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 waypoints in complex workspaces. Compared to two leading tools, DryVR and Flow*, SceneChecker shows 14× average speedup in verification time, even while using those very tools as reachability subroutines[1].

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 [2,3,4,5,6,7,8,9]. The methods and tools developed have been applied to check safety of aerospace, medical, and autonomous vehicle control systems [4,5,10,11]. 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 [12] and rapidly-exploring random trees (RRTs) [13]) 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

* The authors are supported by a research grant from The Boeing Company and a research grant from NSF (FMITF: 1918531). We would like to thank John L. Olson, Aaron A. Mayne, and Michael R. Abraham from The Boeing Company for valuable technical discussions.

[1] SceneChecker can be found on figshare: https://figshare.com/articles/software/CAV2021_reduce_v6_ova/14504352, gitlab: https://gitlab.engr.illinois.edu/sibai2/multi-drone_simulator/-/tree/CAV_artifact, and its website: https://publish.illinois.edu/scenechecker/. An extended version of this paper is available online [1].

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 [14]. 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* [4] 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. Automaton H_v represents another scenario with fewer segments, each representing an equivalence class 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 set 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 set of symmetric segments with one more segment in the abstract scenario, capturing more accurately the original scenario[2].

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 an average of $14\times$ 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 $13\times$ speedup in verification time over not using abstraction-refinement in scenarios with the NN-controlled quadrotor (Section 7).

[2] A figure showing the architecture of SceneChecker can be found in the extended version [1].

Related work The idea of using symmetries to accelerate verification has been exploited in a number of contexts such as probabilistic models [15,16], automata [17,18], distributed architectures [19], and hardware [20,21]. Some symmetry utilization algorithms are implemented in Murϕ [22] and Uppaal [23].

In our context of cyber-physical systems, Bak et al. [24] 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. [25] proposed a symmetry-based dimensionality reduction method for backward reachable set computations for discrete dynamical systems. Majumdar et al. [26] proposed a safe motion planning algorithm that computes a family of reachsets offline and composes them online using symmetry. Bujorianu et al. [27] 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, we presented a modified version of DryVR that utilizes symmetry to cache reachsets aiming to accelerate simulation-based safety verification of continuous dynamical systems [28]. We developed the related tool CacheReach that implements a hybrid system verification algorithm that uses symmetry to accelerate reachability analysis [29]. 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 we presented in [14]. We 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 [14], 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 [29]. 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$\times$ 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 fixed obstacles, a plan, and an agent that is supposed to execute the plan without running into the obstacles (e.g., see Figure 1.B). 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*[3].A general graph allows for nondeterministic and contingency planning.

[3] 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.

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 $\xi : X \times S \times \mathbb{R}^{\geq 0} \rightarrow X$ 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 $f : X \times S \rightarrow X$ is Lipschitz continuous in the first argument. Note that the trajectories only depend on the segment the agent is following (and not on the 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[4].

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 (e.g. see Figure 1.A). 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.

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

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 $\forall s \in S$, $Reach_H(s) \cap O(s) = \emptyset$.

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 [14]. 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:

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

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 [30]. 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 (see Figure 1.D). 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 (Figure 1.D), 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 [1] (or [14]).

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 $\forall s \in S, Reach_H(s) \cap O(s) = \emptyset$ [14].

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

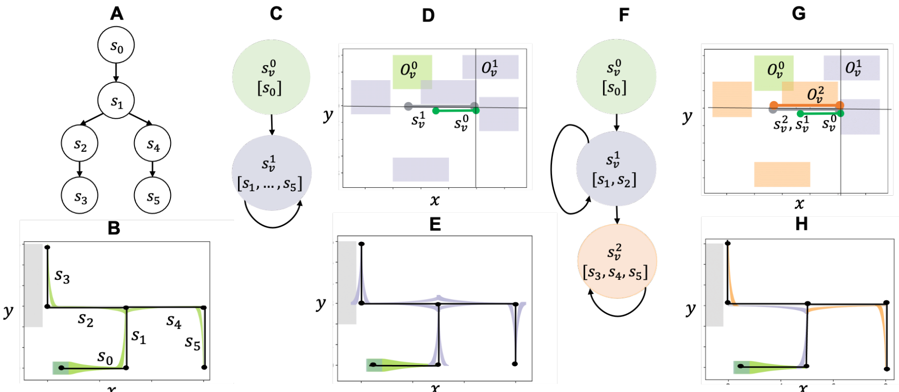

Fig. 1: A simple scenario with a car following a plan with six segments is shown in B. Set of initial positions (green square), unsafe set (grey), and the segments (black lines). The automaton (A) has one mode per segment. Translation and rotation symmetries are used to abstract A to the automaton C. The abstraction translates and rotates each segment of the original scenario to a segment aligned with the x-axis and ends at the origin resulting in the segments (i.e. modes) s_v^0 and s_v^1. The unsafe set is transformed accordingly for each mode as shown in D. SceneChecker computes the reachset of C which turns out to be unsafe; to illustrate the process this abstract reachset transformed to the original scenario is shown in E. The colors refer to a different abstract modes. The algorithm refines C to F adding s_v^2 (same segment as s_v^1 but different guard). The reachset of F is safe and the algorithm terminates (H).

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. An example run is shown in Figure 1.

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

1: $H_v, O_v \leftarrow$ abstract(H, O, Φ)
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: $\quad$ $result, s_v^* \leftarrow$ 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** $rv, H_v, O_v \leftarrow$ splitMode$(s_v^*, rv, H_v, O_v, H, O)$

The core algorithm verify (Algorithm 2) is called iteratively. If verify returns $(safe, \bot)$ or $(unknown, \bot)$, SceneChecker returns the same result. If verify instead results in $(refine, s_v^*)$, splitMode (check the extended version of this paper [1] 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 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 the extended version of this paper [1]). 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: $R_v \leftarrow$ computeReachset$(initset, s_v)$
2: **if** $R_v \cap O_v(s_v) = \emptyset$ **then**
3: **for** $s_v' \in children(s_v)$ **do**
4: $initset' \leftarrow reset_v(guard_v((s_v, s_v')) \cap R_v) \backslash cache[s_v']$
5: **if** $initset' \neq \emptyset$ **then**
6: $result, s_v^* \leftarrow$ verify$(s_v', initset', cache, rv, H_v, O_v)$
7: **if** *result* $=$ *refine* **then return:** *refine*, s_v^*
8: **else if** *result* $=$ *unknown* **then break**
9: **if** $R_v \cap O_v(s_v) \neq \emptyset$ or *result* is *unknown* **then**
10: **if** $|rv^{-1}[s_v]| > 1$ **then return:** *refine*, s_v
11: **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] but modified to be rotation symmetric (check the extended version of this paper [1] 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 [31] 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 SceneChecker and CacheReach, each with two instances, one with DryVR and the other with Flow*, implementing computeReachset. We also use DryVR and Flow* as independent tools to verify the same scenarios. The results of experiments with tools that involve DryVR (i.e., SceneChecker+DryVR, CacheReach+DryVR, and DryVR) are stochastic and change between runs. The reason is that each time DryVR is called, it randomly samples traces of the system from which it computes the requested reachset. We fix the random seed for repeatable results in this section. We show close averaging-based results on SceneChecker's website.

SceneChecker is able to verify all scenarios with PD controllers. The results are shown in Table 1[5] and plotted for C-S1 using SceneChecker+Flow* in Figure 1.

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+DR 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 the other ones 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	1.73	1.28	1	4	0.51	0.52	52	8.20	2.11
C-S2	140	0	1	0.04	0.65	424	19.92	10.57	0	1	0.18	0.79	192	30.95	17.52
C-S3	458	0	1	0.04	4.24	502	19.33	71.41	0	1	0.11	4.34	176	28.64	73.06
C-S4.a	520	2	7	0.26	4.37	404	15.84	94.62	2	7	0.80	4.96	160	25.98	61.53
C-S4.b	520	10	39	1.43	8.69	404	16.06	96.02	10	39	2.83	31.73	160	26.07	60.67
Q-S1	6	1	4	0.04	0.05	NA	TO	0.25	1	4	13.85	14.13	NA	TO	30.17
Q-S2	140	0	1	0.04	0.88	NA	TO	4.97	0	1	3.38	12.62	NA	TO	TO
Q-S3	458	0	1	0.06	5.9	NA	TO	46.34	0	1	4.98	62.66	NA	TO	TO
Q-S4.a	520	0	1	0.06	3.17	NA	TO	56.19	0	1	4.8	34.89	NA	TO	TO
Q-S5	188	0	36	0.85	3.04	NA	TO	8.03	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.

[5] Figures presenting the reachsets of the concrete and abstract automata for different scenarios can be found in the extended version of this paper [1] as well as the machine specifications.

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 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[6]. We tried different Φs. The results are shown in Table 2. When not using abstraction-refinement, SceneChecker took 10.5, 130.95, and 74.15 minutes for the QNN-S2, QNN-S3, and QNN-S4 scenarios, while DryVR took 5.22, 52.56, and 61.31 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× in certain scenarios (QNN-S4 using Φ = T). SceneChecker+DR was more than 10× 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	6	TR	140	1	1	7	17	19	1.51	3.05
CNN-S4	9	TR	520	1	1	10	28	47	3.77	11.25
QNN-S2	3	TR	140	1	1	4	9	9	0.61	3.55
QNN-S3	5	TR	458	1	1	6	16	15	1.51	12.7
QNN-S4	4	TR	520	1	1	5	13	11	1.11	7.43
QNN-S2	0	T	140	7	19	7	19	8	0.53	1.38
QNN-S3	4	T	458	7	30	11	58	29	2.92	16.88
QNN-S4	0	T	520	7	30	7	30	13	1.32	5.34

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 Φ =T resulted in zero mode splits leading to $|S_v|^i = |S_v|^f = 7$, while Φ = TR resulted in 4 mode splits, starting with $|S_v|^i = 1$ modes and ending with $|S_v|^f = 5$, and longer verification time because of refinements. On the other hand, in QNN-S3, Φ = TR resulted in Nref= 5, $|S_v|^f = 6$, and Tt= 12.7 min while Φ =T resulted in Nref= 4, $|S_v|^f = 11$, and Tt= 16.83 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.

[6] Check the extended version [1] 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 $\Phi = \mathrm{T}$. 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 [32], commonroad [33], and Scenic [34].

References

1. Sibai, H., Li, Y., Mitra, S.: Scenechecker: Boosting scenario verification using symmetry abstractions (2021), https://arxiv.org/abs/2011.10713
2. 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 (2011)
3. 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)
4. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: CAV. pp. 258–263. Springer (2013)
5. 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)
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.) CAV (2017)
7. Dutta, S., Chen, X., Jha, S., Sankaranarayanan, S., Tiwari, A.: Sherlock - a tool for verification of neural network feedback systems: Demo abstract. p. 262–263. HSCC '19, ACM, 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.) CAV (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: ACM HSCC (2019)
10. Althoff, M.: An introduction to cora 2015. In: Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
11. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: CAV (2016)
12. 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)
13. Lavalle, S.M.: Rapidly-exploring random trees: A new tool for path planning. Tech. rep. (1998)

14. Sibai, H., Mitra, S.: Symmetry abstractions for hybrid systems and their applications (2020), https://arxiv.org/abs/2006.09485
15. Kwiatkowska, M.Z., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: CAV (2006)
16. 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 TAROS. pp. 26–37 (2015)
17. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Computer Aided Verification, Elounda, Greece, June 28 - July 1, 1993, Proceedings. pp. 463–478 (1993)
18. Clarke, E.M., Jha, S.: Symmetry and induction in model checking. In: Computer Science Today: Recent Trends and Developments, pp. 455–470 (1995)
19. Jacobs, S., Bloem, R.: Parameterized synthesis. Logical Methods in Computer Science [electronic only] 10 (01 2014)
20. 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)
21. 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
22. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Proceedings of the 11th IFIP WG10.2 International Conference. pp. 97–111. CHDL '93, North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (1993)
23. Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., Vaandrager, F.: Adding symmetry reduction to uppaal (2004)
24. 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
25. Maidens, J., Arcak, M.: Exploiting symmetry for discrete-time reachability computations. IEEE Control Systems Letters 2(2), 213–217 (2018)
26. Majumdar, A., Tedrake, R.: Funnel libraries for real-time robust feedback motion planning. The International Journal of Robotics Research 36(8), 947–982 (2017)
27. Bujorianu, M., Katoen, J.P.: Symmetry reduction for stochastic hybrid systems. In: 2008 47th IEEE Conference on Decision and Control : CDC ; Cancun, Mexico, 9 - 2008. - T. 1. pp. 233–238. IEEE, Piscataway, NJ (2008), https://publications.rwth-aachen.de/record/100535
28. 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.) ATVA. pp. 98–114. Springer International Publishing, Cham (2019)
29. 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)
30. Russo, G., Slotine, J.J.E.: Symmetries, stability, and control in nonlinear systems and networks. Physical Review E 84(4), 041929 (2011)
31. Fan, C., Miller, K., Mitra, S.: Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In: Lahiri, S.K., Wang, C. (eds.) CAV (2020)
32. 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)
33. Althoff, M., Koschi, M., Manzinger, S.: Commonroad: Composable benchmarks for motion planning on roads. In: Proc. of the IEEE Intelligent Vehicles Symposium (2017)
34. 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. p. 63–78. PLDI 2019, ACM, New York, NY, USA (2019), https://doi.org/10.1145/3314221.3314633