SceneChecker: Boosting Scenario Verification using Symmetry Abstractions

Hussein Sibai, Yangge Li, and Sayan Mitra

University of Illinois at Urbana-Champaign
Coordinated Science Laboratory
{sibai2,li213,mitras}@illinois.edu

Abstract. We present SceneChecker, a tool for verifying scenarios involving vehicles executing complex plans in large cluttered workspaces. SceneChecker converts the scenario verification problem to a standard hybrid system verification problem, and solves it effectively by exploiting structural properties in the plan and the vehicle dynamics. SceneChecker uses symmetry abstractions, a novel refinement algorithm, and importantly, is built to boost the performance of any existing reachability analysis tool as a plug-in subroutine. We evaluated SceneChecker on several scenarios involving ground and aerial vehicles with nonlinear dynamics and neural network controllers, employing different kinds of symmetries, using different reachability subroutines, and following plans with hundreds of way-points in complex workspaces. Compared to two leading tools, DryVR and Flow*, SceneChecker shows 20× speedup in verification time, even while using those very tools as reachability subroutines.

Keywords: Hybrid systems · Safety verification · Symmetry.

1 Introduction

Remarkable progress has been made in safety verification of hybrid and cyber-physical systems in the last decade [1–9]. The methods and tools developed have been applied to check safety of aerospace, medical, and autonomous vehicle control systems [3, 4, 10–12]. The next barrier in making these techniques usable for more complex applications is to deal with what is colloquially called the *scenario verification problem*. A key part of the scenario verification problem is to check that a vehicle or an agent can execute a plan through a complex environment. A planning algorithm (e.g., probabilistic roadmaps [13] and RRT [14]) generates a set of possible paths avoiding obstacles, but only considering the geometry of the scenario, not the dynamics. The verification task has to ensure that the plan can indeed be safely executed by the vehicle with all the dynamic constraints and the state estimation uncertainties. Indeed, one can view a scenario as a hybrid automaton with the modes defined by the segments of the planner, but this leads to massive models. Encoding such automata in existing tools presents some practical hurdles. More importantly, analyzing such models is challenging as the over-approximation errors and the analysis times grow rapidly with the number of transitions. At the same time, such large hybrid verification problems also have lots of repetitions and symmetries, which suggest new opportunities.

We present SceneChecker, a tool that implements a symmetry abstraction-refinement algorithm for efficient scenario verification. Symmetry abstractions significantly reduce the number of modes and edges of an automaton H by grouping all modes that share symmetric continuous dynamics [15]. SceneChecker implements a novel refinement algorithm for symmetry abstractions and is able to use any existing reachability analysis tool as a subroutine. Our current implementation comes with plug-ins for using Flow* [3] and DryVR [6]. SceneChecker's verification algorithm is sound, i.e., if it returns *safe*, then the reachset of H indeed does not intersect the unsafe set. The algorithm is lossless in the sense that if one can prove safety without using abstraction, then SceneChecker can also prove safety via abstraction-refinement, and typically a lot faster.

SceneChecker offers an easy interface to specify plans, agent dynamics, obstacles, initial uncertainty, and symmetry maps. SceneChecker checks if a fixed point has been reached after each call to the reachability subroutine, avoiding repeating computations. First, SceneChecker represents the input scenario as a hybrid automaton H where modes are defined by the plan's segments. It uses the symmetry maps provided by the user to construct an abstract automaton H_v. H_v represents another scenario with fewer segments, each representing a group of symmetric segments in H. A side effect of the abstraction is that upon reaching waypoints in H_v, the agent's state resets non-deterministically to a set of possible states. For example, in the case of rotation and translation invariance, the abstract scenario would have a single segment for any group of segments with a unique length in the original scenario. SceneChecker refines H_v by splitting one of its modes to two modes. That corresponds to representing a group of symmetric segments with one more segment in the abstract scenario, capturing more accurately the original scenario[1].

We evaluated SceneChecker on several scenarios where car and quadrotor agents with nonlinear dynamics follow plans to reach several destinations in 2D and 3D workspaces with hundreds of waypoints and polytopic obstacles. We considered different symmetries (translation and rotation invariance) and controllers (Proportional-Derivative (PD) and Neural Networks (NN)). We compared the verification time of SceneChecker with DryVR and Flow* as reachability subroutines against Flow* and DryVR as standalone tools. SceneChecker is faster than both tools in all scenarios considered, achieving up to 20× speedup in verification time (Table 1). In certain scenarios where Flow* timed out (executing for more than 120 minutes), SceneChecker is able to complete verification in as fast as 12 minutes using Flow* as a subroutine. SceneChecker when using abstraction-refinement achieved 13× speedup in verification time over not using abstraction-refinement in scenarios with the NN-controlled quadrotor (Section 7).

Related work The idea of using symmetries to accelerate verification has been exploited in a number of contexts such as probabilistic models [16, 17], automata [18, 19], distributed architectures [20], and hardware [21–23]. Some symmetry utilization algorithms are implemented in Murϕ [24] and Uppaal [25].

In our context of cyber-physical systems, Bak et al. [26] suggested using symmetry maps, called *reachability reduction transformations*, to transform reachsets to symmetric reachsets for continuous dynamical systems modeling non-interacting vehicles. Maidens et al. [27] proposed a symmetry-based dimensionality reduction method for backward

[1] A figure showing the architecture of SceneChecker can be found in Appendix A.

reachable set computations for discrete dynamical systems. Majumdar et al. [28] proposed a safe motion planning algorithm that computes a family of reachsets offline and composes them online using symmetry. Bujorianu et al. [29] presented a symmetry-based theory to reduce stochastic hybrid systems for faster reachability analysis and discussed the challenges of designing symmetry reduction techniques across mode transitions.

In a more closely related research, Sibai et al. [30] presented a modified version of DryVR that utilizes symmetry to cache reachsets aiming to accelerate simulation-based safety verification of continuous dynamical systems. The related tool CacheReach implements a hybrid system verification algorithm that uses symmetry to accelerate reachability analysis [31]. CacheReach caches and shares computed reachsets between different modes of non-interacting agents using symmetry. SceneChecker is based on the theory of symmetry abstractions of hybrid automata presented in [15]. They suggested computing the reachset of the abstract automaton instead of the concrete one then transform it to the concrete reachset using symmetry maps to accelerate verification. SceneChecker is built based on this line of work with significant algorithmic and engineering improvements. In addition to the abstraction of [15], SceneChecker 1) maps the unsafe set to an abstract unsafe set and verifies the abstract automaton instead of the concrete one and 2) decreases the over-approximation error of the abstraction through refinement. SceneChecker does not cache reachsets and thus saves cache-access and reachset-transformation times and does not incur over-approximation errors due to caching that CacheReach suffers from [31]. At the implementation level, SceneChecker accepts plans that are general directed graphs and polytopic unsafe sets while CacheReach accepts only single-path plans and hyperrectangle unsafe sets. We show more than 30× speedup in verification time while having more accurate verification results when comparing SceneChecker against CacheReach (Table 1 in Section 7).

2 Specifying Scenarios in SceneChecker

A scenario verification problem is specified by a set of obstacles, a plan, and an agent that is supposed to execute the plan without running into the obstacles. For ground and air vehicles, for example, the agent moves in a subset of the 2D or the 3D Euclidean space called the *workspace*. A *plan* is a directed graph $G = \langle V, S \rangle$ with vertices V in the workspace called *waypoints* and edges S called *segments*[2].A general graph allows for nondeterministic and contingency planning.

An *agent* is a control system that can follow waypoints. Let the state space of the agent be X and $\Theta \subseteq X$ be the uncertain initial set. Let s_{init} be the initial segment in G that the agent has to follow. From any state $x \in X$, the agent follows a segment $s \in S$ by moving along a *trajectory*. A trajectory is a function $\xi : X \times S \times \mathbb{R}^{\geq 0} \to X$ that meets certain dynamical constraints of the vehicle. Dynamics are either specified by ordinary differential equations (ODE) or by a black-box simulator. For ODE models, ξ is a solution of an equation of the form: $\frac{d\xi}{dt}(x,s,t) = f(\xi(x,s,t),s)$, for any $t \in \mathbb{R}^{\geq 0}$ and $\xi(x,s,0) = x$, where $f : X \times S \to X$ is Lipschitz continuous in the first argument. Note that the trajectories only depend on the segment the agent is following (and not on the

[2] We introduce this redundant nomenclature because later we will reserve the term edges to talk about mode transitions in hybrid automata. We use waypoints instead of vertices as a more natural term for points that vehicles have to follow.

full plan G). We denote by $\xi.fstate$, $\xi.lstate$, and $\xi.dom$ the initial and last states and the time domain of the time bounded trajectory ξ, respectively.

We can view the obstacles near each segment as sets of unsafe states, $O : S \rightarrow 2^X$. The map $tbound : S \rightarrow \mathbb{R}^{\geq 0}$ determines the maximum time the agent should spend in following any segment. For any pair of consecutive segments (s, s'), i.e. sharing a common waypoint in G, $guard((s, s'))$ defines the set of states (a hyperrectangle around a waypoint) at which the agent is allowed to transition from following s to following s'.

Scenario JSON file is the first of the two user inputs. It specifies the scenario: Θ as a hyperrectangle; S as a list of lists each representing two waypoints; *guard* as a list of hyperrectangles; *tbound* as a list of floats; and O as a list of polytopes.

Output of SceneChecker is the scenario verification result (*safe* or *unknown*) and a number of useful performance metrics, such as the number of mode-splits, number of reachability calls, reachsets computation time, and total time. SceneChecker can also visualize the various computed reachsets.

3 Transforming Scenarios to Hybrid Automata

The input scenario is first represented as a hybrid automaton by a Hybrid constructor. This constructor is a Python function that parses the Scenario file and constructs the data structures to store the scenario's hybrid automaton components. In what follows, we describe the constructed automaton informally. In our current implementation, sets are represented either as hyper-rectangles or as polytopes using the Tulip Polytope Library[3].

Scenario as a hybrid automaton A hybrid automaton has a set of *modes* (or discrete states) and a set of continuous states. The evolution of the continuous states in each mode is specified by a set of trajectories and the transition across the modes are specified by *guard* and *reset* maps. The agent following a plan in a workspace can be naturally modeled as a hybrid automaton H, where s_{init} and Θ are its initial mode and set of states.

Each segment $s \in S$ of the plan G defines a *mode* of H. The set of edges $E \subseteq S \times S$ of H is defined as pairs of consecutive segments in G. For an edge $e \in E$, $guard(e)$ is the same as that of G. The *reset* map of H is the identity map. We will see in Section 5 that abstract automata will have nontrivial reset maps.

Verification problem An *execution* of length k is a sequence $\sigma := (\xi_0, s_0), \ldots, (\xi_k, s_k)$. It models the behavior of the agent following a particular path in the plan G. An execution σ must satisfy: 1) $\xi_0.fstate \in \Theta$ and $s_0 = s_{init}$, for each $i \in \{0, \ldots, k-1\}$, 2) $(s_i, s_{i+1}) \in E$, 3) $\xi_i.lstate \in guard((s_i, s_{i+1}))$, and 4) $\xi_i.lstate = \xi_{i+1}.fstate$, and 5) for each $i \in \{0, \ldots, k\}$, $\xi_i.dom \leq tbound(s_i)$. The set of *reachable states* is $Reach_H := \{\sigma.lstate \mid \sigma$ is an execution$\}$. The restriction of $Reach_H$ to states with mode $s \in S$ (i.e., agent following segment s) is denoted by $Reach_H(s)$. Thus, the hybrid system verification problem requires us to check whether $\forall s \in S$, $Reach_H(s) \cap O(s) = \emptyset$.

[3] https://pypi.org/project/polytope/

4 Specifying Symmetry Maps in SceneChecker

The hybrid automaton representing a scenario, as constructed by the Hybrid constructor, is transformed into an abstract automaton. SceneChecker uses symmetry abstractions [15]. The abstraction is constructed by the abstract function (line 1 of Algorithm 1) which uses a collection of pairs of maps $\Phi = \{(\gamma_s : X \to X, \rho_s : S \to S)\}_{s \in S}$ that is provided by the user. We describe below how these maps are specified by the user in the Dynamics file. These maps should satisfy:

$$\forall\ t \geq 0, x_0 \in X, s \in S, \gamma_s(\xi(x_0, s, t)) = \xi(\gamma_s(x_0), \rho_s(s), t). \tag{1}$$

where $\forall s \in S$, the map γ_s is differentiable and invertible. Such maps are called *symmetries* for the agent's dynamics. They transform the agent's trajectories to other symmetric ones of its trajectories starting from symmetric initial states and following symmetric modes (or segments in our scenario verification setting). It is worth noting that (1) does not depend on whether the trajectories ξ are defined by ODEs or black-box simulators. Currently, condition (1) is not checked by SceneChecker for the maps specified by the user. However, in the following discussion, we present some ways for the user to check (1) on their own. For ODE models, a sufficient condition for (1) to be satisfied is if: $\forall\ x \in X, s \in S$, $\frac{\partial \gamma_s}{\partial x} f(x, s) = f(\gamma_s(x), \rho_s(s))$, where f is the right-hand-side of the ODE [32]. For black-box models, (1) can be checked using sampling methods. In realistic settings, dynamics might not be exactly symmetric due to unmodeled uncertainties. In the future, we plan to account for such uncertainties as part of the reachability analysis.

In scenario verification, a given workspace would have a coordinate system according to which the plan (waypoints) and the agent's state (position, velocity, heading angle, etc.) are represented. In a 2D workspace, for any segment $s \in S$, an example symmetry ρ_s would transform the two waypoints of s to a new coordinate system where the second waypoint is the origin and s is aligned with the negative side of the horizontal axis. The corresponding γ_s would transform the agent's state to this new coordinate system (e.g. by rotating its position and velocity vectors and shifting the heading angle). For such a pair (γ_s, ρ_s) to satisfy (1), the agent's dynamics have to be invariant to such a coordinate transformation and (1) merely formalizes this requirement. Such an invariance property is expected from vehicles' dynamics–rotating or translating the lane should not change how an autonomous car behaves.

Dynamics file is the second input provided by the user in addition to the Scenario file and it contains the following:

- polyVir(X', s): returns $\gamma_s(X')$ for any polytope $X' \subset X$ and segment $s \in S$.
- modeVir(s): returns $\rho_s(s)$ for any given segment $s \in S$.
- virPoly(X', s): returns $\gamma_s^{-1}(X')$, implementing the inverse of polyVir.
- computeReachset$(initset, s, T)$: returns a list of hyperrectangles over-approximating the agent's reachset starting from *initset* following segment s for T time units, for any set of states $initset \subset X$, segment $s \in S$, and $T \geq 0$.

5 Symmetry Abstraction of the Scenario's Automaton

In this section, we describe how the abstract function in Algorithm 1 uses the functions in the Dynamics file to construct an abstraction of the scenario's hybrid automaton provided by the Hybrid constructor. Given the symmetry maps of Φ, the symmetry abstraction of H is another hybrid automaton H_v that aggregates many symmetric modes (segments) of H into a single mode of H_v.

Modes and Transitions Any segment $s \in S$ of H is mapped to the segment $\rho_s(s)$ in H_v using modeVir. The set of modes S_v of H_v is the set of segments $\{\rho_s(s)\}_{s\in S}$. For any s_v, $tbound_v(s_v) = \max_{s\in S, s_v=\rho_s(s)} tbound(s)$. In the example of Section 4, the segments in H_v are aligned with the horizontal axis and ending at the origin. The number of segments in H_v would be the number of segments in G with unique lengths. The agent would always be moving towards the origin of the workspace in the abstract scenario. Any edge $e = (s, s') \in E$ of H is mapped to the edge $e_v = (\rho_s(s), \rho_{s'}(s'))$ in H_v. The $guard(e)$ is mapped to $\gamma_s(guard(e))$ using polyVir which becomes part of $guard_v(e_v)$ in H_v. For any $x \in X$, $reset(x, e)$, which is equal to x, is mapped to $\gamma_{s'}(\gamma_s^{-1}(x))$ and becomes part of $reset_v(x, e_v)$ in H_v. In our example in Section 4, the $\gamma_s^{-1}(x)$ would represent x in the absolute coordinate system assuming it was represented in the coordinate system defined by segment s. The $\gamma_{s'}(\gamma_s^{-1}(x))$ would represent $\gamma_s^{-1}(x)$ in the new coordinate system defined by segment s'. The $guard_v(e_v)$ would be the union of rotated hyperrectangles centered at the origin that result from translating and rotating the guards of the edges represented by e_v. The initial set Θ of H is mapped to $\Theta_v = \gamma_{s_{init}}(\Theta)$, the initial set of H_v. A formal definition of symmetry abstractions can be found in Appendix D (or [15]).

The unsafe map O is mapped to O_v, where $\forall s_v \in S_v, O_v(s_v) = \cup_{s\in S, \rho_s(s)=s_v} \gamma_s(O(s))$. That means the obstacles near any segment $s \in S$ in the environment will be mapped to be near its representative segment $\rho_s(s)$ in H_v.

A forward simulation relation between H and H_v can show that if H_v is safe with respect to O_v, then H is safe with respect to O. More formally, if $\forall s_v \in S_v, Reach_{H_v}(s_v) \cap O_v(s_v) = \emptyset$, then $\forall s \in S, Reach_H(s) \cap O(s) = \emptyset$ [15].

6 SceneChecker Algorithm Overview

A sketch of the core abstraction-refinement algorithm is shown in Algorithm 1. It constructs a symmetry abstraction H_v of the concrete automaton H resulting from the Hybrid constructor. SceneChecker attempts to verify the safety of H_v using traditional reachability analysis. SceneChecker uses a *cache* to store per-mode initial sets from which reachsets have been computed and thus avoids repeating computations.

The core algorithm verify (Algorithm 2) is called iteratively. If verify returns $(safe, \perp)$ or $(unknown, \perp)$, SceneChecker returns the same result. If verify instead results in $(refine, s_v^*)$, splitMode (check Appendix E for the formal definition) is called to refine H_v by splitting s_v^* into two modes s_v^1 and s_v^2. Each of the two modes would represent part of the set of the segments of S that were originally mapped to s_v in rv. Then the edges, guards, resets, and the unsafe sets related to s_v are split according to their definitions.

The function verify executes a *depth first search* (DFS) over the mode graph of H_v. For any mode s_v being visited, computeReachset computes R_v, an over-approximation

Algorithm 1 SceneChecker($\Phi = \{(\gamma_s, \rho_s)\}_{s \in S}, H, O$)

1: $H_v, O_v \leftarrow \mathsf{abstract}(H, O, \Phi)$
2: $\forall s \in S, rv[s] \leftarrow \rho_s(s)$
3: **while** *True* **do**
4: $\quad cache \leftarrow \{s_v \mapsto \emptyset \mid s_v \in S_v\}$
5: $\quad result, s_v^* \leftarrow \mathsf{verify}(rv[s_{init}], \Theta_v, cache, rv, H_v, O_v)$
6: $\quad$ **if** *result* $=$ *safe* or *unknown* **then return:** *result*
7: $\quad$ **else** $rv, H_v, O_v \leftarrow \mathsf{splitMode}(s_v^*, rv, H_v, O_v, H, O)$

of the agent's reachset starting from *initset* following segment s_v for time $tbound_v(s_v)$. If $R_v \cap O_v(s_v) = \emptyset$, verify recursively calls s_v's children continuing the DFS in line 6. Before calling each child, its initial set is computed and the part for which a reachset has already been computed and stored in *cache* is subtracted. If all calls return *safe*, then *initset* is added to the other initial sets in $cache[s_v]$ (line 12) and verify returns *safe*. Most importantly, if verify returns $(refine, s_v^*)$ for any of s_v's children, it directly returns $(refine, s_v^*)$ for s_v as well (line 7). If any child returns *unknown* or R_v intersects $O_v(s_v)$, verify will need to split s_v. In that case, it checks if $rv^{-1}[s_v]$ is not a singleton set and thus amenable to splitting (line 10). If s_v can be split, verify returns $(refine, s_v)$. Otherwise, verify returns $(unknown, \perp)$ implicitly asking one of s_v's ancestors to be split instead.

Correctness SceneChecker ensures that all the refined automata H_v's are abstractions of the original hybrid automaton H (a proof is given in Appendix E). For any mode with a reachset intersecting the unsafe set, SceneChecker keeps refining that mode and its ancestors until safety can be proven or H_v becomes H.

Theorem 1 (Soundness). *If* SceneChecker *returns safe, then H is safe.*

Algorithm 2 verify($s_v, initset, cache, rv, H_v, O_v$)

1: $R_v \leftarrow \mathsf{computeReachset}(initset, s_v)$
2: **if** $R_v \cap O_v(s_v) = \emptyset$ **then**
3: $\quad$ **for** $s'_v \in children(s_v)$ **do**
4: $\quad\quad initset' \leftarrow reset_v(guard_v((s_v, s'_v)) \cap R_v) \backslash cache[s'_v]$
5: $\quad\quad$ **if** $initset' \neq \emptyset$ **then**
6: $\quad\quad\quad result, s_v^* \leftarrow \mathsf{verify}(s'_v, initset', cache, rv, H_v, O_v)$
7: $\quad\quad\quad$ **if** *result* $=$ *refine* **then return:** *refine*, s_v^*
8: $\quad\quad\quad$ **else if** *result* $=$ *unknown* **then break**
9: **if** $R_v \cap O_v(s_v) \neq \emptyset$ or *result* is *unknown* **then**
10: $\quad$ **if** $|rv^{-1}[s_v]| > 1$ **then return:** *refine*, s_v
11: $\quad$ **else return:** *unknown*, $\perp$
12: $cache[s_v] \leftarrow cache[s_v] \cup initset$
13: **return:** *safe*, $\perp$

If verify is provided with the concrete automaton H and unsafe set O, it will be the traditional safety verification algorithm having no over-approximation error due to abstraction. If such a call to verify returns *safe*, then SceneChecker is guaranteed to return *safe*. That means that the refinement ensures that the over-approximation error of the reachset caused by the abstraction is reduced to not alter the verification result.

Counter-examples SceneChecker currently does not find counter-examples to show that the scenario is *unsafe*. There are several sources of over-approximation errors, namely, computeReachset and guard intersections. Even after all the over-approximation errors from symmetry abstractions are eliminated, as refinement does, it still cannot infer unsafe executions or counter-examples because of the other errors. We plan to address this in the future by combining the current algorithm with systematic simulations.

7 Experimental Evaluation

Agents and controllers In our experiments, we consider two types of nonlinear agent models: a standard 3-dimensional car (C) with bicycle dynamics and 2 inputs, and a 6-dimensional quadrotor (Q) with 3 inputs. For each of these agents, we developed a PD controller and a NN controller for tracking segments. The NN controller for the quadrotor is from Verisig's paper [9] (Appendix F for more details) but modified to be rotation symmetric (Appendix G for more details). Similarly, the NN controller for the car is also rotation symmetric. Both NN controllers are translation symmetric as they take as input the difference between the agent's state and the segment being followed. The PD controllers are translation and rotation symmetric by design.

Symmetries We experimented with two different collections of symmetry maps Φs: 1) translation symmetry (T), where for any segment s in G, γ_s maps the states so that the coordinate system is translated by a vector that makes its origin at the end waypoint of s, and 2) rotation and translation symmetry (TR), where instead of just translating the origin, Φ rotates the xy-plane so that s is aligned with the x-axis, which we described in Section 4. For each agent and one of its controllers, we manually verified that condition (1) is satisfied for each of the two Φs using the sufficient condition for ODEs in Section 4.

Scenarios We created four scenarios with 2D workspaces (S1-4) and one scenario with a 3D workspace (S5) with corresponding plans. We generated the plans using an RRT planner [33] after specifying a number of goal sets that should be reached. We modified S4 to have more obstacles but still have the same plan and named the new version S4.b and the original one S4.a. When the quadrotor was considered, the waypoints of the 2D scenarios (S1-4) were converted to 3D representation by setting the altitude for each waypoint to 0. Scenario S5 is the same as S2 but S5's waypoints have varying altitudes. The scenarios have different complexities ranging from few segments and obstacles to hundreds of them. All scenarios are safe when traversed by any of the two agents.

We verify these scenarios using two instances of SceneChecker, one with DryVR and the other with Flow*, implementing computeReachset. SceneChecker is able to verify all scenarios with PD controllers. The results are shown in Table 1[4].

Observation 1: SceneChecker *offers fast scenario verification and boosts existing reachability tools* Looking at the two total time (Tt) columns for the two instances of SceneChecker with the corresponding columns for Flow* and DryVR, it becomes clear that symmetry abstractions can boost the verification performance of reachability engines. For example, in C-S4.a, SceneChecker with DryVR was around 20× faster than DryVR. In C-S3, SceneChecker with Flow* was around 16× faster than Flow*. In scenario Q-S5, SceneChecker timed out at least in part because a computeReachset call to Flow* timed out. Even when many refinements are required and thus causing several repetitions of the verification process in Algorithm 1, SceneChecker is still faster than DryVR and Flow* (C-S4.b). All three tools resulted in *safe* for all scenarios when completed executions.

Table 1: Comparison between SceneChecker, DryVR (DR), Flow* (F*), and CacheReach (CacheR). Both SceneChecker and CacheReach use reachability tools as subroutines. The subroutines used are specified after the '+' sign. Φ is TR. The table shows the number of mode-splits performed (Nrefs), the total number of calls to computeReachset (Rc), the total time spent in reachset computations (Rt), and the total computation time in minutes (Tt). In scenarios where a tool ran over 120 minutes, we marked the Tt column as 'Timed out'(TO) and when it errored, we marked it as 'Not Available'(NA).

		SceneChecker+DR				CacheR+DR		DR	SceneChecker+F*				CacheR+F*		F*
Sc.	$\|S\|$	Nrefs	Rc	Rt	Tt	Rc	Tt	Tt	NRefs	Rc	Rt	Tt	Rc	Tt	Tt
C-S1	6	1	4	0.14	0.15	46	1.75	1.34	1	4	0.51	0.52	52	8.20	2.11
C-S2	140	0	1	0.04	0.66	453.86	37.42	11.25	0	1	0.18	0.79	192	30.95	17.52
C-S3	458	0	1	0.04	4.26	398.26	33.32	75.35	0	1	0.11	4.34	176	28.64	73.06
C-S4.a	520	2	7	0.26	4.52	276.64	23.23	95.02	2	7	0.80	4.96	160	25.98	61.53
C-S4.b	520	10	39	1.48	8.90	277.10	23.17	95.05	10	39	2.83	31.73	160	26.07	60.67
Q-S1	6	1	4	0.05	0.06	NA	NA	0.25	1	4	13.85	14.13	NA	TO	30.17
Q-S2	140	0	1	0.04	0.88	NA	NA	4.93	0	1	3.38	12.62	NA	TO	TO
Q-S3	458	0	1	0.06	5.9	NA	NA	45.03	0	1	4.98	62.66	NA	TO	TO
Q-S4.a	520	0	1	0.06	3.32	NA	NA	55.99	0	1	4.8	34.89	NA	TO	TO
Q-S5	280	0	36	0.85	3.06	NA	NA	4.91	NA	NA	NA	TO	NA	TO	TO

Observation 2: SceneChecker *is faster and more accurate than* CacheReach Since CacheReach only handles single-path plans, we only verify the longest path in the plans of the scenarios in its experiments. CacheReach's instance with Flow* resulted in unsafe reachsets in C-S1 and C-S4.b scenarios likely because of the caching over-approximation error. In all scenarios where CacheReach completed verification besides C-S4.b, it has more Rc and longer Tt (more than 30× in C-S2) while verifying simpler plans than SceneChecker using the same reachability subroutine. In all Q scenarios, CacheReach's instance with Flow* timed out, while its instance with DryVR terminated with an error.

Observation 3: More symmetric dynamics result in faster verification time SceneChecker usually runs slower in 3D scenarios compared to 2D ones (Q-S2 vs. Q-S5) in part because

[4] Figures presenting the reachsets of the concrete and abstract automata for different scenarios can be found in Appendix J. The machine specifications can be found in Appendix H.

there is no rotational symmetry in the z-dimension to exploit. That leads to larger abstract automata. Therefore, many more calls to computeReachset are required.

We only used SceneChecker's instance with DryVR for agents with NN-controllers[5]. We tried different Φs. The results are shown in Table 2. When not using abstraction-refinement, SceneChecker took 11, 132, and 73 minutes for the QNN-S2, QNN-S3, and QNN-S4 scenarios, while DryVR took 5, 46, and 55 minutes for the same scenarios, respectively. Comparing these results with those in Table 2 shows that the speedup in verification time of SceneChecker is caused by the abstraction-refinement algorithm, achieving more than $13\times$ in certain scenarios (QNN-S4 using $\Phi = $ T). SceneChecker's instance with DryVR was more than $10\times$ faster than DryVR in the same scenario.

Table 2: Comparison between Φs. In addition to the statisitics of Table 1, this table reports the number of modes and edges in the initial and final (after refinement) abstractions ($|S_v|^i$, $|E_v|^i$; $|S_v|^f$, and $|E_v|^f$, respectively)

Sc.	NRef	Φ	$\|S\|$	$\|S_v\|^i$	$\|E_v\|^i$	$\|S_v\|^f$	$\|E_v\|^f$	Rc	Rt	Tt
CNN-S2	7	TR	140	1	1	8	20	35	2.83	5.64
CNN-S4	10	TR	520	1	1	11	32	68	5.57	36.66
QNN-S2	3	TR	140	1	1	4	9	9	0.61	4.01
QNN-S3	7	TR	458	1	1	8	23	21	2.11	13.98
QNN-S4	6	TR	520	1	1	7	20	15	1.51	8.11
QNN-S2	0	T	140	7	19	7	19	9	0.62	1.85
QNN-S3	4	T	458	7	30	11	58	29	2.85	16.72
QNN-S4	0	T	520	7	30	7	30	13	1.3	5.32

Observation 4: Choice of Φ is a trade-off between over-approximation error and number of refinements The choice of Φ affects the number of refinements performed and the total running times (e.g. QNN-S2, QNN-S3, and QNN-S4). Using TR leads to a more succinct H_v but larger over-approximation error causing more mode splits. On the other hand, using T leads to a larger H_v but less over-approximation error and thus fewer refinements. This trade-off can be seen in Table 2. For example, QNN-S4 with $\Phi =$T resulted in zero mode splits leading to $|S_v|^i = |S_v|^f = 7$, while $\Phi =$ TR resulted in 6 mode splits, starting with $|S_v|^i = 1$ modes and ending with $|S_v|^f = 7$, and longer verification time because of refinements. On the other hand, in QNN-S3, $\Phi =$ TR resulted in Nref= 7, $|S_v|^f = 8$, and Tt= 13.98 min while $\Phi =$T resulted in Nref= 4, $|S_v|^f = 11$, and Tt= 16.72 min.

Observation 5: Complicated dynamics require more verification time Different vehicle dynamics affect the number of refinements performed and consequently the verification time (e.g. QNN-S2, QNN-S4, CNN-S2, and CNN-S4). The car appears to be less stable than the quadrotor leading to longer verification time for the same scenarios. This can also be seen by comparing the results of Tables 1 and 2. The PD controllers lead to more stable dynamics than the NN controllers requiring less total computation time for both agents. More stable dynamics lead to tighter reachsets and fewer refinements.

[5] Check Appendix I for a discussion about our attempts for using other verification tools for NN-controlled systems as reachability subroutines.

8 Limitations and Discussions

SceneChecker allows the choice of modes to be changed from segments to waypoints or sequences of segments as well. The waypoint-defined modes eliminate the need for segments of G to have few unique lengths, but only allow Φ =T. SceneChecker splits only one mode per refinement and then repeats the computation from scratch. It has to refine many times in unsafe scenarios until reaching the result *unknown*. We plan to investigate other strategies for eliminating spurious counter-examples and returning valid ones in unsafe cases. In the future, it will be important to address other sources of uncertainty in scene verification such as moving obstacles, interactive agents, and other types of symmetries such as permutation and time scaling. Finally, it will be useful to connect a translator to generate scene files from common road simulation frameworks such as CARLA [34], commonroad [35], and Scenic [36].

References

1. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: Scalable verification of hybrid systems. In: CAV. pp. 379–395 (2011)
2. Bak, S., Duggirala, P.S.: Hylaa: A tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control. pp. 173–178. ACM (2017)
3. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: CAV. pp. 258–263. Springer (2013)
4. Duggirala, P.S., Fan, C., Mitra, S., Viswanathan, M.: Meeting a powertrain verification challenge. In: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. pp. 536–543 (2015)
5. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2e2: A verification tool for stateflow models. In: Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035. p. 68–82. Springer-Verlag, Berlin, Heidelberg (2015), https://doi.org/10.1007/978-3-662-46681-0_5
6. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: Dryvr: Data-driven verification and compositional reasoning for automotive systems. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 441–461. Springer International Publishing, Cham (2017)
7. Dutta, S., Chen, X., Jha, S., Sankaranarayanan, S., Tiwari, A.: Sherlock - a tool for verification of neural network feedback systems: Demo abstract. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. p. 262–263. HSCC '19, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3302504.3313351
8. Tran, H.D., Yang, X., Manzanas Lopez, D., Musau, P., Nguyen, L.V., Xiang, W., Bak, S., Johnson, T.T.: Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 3–17. Springer International Publishing, Cham (2020)
9. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. pp. 169–178 (2019)
10. Althoff, M.: An introduction to cora 2015. In: Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)

11. Roohi, N., Wang, Y., West, M., Dullerud, G., Viswanathan, M.: Statistical verification of toyota powertrain control verification benchmark. In: Proceedings of the 20th international conference on Hybrid systems: computation and control. ACM (2017)
12. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. pp. 531–538 (2016)
13. Kavraki, L.E., Svestka, P., Latombe, J.., Overmars, M.H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation 12(4), 566–580 (1996)
14. Lavalle, S.M.: Rapidly-exploring random trees: A new tool for path planning. Tech. rep. (1998)
15. Sibai, H., Mitra, S.: Symmetry abstractions for hybrid systems and their applications (2020)
16. Kwiatkowska, M.Z., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings. pp. 234–248 (2006)
17. Antuña, L.R., Araiza-Illan, D., Campos, S., Eder, K.: Symmetry reduction enables model checking of more complex emergent behaviours of swarm navigation algorithms. In: Towards Autonomous Robotic Systems - 16th Annual Conference, TAROS 2015, Liverpool, UK, September 8-10, 2015, Proceedings. pp. 26–37 (2015)
18. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings. pp. 463–478 (1993)
19. Clarke, E.M., Jha, S.: Symmetry and induction in model checking. In: Computer Science Today: Recent Trends and Developments, pp. 455–470 (1995)
20. Jacobs, S., Bloem, R.: Parameterized synthesis. Logical Methods in Computer Science [electronic only] 10 (01 2014)
21. Mann, M., Barrett, C.: Partial order reduction for deep bug finding in synchronous hardware. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 367–386. Springer International Publishing, Cham (2020)
22. Pandey, M., Bryant, R.E.: Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(7), 918–935 (1999)
23. Hu, Y., Shih, V., Majumdar, R., He, L.: Exploiting symmetries to speed up sat-based boolean matching for logic synthesis of fpgas. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 27(10), 1751–1760 (2008), `https://doi.org/10.1109/TCAD.2008.2003272`
24. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Proceedings of the 11th IFIP WG10.2 International Conference Sponsored by IFIP WG10.2 and in Cooperation with IEEE COMPSOC on Computer Hardware Description Languages and Their Applications. pp. 97–111. CHDL '93, North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (1993)
25. Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., Vaandrager, F.: Adding symmetry reduction to uppaal (2004)
26. Bak, S., Huang, Z., Abad, F.A.T., Caccamo, M.: Safety and progress for distributed cyber-physical systems with unreliable communication. ACM Trans. Embed. Comput. Syst. 14(4) (Sep 2015), `https://doi.org/10.1145/2739046`
27. Maidens, J., Arcak, M.: Exploiting symmetry for discrete-time reachability computations. IEEE Control Systems Letters 2(2), 213–217 (2018)
28. Majumdar, A., Tedrake, R.: Funnel libraries for real-time robust feedback motion planning. The International Journal of Robotics Research 36(8), 947–982 (2017), `https://doi.org/10.1177/0278364917712421`

29. Bujorianu, M., Katoen, J.P.: Symmetry reduction for stochastic hybrid systems. In: 2008 47th IEEE Conference on Decision and Control : CDC ; Cancun, Mexico, 9 - 11 December 2008. - T. 1. pp. 233–238. IEEE, Piscataway, NJ (2008), https://publications.rwth-aachen.de/record/100535, nebent.: Proceedings of the 47th IEEE Conference on Decision and Control
30. Sibai, H., Mokhlesi, N., Mitra, S.: Using symmetry transformations in equivariant dynamical systems for their safety verification. In: Chen, Y.F., Cheng, C.H., Esparza, J. (eds.) Automated Technology for Verification and Analysis. pp. 98–114. Springer International Publishing, Cham (2019)
31. Sibai, H., Mokhlesi, N., Fan, C., Mitra, S.: Multi-agent safety verification using symmetry transformations. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 173–190. Springer International Publishing, Cham (2020)
32. Russo, G., Slotine, J.J.E.: Symmetries, stability, and control in nonlinear systems and networks. Physical Review E 84(4), 041929 (2011)
33. Fan, C., Miller, K., Mitra, S.: Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 629–652. Springer International Publishing, Cham (2020)
34. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V.: CARLA: An open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning. pp. 1–16 (2017)
35. Althoff, M., Koschi, M., Manzinger, S.: Commonroad: Composable benchmarks for motion planning on roads. In: Proc. of the IEEE Intelligent Vehicles Symposium (2017)
36. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: A language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 63–78. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3314221.3314633
37. Alur, R., Henzinger, C.C.T.A., Ho., P.H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) Hybrid Systems. LNCS, vol. 736, pp. 209–229. Springer-Verlag (1993)
38. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science, Morgan Claypool (November 2005), also available as Technical Report MIT-LCS-TR-917, MIT
39. Mitra, S.: A Verification Framework for Hybrid Systems. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA 02139 (September 2007), http://people.csail.mit.edu/mitras/research/thesis.pdf
40. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability Analysis for Neural Feedback Systems Using Regressive Polynomial Rule Inference, p. 157–168. Association for Computing Machinery, New York, NY, USA (2019), https://doi.org/10.1145/3302504.3311807

A Tool architecture

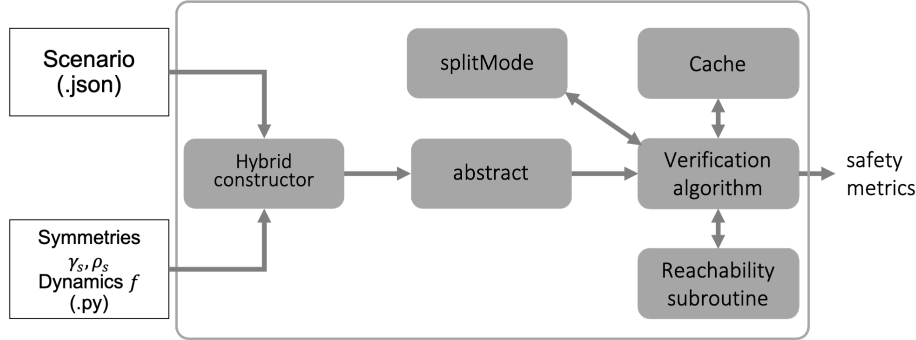

Fig. 1: SceneChecker's architecture. Its inputs are the Scenario file and the Dynamics file. It outputs the safety results and other performance metrics. The Hybrid constructor component constructs a hybrid automaton from the given scenario. The abstract component constructs the symmetry abstraction. The Verification algorithm component implements Algorithms 1 and 2. The splitMode component refines the abstract automaton by splitting a given abstract mode. It implements Algorithm 3 that is presented in Appendix E. The Cache component stores the per-mode initial sets from which reachsets have been computed to avoid repeating computations and for fixed point checking. The Reachability subroutine component is the reachability tool being used for computing reachsets called by computeReachset in the Dynamics file.

B Hybrid Automata definition

In this section, we present a definition of hybrid automata [37–39] that SceneChecker constructs for the given scenario.

Definition 1. *A* hybrid automaton *is a tuple*

$$H := \langle X, P, \Theta, p_{init}, E, guard, reset, f \rangle, \text{ where} \tag{2}$$

(a) *$X \subseteq \mathbb{R}^n$ is the continuous state space, or simply the* state *space, and $P \subseteq \mathbb{R}^d$ is the discrete state space, which we call the* parameter *or* mode *space (it is equal to S in the scenario verification problem),*
(b) *$\langle \Theta, p_{init} \rangle \subseteq X \times P$ is a pair of a compact set of possible initial states and an initial mode ($p_{init} = s_{init}$ in our setting),*
(c) *$E \subseteq P \times P$ is a set of edges that specify possible transitions between modes,*
(d) *$guard : E \to 2^X$ defines the set of states at which a mode transition over an edge is possible,*

(e) $reset : X \times E \rightarrow 2^X$ defines the possible updates of the state after a mode transition over an edge, and
(f) $f : X \times P \rightarrow X$ is the dynamic *function that define the continuous evolution of the state in each mode. It is Lipschitz continuous in the first argument.*

C Symmetries of dynamical systems

In this section, we present formal definitions and sufficient conditions for symmetries of dynamical systems from the literature which are used in Section 4.

Let Γ be a group of smooth maps acting on X.

Definition 2 (Definition 2 in [32]). *We say that $\gamma \in \Gamma$ is a symmetry of the ODE in Section 2 if its differentiable, invertible, and for any solution $\xi(x_0, p, \cdot)$, $\gamma(\xi(x_0, p, \cdot))$ is also a solution.*

Coupled with the notion of symmetries, is the notion of equivarient dynamical systems.

Definition 3 ([32]). *The dynamic function $f : X \times P \rightarrow X$ is said to be Γ-equivariant if for any $\gamma \in \Gamma$, there exists $\rho : P \rightarrow P$ such that,*

$$\forall\, x \in X, \forall\, p \in P,\ \frac{\partial \gamma}{\partial x} f(x,p) = f(\gamma(x), \rho(p)). \tag{3}$$

The following theorem draws the relation between symmetries and equivariance definitions.

Theorem 2 (Theorem 10 in [32]). *If f is Γ-equivariant, then all maps in Γ are symmetries of the ODE in Section 2. Moreover, for any $\gamma \in \Gamma$, map $\rho : P \rightarrow P$ that satisfies equation (3), $x_0 \in X$, and $p \in P$, $\gamma(\xi(x_0, p, \cdot)) = \xi(\gamma(x_0), \rho(p), \cdot)$.*

That means that one can get the trajectory of the system starting from an initial state $\gamma(x_0)$ in mode $\rho(p)$ by transforming its trajectory starting from x_0 in mode p using γ.

D Abstraction definition

In this section, we define formally symmetry abstractions [15]. The abstraction requires a set of symmetry maps which they call the *virtual map*, defined as follows:

Definition 4 (Definition 4 in [15]). *Given a hybrid automaton H, a virtual map is a set*

$$\Phi = \{(\gamma_p, \rho_p)\}_{p \in P}, \tag{4}$$

where for every $p \in P$, $\gamma_p : X \rightarrow X$, $\rho_p : P \rightarrow P$, and they satisfy equation (3).

Given a virtual map Φ, they define a new map $rv : P \rightarrow P$ as follows: $rv(p) = \rho_p(p)$. This is equal to our initialization of the map rv in line 2 of Algorithm 1. It maps modes of the original automaton H to their corresponding ones in the abstract one H_v, defined next.

The idea of the abstraction is to group modes of H that share similar behavior in symmetry terms together in the same mode in the abstract automaton. The trajectories of such modes can be obtained by transforming the trajectories of the corresponding abstract mode using symmetry maps.

Definition 5 (Definition 5 in [15]). *Given a hybrid automaton H, and a virtual map Φ, the resulting* abstract (virtual) hybrid automaton *is:*

$$H_v = \langle X_v, P_v, \Theta_v, p_{init,v}, E_v, guard_v, reset_v, f_v \rangle, \text{ where}$$

(a) $X_v = X$ *and* $P_v = rv(P)$
(b) $X_{init,v} = \gamma_{p_{init}}(\Theta)$ *and* $s_{init,v} = rv(p_{init})$,
(c) $E_v = rv(E) = \{(rv(p_{v,1}), rv(p_{v,2})) \mid e = (p_{v,1}, p_{v,2}) \in E\}$
(d) $\forall e_v \in E_v$,

$$guard_v(e_v) = \bigcup_{e \in rv^{-1}(e_v)} \gamma_{e.src}\big(guard(e)\big),$$

(e) $\forall x_v \in X_v, e_v \in E_v$,

$$reset_v(x_v, e_v) = \bigcup_{e \in rv^{-1}(e_v)} \gamma_{e.dest}\big(reset\big(\gamma^{-1}_{e.src}(x_v), e\big)\big), \text{ and}$$

(f) $\forall p_v \in P_v$, $\forall x \in X, f_v(x, p_v) = f(x, p_v)$.

E Split modes algorithm

In this section, we describe splitMode, the algorithm used to split an abstract mode s_v^* and update the abstract automaton H_v. Then, we prove that its result H_v' is an abstraction of H and that H_v is an abstraction of H_v', using two forward simulation relations.

splitMode **description** The procedure splitMode takes as input the mode to be split s_v^*, the map rv that maps original modes to abstract ones, the original and abstract automata H and H_v, and the original and abstract unsafe maps O and O_v. It outputs the the updated map rv, updated abstract automaton H_v, and the updated abstract unsafe map O_v'.

It starts by obtaining the set of original modes S^* of H represented by s_v^* in line 1. If the number of these modes is less than two, then the algorithm returns the given input to indicate failure to split in line 2. Otherwise, it initializes the output variables rv', H_v', and O_v' by copying rv, H_v,and O_v. It then creates two new abstract modes $s_{v,1}$ and $s_{v,2}$ and adds them to S_v' in line 4. Then, it decomposes S into two disjoint sets S_1^* and S_2^*, in line 5. After that, it updates the map rv to map modes in S_1^* to $s_{v,1}$ and those of S_2^* to $s_{v,2}$ in line 6. It then updates the initial mode of H_v' in case the split mode was the root.

Now that it created the new modes, it proceeds into updating the edges of H_v' and their $guard_v$ and $reset_v$ annotations. It iterates over the edges that connect s_v^* with its *parents*, which may include s_v^* itself, and create for each such edge, two edges connecting that parent with both $s_{v,1}$ and $s_{v,2}$ in line 8. It repeats the same process but for the edges

that connect s_v^* with its *children* in line 10. Finally, it checks if s_v^* had an edge connecting it to itself, and if that is the case, creates two edges connecting each of $s_{v,1}$ and $s_{v,2}$ to themselves.

In line 14, it annotates the created edges with their guards and resets in the same way Definition 5 defined them, but using the update map rv'. In lines 16 and 17, it sets the dynamic function of both modes to be the same as that of s_v^*. Finally, it deletes s_v^* with all edges connected to it in line 18. It then initializes the unsafe maps for the newly created modes by decomposing the unsafe set of s_v^* into those of the two modes. It returns in line 21 the new modes, new rv' and H_v', and the unsafe map O_v'.

Algorithm 3 splitMode($s_v^*, rv, H_v, O_v, H, O$)

1: $S^* \leftarrow rv^{-1}(s_v^*)$.
2: **if** $|P| < 2$ **then return:** rv, H_v, O_v.
3: Create copies of rv, A_v, and O_v, and name them rv', H_v', and O_v'.
4: Create two new virtual modes $s_{v,1}$ and $s_{v,2}$ and add them to S_v'.
5: Split S^* in half to two sets S_1^* and S_2^*.
6: $rv'[S_1^*] \leftarrow s_{v,1}$, $rv'[S_2^*] \leftarrow s_{v,2}$.
7: $s'_{init,v} \leftarrow rv'[s_{init}]$
8: **for** $e_v \in E_v$ such that $e_v.dest = s_v^*$ **do**
9: Create two new edges $(e_v.src, s_{v,1})$ and $(e_v.src, s_{v,2})$.
10: **for** $e_v \in E_v$ such that $e_v.src = s_v^*$ **do**
11: Create two new edges $(s_{v,1}, e_v.dest)$ and $(s_{v,2}, e_v.dest)$.
12: **if** $\exists\ e_v \in E_v$ such that $e_v.src = e_v.dest = s_v^*$ **then**
13: Create two new edges $(s_{v,1}, s_{v,1})$ and $(s_{v,2}, s_{v,2})$.
14: Define the guards and resets of new edges using the virtual map rv'.
15: Remove added edges that have empty guards.
16: Set $f_v'(\cdot, s_{v,1}) = f_v(\cdot, s_v^*)$.
17: Set $f_v'(\cdot, s_{v,2}) = f_v(\cdot, s_v^*)$.
18: Remove s_v^* from S_v' and O_v', and remove all attached edges from E_v'.
19: $O_v'(s_{v,1}) \leftarrow \cup_{s \in S_1} \gamma_s(O_v(s))$
20: $O_v'(s_{v,2}) \leftarrow \cup_{s \in S_2} \gamma_s(O_v(s))$
21: **return:** rv', H_v', O_v'

Correctness guarantees of splitMode In this section, we show that the resulting automaton H_v' from splitMode is still a valid abstraction of H, but it is a tighter one than H_v by showing that H_v is an abstraction of H_v'.

Consider $\mathscr{R}_{rv'}$, the same relation as $\mathscr{R}_{rv}$ defined in Theorem 3 in [15], but using rv' instead of rv. Formally, $\mathscr{R}_{rv'} \subseteq (X \times S) \times (X_v' \times S_v')$ defined as $(x,s)\mathscr{R}_{rv'}(x_v', s_v')$ if and only if:

(a) $x_v' = \gamma_p(x)$, and
(b) $s_v' = rv'(s)$.

Let us refer to $\mathscr{R}_{rv'}$ by $\mathscr{R}_1$ and let $\mathscr{R}_2 \subseteq (X'_v \times S'_v) \times (X_v \times S_v)$ be defined as: $(x'_v, s'_v)\mathscr{R}_2(x_v, s_v)$ if and only if:

(a) $x_v = x'_v$, and
(b)

$$s_v = \begin{cases} s'_v, \text{ if } s'_v \notin \{s_{v,1}, s_{v,2}\}, \\ s^*_v, \text{ otherwise.} \end{cases} \quad (5)$$

The following theorem shows that these two relations are forward simulation relations between H and H'_v and H'_v and H_v, respectively.

Theorem 3. *Fix any abstract mode $s^*_v \in S_v$ of H_v, let*

$$rv', H'_v, O'_v = \mathsf{splitMode}(s^*_v, rv, H_v, O_v, H, O).$$

Then, the resulting relations $\mathscr{R}_1$ and $\mathscr{R}_2$ are FSRs from H to H'_v and H'_v to H_v, respectively, and $H \preceq_{\mathscr{R}_1} H'_v \preceq_{\mathscr{R}_2} H_v$.

Proof. Let us prove the first half first: that $\mathscr{R}_1$ is a FSR from H to H'_v. We do that by showing that H'_v is the result of following Definition 5 to create an abstraction of H using a slightly modified version Φ' of the virtual map Φ, where Φ' itself is another virtual map for H.

By definition, $\forall s \in S, rv(s) = \rho_s(s)$, where $(\gamma_s, \rho_s) \in \Phi$. Let Φ' be equal to Φ for all $s \notin S$, where S is as in line 1 of splitMode. For any $s \in S$, let $\rho'_s(s) = s_{v,1}$, if $s \in S_1$, and $\rho'_s(s) = s_{v,2}$, otherwise. Moreover, as in lines 16 and 17, define the continuous dynamics f'_v to be equal to f_v, for all $s_v \in S_v \setminus \{s_{v,1}, s_{v,2}\}$, and to be equal to $f_v(\cdot, s^*_v)$, otherwise. Then, Φ' is a virtual map of f'_v since any $(\gamma'_s, \rho'_s) \in \Phi'$ satisfies equation (3) for f'_v, because the corresponding $(\gamma_s, \rho_s) \in \Phi$ satisfies it for f_v. The map rv' is just the result of Φ' as rv is the result of Φ.

The edges created in splitMode for $s_{v,1}$ and $s_{v,2}$ in H'_v are a decomposition of the edges connected to s^*_v in H_v, including self edges. Hence, the output of splitMode H'_v is indeed the result of following Definition 5 to construct an abstraction of H using rv'. It follows from Theorem 3 in [15], that H'_v is an abstraction of H and $\mathscr{R}_1 = \mathscr{R}_{rv'}$ is a corresponding FSR.

Now we prove the second half of the theorem: that $\mathscr{R}_2$ is a FSR from H'_v to H_v. We follow similar steps of the proof of the first half in defining a new map, which we name Φ_2, and prove that it is a virtual map of H'_v. Let $\Phi_2 = \{(\gamma_{s'_v}, \rho_{s'_v})\}_{s'_v \in S'_v}$, where $\gamma_{s'_v}(x'_v) = x'_v$ is the identity map and $\rho_{s'_v}(s'_v) = s'_v$, if $s'_v \notin \{s_{v,1}, s_{v,2}\}$, and $\rho_{s'_v}(s'_v) = s^*_v$, otherwise. Because of lines 16 and 17, $(\gamma_{s'_v}, \rho_{s'_v})$ satisfy equation (3) with the RHS dynamic function being f'_v of f. Finally, notice that H_v can be retrieved from H'_v by following Definition 5 using Φ_2. It follows from Theorem 3 in [15], that $\mathscr{R}_2$ is a FSR from H'_v to H_v. Thus, $H \preceq_{\mathscr{R}_1} H'_v \preceq_{\mathscr{R}_2} H_v$.

Usefulness of splitMode **in safety verification** We discuss now the benefits of splitting a mode s^*_v, where $Reach_{H_v}(s^*_v) \cap O_v(s^*_v) \neq \emptyset$. The non-empty intersection with the unsafe set can mean either that:

1. *genuine counterexample*: $\exists s \in S$ of H where $Reach_H(s) \cap O(s) \neq \emptyset$, and thus H is unsafe, or
2. *spurious counterexample*: there exists an execution of H_v that does not correspond to a one of H that is intersecting $O_v(s_v^*)$, and thus the intersection is a result of the abstraction, and not a correct counterexample.

Spurious counter examples could happen because of the guards and resets of the edges incoming to s_v^* being too large that the initial set of states for that mode is being larger than it should. Remember from Definition 5, that the guard and reset of any of these edges e_v is the union of all the transformed guards and resets of the edges in E that get mapped to e_v. If too many of the original edges are mapped to e_v, its guard and reset will get larger, causing more transitions and larger initial set of s_v^* in H_v. This might increase the possibility of spurious counter example. Moreover, by definition, $O_v(s_v)$ is the union of the unsafe sets of all the modes that are mapped to s_v^* under rv. The more the original modes that get mapped to s_v^*, the larger is the unsafe set of s_v^* and the higher is the chance of a spurious counter example.

Upon splitting s_v^* into two abstract modes, the guards of the edges incoming to s_v^*, i.e. where s_v^* is a destination, will have their guards divided between the edges to $s_{v,1}$ and $s_{v,2}$. Additionally, the unsafe sets of s_v^* will be divided between $s_{v,1}$ and $s_{v,2}$. This will make the over-approximation of the behaviors of H by H_v get tighter and safety checking less conservative.

F NN-controlled quadrotor case study

In this section, we will describe a case study of a scenario having a planner, NN controller, and a quadrotor and model it as a hybrid automaton. We use the quadrotor model that was presented in [9] along, its trained NN controller (see Appendix G on how we modify it to be rotation symmetric), and a RRT planner to construct its reference trajectories, independent of its dynamics.

The dynamics of the quadrotor are as follows:

$$\dot{q} := \begin{bmatrix} \dot{p}_x^q \\ \dot{p}_y^q \\ \dot{p}_z^q \\ \dot{v}_x^q \\ \dot{v}_y^q \\ \dot{v}_z^q \end{bmatrix} = \begin{bmatrix} v_x^q \\ v_y^q \\ v_z^q \\ g \tan\theta \\ -g\tan\phi \\ \tau - g \end{bmatrix}, \dot{w} := \begin{bmatrix} \dot{p}_x^w \\ \dot{p}_y^w \\ \dot{p}_z^w \\ \dot{v}_x^w \\ \dot{v}_y^w \\ \dot{v}_z^w \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{6}$$

where q and w are the states of the quadrotor and the planner reference trajectory representing their position and velocity vectors in the 3D physical space, respectively. The variables θ, ϕ, and τ represent the control inputs pitch, roll, and thrust, respectively, provided by the NN controller. The input to the NN controller is the difference between the quadrotor state and the reference trajectory: $q - w$. The $g = 9.81 m/s^2$ is gravity and b_x, b_y, and b_z are piece-wise constant resulting in a piece-wise linear planner trajectory. In our case, these would be determined by the RRT planner as we will explain next.

The NN controller has two hidden layers with 20 neurons each with *tanh* activation units and a linear output layer. It acts as a classifier to choose from a set $\mathbf{U} \subset [-0.1, 0.1] \times$

$[-0.1, 0.1] \times [7.81, 11.81]$ of eight possible control inputs. It was trained to mimic a model predictive control (MPC) controller to drive the quadrotor to follow the planner trajectory. A NN is used for its faster runtime computation and reachability analysis and smaller memory requirements than traditional MPC controllers.

Given an initial set of positions $K \subset \mathbb{R}^3$, a goal set of positions $\mathbf{G} = \cup_i \mathbf{G}_i \subset \mathbb{R}^3$, and a set of 3D obstacles, the planner would generate a directed graph over $\mathbb{R}^3$ that connects K to every G_i with piece-wise linear paths. We denote the set of linear segments in the graph by $\mathbf{R} := \{r_i\}_i$. The planner ensures that the waypoints and segments do not intersect obstacles, but without regard of the quadrotor dynamics.

The Hybrid constructor in SceneChecker models such a scenario as a hybrid automaton:

(a) $X = \mathbb{R}^6$, the space in which the state of the quadrotor q lives, and $S = \mathbf{R}$, the space in which the graph segments live, where the first three and last three coordinates determine the start and end points $s.src$ and $s.dest$ of the segment, respectively,
(b) $\langle \Theta, s_{init} \rangle := \langle [K, [-0.5, 0.5]^3], s_{init} \rangle$, where $[-0.5, 0.5]^3$ are the range of initial velocities of the quadrotor and s_{init} is the initial segment going out of K,
(c) $E := \{(r_i, r_{i+1}) \mid r_i, r_{i+1} \in \mathbf{R},\ r.dest = r_{i+1}.src\}$,
(d) $guard((r_i, r_{i+1}))$ is the 6D ball centered at $[r_i.dest, 0, 0, 0]$ with radius $[1, 1, 1, \infty, \infty, \infty]$, meaning that the quadrotor should arrive within distance 1 unit of the destination waypoint of the first segment, which is equivalent to the source waypoint of the second segment, at any velocity, to be able to transition to the second segment/mode,
(e) $reset(q, (r_i, r_{i+1})) = q$, meaning that there is no change in the quadrotor state after it starts following a new segment, and
(f) $f(q, r) = g(q, h(q, r_i))$, where $g : X \times \mathbf{U} \to X$ is the right hand side of the differential equation of q in equation (6) and $h : X \times S \to \mathbf{U}$ is the NN controller. Without loss of generalization, we assume that $[b_x, b_y, b_z] \in \{-0.125, 0.125\}^3$. b_x is equal to -0.125 if $r.src[0] > r.dest[0]$ and 0.125 otherwise. The same applies for b_y and b_z.

G Symmetry with non-symmetric controllers

G.1 Symmetries of closed loop control systems

In this section, we discuss the property that the controller should satisfy for a closed-loop control system to be symmetric.

Fix an input space $\mathbf{U} \subseteq \mathbb{R}^m$ and consider a right hand side of the ODE in Section 2 of the form:

$$f_c(x, s) := g(x, h(x, s)), \tag{7}$$

where $g : X \times \mathbf{U} \to X$ and $h : X \times S \to \mathbf{U}$ are Lipschitz continuous functions with respect to both of their arguments.

In order to retain symmetry for such systems, we update the notion of equivariance of dynamic functions. But first, let us define symmetric controllers.

Definition 6. *Given three maps $\beta : \mathbf{U} \to \mathbf{U}$, $\gamma : X \to X$, and $\rho : S \to S$. We call the control function h, (β, γ, ρ)-symmetric, if for all $x \in X$ and $s \in S$, $\beta(h(x, s)) = h(\gamma(x), \rho(s))$.*

Definition 6 means that if we transform the input of the controller, the state and the mode, using the maps γ and ρ, respectively, then its output gets transformed with the map β. Such a property formalizes intuitive assumptions about controllers in general. For example, translating the position of the quadrotor and the planned trajectory by the same vector should not change the controller output. The NN controller discussed in Appendix F indeed satisfies this property since its input is the relative state $q-w$. We update the notion of equivariance for closed-loop control systems to account for the controller in the following definition.

Definition 7. *We call the control system dynamic function f_c of equation (7) Γ-equivariant if for any $\gamma \in \Gamma$, there exist $\rho : S \rightarrow S$ and $\beta : \mathbf{U} \rightarrow \mathbf{U}$ such that h is (β, γ, ρ)-symmetric and*

$$\forall\, x \in X, \forall\, u \in \mathbf{U},\ \frac{\partial \gamma}{\partial x} g(x,u) = g(\gamma(x), \beta(u)). \tag{8}$$

The following theorem repeats the results of Theorem 2 for the closed loop control system.

Theorem 4. *If f_c of equation (7) is Γ-equivariant, then all maps in Γ are symmetries. Moreover, for any $\gamma \in \Gamma$, maps $\rho : S \rightarrow S$ and $\beta : \mathbf{U} \rightarrow \mathbf{U}$ that satisfy equation (8), $x_0 \in X$, and $s \in S$, $\gamma(\xi_c(x_0, s, \cdot)) = \xi_c(\gamma(x_0), \rho(s), \cdot)$, where ξ_c is the trajectory of the dynamical system with RHS equation (7).*

Proof. Fix an initial state $x_0 \in X$, a mode $s \in S$, and $\gamma \in \Gamma$ with its corresponding maps ρ and β that satisfy Definition 3 per the assumption of the theorem. For any $t \geq 0$, let $x = \xi_c(x_0, s, t)$ and $y = \gamma(x)$. Then,

$$\begin{aligned}
\frac{dy}{dt} &= \frac{\partial \gamma}{\partial x}\frac{dx}{dt}, \text{ using the chain rule,} \\
&= \frac{\partial \gamma}{\partial x} g(x, h(x,s)), \text{ using equation (7),} \\
&= g(\gamma(x), \beta(h(x,s))), \text{ using equation (8),} \\
&= g(\gamma(x), h(\gamma(x), \rho(s))), \text{using Definition 6,} \\
&= g(y, h(y, \rho(s))), \text{ by substituting } \gamma(x) \text{ with } y, \\
&= f_c(y, \rho(s)).
\end{aligned} \tag{9}$$

Hence, $\gamma(\xi_c(x_0, s, t))$ also satisfies equation (7) and thus a valid solution of the system. Therefore, γ is a symmetry per Definition 2. Moreover, y is a solution starting from $\gamma(x_0)$ in mode $\rho(s)$. This proof is similar to that of Theorem 10 in [32] with is the difference of having a controller h, which requires the additional assumption that h is symmetric.

In Appendix G.2, we discuss how to make non-symmetric controllers symmetric, and apply that to the NN-controller of the quadrotor to make it rotation symmetric.

G.2 From non-symmetric controllers to symmetric ones

In some cases, the controller h is not symmetric. For example, the NN controller of the quadrotor in Section 7 and Appendix F is not symmetric with respect to rotations in the xy-plane. We show a counter example in Figure 2a.

(2a) NN controller of the quadrotor is not rotation symmetric.

(2b) Patching the controller to make it symmetric.

The NN input is the relative state $q - w$, which as we mentioned before, makes it symmetric to translations of the state and the reference trajectory. But, if we rotate the coordinate system of the physical xy-plane, there is no guarantee that the NN will change its outputs θ and ϕ, such that the RHS of $\dot{v}_x^q$, and $\dot{v}_y^q$, $g \tan\theta$ and $-g\tan\phi$, are rotated accordingly.

Such non-symmetric controllers will prevent the dynamics from being equivariant. Equivariance is a desirable, and expected, property of certain dynamical systems. For example, vehicles dynamics are expected to be translation and rotation invariant in the xy-plane. Thus, non-symmetric controllers violate intuition about systems dynamics. Such controllers may not be feasible to abstract using Definition 5. Next, we will suggest a way to make any controller, including NN ones, such as that of the quadrotor, symmetric, leading to better controllers and retrieving the ability to construct abstractions.

Consider again the closed loop control system of equation (7). Let $\Phi = \{(\gamma_s, \rho_s)\}_{s\in S}$ be a set of maps that we want it to be a virtual map of system (7). As before let us define $rv : S \to S$ by $rv(s) = \rho_s(s)$, for all $s \in S$. Assume that for every $s \in S$, there exists $\beta_p : \mathbf{U} \to \mathbf{U}$, such that the open loop dynamic function g is symmetric in the sense that it satisfies equation (8). Moreover, assume that $\gamma_{rv(s)}, \beta_{rv(s)}$, and $\rho_{rv(s)}$ are identity maps for any $s \in S$. This assumption means that applying the same symmetry transformation twice would not change the state nor the mode. Now, let us define a new controller $h' : X \times S \to \mathbf{U}$, that is shown in Figure 2b, as follows:

$$h'(x,s) = \beta_s^{-1}(h(\gamma_s(x), rv(s))). \tag{10}$$

Theorem 5. *For any $s \in S$, the controller h' is (β_s, γ_s, rv)-symmetric.*

Proof. Fix $x \in X$ and $s \in S$. Then, $\beta_s(h'(x,s))$

$$\begin{aligned}
&= \beta_s(\beta_s^{-1}(h(\gamma_s(x), rv(s)))), \text{ using equation (10)}, \\
&= h(\gamma_s(x), rv(s)), \text{ since } \beta_s\beta_s^{-1} \text{ is the identity map}, \\
&= \beta_{rv(s)}^{-1} h(\gamma_{rv(s)}(\gamma_s(x)), \rho_{rv(s)}(rv(s))), \\
&\quad [\text{since } \gamma_{rv(s)} \text{ and } \beta_{rv(s)} \text{ are identity maps, } \rho_{rv(s)}(rv(s)) = rv(s)], \\
&= h'(\gamma_s(x), rv(s)), \qquad (11)
\end{aligned}$$

where the last equality follows from using equation (10) again.

The controller h' ensures that all modes $s \in S$ that get mapped to the same mode s_v by rv have a transformed version, using β_s, of a unique control for the same transformed state $\gamma_s(x)$. That unique control is equal to h with mode s_v. This ensures that all modes that are equivalent under rv have symmetric behavior when the open loop dynamic function g is symmetric as well.

H Machine Specification

- Processor: AMD Ryzen 7 5800X CPU @ 3.8GHz x 8
- Memory: 32GB

I Trying other NN-controlled systems' verification tools as reachability subroutines

1. The state-of-the-art verification tool for NN-controlled systems Verisig needs up to 30 minutes to compute the reachsets for 4 segments in a quadrotor scenario [9]. We tried Verisig and it took similar or longer amount of time for scenarios with less than five segments. We decided to use DryVR for faster evaluation of SceneChecker in this paper.
2. We tried NNV [8], however the resulting reachsets had large over-approximation errors in our scenarios to the point of being not useful. We contacted its developers and they are working on the conservativeness problem.
3. We considered using the tool of "Reachability analysis for neural feedback systems using regressive polynomial rule inference", by Dutta et. al. [40]. We were unable to implement our scenarios in manageable time given that there is no manual to use the tool.

J Reachset and Scenarios Figures

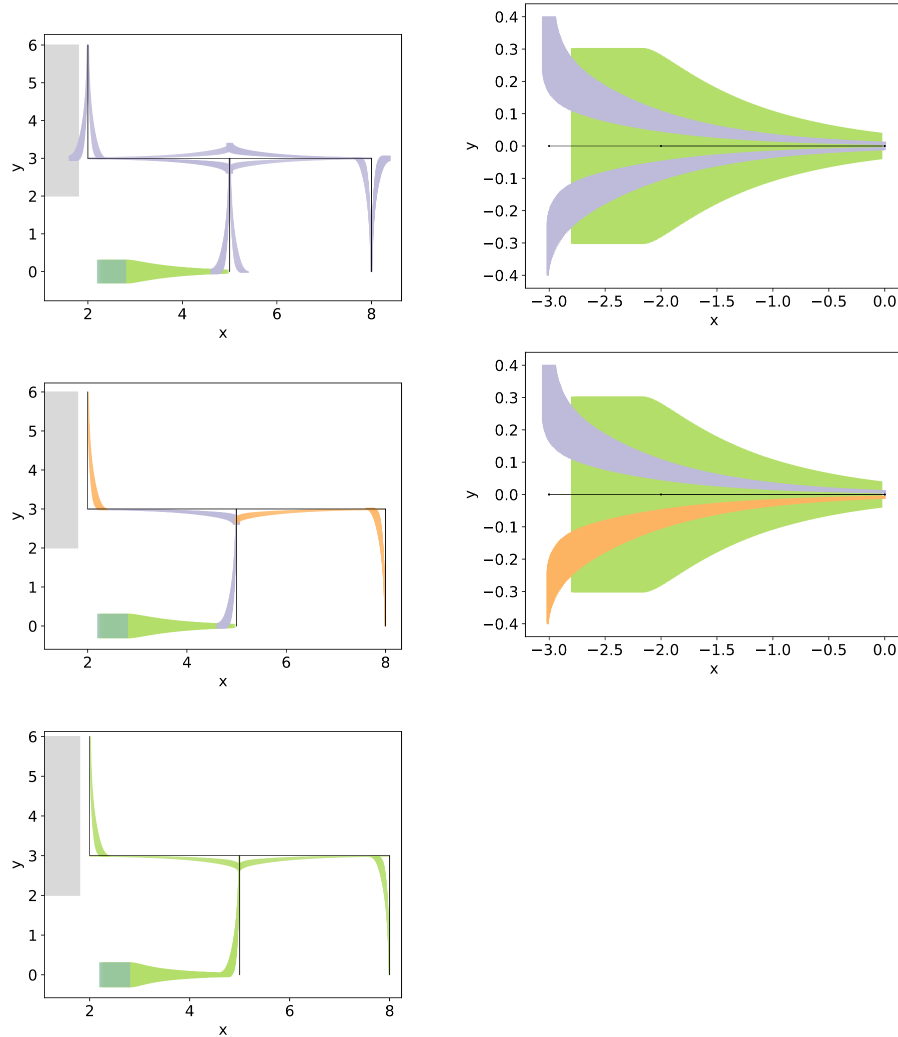

Fig. 3: Scenario verification example C-S1. Uncertain initial positions (blue square). Plan defined by the black segments. Obstacle (grey rectangle). The possible positions the car might pass by (i.e. reachset) while following the plan (green, orange, and violet). The different colors correspond to different abstract segments (defined in Section 5). All figures are generated by SceneChecker's instance with DryVR subroutine using $\Phi =$ TR. The first row is when using abstraction but no refinements are allowed. The second row is where using the abstraction-refinement algorithm (only one refinement needed). The third row is generated without using the abstraction refinement algorithm. The left column represents the concrete scenario with the computed reachsets. This reachset is just used for visualization purposes but not used in the verification process. Only the abstract reachset is used for verification. The right column represents the reachsets of the abstract automaton.

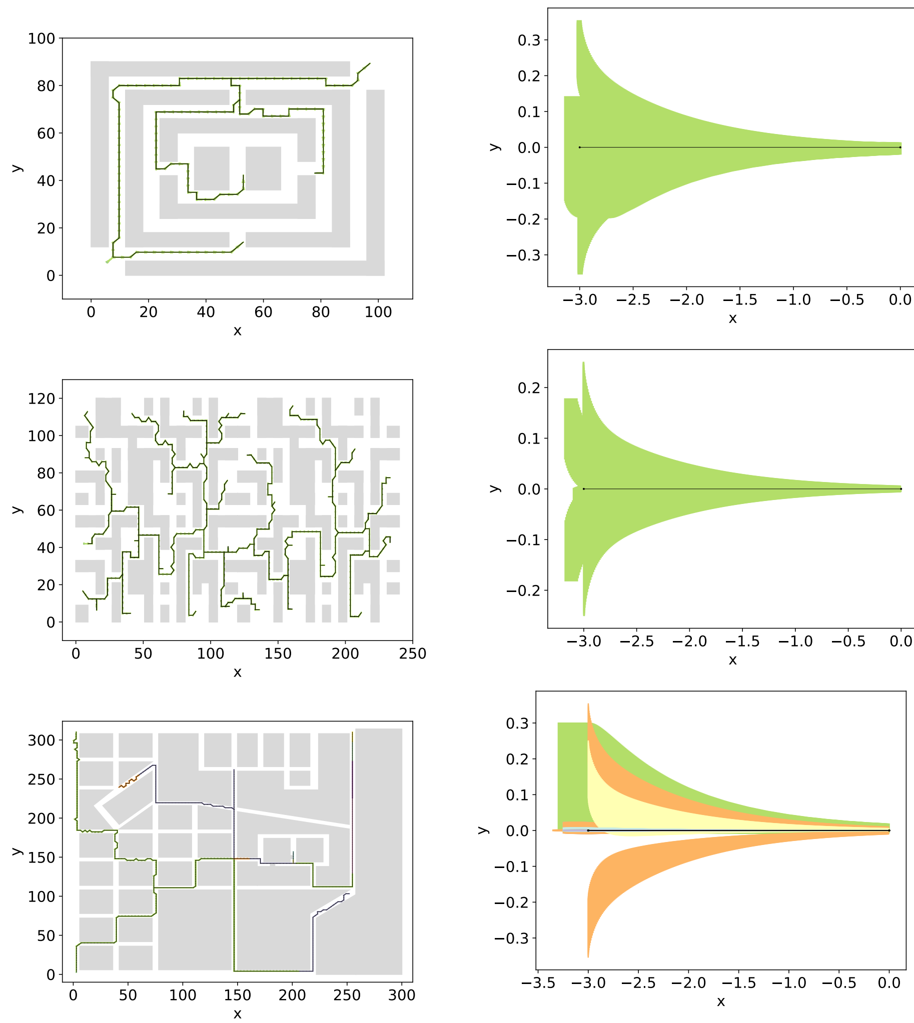

Fig. 4: Reachsets of the car with the PD-controller in scenarios S2 (first row), S3 (second row), S4.b (third row). The left column represents the concrete scenario with the car reachsets. The right column represents the reachsets of the abstract automaton. The different colors correspond to different abstract segments (defined in Section 5). All abstract segments have the same waypoints but represent different segments of the concrete scenario. That means they lead to different reset of the agent's state after mode transitions.

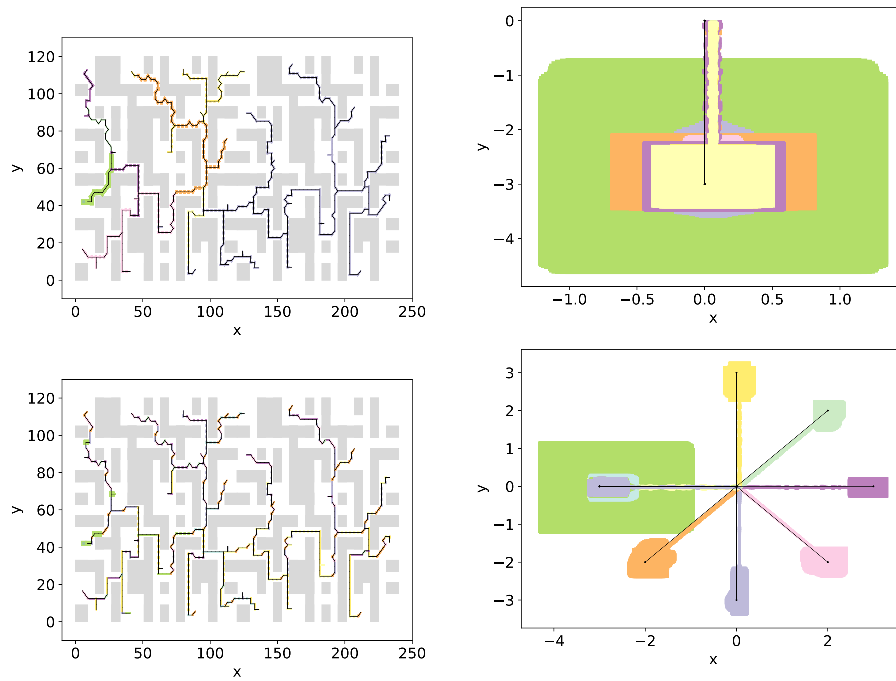

Fig. 5: Reachsets of the quadrotor with the NN-controller while using different Φs. The first row is the quadrotor's reachset when using $\Phi = \mathrm{TR}$. The second row is when using $\Phi = \mathrm{T}$. The left column represents the concrete scenario with the computed reachsets. The right column represents the reachsets of the abstract automaton.