound of
     ↪  each variable],
  "goalSet": target region,
  "unsafeSet": @[mode name]:[unsafe region],
  "timeHorizon": time bound for resulting transition graph,
  "directory": directory of the folder which contains the
     ↪ simulator for black-box system,
  "minTimeThres": minimal dwell time,
  "goal":[[variables in goal set],[lower bound][upper bound]]
}
```

- **unsafeSet** $\mathcal{U}$ (string) is the unsafe set. It can be either specified per mode or all modes.
- **timeHorizon** T (float) is the time bound. The resulting transition graph will end in T seconds.
- **directory** (string) is the file path for the black-box simulator.
- **minTimeThres** t_{min} (float) is the minimum dwell time. The number of transitions is bounded by $[T/t_{min}]$.
- **goal** $\mathcal{T}$ (list of list) is the goal set in a different form. It is a 3-tuple where the first element is the list of variables in the target set, and the second and third elements define the lower and upper bound for each variable.

The Figure 4.3 shows the JSON input file for the example we discussed in Section 4.1. The white-box transition graph found by DryVR's controller synthesis algorithm is shown in Figure 4.4.

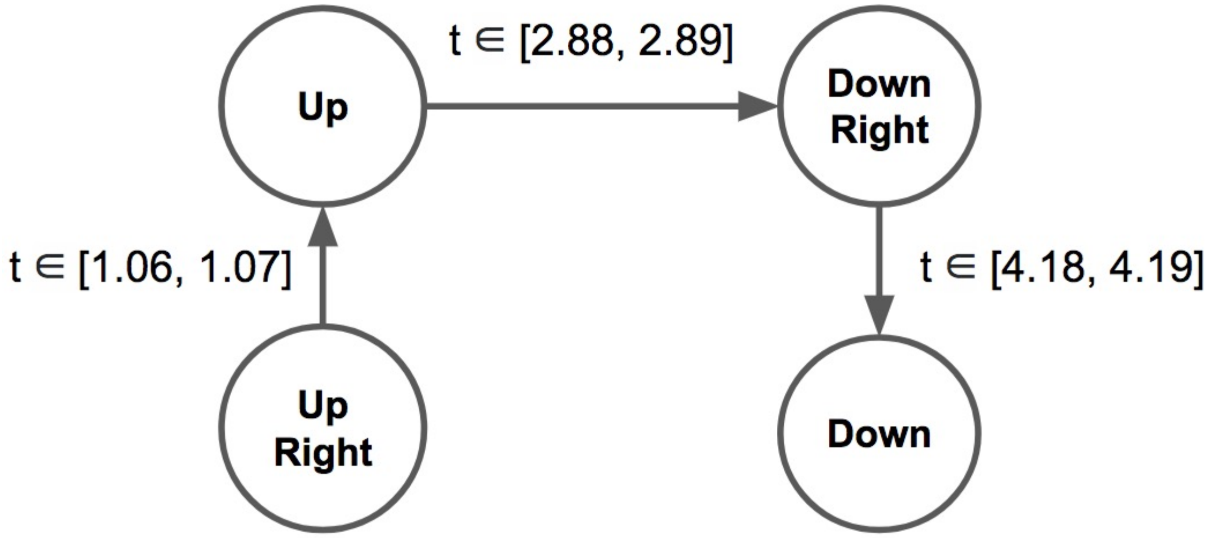
4.3 CONTROLLER SYNTHESIS ALGORITHM

The synthesis algorithm used in DryVR is based on rapidly-exploring random trees algorithm (RRT) [39]. We introduce an algorithm *GraphSearch* (Algorithm 4.1) which takes as an input a list of available modes $\mathcal{L}$, a black-box simulator $\mathcal{TL}$, a current mode $\mathcal{L}_{cur}$, an

Figure 4.3: Input file for robot controller synthesis.

```
{
  "modes":["UP", "DOWN", "LEFT", "RIGHT", "UPLEFT", "UPRIGHT", "
      ↪ DOWNLEFT", "DOWNRIGHT"],
  "variables":["x","y","vx","vy"],
  "initialSet":[[1.0,1.0,1.0,1.0],[1.1,1.0,1.0,1.0]],
  "unsafeSet":"@Allmode:Or(And(x>=2.0, x<3.0, y>=3.0, y<=4.0),
      ↪ And(x>=3.0, x<=4.0, y>=2.0, y<3.0), x<0, x>5, y<0, y>5)",
  "goalSet":"And(x>=3.0, x<=4.0, y>=3.0, y<=4.0)",
  "timeHorizon":10.0,
  "minTimeThres":1.0,
  "directory":"examples/carinmaze",
  "goal":[["x","y"],[3.0,3.0],[4.0,4.0]]
}
```

Figure 4.4: Transition graph for the robot example.



initial set Θ, an unsafe set U, a target set $\mathcal{T}$, a time bound T and a minimal dwell time t_{min}. This algorithm either returns "true" if it finds the transition graph or "false" otherwise.

$getSafeTube(RT, U)$ truncates the reachtube before its first collision with the unsafe set. For each hyper-rectangle in the reachtube RT, Z3 SMT solver [51] checks the intersection between the hyper-rectangle and the unsafe set U. Once a hyper-rectangle intersects with the unsafe set, the function truncates the reachtube, keeps the hyper-rectangles before the first intersected one, and returns the resulting reachtube.

$reachTarget(RT, \mathcal{T})$ checks if the reachtube reaches the target region. For each hyper-rectangle in the reachtube RT, Z3 checks the intersection between the hyper-rectangle and the target set $\mathcal{T}$. The function returns "true" if there is a hyper-rectangle that is fully-contained in the target set.

$randomPick(RT, t_{min})$ randomly picks *RANDSECTIONNUM* number of sets along the reachtube RT, where *RANDSECTIONNUM* is a constant parameter specified by users. Each picked set's time interval is larger than t_{min} second. This function returns a list of randomly picked sets RS.

$sortByDistance(RS, \mathcal{T})$ takes a list of sets RS and sorts the sets based on their Euclidean distance to the target set $\mathcal{T}$ and returns the sorted list.

$shuffle(\mathcal{L})$ shuffles the list of available modes and returns the shuffled list.

The $GraphSearch$ Algorithm 4.1 proceeds as follows:

- check if the remaining time bound T is larger than minimal dwell time t_{min}(Line 1).
- Calculate the reachtube with the given mode $\mathcal{L}_{cur}$, the initial set Θ and the time bound T (Line 3).
- Truncate the reachtube before its first collision with the unsafe set U (Line 4).
- Check if the truncated reachtube still holds the minimal dwell time constraint (Line 5).
- Check if the reachtube reaches the target set $\mathcal{T}$ (Line 7).
- Randomly pick *RANDSECTIONNUM* sets along the reachtube RT (Line 9).

- Sort the list of randomly picked small sets $RS\mathcal{T}$ (Line 10).
- Shuffle all available modes (Line 11).
- For each set in the RS and for each mode in the shuffled modes, calculate the remaining time and call the GraphSearch function recursively to proceed (Line 12).

DryVR uses $GraphSearch$ function with $\mathcal{L}_{cur}$ set to l for all l in $\mathcal{L}$. Therefore, the controller synthesis in DryVR does not require users to give an initial mode.

Algorithm 4.1 $GraphSearch(\mathcal{L}, \mathcal{TL}, \mathcal{L}_{cur}, \Theta, U, \mathcal{T}, T, t_{min})$

```
1: if $T < t_{min}$ then
2:     return false
3: $RT \leftarrow clacReachTube(\mathcal{L}_{cur}, \Theta, T, \mathcal{TL})$
4: $RT \leftarrow getSafeTube(RT, U)$
5: if $RT.ltime < t_{min}$ then
6:     return false
7: if $reachTarget(RT, \mathcal{T})$ is true then
8:     return true
9: $RS \leftarrow randomPick(RT, t_{min})$
10: $RS \leftarrow sortByDistance(RS, \mathcal{T})$
11: $modes \leftarrow shuffle(\mathcal{L})$
12: for $set$ in $RS$ do
13:     for $mode$ in $modes$ do
14:         $remainT \leftarrow T - set.ltime$
15:         if $GraphSearch(\mathcal{L}, \mathcal{TL}, mode, set, U, \mathcal{T}, remainT, t_{min})$ is true then
16:             return true
17: return false
```

The synthesis algorithm is sound if the learned discrepancy function is correct. That is, given the synthesized transition graph, any execution of the hybrid system reaches the target set $\mathcal{T}$ without colliding with the unsafe set U. The algorithm returns "Fail" if it does not find a transition graph. Returning "Fail" does not mean the transition graph does not exist with the given specification since the algorithm is non-deterministic.

4.4 EXPERIMENTS ON CONTROLLER SYNTHESIS

We have tested the performance of the controller synthesis on 6 control synthesis examples, and the results are shown in Table 4.1. These examples include (a) a vehicle collision avoidance system, where the vehicle needs to switch a lane to avoid an obstacle in front

Table 4.1: Control synthesis performance table.

Example	Dim	T(s)	t_{min}(s)	Running time(s)
Vehicle	4	50	2.0	1896.26
Robot	4	10.0	1.0	98.93
Motion plan	3	6.0	1.0	4.55
DC motor	2	1.0	0.1	0.35
Room heating	3	25.0	2.0	2.66
Pendulum	2	2.0	0.2	6.06

of the vehicle and switch back afterward; (b) a robot must find a path in the maze; (c) a robot must reach a target while avoid colliding with a large obstacle[56]; (d) a DC motor where the speed of the motor needs to be regulated in a certain range [56]; (e) a room heating problem, where the controller must keep the temperature of 3 rooms around 21 Celsius [57]; (f) an inverted pendulum must be stabilized around the unstable equilibrium point [57]. The algorithm is able to find transition graphs for these examples, but the algorithm is non-deterministic and the running time varies in different executions for the same example. Typically these examples can be complete in few minutes. We ran each example five times to record the average running time.

4.5 CONCLUSION

In this chapter, we presented the input format for DryVR controller synthesis. We then presented controller synthesis algorithm, which automatically finds the white-box transition graph for a given black-box simulator and a reach-avoid specification. We conclude this chapter with the performance analysis of standard controller synthesis benchmark examples.

CHAPTER 5: INTERACTION WITH DRYVR

5.1 CONNECTING BLACK-BOX SIMULATOR

DryVR is an open source project written in Python programming language. To connect arbitrary black-box simulators, users have to provide a Python simulation function to DryVR. DryVR uses the simulation function to get simulation traces from the black-box simulator. The input to the simulation function is an initial condition, a mode label l and a time bound T, and the output is a simulation trace Sim. The simulation trace is a 2-dimensional list of float numbers, the first index indicates the number of steps, and the second index indicates the variable in the system. The "0" for the second index in the Sim is always the time. For example, $Sim[0][1]$ is a float value of the first variable in the system at the first time step and $Sim[1][0]$ is a float value of the time at the second time step.

The Python simulation function signature is shown in Listing 5.1. Users should provide a function with the same name and input order. DryVR will search this simulation function in the user provided directory. There is no limitation on the implementation of the simulation function as long as it returns the simulation trace Sim with given format. DryVR has successfully verified hybrid systems with different kinds of simulators. For example, the simulation function for the circuits example [53] we showed in section 3.5 uses a shell script to invoke an executable that compiled using ODEINT simulator [58] and returns output traces to DryVR. For a spacecraft model [59], $TC\_Simulate$ function connects a simulink [60] model and generates simulation results using it. Detailed guidance for creating the $TC\_Simulate$ is available at DryVR's manual `http://dryvr-02.readthedocs.io/en/latest/dryvr's_language.html`

```
def TC_Simulate(Modes,initialCondition,time_bound):
    Sim = /*interact with black box simulator*/
    return Sim
```

Listing 5.1: Black-box simulation function

5.2 DRYVR CONFIGURATION

DryVR allows users to tune some parameters to get better verification and controller synthesis results. Available parameters are:

- *SIMUTESTNUM* **(default value:1)** is the number of hybrid simulations, which is introduced in section 3.3;
- *SIMTRACENUM* **(default value:10)** is the number of simulations used to learn the discrepancy function;
- *REFINETHRES* **(default value:10)** is the refinement threshold in the verification algorithm;
- *RANDSECTIONNUM* **(default value:3)** is the number of random sets picked in the reachtube in controller synthesis algorithm.

These parameters can be changed in the configuration file located in DryVR source code folder.

5.3 CONNECTING DRYVR WITH JUPYTER NOTEBOOK

Jupyter Notebook [4] extends the console-based interactive shell, providing a web-based application suitable for capturing the whole computation process and allows users to communicate with the result. It has two components:(a) **A web application**: Jupyter notebook launches a local server so that users can access documents using a browser, where documents contain text, mathematics, computations and rich media representations of objects. (b) **Notebook documents**: a representation of all contents visible in the web application, such as inputs and outputs of the computations, text, mathematics and rich media representations of objects.

Connecting DryVR with Jupyter notebook has several benefits, which including (a) use DryVR as a python library; (b) store and load computation results using notebook documents; (c) visualize the verification process; (d) interact with the reachtube or counter-example after the verification.

An example of notebook document is shown in Listing 5.3. The example proceeds as following:(a) import the verify function from DryVR and the $TC\_Simulate$ function (Line

1); (b) enable the interactive mode of Matplotlib [61] in notebook to visualize the verification process; (c) write the input file we introduced in Section 3.1 as a python dictionary object (Line 4); (d) write the optional configuration parameters as a python dictionary object (Line 13); (e) verify the example with the given input, the simulation function and the configuration parameters (Line 16).

```
 1  from src.core.dryvrmain import verify
 2  from \*simulator directory*\ import TC_Simulate
 3  %matplotlib notebook
 4  args = {
 5     "vertex":["On","Off","On"],
 6     "edge":[[0,1],[1,2]],
 7     "variables":["temp"],
 8     "guards":["And(t>1.0,t<=1.1)","And(t>1.0,t<=1.1)"],
 9     "initialSet":[[75.0],[76.0]],
10     "unsafeSet":"@On:temp>91@Off:temp>91",
11     "timeHorizon":3.5,
12  }
13  config = {
14     "SIMUTESTNUM":2
15  }
16  reach = verify(args, TC_Simulate, paramConfig=config)
```

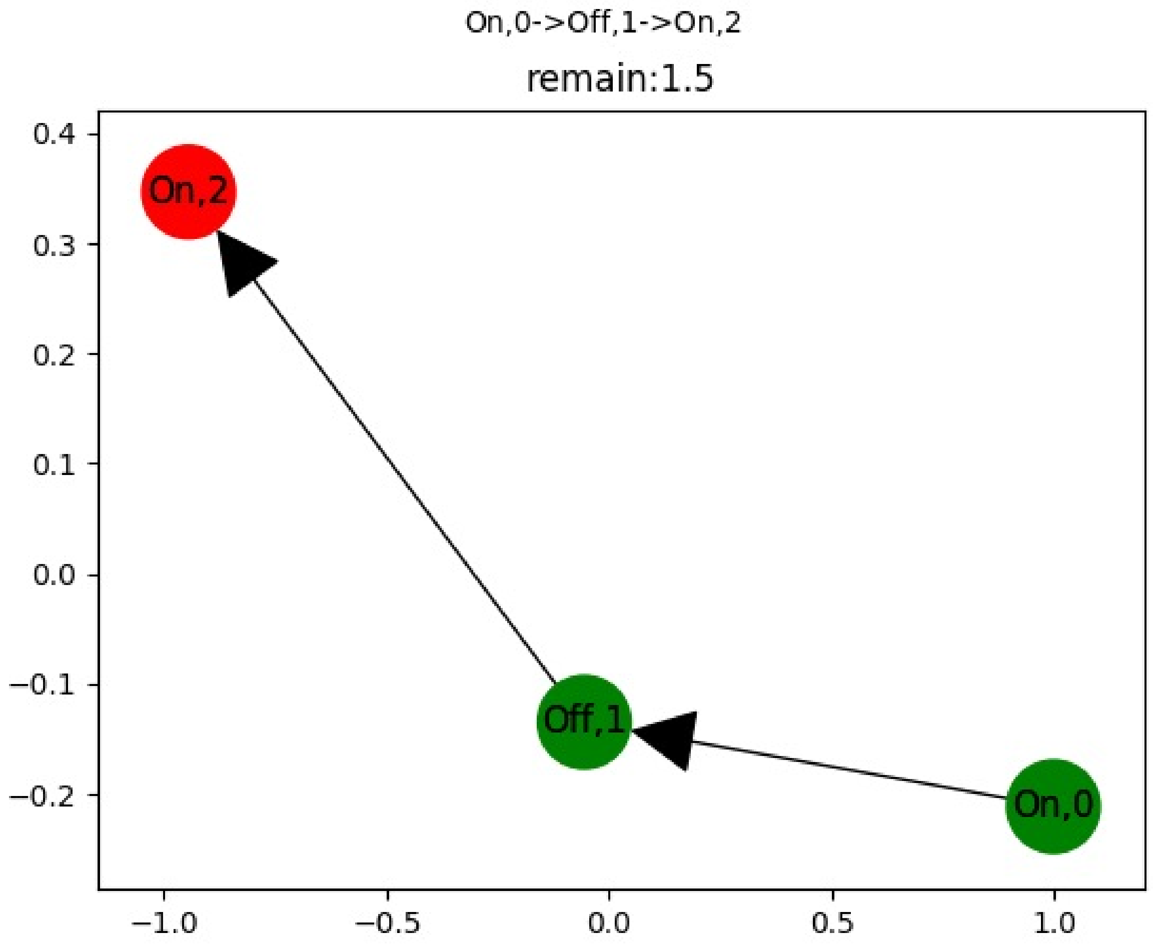
Listing 5.2: DryVR in Jupyter notebook

During the verification, Jupyter notebook will display a live transition graph in its interface with the remaining time bound, which is shown in Figure 5.1. The red vertex is the current vertex, and the graph gets updated when a transition happens. The verify function returns a reachtube object contains the reachtube if the system is safe, or a counter-example if the system is unsafe. The reachtube object has functionalities such as filtering and plotting.

5.4 CONCLUSION

In this chapter, we first showed the way of constructing a Python simulation function that connects DryVR and black-box simulators. By creating the python simulation function, users can use DryVR to verify CPS modeled with arbitrary programming languages,

Figure 5.1: Live transition graph. Red vertex is the vertex DryVR is verifying currently.



software, and frameworks. We then presented user-configurable parameters in DryVR so that users can make changes to these parameters to get better verification and controller synthesis results. We finally showed the way of using DryVR in Jupyter notebook, which allows users to visualize the verification process, interact with the verification results, and store computation results.

CHAPTER 6: CONCLUSION AND FUTURE WORK

We presented DryVR in this thesis, which is a tool that performs verification and controller synthesis on CPS without the need for explicit mathematical models. The hybrid system that DryVR handles is a combination of a white-box transition graph and a black-box simulator. Both verification and controller synthesis rely on the algorithm to find discrepancy functions with pure simulation traces [3]. The correctness of the discrepancy function hold with a high probability. Assuming that the discrepancy function is correct, the verification and synthesis results are sound.

We evaluated the performance of verification and controller synthesis. And we compared the verification performance between DryVR and Flow*. Results show that DryVR outperformed Flow* significantly on most examples, especially on models with higher dimension or with complex dynamics. The performance for controller synthesis is highly non-deterministic, but most scenarios can be completed within few minutes.

We showed how to build Python simulation function to connect DryVR with arbitrary black-box simulators so that users can verify more general and complicated systems. We also presented the way of connecting DryVR with Jupyter Notebook to make DryVR straightforward to use. Users can use DryVR interactively in the Jupyter Notebook, and all the computation results will be stored in the document.

The future directions of DryVR include: (a) extends DryVR's functionality to synthesize for temporal logic; (b) design a web-based user interface and building a web-based backed to utilize the strong computation power of web services; (c) implement RRT* algorithm for controller synthesis to find more optimized transition graphs.

REFERENCES

[1] P. Koopman and M. Wagner, “Challenges in autonomous vehicle testing and validation,” *SAE International Journal of Transportation Safety*, vol. 4, no. 1, pp. 15–24, 2016.

[2] S. Shalev-Shwartz, S. Shammah, and A. Shashua, “On a formal model of safe and scalable self-driving cars,” *arXiv preprint arXiv:1708.06374*, 2017.

[3] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, “Dryvr: Data-driven verification and compositional reasoning for automotive systems,” in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 441–461.

[4] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay et al., “Jupyter notebooks-a publishing format for reproducible computational workflows.” in *ELPUB*, 2016, pp. 87–90.

[5] M. della Cava, “Uber self-driving car kills arizona pedestrian, realizing worst fears of the new tech,” *USA TODAY*. [Online]. Available: https://www.usatoday.com/story/tech/2018/03/19/uber-self-driving-car-kills-arizona-woman/438473002/

[6] B. Raven, “Tesla recalls 123k model s sedans due to corrosion risks in winter conditions,” *MICHIGAN AUTOMOTIVE NEWS*. [Online]. Available: http://www.mlive.com/auto/index.ssf/2018/04/tesla_model_s_recall.html

[7] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[8] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems,” in *Hybrid systems*. Springer, 1993, pp. 209–229.

[9] S. Mitra, “A verification framework for hybrid systems,” Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[10] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, “UppaalâĂŤa tool suite for automatic verification of real-time systems,” in *International Hybrid Systems Workshop*. Springer, 1995, pp. 232–243.

[11] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, “Kronos: A model-checking tool for real-time systems,” in *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 1998, pp. 298–302.

[12] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, “Hytech: A model checker for hybrid systems,” *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 110–122, 1997.

[13] T. T. Johnson and S. Mitra, “Anonymized reachability of rectangular hybrid automata networks,” *Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2014.

[14] T. T. Johnson, *Uniform verification of safety for parameterized networks of hybrid automata*. University of Illinois at Urbana-Champaign, 2013.

[15] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 379–395.

[16] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech," in *International workshop on hybrid systems: computation and control*. Springer, 2005, pp. 258–273.

[17] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *CAV*. Springer, 2013, pp. 258–263.

[18] E. Asarin, T. Dang, and O. Maler, "The d/dt tool for verification of hybrid systems," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 365–370.

[19] A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A. L. Sangiovanni-Vincentelli, "Ariadne: a framework for reachability analysis of hybrid automata," in *In: Proceedings of the International Syposium on Mathematical Theory of Networks and Systems.(2006*. Citeseer, 2006.

[20] S. Kong, S. Gao, W. Chen, and E. Clarke, "dreach: δ-reachability analysis for hybrid systems," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 200–205.

[21] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2e2: a verification tool for stateflow models," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 68–82.

[22] "Aircraft electrical power generation and distribution," https://www.mathworks.com/help/physmod/sps/examples/aircraft-electrical-power-generation-and-distribution.html, accessed: 2018-04-19.

[23] "Model-based design: From concept to code," https://www.mathworks.com/products/simulink.html, accessed: 2018-04-08.

[24] R. Benekohal and J. Treiterer, "Carsim: Car-following model for simulation of traffic in normal and stop-and-go conditions," *Transportation research record*, no. 1194, 1988.

[25] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *International Conference on Automated Deduction*. Springer, 1992, pp. 748–752.

[26] S. Mitra and K. M. Chandy, "A formalized theory for verifying stability and convergence of automata in PVS," in *TPHOLs*, ser. Lecture Notes in Computer Science, vol. 5170. Springer, 2008, pp. 230–245.

[27] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2007.

[28] H. Lim, D. K. Kaynar, N. A. Lynch, and S. Mitra, "Translating timed I/O automata specifications for theorem proving in PVS," in *FORMATS*, ser. Lecture Notes in Computer Science, vol. 3829. Springer, 2005, pp. 17–31.

[29] A. Platzer and J.-D. Quesel, "Keymaera: A hybrid theorem prover for hybrid systems (system description)," in *International Joint Conference on Automated Reasoning*. Springer, 2008, pp. 171–178.

[30] A. Platzer, "Differential dynamic logic for verifying parametric hybrid systems," in *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2007, pp. 216–232.

[31] A. Platzer, "Differential dynamic logic for hybrid systems," *Journal of Automated Reasoning*, vol. 41, no. 2, pp. 143–189, 2008.

[32] A. Platzer and J.-D. Quesel, "European train control system: A case study in formal verification," in *International Conference on Formal Engineering Methods*. Springer, 2009, pp. 246–265.

[33] S. M. Loos, A. Platzer, and L. Nistor, "Adaptive cruise control: Hybrid, distributed, and now formally verified," in *International Symposium on Formal Methods*. Springer, 2011, pp. 42–56.

[34] A. Platzer and E. M. Clarke, "Formal verification of curved flight collision avoidance maneuvers: A case study," in *International Symposium on Formal Methods*. Springer, 2009, pp. 547–562.

[35] J.-B. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer, "A formally verified hybrid system for the next-generation airborne collision avoidance system," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 21–36.

[36] R. Alur and D. L. Dill, "Automata-theoretic verification of real-time systems," *Formal Methods for Real-Time Computing*, pp. 55–82, 1996.

[37] R. Alur, T. A. Henzinger, and P.-H. Ho, "Automatic symbolic verification of embedded systems," *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 181–201, 1996.

[38] C. Le Guernic and A. Girard, "Reachability analysis of linear systems using support functions," *Nonlinear Analysis: Hybrid Systems*, vol. 4, no. 2, pp. 250–262, 2010.

[39] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[40] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[41] E. Frazzoli, M. A. Dahleh, and E. Feron, "Real-time motion planning for agile autonomous vehicles," *Journal of Guidance, Control, and Dynamics*, vol. 25, no. 1, pp. 116–129, 2002.

[42] R. Tedrake, "Lqr-trees: Feedback motion planning on sparse randomized trees," 2009.

[43] J. Kim and J. P. Ostrowski, "Motion planning a aerial robot using rapidly-exploring random trees with dynamic constraints," in *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, vol. 2. IEEE, 2003, pp. 2200–2205.

[44] F. Lamiraux, E. Ferré, and E. Vallée, "Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods," in *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, vol. 4. IEEE, 2004, pp. 3987–3992.

[45] A. Yershova, L. Jaillet, T. Siméon, and S. M. LaValle, "Dynamic-domain rrts: Efficient exploration by controlling the sampling domain," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2005, pp. 3856–3861.

[46] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, vol. 2. IEEE, 2000, pp. 995–1001.

[47] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *Robotics Science and Systems VI*, vol. 104, p. 2, 2010.

[48] J. Nasir, F. Islam, U. Malik, Y. Ayaz, O. Hasan, M. Khan, and M. S. Muhammad, "Rrt*-smart: A rapid convergence implementation of rrt," *International Journal of Advanced Robotic Systems*, vol. 10, no. 7, p. 299, 2013.

[49] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*. IEEE Press, 2013, p. 26.

[50] A. B. Kurzhanski and P. Varaiya, "Ellipsoidal techniques for reachability analysis: internal approximation," *Systems & control letters*, vol. 41, no. 3, pp. 201–211, 2000.

[51] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[52] D. Joyner, O. Čertík, A. Meurer, and B. E. Granger, "Open source computer algebra systems: Sympy," *ACM Communications in Computer Algebra*, vol. 45, no. 3/4, pp. 225–234, 2012.

[53] J. Maier, "Modeling the cmos inverter using hybrid systems," E182 - Institut für Technische Informatik; Technische Universität Wien, Tech. Rep. TUW-259633, 2017. [Online]. Available: http://publik.tuwien.ac.at/files/publik_259633.pdf

[54] H.-D. Tran, L. V. Nguyen, and T. T. Johnson, "Large-scale linear systems from order-reduction (benchmark proposal)," in *3rd Applied Verification for Continuous and Hybrid Systems Workshop (ARCH), Vienna, Austria*, 2016.

[55] "Benchmarks of continuous and hybrid systems," https://ths.rwth-aachen.de/research/projects/hypro/benchmarks-of-continuous-and-hybrid-systems/, accessed: 2018-01-29.

[56] M. Mazo Jr, A. Davitian, and P. Tabuada, "Pessoa: A tool for embedded controller synthesis." in *CAV*, vol. 6174. Springer, 2010, pp. 566–569.

[57] H. Ravanbakhsh and S. Sankaranarayanan, "Robust controller synthesis of switched systems using counterexample guided framework," in *Embedded Software (EMSOFT), 2016 International Conference on*. IEEE, 2016, pp. 1–10.

[58] K. Ahnert and M. Mulansky, "Odeint–solving ordinary differential equations in c++," in *AIP Conference Proceedings*, vol. 1389, no. 1. AIP, 2011, pp. 1586–1589.

[59] N. Chan and S. Mitra, "Verified hybrid lq control for autonomous spacecraft rendezvous," 2017.

[60] M. Grant, S. Boyd, and Y. Ye, "Cvx: Matlab software for disciplined convex programming," 2008.

[61] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in science & engineering*, vol. 9, no. 3, pp. 90–95, 2007.