Verification of Camera-Based Autonomous Systems

Habeeb P, Nabarun Deka, Deepak D'Souza, Kamal Lodaya, Pavithra Prabhakar

Abstract—**We consider the problem of verifying the safety of the trajectories of a camera-based autonomous vehicle in a given 3D-scene. We give a procedure to verify that all trajectories starting from a given initial region reach a specified target region safely without colliding with obstacles on the way. We also give a prioritization-based falsification procedure that collects unsafe trajectories. Both our procedures are based on the key notion of image-invariant regions, which are regions within which the captured images are identical. We evaluate our methods on a model of an autonomous road-following drone in a variety of 3D-scenes; our experimental results demonstrate the feasibility and benefits of our approach for both safety analysis and falsification.**

Index Terms—**Autonomous vehicles, cameras, mobile robots, software verification and validation.**

I. Introduction

SELF-DRIVING or highly autonomous technologies are being embraced by vehicle manufacturers worldwide due to their promise of safer rides (most road accidents are attributed to driver errors), accessibility to challenged users, and ability to maneuver in environments dangerous to humans, to name a few. Such technologies typically rely on inputs from camera, lidar, infrared, and other sensors, which are often processed by neural network based controllers to control the vehicle's trajectory. To realize the promise of these technologies, it is imperative that the closed-loop control system of an autonomous vehicle along with the perception modules are reliable, in order to avoid potential mishaps. Testing the vehicle in real terrains or test tracks helps in debugging and gaining confidence in the reliability of the system, but is expensive and affords very limited coverage. A promising alternative is to reason about a model of the vehicle in simulated (or synthetic) environments. Both safe and unsafe trajectories from a simulated environment are known to transfer well to real environments. Fremont et al [1] report that 62.5% of unsafe simulated behaviours could be reproduced as unsafe behaviours in a real track, while 93.3% of safe simulated behaviours continued to be safe in a real track. Thus, analysis in a simulated environment can give us an effective way to debug and gain confidence in our system.

One could perform a variety of analyses in a synthetic environment, including executing the vehicle model from some initial positions (testing/simulation), looking for unsafe trajectories (directed testing or falsification), and verifying whether *all* trajectories from a given, potentially infinite, initial region are safe (verification). While simulation and falsification approaches have been widely used in this domain (see [2]–[6] to name a few), verification has received much less attention. O'Kelly et al [7] and Sun et al [8] consider the verification problem in the setting of gap and lidar sensors respectively. However, none of these works model camera-based sensors, which are one of the key sensor inputs used by autonomous vehicles, and pose novel challenges when compared to "continuous" sensors like lidars addressed in [8].

In this paper, we address this gap by considering the problem of verifying the safe trajectory of a camera-based autonomous vehicle in a given synthetic 3D-scene, with a specified initial and target region within the scene. We present an algorithm that can verify that all trajectories of the vehicle from the initial region are safe in that they don't collide with an obstacle along the way. We also design an abstraction-refinement based algorithm, which gives us better efficiency in practice. The key idea in our work is the notion of an *image-invariant* region, which is a set of vehicle positions from which the camera captures identical images. Throughout an invariant region the neural network based controller must invariably provide the same control input. This allows us to reason about trajectories at the level of regions, and thereby discretize the continuous state-space of the vehicle. Thus, in contrast with the work on lidars, we exploit the fact that image capture is a *discrete* function of the position of the vehicle.

We also present a directed testing procedure whose objective is to find a large number of spatially distinct unsafe trajectories, that could be used to re-train/re-design the neural network/controller when the verification fails. Both our techniques take advantage of the notion of invariant regions.

We have implemented our techniques in a tool called AirVerif (inspired by the drone simulation tool AirSim [2]), and evaluated its performance on a model of an autonomous drone [9], in a variety of scenes with hundreds of vertices. The experiments show that our basic algorithm is effective in proving safety as well as finding collisions, in environments with around 100 triangles. Our abstraction-based algorithm shows that abstraction techniques can help us scale better for more complex scenes.

II. Camera Model

A digital camera works on a principle similar to a pinhole camera, as shown in Fig. 1. Light rays from an object in the field of view of the camera enter through a small hole called the *aperture* of the camera and strike photosensitive material like film in the case of a classical camera, or a two-dimensional array of "photosites" or *pixels* in the case of a

Habeeb P, Nabarun Deka, and Deepak D'Souza are with the Department of Computer Science And Automation, Indian Institute of Science, Bangalore 560012, Karnataka, India (email: habeebp@iisc.ac.in; nabarundeka@iisc.ac.in; deepakd@iisc.ac.in).

Kamal Lodaya was formerly with the The Institute of Mathematical Sciences, Chennai 600113, Tamil Nadu, India (email: kamal@imsc.res.in).

Pavithra Prabhakar is with the Department of Computer Science, Kansas State University, Manhattan, Kansas 66506, USA (e-mail: pprabhakar@ksu.edu).

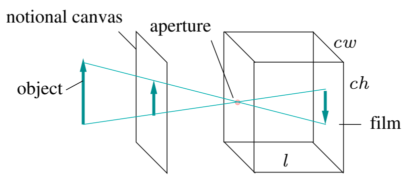


Fig. 1. Pinhole camera

digital camera. The distance from the aperture to the film is called the *focal length* of the camera. The physical dimensions of the film are denoted cw (width) and ch (height) respectively. The image thus captured is inverted and needs to be re-oriented before storing or viewing. In Computer Graphics (CG), this is tackled by considering a notional canvas that lies in front of the aperture at a distance equal to the focal length and parallel to the film plane, onto which the object is projected. We refer the reader to the excellent reference [10], which much of the material in this section draws on, for more details on how images are rendered in CG.

Formally we use a simple model of a digital camera, which we call a *camera model*. For uniformity we use meters as the unit of length thoughout this paper.

Definition 1. *A camera model $\mathcal{C}$ is specified by the following components:*

$$\mathcal{C} = (l, cw, ch, cwp, chp)$$

where

- *l is the* focal length *of the camera (that is the distance from the aperture to the film plane).*
- *cw and ch are the* canvas width *and* height*, respectively.*
- *cwp and chp are the canvas* width in pixels *and canvas* height in pixels*, respectively, in terms of the* number *of pixels. For convenience, we assume these are even numbers.*

We assume that each of entities l, cw, ch, cwp, and chp are positive values.

We will be interested in the images captured by our camera model in a synthetic 3D environment (created using a 3D-design tool like Blender [11]), in which our autonomous vehicle will travel. Following the convention in CG, we represent such an environment as a collection of triangles that represent the triangulated faces of objects in the environment (obstacles which our vehicle must avoid when travelling). Each vertex of a triangle is assumed to have a colour attribute, comprising a triple of numbers (r, g, b) each between 0 and 255 (corresponding to an 8-bit unsigned number), which represent the components of Red, Green and Blue respectively in the colour of the vertex. We call the colour attribute an *RGB value* and denote the set of such triples by RGB.

More formally, let us first fix a 3D coordinate system, popularly called *World Coordinates*, in which our objects and vehicle (along with its camera) will lie. This is a "right-handed" coordinate system which can be visualized by holding the fingers of your right hand (this can be a bit of a twist!) with the middle finger pointing to the right, thumb upwards, and the index finger pointing towards you, denoting the positive directions of the x, y, and z axes respectively.

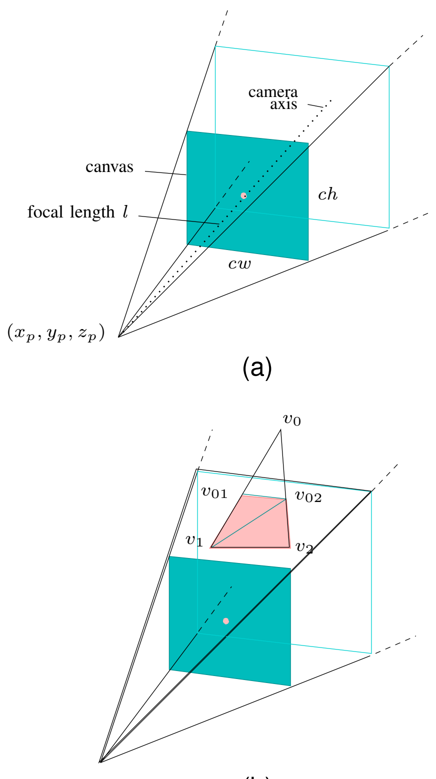


Fig. 2. (a) Camera viewing-frustum (b) Clipping triangle (v_0, v_1, v_2).

An *environment* (or *3D-scene*) can now be represented as a tuple $E = (V, vcol, T)$, where V is a non-empty set of vertices in the world coordinate space, $vcol : V \to RGB$ is a map which assigns an RGB value to each vertex, and $T \subseteq V^3$ is a non-empty set of triangles built from vertices in V.

An *image* in a $cwp \times chp$ grid of pixels, more precisely a (cwp, chp)-image I, is simply a map $I : [(-\frac{cwp}{2}), \frac{cwp}{2}] \times [(-\frac{chp}{2}), \frac{chp}{2}] \to RGB$. Thus, we assign a coordinate $(0, 0)$ to the pixel whose bottom left corner lies at the center of the canvas. Note that we include an "extra" row of pixels to the top and a column of pixels to the right of the canvas, to include the points on the top and right boundaries of the the canvas.

The main purpose of the rest of this section is to describe how our camera model captures an image when placed at a particular position and orientation in a given 3D-scene. At a high level the idea is simple: starting from the triangles in the scene that are farthest away from the camera, (a) project each triangle onto the camera canvas, and (b) colour the pixels lying within this projected triangle, by interpolating the colours from the vertices of the triangle. In what follows we describe these steps in more detail.

Let us fix a 3D-scene $E = (V, vcol, T)$, and a camera position $p = (x_p, y_p, z_p)$. Throughout this paper we make the simplifying assumption that the camera has a *fixed* orientation along the negative z-axis of world space.

Step 1 Transform vertices in scene E to a left-handed *camera coordinate system*, with origin at position of camera p, and z-axis pointing along the camera axis (thus pointing away from you, in the negative z-direction of the world space). This makes subsequent operations like clipping and projection easier to do.

A vertex $v = (x_0, y_0, z_0)$ in world space becomes $WtoC_p(v) = (x_0 - x_p, y_0 - y_p, z_p - z_0)$, and is extended to triangles in the expected way. The last component illustrates that the camera z-axis points in the negative direction of world space. Denote by $E_1 = (V', vcol', T')$ the scene E represented in camera space, where $V' = \{ WtoC_p(v) \mid v \in V\}$, $T' = \{ WtoC_p(t) \mid t \in T\}$, and $vcol(v') = vcol(v)$ where $v' = WtoC(v)$.

Step 2 Next eliminate triangles from E_1 that are outside (i.e. have no intersection with) the field of view of the camera. The field of view of the camera, called its *viewing frustum*, and denoted $F^p_{\mathcal{C}}$, is the unbounded rectangular pyramid with axis same as the z-axis of camera space, and determined by the rectangular $(cw \times ch)$-canvas at $z = l$. See Fig. 2a. In camera space, the viewing frustum $F^p_{\mathcal{C}}$ can be characterized by the set of points (x, y, z) satisfying:

$$-\frac{cw}{2l}z \le x \le \frac{cw}{2l}z, \; -\frac{ch}{2l}z \le y \le \frac{ch}{2l}z, \; 0 \le z. \quad (1)$$

We can check whether a triangle $t = (v_0, v_1, v_2)$ in E_1, with each $v_i = (x_i, y_i, z_i)$, has non-empty intersection with the viewing frustum, by checking if some point v that is a convex combination of v_0, v_1, and v_2 (equivalently, lying in the triangle t) satisfies the constraints in (1). That is, we check if there exist non-negative reals $\alpha_0, \alpha_1, \alpha_2$ with $\alpha_0 + \alpha_1 + \alpha_2 = 1$, such that $v = \alpha_0 v_0 + \alpha_1 v_1 + \alpha_2 v_2$ satisfies the constraints in (1). We define E_2 to be the scene obtained from E_1 by dropping triangles that do not intersect $F^p_{\mathcal{C}}$.

Step 3 Next we address the problem of a triangle being *partially* contained in the viewing frustum. The region of intersection of such triangles and viewing frustum may have to be retriangulated. Consider a triangle $t = (v_0, v_1, v_2)$ in E_2 which is partially contained in $F^p_{\mathcal{C}}$. A representative case is where vertex v_0 is outside the upper plane of the frustum, while v_1 and v_2 are within the frustum, as shown in Fig. 2b. Find the points of intersection v_{01} and v_{02} of the lines (v_0, v_1) and (v_0, v_2) with the upper plane of $F^p_{\mathcal{C}}$, and add the triangles $t_{01} = (v_{01}, v_1, v_{02})$, $t_{02} = (v_1, v_{02}, v_2)$ to E_2 in place of t. The equation of the plane containing the upper plane of $F^p_{\mathcal{C}}$ is:

$$ly = \frac{ch}{2}z. \quad (2)$$

The equation of the line (v_0, v_1) is

$$\frac{x - x_0}{x_1 - x_0} = \frac{y - y_0}{y_1 - y_0} = \frac{z - z_0}{z_1 - z_0}. \quad (3)$$

Eqs. (2) and (3) can be solved to obtain the coordinates of v_{01}. We can similarly compute v_{02}.

We now remove triangle t from E_2 and add triangles t_{01} and t_{02} to it. Let E_3 be the resulting set of triangles obtained by performing this replacement for all triangles in E_2 that are partially contained in $F^p_{\mathcal{C}}$.

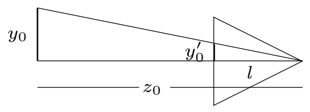


Fig. 3. Perspective division to obtain the projection onto the canvas

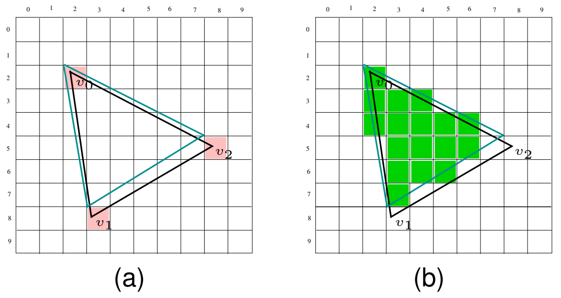


Fig. 4. Colouring pixels in a triangle.

Step 4 Perform "perspective division" to obtain projections of the vertices in triangles in E_3 onto the canvas.

The projection of $v_0 = (x_0, y_0, z_0)$ within the viewing frustum of the camera to the canvas can be seen to be the two-dimensional point $v'_0 = (\frac{lx_0}{z_0}, \frac{ly_0}{z_0})$. Fig. 3 illustrates how this value is derived for the y coordinate. By similarity of the two right-angled triangles, we have $\frac{y_0}{z_0} = \frac{y'_0}{l}$. For convenience, v'_0 is often viewed as a 3-dimensional point with the z value being retained as z_0, which gives us the depth of the vertex in the z-direction. Let us call the resulting scene obtained by projection of each vertex of E_3 in this way, E_4.

Step 5 To colour the pixels on the canvas, first note that a vertex $v_0 = (x_0, y_0, z_0)$ in E_4 falls within the pixel (a, b) given by:

$$a = \lfloor \frac{x_0}{pw} \rfloor, \; b = \lfloor \frac{y_0}{ph} \rfloor, \quad (4)$$

where $pw = \frac{cw}{cwp}$ and $ph = \frac{ch}{chp}$.

Consider a triangle t in E_4. We illustrate how t is coloured with the help of Fig. 4. First find the pixels the vertices of t lie in. In part (4a) of the figure, the triangle t is shown in black, and the pixels each of the vertices lie in (namely $(-3, 2)$, $(-2, -4)$, $(3, -1)$) are shaded pink. Now construct a notional triangle (shown lightly in cyan edges) with vertices as the *top-left corners* of the pink pixels. Colour each pixel whose center lies *within* the cyan triangle, with a colour obtained by interpolating the colours of the vertices of the cyan triangle (which are inherited from the black triangle's vertices). This is shown in Fig. 4b. Finally, for each pixel coloured we also keep track of the depth of the pixel, which is obtained by interpolating the depths of the vertices of the original black triangle.

To handle overlapping triangles, we do the following. A pixel may get colour and depth from several triangles in E_4. The colour assigned to the pixel is the one with the least associated depth.

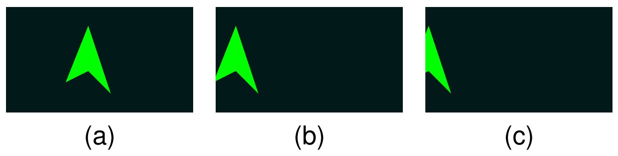


Fig. 5. Camera images of an example scene at positions (a) $(1, 5, 110)$ (b) $(6.5, 5, 110)$ and (c) $(8, 5, 110)$

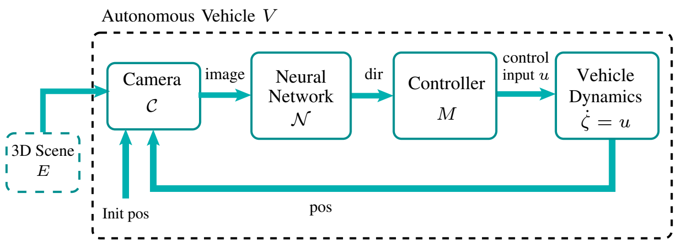


Fig. 6. Camera-Based autonomous vehicle model

This completes the description of the image capture process. To illustrate the end-product of this process, consider an environment with four vertices $v_0 = (0, 8, 100)$, $v_1 = (-2, 3, 100)$, $v_2 = (0, 4, 100)$, and $v_3 = (2, 2, 100)$; and two triangles $t_0 = (v_0, v_1, v_2)$ and $t_1 = (v_0, v_2, v_3)$. All vertices have the same colour $(0, 255, 0)$. Fig. 5 shows the images generated using the above procedure, by a camera model $\mathcal{C}_0 = (0.035\,m, 0.02507488\,m, 0.018669\,m, 1920, 1080)$, at positions $(1, 5, 110)$, $(6.5, 5, 110)$ and $(8, 5, 110)$ respectively.

A camera model $\mathcal{C}$ thus induces a function $img_{\mathcal{C}}$ which takes a 3D-scene E and a position p, and returns the (cwp, chp)-image seen from point p.

III. Autonomous Vehicle Model

We model a camera-based autonomous vehicle in a given 3D-scene, as a simple closed-loop continuous-time sampled control system. The components of the model are shown in Fig. 6. At the beginning of a sample period (represented by τ), the vehicle and the camera mounted on it are in a certain position (with fixed orientation along z-axis), which determines the image of the environment captured by the camera. The image is then fed to the controller consisting of a neural network and a transform matrix wherein the neural network processes the image to determine the action and the transform matrix transforms the action into an input to the vehicle dynamics. The vehicle then updates its state for a sample period based on the input received.

The state of the vehicle at time t is modelled by a real-valued 3-dimensional vector $\zeta(t)$ corresponding to the 3-D position of the vehicle, and the state trajectory changes continuously with time in accordance with a simple vehicle dynamics of the form

$$\dot{\zeta} = u,$$

in which the components of the state vector grow at a rate specified by a 3-dimensional control input vector u. Given a start state $\zeta_0 \in \mathbb{R}^3$ at time $t_0 \in \mathbb{R}_{\geq 0}$, and a control input $u \in \mathbb{R}^3$, the trajectory of the vehicle in the interval $[t_0, \infty)$ is given by a function $\zeta : [t_0, \infty) \to \mathbb{R}^3$, defined as

$$\zeta(t_0 + t) = \zeta_0 + t \cdot u.$$

Note that for clarity we use "$\cdot$" for scalar multiplication above. In the sequel, we will use it for matrix multiplication as well as the dot product of two vectors. The specific operation should be clear from the type of arguments.

Consider a time point t which is a multiple of the sampling period τ with the current state of the system being $\zeta(t)$. The camera component first captures an image *Im* $=$ $img_{\mathcal{C}}(E, \zeta(t))$ of the scene E as seen at the current position $\zeta(t)$ of the vehicle. This image is fed to a neural network $\mathcal{N}$, which for our purpose is simply a map $f_{\mathcal{N}}$, taking as input a vector of dimension k representing the image *Im*, and returning another vector of dimension l representing the suggested action. The l-dimensional action vector is then transformed (multiplied) by the $(3 \times l)$-dimensional controller (transform) matrix M resulting in the control input u that is fed to the vehicle dynamics for a period τ, thereby determining the trajectory of the vehicle in the interval $[t, t+\tau]$. The whole process then repeats for another sample period.

More formally,

Definition 2. *A* (camera-based) autonomous vehicle *V of dimension (k, l) is a tuple of the form*

$$V = (\mathcal{C}, \mathcal{N}, M, \tau),$$

where

- *$\mathcal{C} = (fl, cw, ch, cwp, chp)$ is a camera model, with $k = cwp \cdot chp \cdot 3$,*
- *$\mathcal{N}$ is a neural network with input and output layers of dimension k and l respectively,*
- *$M \in \mathbb{R}^{3 \times l}$ is the controller transformation matrix of dimension $3 \times l$*
- *$\tau \in \mathbb{R}_{>0}$ is the sampling period of the controller.*

A *trajectory* of the vehicle V above, from a given initial state $\zeta_0 \in \mathbb{R}^3$, in a given 3D-scene E, is defined as a function $\zeta : [0, \infty) \to \mathbb{R}^n$, where $\zeta(0) = \zeta_0$, and for each $i \in \mathbb{N}$, ζ in the interval $(i\tau, (i+1)\tau]$ is defined as follows. For any $t \in (0, \tau]$

$$\zeta(i\tau + t) = \zeta(i\tau) + t \cdot u, \text{ where}$$
$$u = M \cdot f_{\mathcal{N}}(img_{\mathcal{C}}(E, \zeta(i\tau))).$$

We are interested in trajectories of the vehicle V in a scene E that start from an *initial* region of states $I \subseteq \mathbb{R}^3$ and eventually reach a *target* region $T \subseteq \mathbb{R}^3$. A trajectory ζ of V in a scene E is *safe* w.r.t. a target region $T \subseteq \mathbb{R}^3$ if there exists $i \in \mathbb{N}$ such that:

- $\zeta(i\tau) \in T$, and
- for $t \in [0, i\tau]$, $\zeta(t)$ does not intersect any triangle in E.

Definition 3. *The* reach-avoid *verification problem is the following: Given an autonomous vehicle $V = (\mathcal{C}, \mathcal{N}, M, \tau)$, a 3D-scene E, an initial region I, and a target region T; are* all *trajectories of V in E starting from I safe?*

We note that the problem *cannot* be solved by enumerating the positions in I and simulating trajectories from each of them, as there are an (uncountably) infinite number of points in I.

We will make some simplifying assumptions going forward. Firstly, we will assume that the target region T is specified as the region beyond an unbounded plane parallel to the xy-plane. In other words, T is given by a constraint of the form $z \leq t$ (recall that the camera is assumed to be oriented in the negative z-direction of world space). Secondly, we assume the vehicle dynamics is *progressive* in that there exists a positive constant c such that the control vector u computed at each step satisfies $u(z) < -c$. Thus in each sample period the vehicle makes a minimum progress of $c \cdot \tau$ in the negative z-direction. Finally, for convenience we have ignored the volume of the vehicle, assuming it to be a point. Our techniques can be extended easily to handle a specified polyhedral volume for the vehicle.

IV. Invariant Regions

In this section we introduce the notion of an invariant region of a position in a scene (w.r.t. a given camera model). This notion will play an important role in our verification and falsification algorithms for the reach-avoid problem.

Definition 4. *Let E be a 3D-scene, $\mathcal{C}$ a camera model, and p a point in world space. Then a set of points X in world space is an* invariant region *around p if $p \in X$ and for each $p' \in X$ we have $img_{\mathcal{C}}(E, p') = img_{\mathcal{C}}(E, p)$ (images seen at all points in X coincide with that seen at p).*

Given a camera model $\mathcal{C} = (l, cw, ch, cwp, chp)$, an environment E, and a point p, how does one come up with an invariant region for p? The notion of a "pixel torch" will help in understanding invariant regions and in coming up with an invariant region around p, which we call $inv_{\mathcal{C}}(E, p)$.

Consider a fixed pixel (a, b) in the canvas, and a viewing point p. Then the (a, b)-*pixel torch* at p is the viewing frustum defined by the point p and the pixel (a, b); equivalently one can also think of it as the set of points in the camera's viewing frustum whose "projected pixel" is (a, b). More precisely, for a point $v_0 = (x_0, y_0, z_0)$ and a camera position $p = (x_p, y_p, z_p)$ (both in world space), such that v_0 is within the viewing frustum of p, we define its *pixel projection* w.r.t. p to be the pixel in the canvas of p, in which the projection of v_0 lies. Following the development in Sec. II the projected pixel is given by the expression:

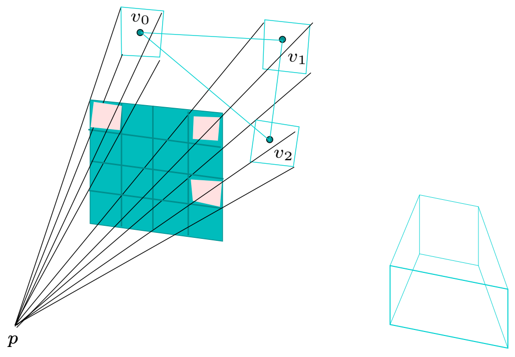
$$proj\text{-}pixel_{\mathcal{C}}(v_0, p) = (\lfloor \frac{l(x_0 - x_p)}{pw(z_p - z_0)} \rfloor, \lfloor \frac{l(y_0 - y_p)}{ph(z_p - z_0)} \rfloor). \quad (5)$$

In camera space, this simplifies to

$$proj\text{-}pixel\text{-}cs_{\mathcal{C}}(v_0) = (\lfloor \frac{l x_0}{pw z_0} \rfloor, \lfloor \frac{l y_0}{ph z_0} \rfloor). \quad (6)$$

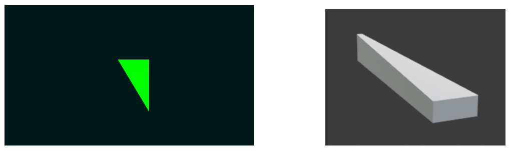
Fig. 7a shows the torches for three different pixels.

To illustrate why pixel torches help us identify invariant regions, consider a simple scene E comprising a single triangle $t = (v_0, v_1, v_2)$, which is fully within the viewing frustum at a position p, as shown in Fig. 7a. Let the pixel projection of v_0, v_1, v_2 be the pixels $(a_0, b_0), (a_1, b_1), (a_2, b_2)$ respectively.



(a) (b)

Fig. 7. (a) Pixel torches, and (b) Frustum shape of invariant region.



(a) (b)

Fig. 8. (a) Image at $p = \langle 0.1, 4.5, 194.5 \rangle$, (b) invariant region around p.

The vertices v_0, v_1, v_2 are each covered by the torch beams of the pixels $(a_0, b_0), (a_1, b_1), (a_2, b_2)$ respectively. Now if we move the camera position (and the pixel torches with it) slightly from p to p', the projected pixels of the vertices of t would continue to be the same as at p. The way we described images being captured in Sec. II, the image captured only depends on the projected pixels of the triangle, and hence the image captured at points p and p' will be identical. Thus, the region in which we can move the camera position around p such that the pixel torches cover the corresponding vertices of the triangle, is an invariant region around p w.r.t. E. This region can be seen to be frustum shaped as shown in Fig. 7b. We further illustrate this in Fig. 8 which shows the image of a triangle on vertices $(-1, 6, 180)$, $(2, 6, 180)$, and $(2, 1, 180)$, by camera model $\mathcal{C}_0$, at position $p = (0.1, 4.5, 194.4)$, along with the invariant region of this image around p, as visualized in Blender [11].

We continue our illustration of the invariant region for a single triangle environment, and derive the invariant region in the plane parallel to the XY-plane, obtained by fixing the z-coordinate to be that of p. Returning to our scene E comprising a single triangle $t = (v_0, v_1, v_2)$, which is fully within the viewing frustum at a position $p = (x_p, y_p, z_p)$, let the pixel projection of v_0, v_1, v_2 be the pixels $(a_0, b_0), (a_1, b_1), (a_2, b_2)$ respectively. Then from Eq. (5):

$$a_0 = \lfloor \frac{l(x_0 - x_p)}{pw(z_p - z_0)} \rfloor, \quad b_0 = \lfloor \frac{l(y_0 - y_p)}{ph(z_p - z_0)} \rfloor$$
$$a_1 = \lfloor \frac{l(x_1 - x_p)}{pw(z_p - z_1)} \rfloor, \quad b_1 = \lfloor \frac{l(y_1 - y_p)}{ph(z_p - z_1)} \rfloor$$
$$a_2 = \lfloor \frac{l(x_2 - x_p)}{pw(z_p - z_2)} \rfloor, \quad b_2 = \lfloor \frac{l(y_2 - y_p)}{ph(z_p - z_2)} \rfloor.$$

Let us call the plane parallel to the XY-plane corresponding

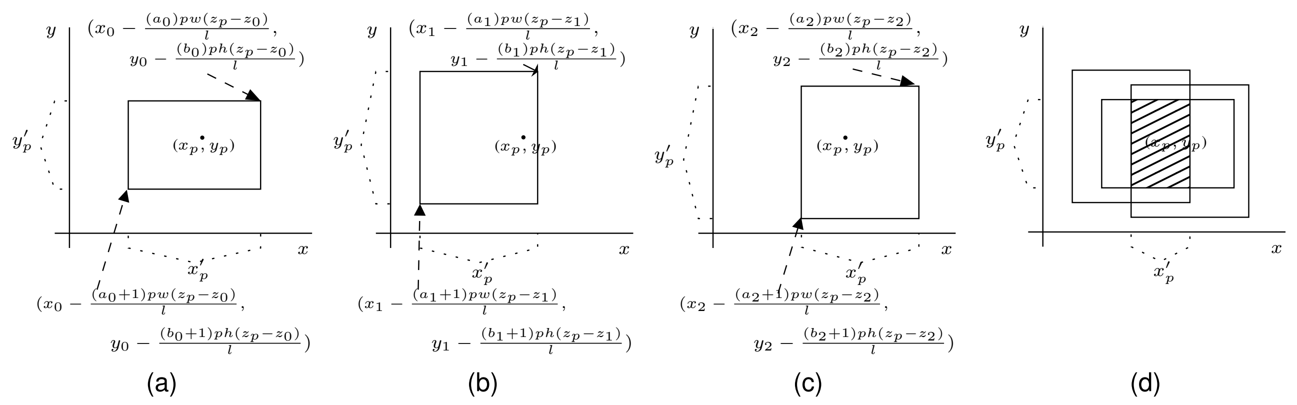

$(x_0 - \frac{(a_0)pw(z_p-z_0)}{l}, y_0 - \frac{(b_0)ph(z_p-z_0)}{l})$
$(x_0 - \frac{(a_0+1)pw(z_p-z_0)}{l}, y_0 - \frac{(b_0+1)ph(z_p-z_0)}{l})$
$(x_1 - \frac{(a_1)pw(z_p-z_1)}{l}, y_1 - \frac{(b_1)ph(z_p-z_1)}{l})$
$(x_1 - \frac{(a_1+1)pw(z_p-z_1)}{l}, y_1 - \frac{(b_1+1)ph(z_p-z_1)}{l})$
$(x_2 - \frac{(a_2)pw(z_p-z_2)}{l}, y_2 - \frac{(b_2)ph(z_p-z_2)}{l})$
$(x_2 - \frac{(a_2+1)pw(z_p-z_2)}{l}, y_2 - \frac{(b_2+1)ph(z_p-z_2)}{l})$
(x_p, y_p)
x'_p
y'_p
x
y
(a)
(b)
(c)
(d)

Fig. 9. Regions in the z_p-plane corresponding to (a) v_0 mapping to (a_0, b_0), (b) v_1 mapping to (a_1, b_1), (c) v_2 mapping to (a_2, b_2); and the invariant region (d), for triangle $t = (v_0, v_1, v_2)$ and camera position (x_p, y_p, z_p).

to $z = z_p$, the z_p-plane. Now the required invariant region in the z_p-plane, with respect to the camera position (x_p, y_p, z_p), is the set of positions (x'_p, y'_p, z_p) satisfying:

$$
\begin{aligned}
a_0 &\le \frac{l(x_0 - x'_p)}{pw(z_p - z_0)} < a_0 + 1, & b_0 &\le \frac{l(y_0 - y'_p)}{ph(z_p - z_0)} < b_0 + 1,\\
a_1 &\le \frac{l(x_1 - x'_p)}{pw(z_p - z_1)} < a_1 + 1, & b_1 &\le \frac{l(y_1 - y'_p)}{ph(z_p - z_1)} < b_1 + 1,\\
a_2 &\le \frac{l(x_2 - x'_p)}{pw(z_p - z_2)} < a_2 + 1, & b_2 &\le \frac{l(y_2 - y'_p)}{ph(z_p - z_2)} < b_2 + 1.
\end{aligned}
$$

Fig. 9(a) shows the region corresponding to v_0 mapping to (a_0, b_0), in that if we move the pixel torch corresponding to the pixel (a_0, b_0) in this region, the beam will still contain the vertex v_0. Similarly, Fig. 9(b) and (c) show the regions corresponding to v_1 mapping to (a_1, b_1) and v_2 mapping to (a_2, b_2) respectively. Finally, the shaded region in Fig. 9(d) shows the intersection of these regions, which is the required invariant region in the z_p-plane. One can now imagine moving the z-plane closer to the triangle and obtaining a similar, smaller sized, rectangular invariant region, thus giving us a frustum-shaped 3D invariant region.

Let us now consider the general case where we have multiple triangles in E, not all of which are within the viewing frustum at p. We need to correctly handle the case of triangles being partially within the viewing frustum, as in Fig. 2b. The image formed now depends on the projected pixel of the added vertex v_{01}, which in turn is the point of intersection of the line (v_0, v_1) and one of the frustum boundary planes.

Definition 5. *Let $\mathcal{C}$ be a camera model, E a 3D-scene, and p a camera position. We define the region $inv_{\mathcal{C}}(E, p)$ to be the set of points p' satisfying the following conditions:*

1) *For each vertex v_0 in E,*
   a) *v_0 is in $F^p_{\mathcal{C}}$ iff v_0 is in $F^{p'}_{\mathcal{C}}$.*
   b) *If $v_0 \in F^p_{\mathcal{C}}$, the projected pixel of v_0 w.r.t. camera positions p and p' coincide.*
2) *For each triangle side $\overline{v_o v_1}$ in E:*
   a) *$\overline{v_0 v_1} \cap Ftop^p_{\mathcal{C}} \ne \emptyset$ iff $\overline{v_0 v_1} \cap Ftop^{p'}_{\mathcal{C}} \ne \emptyset$.*
   b) *If $\overline{v_0 v_1} \cap Ftop^p_{\mathcal{C}} \ne \emptyset$ then the pixel projection of v w.r.t. p, and v' w.r.t. p', where v and v' are the points of intersection of $\overline{v_0 v_1}$ with $Ftop^p_{\mathcal{C}}$ and $Ftop^{p'}_{\mathcal{C}}$ respectively, coincide;*

and similarly for $Fbot_{\mathcal{C}}$, $Fleft_{\mathcal{C}}$, and $Fright_{\mathcal{C}}$.

We now claim that:

Theorem 1. *Let $\mathcal{C}$ be a camera model, E a 3D-scene, and p a point. Then the set of points $inv_{\mathcal{C}}(E, p)$ forms an invariant region around p w.r.t. E.*

Proof. Let p' be a point in $inv_{\mathcal{C}}(E, p)$. By design, projections of the vertices of E and intersection points of the lines in E with boundary planes of the viewing frustum at p and p', fall into the same pixel in the respective canvases at p and p'. From the image capture procedure described in Sec. II it follows that the images captured at p and p' will be identical. □

We now show how we can describe the region $inv_{\mathcal{C}}(E, p)$ as a conjunction of constraints over variables (x'_p, y'_p, z'_p) representing the coordinates of p', in terms of $p = (x_p, y_p, z_p)$, the vertices in E, and camera parameters. We do this by taking the conjunction of constraints corresponding to each of the conditions in Def. 5. This formulation allows us to represent invariant regions logically as polyhedra, and to query and manipulate them using polyhedral libraries and solvers.

Let the coordinates of the points v_0 and v_1 be (x_0, y_0, z_0) and (x_1, y_1, z_1) respectively. For the condition (1a) corresponding to vertex v_0 in the scene E, we have the constraint $in\text{-}vf_{\mathcal{C}}(v_0, p) \equiv in\text{-}vf_{\mathcal{C}}(v_0, p')$ where $in\text{-}vf_{\mathcal{C}}(v_0, p)$ says that point v_0 is in the viewing frustum of p:

$$
\begin{aligned}
x_p - \frac{cw}{2l}(z_p - z_0) \le x_0 \le \frac{cw}{2l}(z_p - z_0) + x_p\\
y_p - \frac{ch}{2l}(z_p - z_0) \le y_0 \le \frac{ch}{2l}(z_p - z_0) + y_p\\
z_0 \le z_p
\end{aligned}
\tag{7}
$$

Here we have transformed v_0 to camera space w.r.t. p, and then used Eq (1). Condition (1b) can be captured by:

$$in\text{-}vf_{\mathcal{C}}(v_0, p) \implies proj\text{-}pixel(v_0, p) = proj\text{-}pixel(v_0, p').$$

Condition (2a) can be phrased as follows. We note that the top boundary (unbounded triangle) of the viewing frustum can be characterized as a convex combination of the points $q_l = (-\frac{wz}{2l}, \frac{hz}{2l}, z)$ and $q_r = (\frac{wz}{2l}, \frac{hz}{2l}, z)$ (for different values of $z \geq 0$). Using this we can phrase the constraint for (2a) as:

$$\begin{aligned}
&\exists\alpha\exists\beta\exists z((0 \leq \alpha \leq 1 \wedge 0 \leq \beta \leq 1) \\
&\wedge\ (\alpha(x_0 - x_p) + (1-\alpha)(x_1 - x_p) = \\
&\beta(-wz/2l) + (1-\beta)wz/2l) \\
&\wedge\ (\alpha(y_0 - y_p) + (1-\alpha)(y_1 - y_p) = \\
&\beta hz/2l + (1-\beta)hz/2l) \\
&\wedge\ (\alpha(z_p - z_0) + (1-\alpha)(z_p - z_1) = z) \quad \mathit{iff} \\
&\exists\alpha'\exists\beta'\exists z((0 \leq \alpha' \leq 1 \wedge 0 \leq \beta' \leq 1) \\
&\wedge\ (\alpha'(x_0 - x'_p) + (1-\alpha')(x_1 - x'_p) = \\
&\beta'(-wz/2l) + (1-\beta')wz/2l) \\
&\wedge\ (\alpha'(y_0 - y'_p) + (1-\alpha')(y_1 - y'_p) = \\
&\beta' hz/2l + (1-\beta')hz/2l) \\
&\wedge\ (\alpha'(z'_p - z_0) + (1-\alpha')(z'_p - z_1) = z).
\end{aligned} \tag{8}$$

Finally condition (2b) can be phrased as:

$$\begin{aligned}
&(\exists\alpha\exists\beta\exists z((0 \leq \alpha \leq 1 \wedge 0 \leq \beta \leq 1) \\
&\wedge\ (\alpha(x_0 - x_p) + (1-\alpha)(x_1 - x_p) = \\
&\beta(-wz/2l) + (1-\beta)wz/2l) \\
&\wedge\ (\alpha(y_0 - y_p) + (1-\alpha)(y_1 - y_p) = \\
&\beta hz/2l + (1-\beta)hz/2l) \\
&\wedge\ (\alpha(z_p - z_0) + (1-\alpha)(z_p - z_1) = z)) \\
&\Longrightarrow\ \mathit{proj\text{-}pixel\text{-}cs}_{\mathcal{C}}(v) = \mathit{proj\text{-}pixel\text{-}cs}_{\mathcal{C}}(v'),
\end{aligned}$$

where v and v' are given by:

$$\begin{aligned}
v = \quad & (\alpha_1(x_0 - x_p) + (1-\alpha_1)(x_1 - x_p), \\
& \alpha_1(y_0 - y_p) + (1-\alpha_1)(y_1 - y_p), \\
& \alpha_1(z_p - z_0) + (1-\alpha_1)(z_p - z_1)), \\
v' = \quad & (\alpha'_1(x_0 - x'_p) + (1-\alpha'_1)(x_1 - x'_p), \\
& \alpha'_1(y_0 - y'_p) + (1-\alpha'_1)(y_1 - y'_p), \\
& \alpha'_1(z'_p - z_0) + (1-\alpha'_1)(z'_p - z_1)),
\end{aligned}$$

α_1, β_1 and α'_1, β'_1 being the values of $\alpha, \beta, \alpha', \beta'$ satisfying Eq. (8).

V. Safety Checking Algorithm

In this section we describe a decision procedure based on invariant regions, that solves the reach-avoid problem for an autonomous vehicle in a given scene. We then propose an abstraction-based version of this procedure that tries to avoid keeping track of an exponential number of regions, by grouping together regions that generate similar control inputs.

A. Safety Checking Procedure

Our basic algorithm for the reach-avoid problem consists of a symbolic state-space exploration through the closed-loop control system representing the autonomous vehicle. The key idea is to represent reachable sets of states as invariant regions. The initial region I is decomposed into a finite set of invariant regions. In general each invariant region R has a set of successor regions that we obtain by propagating R based on the *common* control input u that applies to all states in R. We can compute u by taking any image *Im* corresponding to a state in R, computing the output of the neural network on *Im*, and multiplying it by the transform matrix M. We then propagate R through the vehicle dynamics with input u, to obtain a region R'. We finally decompose R' into invariant regions, to obtain the successor regions of R. The algorithm basically does a depth-first-search of this implicit graph. The regions computed are considered for further exploration only if they do not collide with obstacles in the environment, and only those portions of the regions are retained that have not reached the target. The algorithm terminates when there are no more regions to explore or a collision is detected.

The procedure is summarized in Algorithm 1. Let us fix a vehicle $V = (\mathcal{C}, \mathcal{N}, M, \tau)$, scene E, initial region I and target region T, for rest of this section. Given a region R, a *decomposition of R into image-invariant pairs* is the set of pairs (Im, R_1) where R_1 is a non-empty set of points p in R such that $\mathit{img}_{\mathcal{C}}(E, p) = \mathit{Im}$. This is done as follows. The solver is used to generate a position p in the given region R, and a polyhedral library is used to generate constraints corresponding to the invariant region R_1 at that position. The negation of the invariant region R_1 is added to the solver constraint set, and the solver is asked for a new camera position p from the region R. This process continues until all images in the given region R are generated.

For a region R and a control input u, we compute the post and pre with respect to the dynamics as follows:

$$\mathit{Post}_V(R, u) = \{\zeta + \tau \cdot u \mid \zeta \in R\},$$
$$\mathit{Pre}_V(R, u) = \{\zeta - \tau \cdot u \mid \zeta \in R\}.$$

Algorithm 1 *CheckSafety*

```
Require: Autonomous Vehicle $V$, Environment $E$, Initial Region $I$, Target Region $T$
Ensure: SAFE if all trajectories of $V$ starting from $I$ in $E$ are safe; UNSAFE otherwise.
 1: return CHECKSAFETYDFS($I$)
 2:
 3: function CHECKSAFETYDFS($R$)
 4:     Compute the invariant decomposition $\mathcal{D}$ of $R$
 5:     for each $(Im, R_1)$ in $\mathcal{D}$ do
 6:         $[u, R_2, H]$ := $ComputeTrajectory(Im, R_1, V)$
 7:         if ($H$ intersects with $E$) then
 8:             return UNSAFE
 9:         else if ($R_2 \setminus T$ is non-empty) then
10:             CHECKSAFETYDFS($R_2 \setminus T$)
11:         end if
12:     end for
13:     return SAFE
14: end function
15:
16: function COMPUTETRAJECTORY($Im$, $R$, $V$)
17:     $u := M \cdot f_{\mathcal{N}}(Im)$
18:     $R' := Post_V(R, u)$
19:     $H$ := convex hull of $R$ and $R'$
20:     return $[u, R', H]$
21: end function
```

Theorem 2. *Algorithm CheckSafety returns SAFE iff all trajectories of V starting from I in E are safe, and returns UNSAFE otherwise.*

Proof. We first argue that the procedure terminates. We can view the procedure as starting with an initial worklist comprising the invariant regions constituting I, and then repeatedly removing a region R from this worklist, and adding its successor regions to the worklist. Let us associate a positive real number $d(R)$ with each region R in the worklist, as the maximum distance of a point in R from T. Then each time we remove a region R and add its successor R' to the list, by the progressiveness assumption on the vehicle dynamics, it follows that $d(R') < d(R)$. Now by an argument similar to the one for termination of Smullyan's ball game in [12], it follows that the worklist must become empty (or we return with a collision).

For the correctness part, suppose there is an unsafe trajectory ρ of V in E starting from I. Then it is easy to see (by induction) that there is a sequence of successive invariant regions, with each one containing the respective point from ρ. It follows that when the algorithm explores this sequence of regions it will find a collision and return UNSAFE. Conversely, suppose the procedure finds a sequence of regions that lead to a collision and returns UNSAFE. Since each region visited by the procedure can be seen to contain only points that are reachable by some trajectory from a point in I, it follows that the vehicle must have a colliding trajectory. $\square$

In Section VI we present another version of this algorithm which is directed towards finding collisions.

B. Abstraction-Based Safety Verification

The *CheckSafety* algorithm does a brute-force exploration of the closed-loop system's statespace and the number of invariant regions visited grows exponentially with the number of steps taken. A possible approach to alleviate this is to group together invariant regions that result in the *same* controller outputs (even though the images seen in them may be different), and to propagate them together, by first taking their convex hull. This technique would thus *overapproximate* the set of reachable states. In particular if we find a collision, we need to check if the collision is actually possible or is it spurious because of our overapproximation.

Algorithm 2 shows our abstraction-based algorithm called *AbstractCheckSafety*. The algorithm implicitly explores a successor graph in a similar way to the *CheckSafety* algorithm, except that the nodes of the graph here comprise *groups* of invariant regions, and the one-step successor of a group G is obtained as follows: We first take the convex hull H of the regions in G, propagate it according to the common control output u to obtain a region H'. We now decompose H' into invariant regions and group them together into groups $G'_1, G'_2, \ldots$, and add each of these as successor groups for G.

As before, while propagating the hull R_1 of a group G we check if its sweep intersects with a triangle from the given scene. If it does, we need to check whether it is a real collision or a spurious one. This is done in Lines 11–19 of the algorithm as follows. For each invaraint region (Im', R') in G, we check if its forward propagation intersects with a triangle in the given scene. If it does, we compute the portion R'' of R' whose post image intersects the scene. We now call the CHECKREACH function to check if R'' is reachable from a point in the initial region I, via the sequence of groups that led us to G. If CHECKREACH returns TRUE we have found a true collision and return UNSAFE; if not, we proceed with the search from G.

Algorithm 2 *AbstractCheckSafety*

```
Require: Autonomous Vehicle $V$, Environment $E$, Initial Region $I$,
    Target Region $T$
Ensure: SAFE if all trajectories of $V$ starting from $I$ in $E$ are safe;
    UNSAFE otherwise.
 1: Stack $\mathcal{S} := \{\}$
 2: Compute the invariant decomposition $\mathcal{D}$ of $I$
 3: $\mathcal{G} := \mathit{GroupRegions}(\mathcal{D})$
 4: Push each $G \in \mathcal{G}$ to $\mathcal{S}$
 5: while $\mathcal{S}$ is not empty do
 6:     Pop a group $G$ from $\mathcal{S}$
 7:     $R_1 := \mathit{ConvexHull}(G)$
 8:     Let $(\mathit{Im}, R) \in G$ for some $R$.
 9:     $[u, R_2, H]$ := COMPUTETRAJECTORY($\mathit{Im}, R_1, V$)
10:     if ($H$ intersects with $E$) then
11:         for each $(\mathit{Im}', R') \in G$ do
12:             $[u, R_3, H']$ := COMPUTETRAJECTORY($\mathit{Im}, R', V$)
13:             if $H'$ intersects with $E$ then
14:                 $R'' = \{w | w \in R' \wedge \mathit{Post}_V(w, u)$ intersects $E\}$
15:                 if CHECKREACH($G, R'', I, V$) then
16:                     return UNSAFE
17:                 end if
18:             end if
19:         end for
20:     end if
21:     if $R_2 \backslash T$ not empty then
22:         Compute the invariant decomposition $\mathcal{D}$ of $R_2 \backslash T$
23:         Compute $\mathcal{G} := \mathit{GroupRegions}(\mathcal{D})$
24:         Push each $G \in \mathcal{G}$ to $\mathcal{S}$
25:     end if
26: end while
27: return SAFE
28:
29: function CHECKREACH($G, R, I, V$)
30:     Compute $G_p := \mathit{PreviousGroup}(G)$
31:     Compute $u_p := \mathit{ControlInput}(G_p)$
32:     Compute $R_2 := \mathit{Prev}_V(R, u_p)$
33:     for each $(\mathit{Im}, R_1) \in G_p$ do
34:         if $R_1$ intersects with $R_2$ then
35:             if $R_1 \subseteq I$ then
36:                 return TRUE
37:             else
38:                 if CHECKREACH($G_p$, $R_1 \cap R_2$, $I$, $V$) then
39:                     return TRUE
40:                 end if
41:             end if
42:         end if
43:     end for
44:     return FALSE
45: end function
```

The algorithm uses the following helper functions. The *GroupRegions*$(\mathcal{D})$ function partitions $\mathcal{D}$ into groups such that $(\mathit{Im}_1, R_1) \in \mathcal{D}$ and $(\mathit{Im}_2, R_2) \in \mathcal{D}$ are in same group if and only if $f_{\mathcal{N}}(\mathit{Im}_1) = f_{\mathcal{N}}(\mathit{Im}_2)$. The *ConvexHull*$(G)$ function computes the convex hull of regions in the group G. We keep track of all the regions in a path and the *PreviousGroup*(G) function returns the group whose post region contains G. The *ControlInput*(G_p) function returns

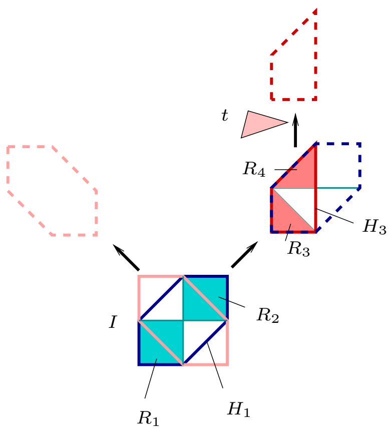


Fig. 10. Illustrating the *AbstractCheckSafety* algorithm

$M \cdot f_{\mathcal{N}}(Im)$ st $(Im, R_1) \in G_p$ and R is in $Post_V(R_1, u_p)$.

Fig. 10 illustrates the idea of the *AbstractCheckSafety* algorithm. Let us say the initial region I is decomposed into four regions including R_1 and R_2 which form one group G_1, while the other two regions form another group G_2. We first form the convex hulls H_1 and H_2 around G_1 and G_2 respectively. Let us say we choose to propagate H_1 to get the dashed region shown to the right of the figure, and suppose its region decomposition yielded R_3 and R_4 as one group G_3. We now take the hull H_3 of G_3 and propagate it to get the dashed region shown on the top of the figure.

Suppose that this propagation resulted in a collision with the triangle t in the scene. We now need to check whether this is a real collision or a spurious one. To do this we check if the propagation of either R_3 or R_4 (the two regions in the group G_3) results in a collision with t. Let us say R_4 resulted in a collision. We then compute the portion of R_4 which collides with t, say R, and ask if R is reachable from some point in I (this is done by the call CHECKREACH(R). This procedure backtracks to the previous group of G_3, namely G_1, and checks if any of the regions R_1 or R_2 in G_1 can reach R. In this case it finds that none of them can reach R, and since this is an initial group, it returns saying that R was not reachable by this sequence of groups. On the other hand, if R_3 was the region in G_3 which collided with t, then CHECKREACH would find that R_1 can reach this part of R_3 and return saying that it is reachable from a point in the initial region I. The algorithm would then return UNSAFE.

Theorem 3. *The AbstractCheckSafety algorithm returns SAFE iff all trajectories of V in E, starting from a position in I, are safe; and returns UNSAFE otherwise.*

Proof. The proof of termination is similar to that of Algorithm *CheckSafety*, where we associate a measure $d(G)$ with a group of invariant regions G as the maximum distance of points in a region of G from the target region T. For correctness, we first argue correctness of the CHECKREACH procedure which checks whether a region R is reachable from a point in the initial region I, along a given path of groups explored by the algorithm. This is not difficult to do by induction on the number of steps in the path. Now let us consider the case when there is an unsafe trajectory ρ of V. Once again we can argue that there must be a sequence of region groups traversed by our algorithm, that cover the corresponding points along ρ. It then follows that we will find a collision in Line 10, and go on to verify it as a true collision using the call to CHECKREACH. Conversely, if our algorithm reports UNSAFE, it must have found a true collision, once again by the correctness of CHECKREACH. □

VI. FALSIFICATION ALGORITHM

In this section we present a "prioritized" version of Algorithm 1, which tries to find as many collisions as possible in a given environment. It is a modification of Algorithm 1, where we try to prioritize the search along paths which are more likely to collide with obstacles in the environment.

For this, we first compute the invariant decomposition $\mathcal{D}$ of the initial region I. For each pair (Im, R_1) in $\mathcal{D}$ we compute a *priority* which is a measure of the likelihood of a point in region R_1 to collide with obstacles in the environment. We now have a choice of which region from $\mathcal{D}$ to propagate first. For this, we choose the region which has the highest priority as computed above and proceed similarly as in the CheckSafety algorithm (Algorithm 1) using a depth-first search (DFS).. Unlike Algorithm 1, whenever we encounter a collision, we record it and continue our search on the remaining state space.

The priority for a pair (Im, R_1) is computed based on the environment and the DNN output for the image *Im*. More precisely, we first assign a potential (or "heat map") to the environment in such a manner that the obstacles of interest are regions of high potential. We do this by first identifying the "obstacles" in the environment which are essentially triangles in the scene that are between the initial and target regions. To compute the potential at a point p, we first identify obstacles that are "close" to p. We compute the distances from p to these obstacles, say $d_1, d_2, \ldots, d_n$. The potential at p is taken to be the sum of $1/d_1$ to $1/d_n$. The gradient of the potential map is obtained by taking the partial derivative of the expression for the potential, along the x, y, and z-directions. This results in a vector field Pot such that for any point p in our environment, $Pot(p)$ points towards the obstacles in the environment. Given a pair (Im, R_1), we first choose a point p in R_1. The priority of (Im, R_1) is a measure of how close the DNN output of *Im*, say u, is to the vector $Pot(p)$, using the angle between the DNN output and $Pot(p)$. More precisely we set the priority of (Im, R_1) to be $\cos\theta$, where θ is the angle between the vectors u and $Pot(p)$. This is computed using the formula $\cos\theta = (u \cdot Pot(p))/(|u||Pot(p)|)$. Our falsification algorithm *FindCollisions* is given in Algorithm 3.

VII. IMPLEMENTATION

We have implemented the three algorithms described in the previous section in a tool called AIRVERIF. Fig. 11 shows a schematic of the workflow within AIRVERIF for the *AbstractCheckSafety* algorithm.

Environments are designed using Blender v2.83 [11], an open-source 3D CG software toolset. Blender's triangulation

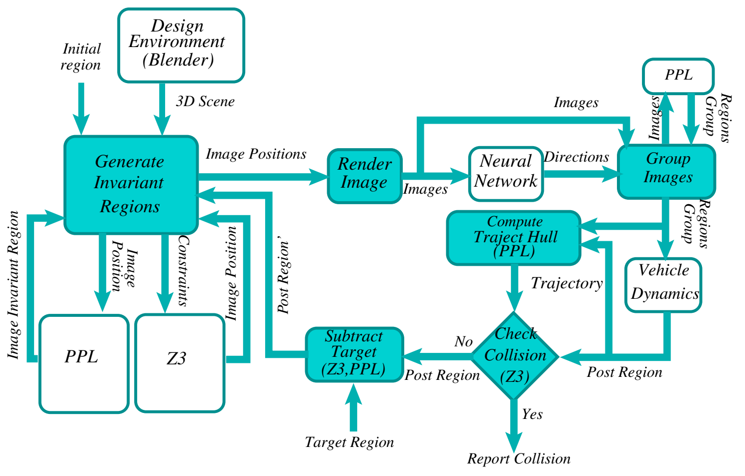


Fig. 11. Schematic of workflow in AIRVERIF for the *AbstractCheckSafety* algorithm.

Algorithm 3 *FindCollisions*

```
Require: Autonomous Vehicle $V$, Environment $E$, Initial Region $I$,
    Target Region $T$
 1: Stack $\mathcal{S}$ := {}
 2: $n$ := 0   // Initialize number of collisions to 0
 3: FINDCOLLISIONS($I$)
 4:
 5: function FINDCOLLISIONS($R$)
 6:     Compute the invariant decomposition $\mathcal{D}$ of $R$
 7:     Compute priority of regions in $\mathcal{D}$
 8:     Push regions of $\mathcal{D}$ to $\mathcal{S}$ in ascending order of priority
 9:     while $\mathcal{S}$ is not empty do
10:         Pop a pair ($Im, R_1$) from $\mathcal{S}$
11:         $[u, R_2, H]$ := COMPUTETRAJECTORY($Im, R_1, V$)
12:         if $H$ intersects with $E$ then
13:             $n := n + 1$
14:         end if
15:         if $((R_2 \setminus T) \neq \emptyset)$ then
16:             FINDCOLLISIONS($R_2 \setminus T$)
17:         end if
18:     end while
19:     return Number of collisions found: $n$
20: end function
```

modifier is used to convert the environments in Blender format to the corresponding set of triangles, and the vertex coordinates are retrieved using Blender's Python API.

We used the Z3 solver [13] to check satisfiability and to find solutions to constraints. AIRVERIF also uses the Parma Polyhedral Library (PPL) [14] to generate and process constraints representing invariant regions. PPL contains a set of numerical abstractions for analyzing and verifying hardware and software systems. These abstractions include convex polyhedra, defined as the intersection of a finite number of (open or closed) halfspaces, each described by a linear inequality (strict or non-strict) with rational coefficients.

The vertex coordinates and the constraints representing the initial region are input to the tool. The *Generate Invariant Regions* module partitions the initial region into invariant regions using the Z3 solver to generate a position in the given region, and the PPL module to generate constraints corresponding to the invariant region at that position.

The *Render Image* module takes a position and renders the image seen from that position. The *Neural Network* module takes these images and outputs the control directions. The images are grouped together based on their neural network output and PPL is used to generate a convex hull of the invariant regions in a group. The *Vehicle Dynamics* module takes a group of regions as input and generates the *post region* according to the region's neural network output and the vehicle dynamics.

The *Collision Check* module's input is the convex hull of the image group region and corresponding post region. This module checks intersection between the convex hull region and obstacle regions. The program terminates if there is a valid collision. If the collision check passes, the post region is passed to the *Target Subtract* module, which computes the difference between the target region and the post region and passes the result to the invariant region generation module. This loop continues until all the paths from the initial region reach the target region or a collision occurs.

VIII. Experimental Evaluation

In this section, we describe our experimental set up, including the autonomous drone case study, and the results of evaluating our verification and falsification algorithms in the context of this case study.

A. Autonomous Drone Case Study

The verification and falsification algorithms implemented in AIRVERIF will be evaluated on an autonomous quadcoptor (drone) navigating along a road with obstacles [9]. The case study consists of a Parrot Bebop 2 quadcopter with an Odroid XU4 board implemented with a "road-following" algorithm that uses a camera as a perception sensor at a sampling period of approximately 33 ms, and a neural network that classifies images and outputs one of the control directives Turn-Left, Go-Straight, or Turn-Right. The commands are then transformed into an input to the quadcopter motors. The neural network

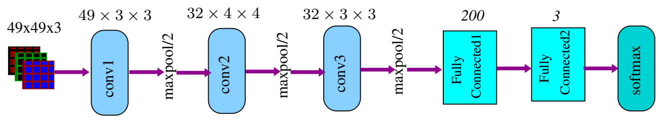


Fig. 12. CNN used in the autonomous quadcopter.

controller was trained on several images of roads, and the vehicle was successfully flown on several roads.

We now provide some more details of each of the vehicle's modules. The camera mounted on the quadcopter has a focal length l of 35 mm, canvas width and height $cw = 0.9872$ in and $ch = 0.735$ in, and width and height in pixels $cwp = 49$ and $chp = 49$. The neural network takes RGB images of size 49×49 from the camera module and outputs control commands consisting of Turn-Left, Go-Straight, and Turn-Right. The neural network has five layers beginning with three convolutional layers, followed by two fully connected layers and ending with a softmax layer as shown in Fig. 12.

We use a simplified version of the quadcoptor's dynamics, by assuming the commands Turn-Left, Go-Straight and Turn-Right are transformed to the control input u using the following controller transformation matrix.

$$M = \begin{bmatrix} -\sin\theta & 0 & \sin\theta \\ 0 & 0 & 0 \\ \cos\theta & 1 & \cos\theta \end{bmatrix}$$

where $\theta = 30°$.

B. Experiments and Results

We analysed this autonomous vehicle model using our safety checking and falsification algorithms, on a variety of 3D-scenes depicting a road and tree-like obstacles. Fig. 13 and Fig. 14 show the different environments, which contain scenes ranging from 24 vertices, 36 edges, and 16 triangles (env1) to 786 vertices, 786 edges, and 286 triangles (env8). Each of these scenes contain obstacles ranging from the periphery of the road to the center of the road.

The following two subsections explain our evaluation results for the verification and falsification algorithms separately. The results are summarized in Table II and Table III. Column "#Edges" in the tables indicates the number of edges in the environment. Column "I" indicates the initial regions used. These are typically 1cm-cubes, with the coordinates of the bottom left corner of the front face of the cube given in Table I. The target region is all points whose z-coordinate value is less than 9.5 m from the initial region (the vehicle moves in the negative z-direction). All experiments were conducted on a machine with an Intel(R) Core(TM) i7-8700 3.20 GHz CPU and 64GB RAM.

1) Verification: The *CheckSafety* algorithm runs with a timeout of 30 min, and the *AbstractCheckSafety* algorithm runs without any timeout. The *CheckSafety* algorithm proved safety for environments with a small number of edges, and even though it timed out for environments with large number of triangles, it could still prove several paths from the initial

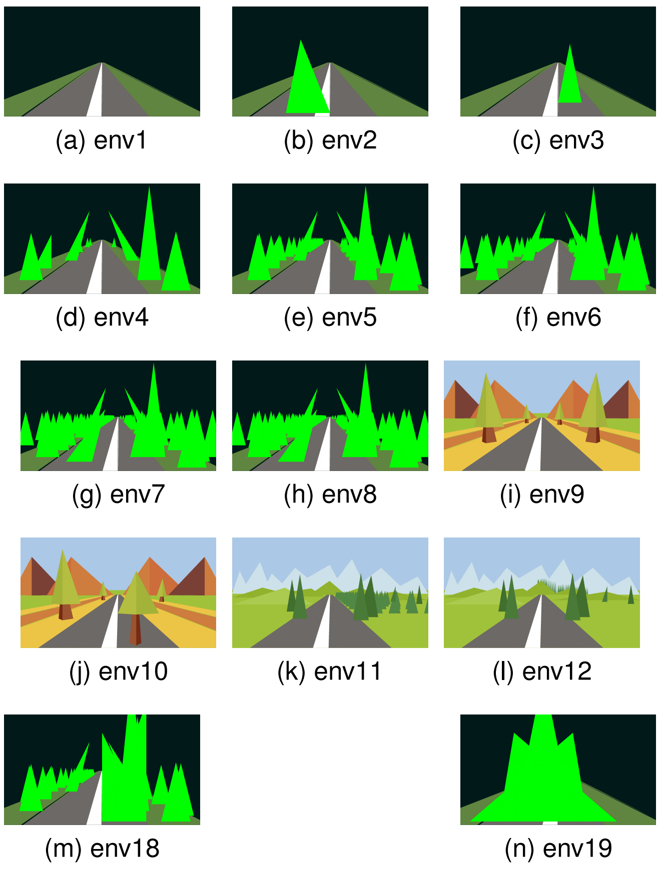


Fig. 13. Environments used. Images captured with the camera $\mathcal{C}_0$ at position (1, 4.5, 200).

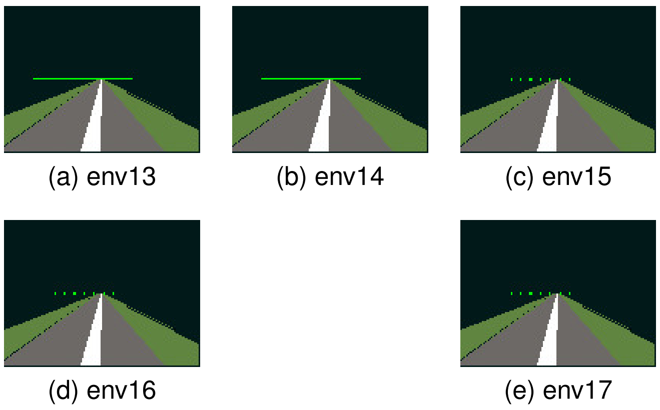


Fig. 14. Environments used. Images captured with the camera $\mathcal{C}_1 = (0.035\,m, 0.02507488\,m, 0.018669\,m, 128, 96)$ at position (1, 4.5, 200).

TABLE I
INITIAL REGIONS

Initial Region	Bottom Left Corner	Volume
I_1	0.1, 4.45, 194.5	1.00 cm^3
I_2	-0.95, 4.45, 194.5	1.00 cm^3
I_3	2.5, 4.45, 194.5	1.00 cm^3
I_4	0.1, 4.45, 194.5	0.25 cm^3

TABLE II
SAFETY CHECKING

			CheckSafety		*AbstractCheckSafety*	
Env	**#Edges**	**I**	**Time (#Paths)**	**Safe**	**Time**	**Safe**
env1	36	I_1	0m 54s (All)	Yes	2m 58s	Yes
env2	39	I_2	0m 13s (1)	No	0m 50s	No
env3	39	I_1	3m 42s (All)	Yes	3m 15s	Yes
env4	66	I_1	30 m 0s (86)	-	4m 0s	Yes
env5	186	I_1	30m 0s(47)	-	18m 16s	Yes
env6	336	I_1	30m 0s(53)	-	17m 18s	Yes
env7	636	I_1	30m 0s(15)	-	27m 38s	Yes
env8	786	I_4	30m 0s(9)	-	16m 0s	Yes
		I_3	1m 9s (1)	No	45m 20s^{+}	No
env9	100	I_1	30m 0s(24)	-	3m 45s	Yes
env10	100	I_1	30m 0s(30)	-	5m 9s	Yes
env11	310	I_1	30m 0s(42)	-	16m 0s	Yes
env12	313	I_1	30m 0s(38)	-	40m 9s	Yes
env13	42	I_1	0m 6s (1)	No	0m 32s	No
env14	51	I_1	0m 7s (1)	No	0m 50s	No
env15	57	I_1	0m 4s (1)	No	0m 57s	No
env16	78	I_1	5m 34s (16)	No	1m 3s	No
env17	171	I_1	3m 10s (12)	No	2m 50s	No

region to the target region to be safe. The number of paths it found before concluding safety or finding a violation is given in the fourth column of Table II along with the time it takes to make a decision. Environments on which it times out, that is, it is not able to conclude safety or find violation, have a "-" entry. For some environments, the *CheckSafety* algorithm in fact finds collisions faster than the *AbstractCheckSafety* algorithm, the reason being that the former generates a single image from the initial region and propagates it toward the target region, whereas the abstraction based algorithm generates *all* the images in each region before moving to the next step.

The *AbstractCheckSafety* algorithm successfully completed (either proved safety or found a collision) on all the considered environments. As can be seen, the time to prove safety gradually increases with the number of edges in the environment. The Z3 solver to which AIRVERIF makes calls to generate images iteratively for a region is given a timeout of 3 min per call. For example, for env8 and initial region I_3, the solver timed out while trying to generate the complete set of images corresponding to the region. In such cases, we proceed with the partial set of generated images, find corresponding invariant regions and propagate them to the next step. Note that even in this case, a collision found is indeed valid. If the forward propagation with the partial set of images does not find a collision, we partition the initial region into smaller regions and restart the algorithm from each of the smaller regions, with the hope to conclude safety on some subset of the original initial region. For example, in env8, the solver timed out on initial region I_1, so we partitioned the initial region into smaller regions and reran the program on these sub-regions; the time to prove safety of a smaller $0.5\times0.5\times1$ cm cuboid region I_4, is reported in the table.

The results show that abstraction-based techniques are likely to be effective in helping the basic algorithm scale to more complex environments.

2) Falsification: We ran our *FindCollisions* algorithm on several example environments with obstacles of varying sizes, and compared it against a simulation algorithm that uses prioritization but does not compute invariant regions. The goal of these experiments was to see whether invariant regions provide any advantage in finding collisions over a simple random simulation based algorithm.

The simulation algorithm works as follows. We divide the initial region into several smaller regions, chose a point uniformly at random inside each of these smaller regions, compute the priority of each of these points similar to the *FindCollisions* algorithm (Sec. VI), arrange them in descending order of priority, and then propagate each of them forward until a collision is found or the target region is reached.

The results are reported in Table III. Columns "1st Coll" report the time taken to find the first collision in the simulation and *FindCollisions* algorithms respectively. A '-' entry indicates that the algorithm found no collisions in 30 min. Columns "Tot Coll" report the total number of collisions discovered by the algorithms in 30 min.

We observed that for these environments with very few collisions, the falsification algorithm performed significantly better than the simulation algorithm, even though both were prioritized to detect collisions. Due to the low chance of a path colliding with environment obstacles, the simulation algorithm was unable to detect any collisions with the environment. On the other hand, the falsification algorithm was able to detect these collisions because it propagates entire invariant regions. For environments with larger number of collisions (env18 and env19), the simulation algorithm detected much more collisions in 30 min as compared to the falsification algorithm. We also observed that the falsification algorithm always found a collision in the *first* path it explored, indicating that the combination of prioritization with invariant regions is effective.

We also compared the falsification algorithm with our *CheckSafety* algorithm. We observe that in several cases the *CheckSafety* algorithm finds the first collision quicker. This is because the *CheckSafety* algorithm does not generate all images in a region before propagating. Hence, if it finds a collision in the first path it explores, it performs better than the *FindCollisions* algorithm. However, as we can see from env16, when the *CheckSafety* algorithm has to explore several paths before detecting the collision, the *FindCollisions* algorithm performs significantly better.

IX. DISCUSSION

In this section we discuss the challenges faced by the current algorithm and how we could relax some of the assumptions made.

TABLE III
FindCollisions VS SIMULATION

			Simulation		*FindCollisions*	
Env	**#Edges**	**I**	**1st Coll**	**Tot Coll**	**1st Coll**	**Tot Coll**
env2	39	I_2	-	-	291 s	1
env8	786	I_3	-	-	709 s	3
env13	42	I_1	-	-	27 s	4
env14	51	I_1	-	-	141 s	6
env15	57	I_1	-	-	88 s	12
env16	78	I_1	-	-	55 s	7
env17	171	I_1	-	-	273 s	3
env18	186	I_3	1 s	23	357 s	8
env19	45	I_1	1 s	15	97 s	3

a) Scalability: Scalability is clearly a major challenge for our verification algorithm. The factors contributing to time taken include the size of the initial region, and the complexity of the given 3D-scene in terms of the number of vertices and edges present, which impacts the solver's ability to generate new invariant regions. Abstraction techniques are a promising way forward, and our *AbstractCheckSafety* algorithm demonstrates the potential benefits. However our abstraction algorithm still requires us to generate all the invariant regions, which impacts scalability. Instead, it would be interesting to investigate the use of abstract image representations (like "interval" images) in conjunction with techniques like [15], [16] for abstractly interpreting neural networks on these images.

b) Availability of Triangulations: We are using a synthetic model of the 3D environment where all objects are represented using triangle meshes. We can create triangulated models of real environments using urban reconstruction methods (see [17] for a comprehensive overview of these techniques). In this way we can build triangulated models of test tracks and do our analysis on the generated synthetic environment.

c) Camera Orientation: For simplicity we have assumed that the autonomous vehicle has a fixed orientation along the negative z-axis of world space. Since the camera is mounted on the vehicle, it will have the same orientation as the vehicle. In the camera model section, we use the function to convert a vertex from world-coordinates to camera coordinates. To handle a dynamic orientation of the vehicle, our system state will now need to be a six-dimensional vector, with the first three components representing the vehicle's position in world coordinates, and the last three components representing the vehicle's orientation, i.e., angles relative to the world coordinate axes. Our world-to-camera conversion function $WtoC$ can be easily extended to consider the orientation by using the standard 4×4 geometric transformation matrix (see [18]).

We will also need to update the invariant region computation to take into account the vehicle's orientation. Initially we have a given orientation, and each time we move the vehicle left or right, the orientation will change according to the vehicle dynamics. If we assume that the change in orientation in a single time step does not depend on the position of the vehicle, then each invariant region will have a common orientation. We can now compute invariant regions similar to what we do in the invariant region section, with a slight modification in the world-to-camera conversion which considers the camera's orientation.

d) Vehicle Dynamics: Even though we consider a simple dynamics in our experiments, our algorithm extends to systems with general linear or non-linear dynamics. We can use standard approximation techniques to get a safe over-approximation of the position of the vehicle at each sampling step. We can then consider this approximate region as the one step successor and partition it into invariant regions. The rest of the algorithm will remain the same.

X. Related Work

We discuss related work under verification, testing and simulation based approaches.

a) Verification: Among verification approaches, the most closely related are that of O'Kelly et al [7] and Sun et al [8], both of which consider the problem of verifying safety in a given environment. O'Kelly et al [7] consider dynamic objects like other vehicles and pedestrians, and make use of testing and verification tools S-TaLiRo [19] and dReach [20] to carry out their analysis. However they don't handle camera sensors, and their scenes are not generic 3D-scenes.

Sun et al [8] give an abstraction based technique to verify lidar-based vehicles in a given 2D-scene. To handle the non-linearity of the lidar readings w.r.t. the vehicle positions they use a partitioning of vehicle space into "image-adapted" regions in which the lidar image can be described by affine constraints. They abstract the overall system (including the neural network) and check feasibility of transitions between abstract regions using an SMC solver. Our work differs in several ways. The lidar image is continuous function of the position of vehicle, while the camera image is a discrete function of the position of the vehicle; and we exploit the latter property. Our invariant regions guarantee a common output from the neural network, and this lets us avoid reasoning about the neural network which can be expensive due to its non-linearity and size. Further (with some assumptions on the vehicle dynamics), our approach gives a decision procedure while theirs is an abstraction-refinement approach with no guarantee of termination. Finally, we consider 3D-scenes while they consider only 2D-scenes.

Ivanov et al [21] present a verification case study in which an autonomous racing car with a NN controller navigates a structured environment using LiDAR measurements only. They encode the neural network with sigmoid/tanh activation functions as a hybrid system and compose it along with the plant's hybrid system, and verify properties using the $Flow^*$ tool. This method can only handle networks with a small number of inputs and nodes. The paper also presents results that compare real-world systems and the simulation model, showing that real-world behavior can also be captured in the simulation. In contrast our work exploits the properties of camera image generation to solve the reach-avoid verification problem.

Corsi et al. [22] introduce a novel approach based on interval algebra to verify the behavioral properties of a neural network controller defined over a large input space. The idea is to divide the input domain intervals iteratively (by exploiting matrix operations) and check each sub-interval for property violation. The tool's output is a percentage of the neural network input area that violated the property. This work specifically looks for property violation of the neural network component, which can help train a safe neural network.

b) Directed Testing: In directed testing for finding bugs, Dreossi et al [23] consider the setting of an vehicle with ML components (like neural network controllers based on camera sensor inputs), and suggest the idea of compositionally analyzing the vehicle abstract model and machine learning components, to come up with a sequence of inputs that violate a given Signal Temporal Logic (STL) specification. In a similar setting, Zhang et al [24] give an optimization-based technique for falsification of STL properties of a hybrid system model. Unlike our setting, these works don't have a given 3D-scene, and are free to use images/inputs from a pre-specified set U, to drive the system to violate the given specification. Tuncali et al [6] consider the falsification of STL properties in a camera-based perception setting with a 3D environment similar to ours. They use the S-TaLiRo tool to find a trajectory that optimizes a cost-function. In contrast to our work, they do not benefit from using invariant regions and cannot verify safety.

c) Testing and Simulation: The DeepTest tool [4] focuses on testing of DNN models used by camera-based self-driving cars. Their aim is to generate test inputs that provide good neuron-coverage, using changing of contrast, brightness, blurring, and rotation of images. In a similar setting, DeepRoad [5] focuses on generating realistic transformations of driving scenes. In contrast, in our setting we have a fixed environment, and no leeway to transform images seen.

The VerifAI software toolkit [25] focuses on simulation-based verification of AI-based cyber-physical systems, where one is given a closed-loop system including its environment in form of a dynamic 3D-scene specified in Scenic [26]. The toolkit includes various kinds of analysis on the model, like temporal logic based falsification, fuzz testing, and parameter synthesis. However, unlike our technique, it cannot prove safety of the system, and its falsication technique does not exploit invariant regions.

Finally, AirSim [2] and LGSVL [3] simulate the flight of autonomous drones and autononmous vehicles in a given synthetic 3D-scene. While these are useful tools for testing and visualization, they do not provide any verification guarantees.

XI. Conclusion and Future Work

In this paper, we have given effective procedures for reasoning about the safe trajectory of camera-based autonomous vehicles in given 3D-scenes, based on the notion of image-invariant regions. Our initial experiments are encouraging and point towards the benefits of effective abstraction-based techniques.

There are several interesting directions going ahead. Apart from investigating other abstraction techniques mentioned earlier, we would like to consider more realistic dynamics, again with the help of abstraction techniques. Finally, it would also be interesting to consider *dynamic* environments in which obstacles are moving, and extend our techniques to this setting.

References

[1] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," in *23rd Intl. Conf. Intelligent Transportation Systems (ITSC 2020), Rhodes, Greece*. IEEE, 2020, pp. 1–8.

[2] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles," in *Res. 11th Intl. Conf. on Field and Service Robotics (FSR 2017), Zurich, Switzerland*. Springer, 2017, pp. 621–635.

[3] LG Electronics America R&D Lab, "SVL Simulator," https://www.svlsimulator.com/, last accessed: 2022-06-07.

[4] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars," in *Proc. 40th Intl. Conf. on Software Engineering (ICSE 2018), Gothenburg, Sweden*. ACM, 2018, pp. 303–314.

[5] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems," in *Proc. 33rd ACM/IEEE Intl. Conf. on Automated Software Engineering (ASE 2018), Montpellier, France*. ACM, 2018, pp. 132–142.

[6] C. E. Tuncali, G. E. Fainekos, H. Ito, and J. Kapinski, "Sim-ATAV: Simulation-Based Adversarial Testing Framework for Autonomous Vehicles," in *Proc. 21st Intl. Conf. on Hybrid Systems: Computation and Control (HSCC 2018), Porto, Portugal*, 2018, pp. 283–284.

[7] M. O'Kelly, H. Abbas, and R. Mangharam, "Computer-Aided Design for Safe Autonomous Vehicles," U. Pennsylvania, Tech. Rep., May 2017. [Online]. Available: https://repository.upenn.edu/mlab_papers/99

[8] X. Sun, H. Khedr, and Y. Shoukry, "Formal verification of neural network controlled autonomous systems," in *Proc. 22nd ACM Intl. Conf. on Hybrid Systems: Computation and Control (HSCC 2019), Montreal, Canada*, 2019, pp. 147–156.

[9] P. Prakash, C. Murti, J. S. Nath, and C. Bhattacharyya, "Optimizing DNN Architectures for High Speed Autonomous Navigation in GPS Denied Environments on Edge Devices," in *Proc. 16th Pac. Rim Intl. Conf. on Artificial Intelligence (PRICAI 2019), Fiji*, 2019, pp. 468–481.

[10] J.-C. Prunier, "Scratchapixel: An Overview of the Rasterization Algorithm," https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation, last accessed: 2020-09-28.

[11] Blender 3D Creation Suite, https://www.blender.org/, last accessed: 2022-03-08.

[12] M. Fitting, *First-Order Logic and Automated Theorem Proving, Second Edition*, ser. Graduate Texts in Computer Science. Springer, 1996.

[13] L. M. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proc. 14th Intl. Conf. Tools and Alg. Constr. Anal. Systems (TACAS)*. Springer, 2008, pp. 337–340.

[14] R. Bagnara, P. M. Hill, and E. Zaffanella, "The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 3–21, 1 June 2008.

[15] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. T. Vechev, "AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation," in *Proc. IEEE Symp. Security and Privacy (SP 2018), San Francisco, USA*, 2018, pp. 3–18.

[16] G. Singh, T. Gehr, M. Püschel, and M. T. Vechev, "An abstract domain for certifying neural networks," *PACMPL*, vol. 3, no. POPL, pp. 41:1–41:30, 2019.

[17] P. Musialski, P. Wonka, D. G. Aliaga, M. Wimmer, L. Van Gool, and W. Purgathofer, "A survey of urban reconstruction," in *Computer graphics forum*, vol. 32, no. 6. Wiley Online Library, 2013, pp. 146–177.

[18] J.-C. Prunier, "Mathematics of computing the 2D coordinates of a 3D point," https://www.scratchapixel.com/lessons/3d-basic-rendering/computing-pixel-coordinates-of-3d-point/mathematics-computing-2d-coordinates-of-3d-points.

[19] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A Tool for Temporal Logic Falsification for Hybrid Systems," in *Proc. 17th Intl. Conf. Tools and Alg. Constr. Anal.s of Systems (TACAS 2011), Saarbrücken*. Springer, 2011, pp. 254–257.

[20] S. Kong, S. Gao, W. Chen, and E. M. Clarke, "dReach: δ-Reachability Analysis for Hybrid Systems," in *Proc. 21st Intl. Conf. Tools and Alg. Constr. Anal. Systems (TACAS 2015), London*. Springer, 2015, pp. 200–205.

[21] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Case study: verifying the safety of an autonomous racing car with a

neural network controller," in *Proc. 23rd International Conference on Hybrid Systems: Computation and Control (HSCC 2020)*, 2020, pp. 1–7.

[22] D. Corsi, E. Marchesini, and A. Farinelli, "Formal verification of neural networks for safety-critical tasks in deep reinforcement learning," in *Uncertainty in Artificial Intelligence*. PMLR, 2021, pp. 333–343.

[23] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional Falsification of Cyber-Physical Systems with Machine Learning Components," in *Proc. 9th NASA Formal Methods (NFM 2017), USA*, 2017, pp. 357–372.

[24] Z. Zhang, G. Ernst, S. Sedwards, P. Arcaini, and I. Hasuo, "Two-Layered Falsification of Hybrid Systems Guided by Monte Carlo Tree Search," *Trans. Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 37, no. 11, pp. 2894–2905, 2018.

[25] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems," in *Proc. 31st Intl. Conf. on Computer Aided Verification (CAV 2019), New York City, USA*, 2019, pp. 432–442.

[26] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: a language for scenario specification and scene generation," in *Proc. 40th Conf. Programming Language Design and Implementation (PLDI 2019), Phoenix, USA, June 22-26, 2019*. ACM, 2019, pp. 63–78.