SOFTWARE TOOLS FOR SCENARIO VERIFICATION OF AUTONOMOUS SYSTEMS EXPLOITING DYNAMICAL SYMMETRIES

BY

YANGGE LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Sayan Mitra

ABSTRACT

In this thesis, we discuss using formal verification techniques to ensure the safety of autonomous systems. We present a particular type of verification problem called scenario verification, which involves vehicles executing complex plans in large cluttered workspaces.

To solve the scenario verification problem, we present the tool SceneChecker. 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 implements symmetry abstractions, a novel refinement algorithm, and is built to enhance the performance of existing reachability analysis tools as a plug-in subroutine. We evaluated SceneChecker on several complicated scenarios with different types of agents. Compared to two leading tools, DryVR and Flow*, SceneChecker shows 20x speedup in verification time, even while using those tools as reachability subroutines.

We further look into a variation of the scenario verification problem, with multiple agents running independently in the shared workspace. In addition, the plan is generated as the agent executing the scenario, which requires safety checking to be performed during runtime. To solve this problem, we present SWERVE, an open-source cloud computing toolkit for efficient runtime collision checking for multi-agent autonomous systems. SWERVE implements a remote server to check safety for different agents by using bounded-time reachability analysis. In addition, SWERVE implements a cache to store already computed reachable sets and reuses them to avoid repeated computations. We evaluate SWERVE on several scenarios and are able to show that SWERVE is able to properly detect potential collisions between agents and static obstacles. In addition, we show that with symmetry and caching, SWERVE is able to obtain 16x average speedup in service response time.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First, I would like to express my most sincere gratitude to my advisor Professor Sayan Mitra. I started working in the group in early 2018 as an undergraduate research assistant and, in the past three years, he patiently guided me to apply the knowledge that I learned in the classroom to solve real-world problems in safe autonomy. Professor Mitra not only taught me how to do research but also helped me with my difficulties in life. All the work in this thesis wouldn't be possible without his generous and patient guidance and encouragement.

I would like to sincerely thank my labmate Hussein Sibai for his enlightened insights and generous help that solved many difficulties in my research. I am also grateful to all other members of the Reliable Autonomy Group for their continuous support.

Last but not least, I would like to express my deepest gratitude to my parents for their financial and moral support since 2015. Their love, support and encouragement, are essential for my life.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivation

Autonomous systems have gained significant attention and have been successfully deployed in various environments. Waymo has already opened a driverless robo-taxi service in Phoenix [1]. It is predicted that by the year 2050, more than 50% of all vehicles on the road will be fully autonomous [2]. However, Uber's fatal accident in 2018 and several other incidents have sharply raised the question: Can autonomous systems be assured safe in the vast number of scenarios that arise in the real world?

A large body of research exists in checking safety for autonomous systems [3, 4, 5, 6, 7, 8, 9, 10]. One essential and common practice is through real-world test driving. For example, Waymo reported driving around 0.8 million miles in 2020 [11] and a total of 20 million miles of real-world testing [12]. Real-world testing is necessary to validate the whole autonomous systems in their actual operating environments. However, real-world testing can become expensive. In addition, it can be hard to create extreme scenarios that can rarely happen in the real world.

An alternative to real-world testing is to perform high-fidelity simulations using photo-realistic simulators. For example, the authors of [13] use the flight simulator X-Plane to find environmental conditions that can cause an autonomous aircraft taxiing system to fail and help improve the system.

However, as argued by Shalev-Shwartz [14], "It will require us to drive 30 billion miles to have enough statistical evidence to argue that the probability of the AV making an unsafe decision that will lead to a fatality per one hour of driving is 10^9". The set of behaviors of most interesting autonomous systems can be too large to the extent that it can even be uncountably infinite. Indeed, we believe that it is not feasible for a reasonable number of

tests, either on the road or in simulations, to establish the absence of bugs.

1.2 Formal Verification Approach to Safe Autonomy

In contrast to testing, a verification method mathematically shows that a model of the system meets the requirements. Currently, formal verification techniques are widely used in the semiconductor design community, also known as the electronic design automation (EDA) community [15]. The EDA community believes that the verification approach can reduce the cost and delays incurred by labor-intensive manual testing. In addition, formal verification methods are used to eliminate bugs that can be hard to find by testing. Besides the EDA community, formal verification has also been used extensively in many areas, including aerospace [16, 17, 18, 19] and avionics [20, 21, 22]. Researchers from NASA [19] use formal verification, especially model checking techniques, to perform various tasks, including verification of Remote Agent Executive (RA EXEC), searching for the Remote Agent Experiment (RAX) anomaly, verification of remote agent planner/scheduler models and consistency checking of remote agent traces. Honeywell [20] uses model-based approaches for the automated verification of complex avionics applications, including flight controls and engine controls.

With all these successes of formal verification, researchers are beginning to explore how formal methods can help assure the safety of autonomous systems. Currently, the researchers are focusing on verifying the software of the autonomous systems with physical constraints from those systems. People are hoping that with the help of formal verification techniques, they can not only detect bugs in the systems but also gain a better understanding of the autonomous systems to improve the design with better efficiency, less energy use, or better performance. In the following subsections, we discuss various approaches that the verification community used to prove the safety of autonomous systems and to understand them better.

1.2.1 Falsification

One common practice for finding bugs in autonomous systems is through falsification. The goal of falsification is to search for a configuration of the

environment and an input to the autonomous system so that the given safety property will be violated. Various tools and algorithms are developed to help finding counter examples [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33].

In [31, 32], the authors create a toolkit VerifAI for the formal design and analysis of artificial intelligence-based systems. The authors designed Scenic, a probabilistic programming language, to write environment models. With Scenic, people can specify distributions encoding interesting types of inputs to the autonomous system and also the environment that the autonomous system is running. Then by using algorithms from [34, 35, 36, 37, 38, 39], the toolkit can efficiently search for inputs and environments that will make the autonomous systems violate the specified safety requirements. In [13], the toolkit is applied to an experimental autonomous aircraft taxiing system, which uses a neural network to track the centerline of a runway. By using the toolkit, they are able to automatically find environment conditions causing the system to violate its specification by deviating significantly from the centerline. The counterexamples found through falsification are then used to identify distinct failure cases and their root causes, which enables them to improve the performance of the autonomous system.

1.2.2 Controller Synthesis

Another approach for assuring the safety of autonomous systems is through controller synthesis. Controller synthesis aims to provide correct-by-construction controllers that can guarantee that the system under control meets certain requirements. Many algorithms are developed to perform controller synthesis [40, 41, 42, 43, 44].

In [42], the authors address the problem of synthesizing a controller for various systems with reach-avoid requirements. They treated the controller as a reference controller and a tracking controller that drives the actual trajectory to follow the reference trajectory. In addition, they find the bound of tracking error between the actual trajectory of the autonomous system following the reference trajectory and use this bound on the tracking error to identify reference trajectories that are guaranteed safe by solving a satisfiability problem over linear constraints.

1.2.3 Reachability Analysis

One common practice for performing formal verification on autonomous systems is through reachability analysis. Reachability analysis is able to generate over-approximations of the set of possible behaviors of the autonomous systems starting from initial conditions given inputs and is able to use this set of possible behaviors to prove that they all satisfy a given property. In the past decades, various tools and algorithms have been developed to perform reachability analysis for both linear [45, 46, 47] and nonlinear [48, 49, 50, 51, 52, 53] hybrid systems.

In addition to proving properties for autonomous systems, reachability analysis can also be used in the falsification [54, 55, 56] and controller synthesis [43, 42, 40, 57] techniques discussed in the previous sections.

1.2.4 Neural Network Verification

Since neural networks are now widely used in various autonomous systems, especially the perception module of those systems, the researchers want to automatically prove that a given network satisfies a specific property of interest, such as robustness to certain perturbations or pre/post conditions.

Currently, many research projects focus on verifying behaviors of neural networks with respect to the input adversarial region. State-of-the-art works in neural network verification include methods based on SMT solving [58], linear approximations [59], and abstract interpretation [60, 61, 62, 63].

In addition, in [64], the author successfully verified neural networks with respect to more complex types of perturbation such as rotation.

1.2.5 Verification of Autonomous Systems with Neural Networks

With the progress in verifying the neural networks, the researchers are directing their attention to verifying the safety properties of closed-loop autonomous systems with neural network components. Verifying autonomous systems with neural network components can be more complicated than verifying the neural networks themselves because the interaction between the

neural network and other components of the system, as well as the nonlinearity of plant dynamics, can introduce extra complexity.

In [65], the authors transform the neural network into an equivalent hybrid system. In this case, they are able to verify the safety properties of systems with neural network controllers using reachability analysis.

In [66], the authors present a method for creating a sound abstraction of the neural network function by abstracting the feedback of the neural network by a polynomial of a given degree plus some error bound. The algorithm is able to verify benchmark with up to 17 dimensions and networks involving around 500 neurons and six layers deep.

In [67], the authors designed a technique for inferring intelligible and safe abstractions for neural network based perception models from system-level safety requirements, data, and program analysis of the modules that are downstream from perception. The technique is applied to various case studies, including a vision-based lane-keeping controller with LaneNet [68] and a corn-row following agriculture robot (AgBot).

1.2.6 Challenges in Formal Verification

Despite all these success stories, applying formal verification techniques to autonomous systems persists as a challenging problem. In this section, we will discuss several challenges that people face while verifying autonomous systems and how people are trying to overcome these challenges.

Hard to Create Specifications One challenge for formal verification to be implemented for autonomous systems is the reliance of most techniques on expressively weak and hard-to-use formal specification languages. In fact, creating correct specifications is challenging. In many cases, a violation of a specification is due to a mistake in the formalization of the specification and not in the actual algorithm or implementation [15]. To address this issue, Chong et al. [69] believe that making specifications explicit in source code, writing proofs-harness in declarative styles, and integrating proof artifacts into the development workflow can help developers correctly and efficiently create specifications.

Sim2Real Gap Because verification of autonomous systems relies on mathematical models that describe the systems, the sim2real gap, which is the difference between a system's model and real performance, becomes a problem for verifying real systems. Overcoming this challenge allows developers to design and test software in simulations with the guarantee that safety properties that hold in the simulation would still hold in real world. Currently, researchers are analyzing the sim2real gap [70, 71, 72] and developing sophisticated simulators [73, 74] and algorithms [75, 76] to close this gap.

Scalability Another major challenge in verifying autonomous systems is scalability with respect to the complexity of the system itself, the operating environment of the system, and the complexity of the property under verification. One effort to overcome this scalability problem is through exploring symmetric properties in agent dynamics. Sibai et al., in work Multi-agent Safety Verification Using Symmetry Transformations [77], uses symmetries to avoid computing reachable sets by transforming previously computed ones to improve the performance of reachability analysis.

1.3 Thesis Contributions

In this thesis, we will look at a particular type of verification problem, the scenario verification problem, which is specified by a set of obstacles, a plan, and an agent. The agent is supposed to execute the plan through the complex environment. The verification task has to ensure that the plan can indeed be safely executed by the vehicle with all the dynamics constraints and the state estimation uncertainties. An example of such problems is shown in Figure 1.1. Verifying such problems with existing techniques can be challenging. First, specifying such scenario verification problems in existing tools presents some practical hurdles. More importantly, analyzing such scenarios can be challenging as the over-approximation errors and the analysis times grow rapidly as the scale of the scenario increases.

On the other hand, scenario verification problems can have lots of repetitions and symmetries, which suggest new opportunities. This thesis introduces SceneChecker [79] and SWERVE [80] that we designed to solve the scenario verification problem.

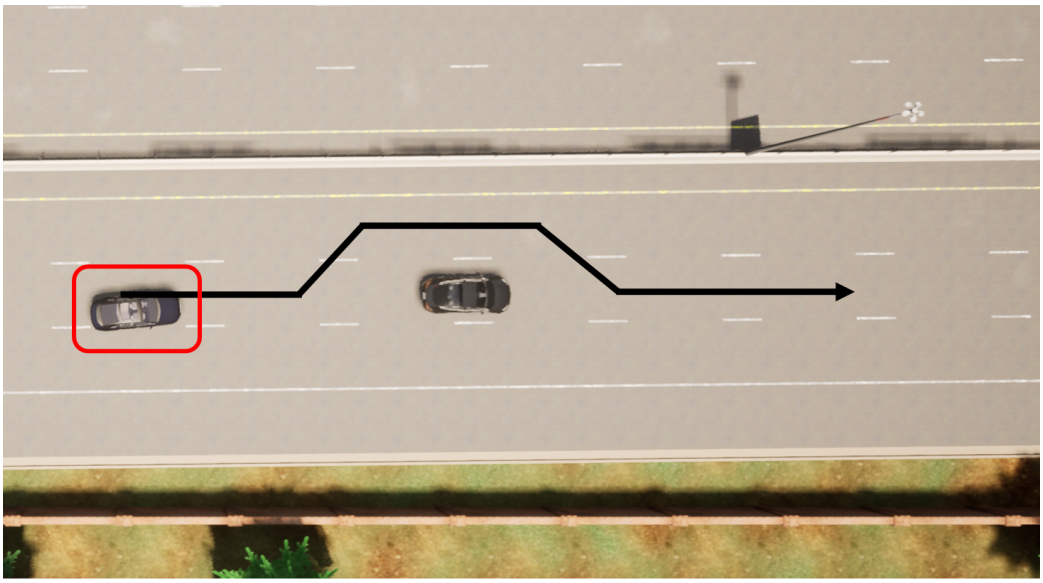

Figure 1.1: This picture shows an example scenario verification problem. The ego vehicle (marked by the red box) is trying to overtake the NPC vehicle stopped in the middle of the lane following the plan specified by the black arrow. We want to show the ego car is able to execute the plan without colliding with obstacles. The scenario is rendered in CARLA [78].

SceneChecker**: Boosting Scenario Verification Using Symmetry Abstractions** In this work, we present SceneChecker [79], a tool that implements a symmetry abstraction-refinement algorithm for efficient scenario verification. We treat the scenario as a hybrid automaton with the modes defined by the segments of the planner, and symmetry abstractions significantly reduce the number of modes and edges of the automaton H representing the scenario by grouping all modes that share symmetric continuous dynamics [81]. SceneChecker offers an easy interface to specify scenarios and is able to use any existing reachability analysis tool as a subroutine.

We evaluated SceneChecker on several scenarios and 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.

SWERVE**: Efficient Runtime Verification of Multi-Agent Systems Using Dynamical Symmetries and Cloud Computing** In this work, we extend the scenario verification problem by having multiple agents run-

ning independently in the shared scenario. In addition, instead of having access to the whole plan, the plan is generated as each agent moves in the workspace. Therefore the safety checking has to happen during runtime. In the rest of this thesis, we will denote this variation of scenario verification problem as the runtime multi-agent verification problem.

We present SWERVE [80], which is an open-source cloud computing toolkit for efficient runtime collision checking of multi-agent autonomous systems. Given a scenario with multiple autonomous systems operating in the same environment, SWERVE utilizes a remote server to perform bounded-time reachability analysis-based verification of safety by checking the absence of collisions for the different agents at runtime. More importantly, building on results demonstrating the effectiveness of symmetries in improving offline verification [79, 82, 77, 83, 84, 85, 86], our Verification Server implements a cache to store reachable sets and reuse them to avoid repeated computations.

The toolkit offers an easy interface for the user to specify the static obstacles in the environment and can check if the reachable sets of the different agents intersect with each other or with other obstacles to check if a collision might happen.

We test SWERVE on several scenarios where more than 20 cars and drones follow independent plans in realistic environments with hundreds of static obstacles. SWERVE is able to properly detect collision among agents and between the agents and the statics obstacles. For 20 closely flying agents, for example, the collision check for a time horizon of 15 s typically takes 0.25 s. This suggests that this client-server approach with caching is a feasible strategy for online collision detection.

1.4 Thesis Structure

The remainder of the thesis is structured as follows. Chapter 2 introduces the scenario verification problem and talks about how we explore symmetry in dynamical systems to solve this problem. This chapter also presents SceneChecker, the tool we implemented to solve the scenario verification problem. Chapter 3 discusses the online multi-agent verification problem and introduces SWERVE, the tool kit we developed to solve this problem. Chapter 4 concludes this thesis and discusses potential research directions.

CHAPTER 2

SCENECHECKER: BOOSTING SCENARIO VERIFICATION USING SYMMETRY ABSTRACTIONS

In this chapter, we describe the scenario verification problem and the difficulties in solving this type of problem. Then, we talk about how we explore symmetry in dynamical systems to help solve this problem. In addition, we present SceneChecker, the tool we implemented to help solve the scenario verification. In the last section, we talk about the experimental evaluation and some observations we have about SceneChecker. The content of this chapter is based on joint work with Sibai et al. in [79].

2.1 Overview

SceneChecker is 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 [81]. 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* [87] and DryVR [49]. 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 much 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 into 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.

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 2.1). 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 2.5).

2.2 Related Works

The idea of using symmetries to accelerate verification has been exploited in a number of contexts, such as probabilistic models [88, 89], automata [90, 91], distributed architectures [92], and hardware [93, 94, 95]. Some symmetry utilization algorithms are implemented in Murϕ [84] and Uppaal [83].

In our context of cyber-physical systems, Bak et al. [85] 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. [96] proposed a symmetry-based dimensionality reduction method for backward reachable set computations for discrete dynamical systems. Majumdar et al. [97] proposed a safe motion planning algorithm that computes a family of reachsets offline and composes them online using symmetry. Bujorianu et al. [86] 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 work, Sibai et al. [82] 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 [77]. 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 [81]. They suggested computing the reachset of the abstract automaton instead of the concrete one then transforming 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 [81], 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 [77]. 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 2.1 in Section 2.5).

2.3 Background

In this section, we formally define the scenario verification problem. We discuss how to construct a hybrid automaton through the scenario to convert the scenario verification problem to a hybrid automaton verification problem. We introduce reachability analysis, a popular method for performing hybrid automaton verification, and the challenges of using reachability analysis to solve scenario verification problems. In the last part of this section, we discuss the definition of symmetry of dynamical systems and how it can help boost performance for verifying scenarios.

2.3.1 Scenario Verification Problem

We are attempting to solve the scenario verification problem. A scenario is defined by a set of obstacles, a plan, and an agent executing the plan. An example scenario is shown in Figure 2.1. The plan, which can be generated by any planner, is defined by a directed graph $G = \langle V, S \rangle$, where the vertices V in the graph are the waypoints in the workspace and the edges S in the graph are segments formed by consecutive waypoints.

The agent is a control system that can execute the plan by following segments in the plan. Let the state space of the agent be X, with initial uncertainty $\Theta \in X$. Let s_{init} be the initial segment to follow. Given the state of the vehicle $x \in X$ and the segment $s \in S$, the vehicle is able to follow the segment along some trajectory ξ. The trajectory is a function $\xi : X \times S \times \mathbb{R}^{\geq 0} \to X$ under certain dynamic constraints. The dynamics of the agent can be specified either by black-box simulators or through ordinary differential equations (ODE). When the agent dynamics is defined by ODE, ξ will be a solution of differential equation $\frac{d\xi}{dt}(x, s, t) = f(\xi(x, s, t), s)$, for any $t \geq \mathbb{R}^{\geq 0}$, $\xi(x, s, 0) = x$ and $f : X \times S \to X$ is Lipschitz continuous with respect to the state. We denote $\xi.fstate$ as the initial state, $\xi.lstate$ the last state, and $\xi.dom$ the time domain of the trajectory ξ.

The obstacles can be viewed as a set of unsafe states $O : S \to 2^X$. The map $tbound : S \to \mathbb{R}^{\geq 0}$ defines the maximum time that the agent follows a given segment. For any pair of segments (segments that share a common waypoint in graph G), $guard((s, s'))$ defines a set of states that the agent can transit from following segment s to s'.

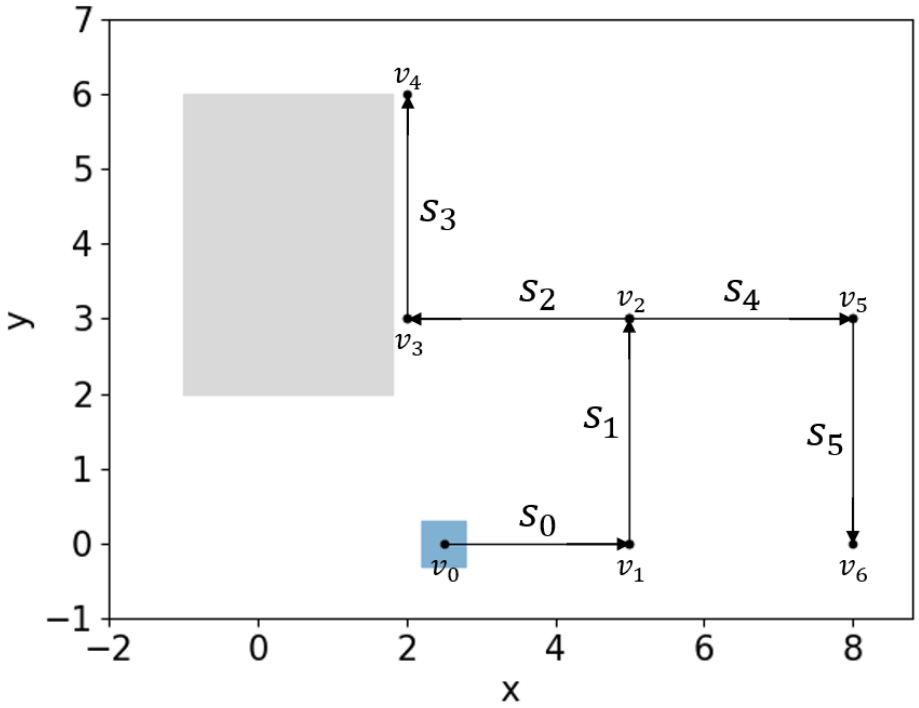

Figure 2.1: An example scenario, the plan is represented by the graph defined by waypoints v_0 to v_6 and segments s_0 to s_5. The obstacle is specified by the gray rectangle. The set of the initial state for the agent is specified by the blue rectangle.

2.3.2 Converting Scenarios to Hybrid Automata

The scenario can be converted to a hybrid automaton. Here we present the definition of hybrid automaton that SceneChecker constructs for the given scenario [79].

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

$$H := \langle X, P, \Theta, p_{init}, E, guard, reset, f \rangle \tag{2.1}$$

1. *$X \subseteq \mathbb{R}^n$ 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).*

2. *$\langle \Theta, p_{init} \rangle \subseteq X \times P$ is a pair of a compact set of possible initial states and an initial mode ($p_{init} = s_{init}$ in our setting).*

3. *$E \subseteq P \times P$ is a set of edges that specify possible transitions between modes.*

4. *$guard : E \to 2^X$ defines the set of states at which a mode transition over an edge is possible.*

5. *$reset : X \times E \to 2^X$ defines the possible updates of the state after a mode transition over an edge.*

6. *$f : X \times P \to X$ is the* dynamic *function that defines the continuous evolution of the state in each mode. It is Lipschitz continuous in the first argument.*

In our case, the agent following a plan in the workspace can be easily modeled as a hybrid automaton H. The discrete modes of the hybrid automaton are defined by the segments S in the plan G. The set of possible initial states and the initial mode are defined by Θ, the set initial position of the agent, and s_{init}, the initial segment that the agent is going to follow. The set of edges of H $E \subseteq S \times S$ can be defined by a pair of consecutive segments in the original plan G. The guard function for each edge $e \in E$, $guard(e)$ for H will be the same as that for the original scenario, which can be the agent arriving at the end waypoint of the segment. The reset for H, in this case, will be the identity map. The dynamic function f will be the agent dynamics specified by either a black-box simulator or ODE. By converting the scenario to a hybrid automaton, we can formally formulate the verification problem.

Verification problem A definition of the verification problem [79] can be the following. An execution of length k is a sequence $\sigma := (\xi_0, s_0), \ldots, (\xi_k, s_k)$, which 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. $\xi_i.lstate \in guard((s_i, s_{i+1}))$

4. $\xi_i.lstate = \xi_{i+1}.fstate$,

5. for each $i \in \{0, \ldots, k\}$, $\xi_i.dom \leq tbound(s_i)$.

The set of *reachable states* is $Reach_H := \{\sigma.lstate \mid \sigma \text{ is an execution}\}$. 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 $\forall s \in S$, $Reach_H(s) \cap O(s) = \emptyset$.

2.3.3 Reachability Analysis

Such a problem can be solved with reachability analysis. Reachability analysis can determine the set of states $Reach_H(s), s \in S$ that the agent can reach starting from a set of initial states while executing the plan. Therefore, as mentioned in the previous section, if the computed reachset does not intersect with any of the obstacles, we can show that all possible executions of the agent in this scenario starting from possible initial states will not collide with obstacles. Many different reachability analysis algorithms and tools have been developed. The key challenge in analyzing autonomous systems using reachability analysis is scalability. The current existing algorithms and tools may have difficulty solving complicated scenarios with a large number of obstacles, complex plans, or highly nonlinear dynamics.

2.3.4 Symmetry for Dynamical Systems

A solution we proposed to overcome the scalability issue is to use the symmetry property of agent dynamics. The symmetry we are using is a pair of maps $\Phi = \{(\gamma_s : X \to X, \rho_s : S \to S)\}$ which satisfies the condition

$$\forall t \geq 0, x \in X, s \in S, \gamma_s(\xi(x, s, t)) = \xi(\gamma_s(x), \rho_s(s), t) \tag{2.2}$$

where $\forall s \in S$, the map γ_s is differentiable and invertible.

When the dynamics of the system are defined by ODE as $\dot{x} = f(x, s)$, a map Φ is a symmetry map if $\forall x \in X$, $\forall s \in S$

$$\frac{\partial \gamma_s}{\partial x} f(x, s) = f(\gamma_s(x), \rho_s(s)) \tag{2.3}$$

This definition of symmetry allows us to transform the agent's trajectories to other symmetric ones starting from symmetric initial states. An example of how symmetry map $\Phi = \{(\gamma, \rho)\}$ can work is shown in Figure 2.2, where

ρ can translate the endpoint of the segment to the origin and rotate the direction of the segment to align with positive x-direction and γ will transform the trajectory of agent accordingly. Note that, in order for a map to satisfy Equation 2.2, the dynamics of the agent have to be invariant under such transformation, which is expected for many vehicle dynamics since the behavior of the vehicle will not be changed under a translation/rotation of lanes. With this definition of symmetry, we want to transform the computed trajectories and reuse them to reduce the total amount of computation.

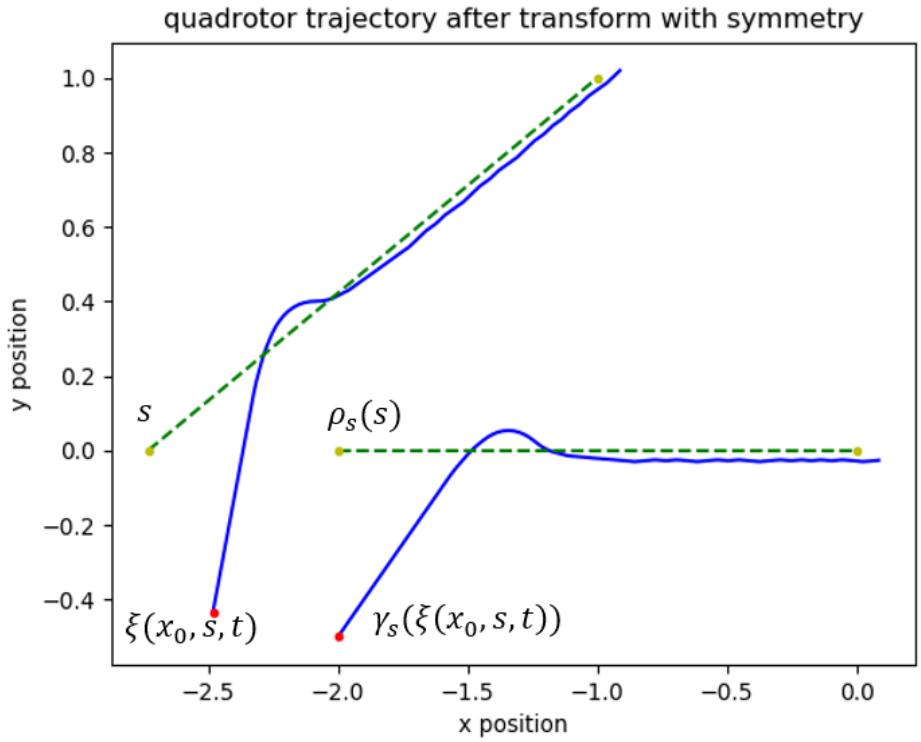

Figure 2.2: The picture shows an example of the symmetry of agent dynamics. The planned segment s (green dashed line on the left) is first rotated and then translated so that its direction is aligned with the positive x and end waypoint at origin to obtain the transformed segment $\rho_s(s)$. The trajectory of the quadrotor $\xi(x_0, s, t)$ is transformed correspondingly to $\gamma_s(\xi(x_0, s, t))$.

2.4 SceneChecker: Algorithm Overview

A tool that we built using the idea of symmetry is SceneChecker [79]. The tool is able to verify scenarios involving vehicles starting from uncertain initial states and executing complicated plans in large cluttered workspaces with fixed obstacles. The tool is built to improve the performance of existing reachability analysis tools by using them as subroutines. The architecture of the tool is shown in Figure 2.3. The inputs to SceneChecker are a scenario JSON file that specifies the scenario and a dynamics python file that contains the symmetry map and the dynamics of the agent. The output of the tool is the verification result (safe or unknown). SceneChecker is also able to visualize various computed reachsets. The output from SceneChecker is sound, which means if SceneChecker returns safe, then the scenario is indeed safe. SceneChecker implements a symmetry abstraction refinement algorithm [98]. The detail about this algorithm will be discussed in the following section.

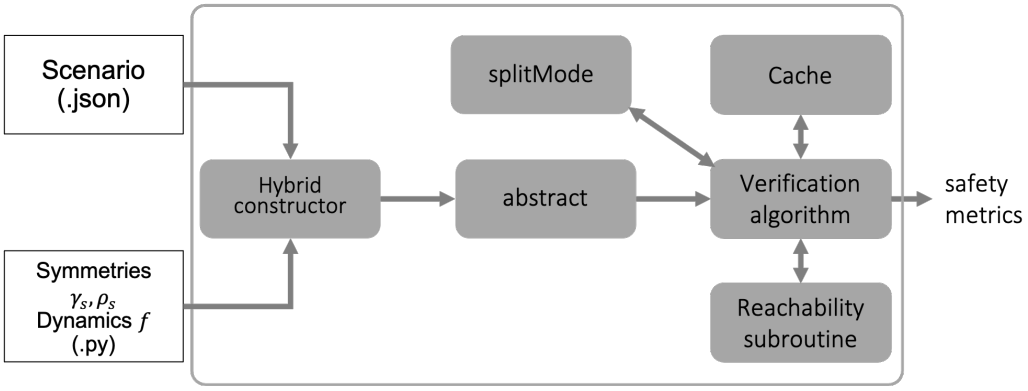

Figure 2.3: High-level architecture of SceneChecker.

2.4.1 Symmetry Abstraction of Hybrid Automaton

The symmetry abstraction part of the algorithm will construct an abstraction hybrid automaton H_v from the original hybrid automaton H created from the scenario using the symmetry map Φ user provided. The mapping between H and H_v works as follows.

1. The mode $s \in S$ for H will now be mapped to $s_v = \rho_s(s)$ in H_v.
2. $\langle \Theta, s_{init} \rangle$, the set of initial states and initial segment the agent should follow in H will be mapped to $\langle \gamma_{s_{init}}(\Theta), \rho_{s_{init}}(s_{init}) \rangle$.

3. The edges $e = (s, s')$ in H will now be mapped to $e_v = (\rho_s(s), \rho_{s'}(s'))$.

4. The guard $guard(e)$ for edge $e = (s, s')$ in H will now be mapped to $\gamma_s(guard(e))$, which will be part of $guard_v(e_v)$ where $e_v = (\rho_s(s), \rho_{s'}(s'))$.

5. The reset map $reset_v(x, e_v)$ for $e_v = (\rho_s(s), \rho_{s'}(s'))$ in H_v will now return $\gamma_{s'}(\gamma_s^{-1}(x))$ instead of the identity map in H.

6. The dynamic function of the agent in each mode will not be changed.

The unsafe set O will be mapped to O_v, following the rule specified in Equation 2.4.

$$\forall s_v \in S_v, O_v(s_v) = \cup_{s \in S, \rho_s(s) = s_v} \gamma_s(O(s)) \tag{2.4}$$

In this case, obstacles of an abstract mode s_v in H_v will be the union of obstacles of each concrete mode s that can be mapped to s_v.

With this mapping, the modes in H that share the same symmetric continuous dynamics can be grouped together in the abstract automaton H_v. An example of this mapping is shown in Figure 2.4. We can show that if H_v is safe, then H is safe, or more formally, if $\forall s_v \in S_v$, $Reach_{H_v}(s_v) \cap O_v(s_v) = \phi$, then $\forall s \in S$, $Reach_H(s) \cap O(s) = \phi$. In this case, we can verify the abstract automaton H_v which may have fewer modes instead of the concrete automaton H.

2.4.2 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 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

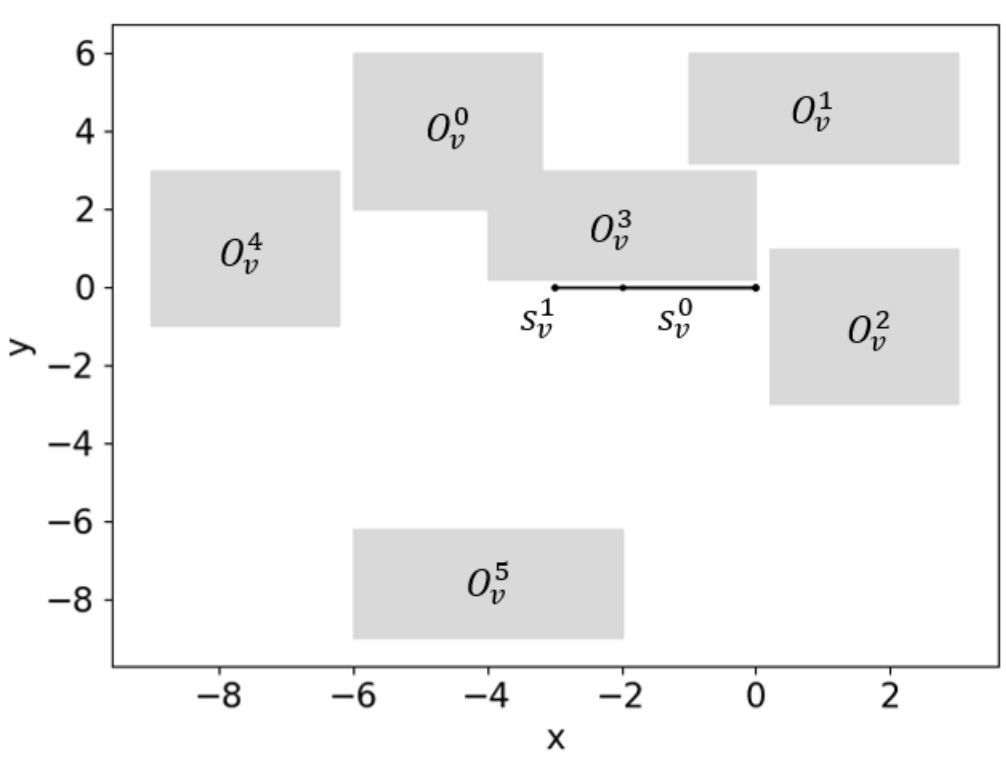

Figure 2.4: Scenario in 2.1 after symmetry abstraction. All the segments in the original scenario are now transformed to align with the x-axis with their endpoints at origin through symmetry mapping. Segments s_1 to s_5 have the same length, so they are mapped to a single mode s_v^1. The obstacles in the transformed scenario will be the union of obstacles of the original segments where $O_v(s_v^0) = \{O_v^0\}$ and $O_v(s_v^1) = \{O_v^1, O_v^2, O_v^3, O_v^4, O_v^5\}$.

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 18) 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

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** $rv, H_v, O_v \leftarrow \mathsf{splitMode}(s_v^*, rv, H_v, O_v, H, O)$
8: $\quad$ **end if**
9: **end while**

$rv^{-1}[s_v]$ is not a singleton set and thus amenable to splitting (line 14). If s_v can be split, verify returns (refine, s_v). Otherwise, verify returns $(\mathit{unknown}, \perp)$ implicitly asking one of s_v's ancestors to be split instead.

One side effect of the symmetry abstraction algorithm is spurious counterexamples. Spurious counterexamples are executions of H_v that do not correspond to any of H that are intersecting $O_v(s_v^*)$. Thus these intersections are results of the abstraction instead of a real counterexample. Since according to the definition of abstract automaton H_v, the guard for edge e_v that corresponds to s_v will be the union of transformed guards for all edges that is mapped to e_v. Therefore, if the guards for coming into a mode s_v^* become too large, the initial set for the abstract mode s_v^* can be too large which may cause over-approximation for the reachset computed for mode s_v^*. In addition, the unsafe set for mode s_v^* will be the union of unsafe sets of all modes that map to s_v^*. Therefore, the more the concrete mode get map to $s_v^*$the larger the unsafe set of s_v^*. These two factors together may cause a high chance of spurious counterexamples.

To solve this problem, we introduce the refinement algorithm [79] that is implemented by function splitMode. If verify function in line 5 in Algorithm 1 results in (refine, s_v^*), splitMode is called to refine H_v by splitting s_v^* into two modes s_v^1 and s_v^2. Each of s_v^1 and s_v^2 will represent part of the set of concrete modes that represent s_v. The edges, guards, resets, and the unsafe sets related to s_v will also be split according to their definition. By doing this, the initial set for each of s_v^1 and s_v^2 will be reduced compared with s_v^* and the unsafe set of s_v^* will also be divided between s_v^1 and s_v^2. This will reduce the over-approximation of the behaviors of H by H_v, and the safety checking result will be less conservative. splitMode can happen until the mode cannot

be split further, i.e., only a single concrete mode is mapped to the current abstract mode. In this case, SceneChecker will return the result *unknown*.

Correctness SceneChecker ensures that all the refined automata H_v's are abstractions of the original hybrid automaton H. 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.*

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.

Counterexamples SceneChecker currently does not find counterexamples 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 counterexamples because of the other errors.

2.5 Experimental Evaluation

Agents and Controllers In our experiments, we consider two types of nonlinear agent models: a standard three-dimensional car (C) with bicycle dynamics and two inputs and a six-dimensional quadrotor (Q) with three 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 [65] (Appendix A for more details) but modified to be rotation symmetric. 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.

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

$R_v \leftarrow$ computeReachset$(initset, s_v)$
if $R_v \cap O_v(s_v) = \emptyset$ **then**
  for $s'_v \in children(s_v)$ **do**
    $initset' \leftarrow reset_v(guard_v((s_v, s'_v)) \cap R_v) n cache[s'_v]$
    if $initset' \neq \emptyset$ **then**
      $result, s^*_v \leftarrow$ verify$(s'_v, initset', cache, rv, H_v, O_v)$
      if $result = refine$ **then return:** $refine, s^*_v$
      else if $result = unknown$ **then break**
      end if
    end if
  end for
end if
if $R_v \cap O_v(s_v) \neq \emptyset$ or $result$ is $unknown$ **then**
  if $|rv^{-1}[s_v]| > 1$ **then return:** $refine, s_v$
  else return: $unknown, \perp$
  end if
end if
$cache[s_v] \leftarrow cache[s_v] \cup initset$
return: $safe, \perp$

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 2.4.1. For each agent and one of its controllers, we manually verified that condition (2.2) is satisfied for each of the two Φs using the sufficient condition for ODEs in Section 2.4.1.

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 [42] after specifying the 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 a 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 2.1. In Table 2.1, both SceneChecker and CacheReach use reachability tools as subroutines. The subroutines used are specified after the '+' sign. The symmetry Φ used 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).

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.

Observation 2: SceneChecker is faster and more exact 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 in-

Table 2.1: Comparison between SceneChecker, DryVR (DR), Flow* (F*), and CacheReach (CacheR).

		SceneChecker+DR				CacheR+DR		DR
Sc.	$\|S\|$	Nrefs	Rc	Rt	Tt	Rc	Tt	Tt
C-S1	6	1	4	0.14	0.15	46	1.75	1.34
C-S2	140	0	1	0.04	0.66	453.86	37.42	11.25
C-S3	458	0	1	0.04	4.26	398.26	33.32	75.35
C-S4.a	520	2	7	0.26	4.52	276.64	23.23	95.02
C-S4.b	520	10	39	1.48	8.90	277.10	23.17	95.05
Q-S1	6	1	4	0.05	0.06	NA	NA	0.25
Q-S2	140	0	1	0.04	0.88	NA	NA	4.93
Q-S3	458	0	1	0.06	5.9	NA	NA	45.03
Q-S4.a	520	0	1	0.06	3.32	NA	NA	55.99
Q-S5	280	0	36	0.85	3.06	NA	NA	4.91

(a) Comparison between SceneChecker, DryVR (DR), and CacheReach.

		SceneChecker+F*				CacheR+F*		F*
Sc.	$\|S\|$	NRefs	Rc	Rt	Tt	Rc	Tt	Tt
C-S1	6	1	4	0.51	0.52	52	8.20	2.11
C-S2	140	0	1	0.18	0.79	192	30.95	17.52
C-S3	458	0	1	0.11	4.34	176	28.64	73.06
C-S4.a	520	2	7	0.80	4.96	160	25.98	61.53
C-S4.b	520	10	39	2.83	31.73	160	26.07	60.67
Q-S1	6	1	4	13.85	14.13	NA	TO	30.17
Q-S2	140	0	1	3.38	12.62	NA	TO	TO
Q-S3	458	0	1	4.98	62.66	NA	TO	TO
Q-S4.a	520	0	1	4.8	34.89	NA	TO	TO
Q-S5	280	NA	NA	NA	TO	NA	TO	TO

(b) Comparison between SceneChecker, Flow* (F*), and CacheReach.

stance 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 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. We tried different Φs. The results are shown in Table 2.2. In addition to the statistics of Table 2.1, 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).

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 2.2 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's instance with DryVR was more than 10× faster than DryVR in the same scenario.

Table 2.2: Comparison between Φs.

Sc.	NRef	Φ	$\lvert S\rvert$	$\lvert S_v\rvert^i$	$\lvert E_v\rvert^i$	$\lvert S_v\rvert^f$	$\lvert E_v\rvert^f$	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 2.2. 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 six 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, Φ = TR resulted in Nref= 7, $|S_v|^f = 8$, and Tt= 13.98 min while Φ =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 a longer verification time for the same scenarios. This can also be seen by comparing the results of Tables 2.1 and 2.2. 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.

2.6 Conclusions and Future Work

The idea of exploiting symmetries in dynamics has proven to be feasible and, as mentioned in the previous section, can really boost the performance of reachability analysis. However, there still exists room for improvement. One possible future direction is to work on better strategies for eliminating the spurious counterexamples and returning actual counterexamples while the scenario is indeed unsafe. Another possible direction is to introduce more uncertainty into the scenarios, including dynamic obstacles or interactive agents. Currently, SceneChecker only explores translation and rotation symmetry, and in the future, we can try to explore more types of symmetries, including permutation or time scaling. Last but not least, it would be valuable to connect SceneChecker with one of the common road simulators, such as CARLA [78], to verify more photo-realistic scenarios.

CHAPTER 3

SWERVE: EFFICIENT RUNTIME VERIFICATION OF MULTI-AGENT SYSTEMS USING DYNAMICAL SYMMETRIES AND CLOUD COMPUTING

In this section, we discuss how we extend the idea of exploring symmetry in dynamical systems to help solve the online multi-agent verification problem. We present SWERVE, a toolkit we implemented to help solve the runtime multi-agent verification problem. In the last part of this chapter, we apply the toolkit to several experiment scenarios and discuss the observations we had during the experiments. The content of this chapter will be based on joint work with Sibai et al. in [80].

3.1 Overview

SWERVE is an open-source cloud computing toolkit for efficient runtime collision checking of multi-agent autonomous systems. Given a scenario with multiple autonomous systems operating in the same environment, SWERVE utilizes a remote server to perform bounded-time reachability analysis-based verification of safety by checking the absence of collisions for the different agents at runtime. SWERVE uses ROS-based communication between the agents and the server. We use the Robot Operating System (ROS) as it has a community of users, interfaces to simulators like Gazebo [99], CARLA [78], and NVIDIA ISAAC [100], many industrial applications [101], and buy-in from the industry (see for example Baidu's Apollo [102], BlueRobotics' BlueROV2 [103]).

The SWERVE can use any existing reachability algorithm as a subroutine to compute reachable sets, which can then be used to check inter-agent collisions. More importantly, building on results demonstrating the effectiveness of symmetries in improving offline verification [79, 82, 77, 83, 84, 85, 86], our Verification Server implements a cache to store reachable sets and reuse them to avoid repeated computations. The idea behind Verification Server is the

same as that of the tool CacheReach [82, 77]. SWERVE implement symmetry transformation and caching to store already computed reachable sets and reuse those reachable sets for later computation.

The toolkit offers an easy interface for the user to specify the static obstacles in the environment for the Verification Server, for example, 3D city maps. The Verification Server checks if the reachable sets of the different agents intersect with each other or with other obstacles to check if a collision might occur.

We test SWERVE on several scenarios where more than 20 cars and drones follow independent plans in realistic environments with hundreds of static obstacles. SWERVE is able to properly detect collision among agents and between the agents and the statics obstacles. With symmetry and caching, SWERVE is able to obtain 16x speed up in average response time. For 20 closely flying agents, for example, the collision check for a time horizon of 15 s typically takes 0.25 s. The results suggest that this client-server approach with caching is a feasible strategy for online collision detection

3.2 Related Works

Significant research has been conducted to develop efficient algorithms and frameworks for runtime verification of autonomous systems [104, 105, 106, 107, 108, 109, 110]. Rozier and Schumann developed the framework R2E2 for runtime verification of autonomous systems [107]. R2E2 focused on the hardware and software components instead of the agent's continuous dynamics. In [111], Majumdar and Tedrake used a previously computed set of reachable sets for safe online motion planning. They used translation and rotation symmetries to compose the offline computed reachable sets in an online manner to over-approximate future behavior and avoid collisions. They did not compute new reachable sets at runtime, nor did they consider multi-agent systems. The closest work to this chapter is CacheReach, developed by Sibai et al. in [77]. They utilize symmetries to accelerate offline safety verification of scenarios with multiple agents. The symmetries are used, as in the CacheReach paper, to avoid computing reachable sets by transforming previously computed ones. A follow-up to CacheReach is SceneChecker [79] which implements a symmetry-based abstraction refinement for efficient offline ver-

ification for an agent following a predefined plan. However, SceneChecker does not consider multi-agent systems. None of the aforementioned works utilize cloud computing for faster reachability analysis computations. In a recent work, Khaled and Zamani presented the tool *pfaces* that utilizes parallel cloud computing to accelerate formal methods, but not runtime verification [112].

3.3 Toolkit Architecture

SWERVE is aimed to help online collision checking for multiple agents in a shared *workspace*. The workspace is a three-dimensional (3D) zone with a map that defines static obstacles such as buildings and unsafe corridors, as shown in Figure 3.1. An *agent* is an entity independently moving in this space. Each agent has a local planner that periodically generates a path for the agent to move from its current position to its next destination. However, the planner does not take into account the positions or the plans of other agents. The agent's actual motion following the plan is governed by its dynamics, and it is affected by various disturbances and uncertainties such as positioning error and wind. The role of the collision checker is to check whether any two such agents in the workspace are likely to collide with each other or with the static obstacles. This collision checking is the key functionality needed in smart intersections [113, 114, 115] and urban air-traffic management (UTM) [116, 117].

Architecture The architecture of SWERVE is shown in Figure 3.2. It consists of the following components: the Verification Server, a set of agents, and the map of the environment with its static obstacles. The Verification Server has two ROS Services: *initialize* and *verifyQuery*, which are presented to the user and the agents as remote procedure calls. The Verification Server has *verify*, which calls the Reachability Subroutine to compute the queried reachable set or retrieve it from the cache. In addition, *verify* checks if any of the intersections between the computed reachable set with the reachable sets of other agents or the static obstacles is non-empty. The map of the environment is provided to the Verification Server through the *initialize* service and will be stored, together with the computed reachable sets for all agents,

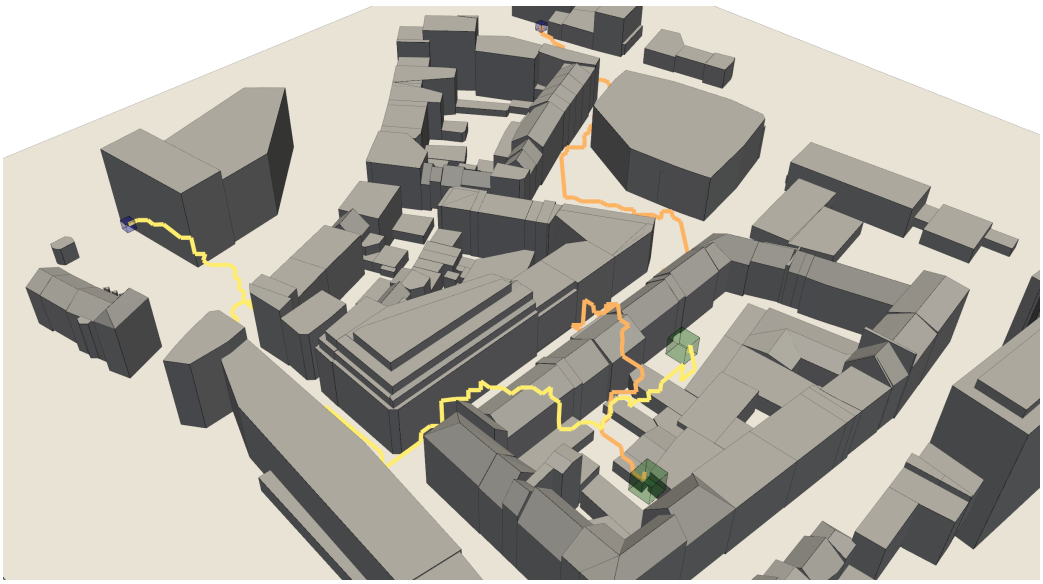

Figure 3.1: 3D map of Cologne with two quadrotors. Each agent is supposed to move from the start position (small blue rectangle) to the goal (green) following plan segments shown by the orange and yellow lines.

in the *UnsafeSet*. The agents interact with the Verification Server through the *verifyQuery* service. A query to the *verifyQuery* service is fulfilled by calling *verify*, which in turn uses the static obstacle map received from the *initialize* service, the cache of reachable sets, the Reachability Subroutine, and the collision checker to answer the query.

As an agent moves in the workspace, it checks if following the next segment in its planned path would lead to a collision by querying the *verifyQuery* service of Verification Server. The time elapsed between the query to the *verifyQuery* service and its response is called the *response time*. The response time can be used to measure the performance of the Verification Server.

A query to the *verifyQuery* ROS Service includes its identifier *id*, the segment to be followed s, the set of possible current states, the dynamics of the agent, the symmetry map Φ, and the look-ahead time T. The dynamics of the agent can either be specified by ordinary differential equations (ODE) or by a black-box simulator. The details about the symmetry map Φ will be discussed in Section 3.4.2.

The Verification Server checks for collision by computing the reachable sets [118] of all querying agents. Given the set of possible current states of the agent Θ, the segment in the workspace it plans to follow s, and the look-ahead time T, the reachable set over-approximates the set of states that the

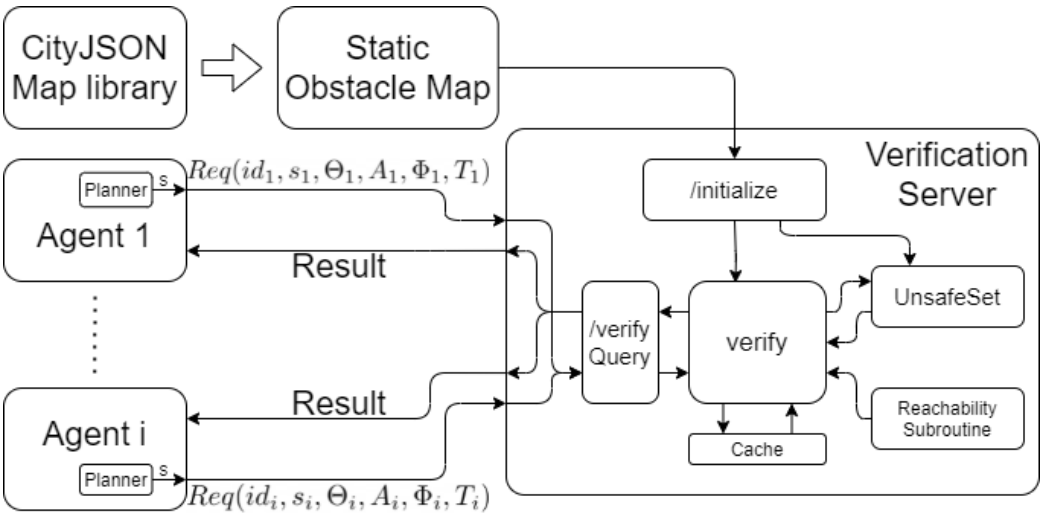

Figure 3.2: The figure shows the architecture of SWERVE.

agent might reach within time T starting from any state in Θ and following the segment s. Therefore, if the reachable set of an agent is not intersecting with any static obstacle or reachable set of another agent, then that agent is guaranteed to not reach a state in which it collides with static obstacles or other agents within time T. The collision checker in the Verification Server stores the reachable set of the most recent query of each agent. The collision checker checks if a newly computed reachable set of an agent intersects with the stored reachable sets of the other agents or with the static obstacles. The Verification Server then replies with a Safe or Unsafe answer to the agent's query as well as with the computed reachable set. More details about the implementation of the server are discussed in Section 3.4.

Initialization The server has to be initialized with the map of the environment listing the static obstacles. This is done in SWERVE by calling the *initialize* ROS Service. The static obstacle map can be specified in three possible ways: (1) a 3D city model in the `CityJSON` format [119], (2) a list of vertices defining polytopes in space-time, or (3) a list of linear inequalities also defining polytopic obstacles.

3.4 Verification Server

In this section, we will discuss the implementation of the Verification Server and its different components.

3.4.1 *verify* Overview

A sketch of the verification algorithm *verify* implemented on the server is shown in Algorithm 3. The execution time of *verify* is the major contributor to the response time described in Section 3.3. *verify* uses a *cache* and a *lock*. The *lock* is used to prevent multiple service calls from modifying the shared contents in the *cache* and the *UnsafeSet* at the same time. Each call to *verify* will have to obtain the *lock* first before executing the service (line 2) and will release the lock only after finishing all the computations (lines 7 and 14). Therefore, only one service request can be fulfilled at a time.

In line 5, *verify* calls the *verifySegment* function described in Algorithm 4. The *verifySegment* function first obtains the reachable set for the agent starting from Θ following segment s for the given time bound T. Then, it checks for collision by checking the non-emptiness of the intersections between the obtained reachable set of the querying agent and the static obstacles and the stored reachable sets of the other agents in *UnsafeSet*. The *verifySegment* function then returns the verification result. If the verification result is *safe* or *unsafe*, *verify* returns it (line 8). Otherwise, *verify* calls the refine function, which is described in Section 3.4.4, to obtain a more accurate reachable set from the cache (line 10). If the *cache* does not store such a reachable set, *verify* sets its `c` flag to True. That means that in the next call to *verifySegment* in the following iteration of the for loop, the reachable set is computed using the reachability subroutine instead of retrieved from the cache (line 11). The for loop terminates after the refine threshold is reached.

3.4.2 Symmetry and Caching

The reachability subroutine in SWERVE can be any of the existing tools such as Flow* [87], CORA [51], DryVR [49], C2E2 [50], and HyLAA [47]. Our current implementation of SWERVE uses DryVR. In addition, SWERVE

Algorithm 3 *verify*$(id, s, \Theta, A, T, \Phi)$

```
1: Global cache, lock
2: lock.acquire()
3: c = False
4: for (i = 0; i < threshold; i++) do
5:     result = verifySegment(id, s, Θ, T, A, Φ, c)
6:     if result = safe or result = unsafe then
7:         lock.release()
8:         return: result
9:     end if
10:    success = cache.refine(key, Θ, Φ, A)
11:    if not success then c = True
12:    end if
13: end for
14: lock.release()
15: return: unsafe
```

utilizes the underlying symmetries in the physics models of the vehicles and a *cache* to speed up the collision checks.

Symmetry The symmetry we used for SWERVE is similar to the one described in Section 2.3.4. For a dynamical system described by the differential equation (3.1) and a corresponding symmetry map Φ, if x is a solution to (3.1), then $\Phi(x)$ will also be a solution to (3.1) [82].

$$\dot{x} = f(x, s), \tag{3.1}$$

where $x \in \mathbb{R}^n$ and $s \in \mathbb{R}^m$. In [77], it has been shown how symmetries can be used to transform previously computed reachable sets to new ones starting from symmetric initial sets of states and following symmetric segments. In SWERVE, we require the user to provide a family of symmetries for the agents' dynamics. The family of symmetries would be a set of pairs of maps $\Phi = \{(\gamma_s, \rho_s)\}_{s \in S}$, where S is the set of segments that might be followed by any of the agents in the workspace. For any $s \in S$, ρ_s transforms the agent's planned segment s to a representative symmetric segment s_v, and γ_s transforms a given agent's state or trajectory following s to its trajectory following s_v. This input requirement is the same as that of SceneChecker in [79] and CacheReach in [77].

Caching SWERVE uses a cache to store computed reachable sets. The implementation of the cache is similar to the one used in CacheReach [79]. For a given query by an agent to the Verification Server to compute the reachable set following segment s, the key to the cache is the representative segment s_v of s, i.e., $s_v := \rho_s(s)$. Since a scenario might have agents with different dynamics, an identifier to the agents' dynamics A is added as part of the key of the cache. After the cache entry is selected, the initial set of states Θ in the query is transformed using γ_s resulting in a new symmetric set of states Θ_v.

In a given entry in *cache*, there might be multiple stored reachable sets starting from different sets of states. The tool CacheReach [77] returns the $\gamma_{s_v}^{-1}$-transformation of the union of all reachable sets stored at that entry with initial sets intersecting Θ_v. This might result in over-approximation errors in the retrieved reachable set [77]. SWERVE tackles this problem by a refinement algorithm described in Section 3.4.4.

3.4.3 *verifySegment* Overview

The *verifySegment* function implements the core algorithm for obtaining the reachable set and performing safety checking against static obstacles and other agents.

The input to the *verifySegment* function is the same as those to the *verify*, except for an additional flag c. It also has access to *cache*, the map of static obstacles O, and the reachable sets corresponding to the most recent request from each agent R.

The algorithm first transforms the initial set of states Θ and plan s to their symmetric representatives Θ_v and s_v using γ_s and ρ_s (line 2). Then, *verifySegment* checks if for agent dynamics A and transformed segment s_v, the transformed initial states Θ_v are already in the cache. If yes, the union of already computed reachable set r_v is retrieved from the cache (line 4) and transformed using γ_s^{-1} to get the reachable set r for the agent following plan s starting from Θ. Otherwise, the reachability subroutine is called to compute the reachable set r (line 8). The computed reachable set will then be transformed using γ_s and stored in the cache (line 9).

The algorithm checks the intersection between r and static obstacles O

(line 12). If the intersection is non-empty and r is retrieved from the cache, the function will return *unknown*. Otherwise, the function will return *unsafe*. The next step is to update the reachable sets in R, which is the dynamic part of *UnsafeSet* (line 17). Since R stores the reachable sets corresponding to the current behaviors of the agents, it can naturally be used to predict near-future collisions between agents. Accordingly, *verifySegment* computes the intersection of r with all the reachable sets stored in R (line 18). If any non-empty intersection exists and r was retrieved from *cache*, *verifySegment* returns *unknown*. If r was computed using the reachability subroutine, *verifySegment* returns *unsafe* instead.

Algorithm 4 *verifySegment*$(id, s, \Theta, A, T, \Phi = \{(\gamma_s, \rho_s)\}, c)$

1: Global *cache*, O, R
2: $\Theta_v = \gamma_s(\Theta), s_v = \rho_s(s)$
3: **if** $cache.in\_cache(\Theta_v, s_v, A)$ and $!c$ **then**
4: $\quad r_v = cache.get(\Theta_v, s_v, A)$
5: $\quad r = \gamma_s^{-1}(r_v)$
6: $\quad from\_cache = True$
7: **else**
8: $\quad r = computeReachSet(\Theta, s, A, T)$
9: $\quad cache.add(\Theta_v, s_v, A, \gamma_s(r))$
10: $\quad from\_cache = False$
11: **end if**
12: **if** $r \cap O \neq \phi$ **then**
13: $\quad$ **if** $from\_cache$ **then return:** *unknown*
14: $\quad$ **elsereturn:** *unsafe*
15: $\quad$ **end if**
16: **end if**
17: R[id] = r
18: **if** $\bigcup_{i!=idx}(r \cap R[i]) \neq \phi$ **then**
19: $\quad$ **if** $from\_cache$ **then return:** *unknown*
20: $\quad$ **elsereturn:** *unsafe*
21: $\quad$ **end if**
22: **end ifreturn:** *safe*

3.4.4 Refinement

Since the returned reachable sets from *cache* have an initial set that contains Θ_v but does not necessarily match it exactly, *verifySegment* might result in

spurious counterexamples.

To solve this problem, we introduce a refinement algorithm *cache.refine* to decrease the over-approximation errors in the retrieved reachable sets from *cache*. This refinement algorithm decomposes the union of reachable sets stored at a given entry in *cache* into different subsets.

When a query matches an entry in *cache*, it computes the Euclidean distances between the center of Θ_v and the centers of the initial sets of these different unions of the stored reachable sets. It selects the reachable sets corresponding to the one with the closest distance. If its initial set contains Θ_v, *cache* transforms it using $\gamma_{s_v}^{-1}$ and returns it as a response to the agent's query. Otherwise, it asks the reachability subroutine to compute it from scratch. This is a heuristic to retrieve more accurate reachable sets from the cache.

The refinement of a *cache* entry can be repeated until the union of stored reachable sets becomes a list of reachable sets. In this case, *cache.refine* returns unsuccess, which will force the *verifySegment* function to compute the reachable set using the reachability subroutine.

3.5 Experimental Evaluation

We test the performance of SWERVE on a number of scenarios with 2D and 3D workspaces and different agent densities and dynamics.

Agents and Dynamics We use two types of agents: a ground vehicle with 3D bicycle dynamics and two inputs (C) and a 6D quadrotor (Q) with six inputs. The bicycle is controlled using a PD controller [120], which is translation and rotation symmetric by design. The quadrotor is controlled by a Neural Network controller from the Verisig paper [65]. This controller is translation symmetric since it takes the difference between the agent state and the segment that the agent follows. Furthermore, we modified the Neural Network controller so that it's also rotation symmetric. Therefore, we are using both translation and rotation symmetries with caching.

Scenarios We experimented with a number of different scenarios. We name scenarios as $Mapi{-}D{-}N$, where (1) *Mapi* indicates the static obstacle map

used; maps with a higher number are more complex and have more complex plans. *Map4* is a partial 3D map of Cologne in CityJSON format. (2) $D \in \{2D, 3D\}$ indicates the dimension of the workspace. (3) N is the number of agents in the scenario; different types of agents may be mixed. We run the above scenarios with different time horizons to vary the load on the verification server.

Recall that the plan from the initial position to goal for each agent is generated independently (using an RRT planner). Therefore, the plans from different agents will indeed intersect in space and time. While running a scenario, the load of the Verification Server can be determined by the number of agents in the scenario together with the time horizon for each plan segment. As the number of agents increases or the time horizon for each plan segment decreases, the Verification Server will be called more frequently, adding loads to the Verification Server.

The plans are fed to the agents segment by segment, which means each agent receives the new segment to follow when it finishes following its current segment. An agent will wait and retry following the segment after 15 s when the plan is detected by *verify* to be unsafe. The initial set of states of verification will be an L_∞ ball centered at the agent's current state, representing sensor and state estimation uncertainty. The reachability tool DryVR [49] is used in the $VerificationServer$ to compute reachable sets. In addition, we add a constant bloating factor to the reachable sets computed by DryVR to simulate the effect of sensor noise while the agent is executing the plan.

We ran each of the scenarios with and without symmetry and caching. The results are shown in Table 3.1. Table 3.1 shows SWERVE's response time (in seconds) in these two cases. In both cases, we are using reachability analysis tool DryVR to compute the reachtubes. The symmetry map Φ used is translation and rotation symmetry. In the table, we present the number of agents (N), the types of agents (Agent), the number of obstacles ($|O|$), the total number of plan segments ($|S|$), the number of calls to the reachability subroutine (Rc), average response time for each service call (ARt), the max response time (MRt), the max response time of 90% of agents ($MRt - 90$), and the average traversal time for each path segment (ASt). From these results, we make the following observations.

Table 3.1: SWERVE response time (in seconds).

				SWERVE				
Sc.	Agent	$\|O\|$	$\|S\|$	Rc	ARt	MRt-90	MRt	ASt
Map1-2D-50	C	100	300	6	0.25	0.25	8.07	15.61
Map2-2D-12	C	236	1457	133	0.95	2.22	4.95	18.30
Map2-2D-17	C	236	1457	197	1.14	2.93	8.62	19.35
Map3-2D-34	C	462	4810	616	2.30	5.48	14.49	21.40
Map1-3D-50	Q	100	300	1	0.18	0.25	2.19	26.84
Map2-3D-17	Q	236	1457	1	0.21	0.30	1.57	16.65
Map4-3D-20	Q	252	2724	25	0.25	0.34	2.67	17.06
Map4-3D-20	C&Q	252	2724	163	0.71	2.09	8.19	18.15

(a) Experiment results for SWERVE with symmetry and caching.

				SWERVE Without Caching		
Sc.	Agent	$\|O\|$	$\|S\|$	ARt	MRt	ASt
Map1-2D-50	C	100	300	75.21	90.33	97.80
Map2-2D-12	C	236	1457	3.63	10.76	21.00
Map2-2D-17	C	236	1457	9.93	13.41	27.84
Map3-2D-34	C	462	4810	35.72	51.22	54.89
Map1-3D-50	Q	100	300	71.16	81.27	96.63
Map2-3D-17	Q	236	1457	6.48	8.34	21.83
Map4-3D-20	Q	252	2724	2.51	5.15	22.91
Map4-3D-20	C&Q	252	2724	10.84	30.05	45.51

(b) Experiment results for SWERVE without symmetry and caching.

SWERVE is able to perform online multi-agent collision checks Table 3.1 shows that the system is able to check potential collisions in both 2D and 3D scenarios with different types of agents (Cols 1-2). In the best case (Row 4), even with 50 agents, the average response time for each service call is only 0.18 s.

Figure 3.3 shows the visualization of computed reachable sets for five agents in a 30 s time interval. The reachable sets shown in orange represent the reachable sets computed in the first 10 s of the 30 s time interval. The reachable sets shown in yellow and light yellow represent the reachable sets computed in the 10-20 s interval and the 20-30 s interval, respectively. The reachable set shown in red represents an unsafe reachable set detected by the Verification Server. We can see that the tool is able to detect a potential col-

lision between Agent3 and Agent4, and therefore, marks the plan for Agent4 to be unsafe. On the other hand, although the reachable sets for Agent1 and Agent2 occupy the same physical space, the Verification Server decided that their planned segments are safe to follow since Agent1 and Agent2 are disjoint in time.

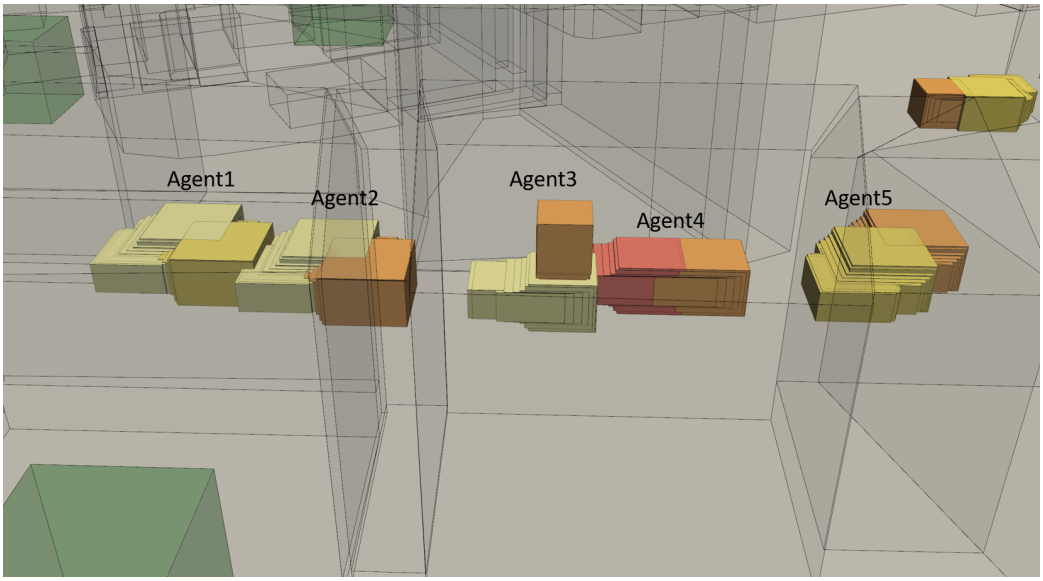

Figure 3.3: The picture shows the computed reachable set of 5 agents in scenario Comp3D-20 in a 30 s time interval.

Symmetry caching can significantly speed up response times If we compare running the scenarios with and without symmetry caching, we can see that the average response time is improved by an average 16x and, in some cases, by more than 300x. In scenario Map1-3D-50, the average response time is only 0.18 s, which provides a maximum of 395x saving in average verification time while using symmetry-based caching. The worst-case response time is also improved with symmetry-based caching. The worst-case response time is improved by an average of 27x. In the same scenario Map1-3D-50, the worst-case response time is 2.19 s, which is 44x faster than without caching.

In addition, we can observe that as the number of agents in the scenario increases, savings from using symmetry-based caching increase. If we compare the average response time for scenario Map2-2D-12 and scenario Map2-2D-17, we can see that as the number of agents increases from 12 to 17, the

saving from using symmetry-based caching increases from 3.8x to 8.7x. A similar trend can also be observed in most other scenarios.

From Table 3.1, we observe that with symmetry and caching, the number of calls to the reachability subroutine is much smaller than the total number of segments in the scenario since many reachable sets were obtained by transforming stored reachable sets in *cache*. In the extreme case, as shown in scenario Map1-3D-50 and Map2-3D-17, only one reachable set was computed, and the rest of the reachable sets were retrieved from *cache*.

The Verification Server is able to finish verification tasks in near real time If we compare the average response time with the average time taken by an agent to traverse a planned segment with typical velocities, we can see that the average response time is around 1/10 of the time for the agent to follow the segment. Even in complicated scenarios with multiple agents with different dynamics (Row 7), the average verification time is only 1/25 of the time it takes for the agent to traverse a segment.

Column 7 in Table 3.1 measures the maximum verification time for 90% (MRt-90) of the agents, which means 90% of the verification requests can be fulfilled before this time. We can see that this time is still much shorter than the total amount of time for the agent to follow the segment. Even in scenario Map3-2D-34, which has the largest value of MRt-90, the value is still only 1/4 of the average total time for the plan to be followed. This suggests that the SWERVE approach is a feasible option for online collision checking.

The performance of Verification Server may decrease as the load increases To further understand the performance of the Verification Server, we varied its load by running scenario map2-2D with different numbers of agents and different time horizons for each of the planned segments. The results are shown in Table 3.2. Besides the values shown in Table 3.1, this table also shows the time horizon for each segment (Ts). From rows 1-3 in Table 3.2, we can clearly see that as the number of agents in the workspace increase from 6 to 17, the average response time increases gradually from 0.71 s to 1.14 s. The max response time and 90% max response time increase as well. This decrease in performance happens not only because Verification Server has to fulfill more verify requests, but also since each verify request may take

Table 3.2: Response time (ARt) with different agent densities in space-time.

Sc.	Ts	Rc	ARt	MRt-90	MRt
Map2-2D-17	15	197	1.14	2.93	8.62
Map2-2D-12	15	133	0.95	2.22	4.95
Map2-2D-6	15	85	0.71	1.96	3.95
Map2-2D-17	20	185	0.90	2.19	8.23
Map2-2D-17	10	173	1.42	3.70	11.69
Map2-2D-17	5	189	2.24	4.97	10.77

longer to fulfill since the collision checker has to check more intersections with other agents.

From rows 1, 3, and 4 in Table 3.2, we can see that as we decrease Ts from 20 to 10, the average response time increases from 0.90 s to 1.42 s, which also shows the decrease of performance in the Verification Server.

Row 5 in Table 3.2 shows an extreme case for the Verification Server. In this case, the number of agents is too large, or Ts is too small so that the verification request is generated faster than the Verification Server can fulfill. In this case, we can see the response time increasing significantly, and the verification tasks can no longer be performed in real time.

The decrease in performance is coming from the fact that the Verification Server can only handle a single verification request at a time. Therefore, when we increase the number of agents or decrease Ts, which will increase the frequency of verification requests, the verification server may not be able to handle the request fast enough, and in this case, some verification requests have to wait until other requests finish before it can be handled. In addition, in some cases, some verification requests that can be fulfilled quickly will have to wait for slower requests to finish first before they can be handled, which can also influence the response time.

3.6 Discussion and Future Work

We present SWERVE, an open-source toolkit for online safety checking of multi-agent systems. It provides a ROS-based communication framework and an implementation of reachable set caching for rapid online collision checking. We apply the toolkit to scenarios with multiple agents with different

dynamics. Our experiments suggest that SWERVE is a promising approach for online checking of inter-agent and static obstacle collisions. In addition, we are able to identify the influence of load on the performance of SWERVE and the reasons for the observations. In the future, it would be beneficial to explore how the sequence of handling verification requests can influence the performance of SWERVE and to further boost the verification speed of the server. Finally, it will be interesting to experiment with SWERVE in scenarios simulated in photo-realistic simulators and in real-world deployments.

CHAPTER 4

CONCLUSION AND FUTURE WORK

In this thesis, we propose solutions to the scenario verification problem and online multi-agent verification problem.

To solve the scenario verification problem, we present the tool SceneChecker. The tool is able to convert the scenario verification problem to a standard hybrid system verification problem and solves it effectively by exploiting symmetric properties in the plan and the vehicle dynamics. SceneChecker implements symmetry abstractions, a novel refinement algorithm, and more importantly, can boost the performance of any existing reachability analysis tool as a plug-in subroutine. We evaluated SceneChecker on several scenarios and compared results with the other two leading tools, DryVR and Flow*. SceneChecker shows significant speedup in verification time, even while using those tools as reachability subroutines.

To solve the online multi-agent verification problem, we present Swerve, an open-source cloud computing toolkit for efficient runtime collision checking for multi-agent autonomous systems. Swerve implements a remote server to check safety for different agents running in a shared workspace using bounded time reachability analysis. In addition, by exploring symmetry in agent dynamics, Swerve implements a cache to store already computed reachable sets and reuses them to avoid repeated computations. We tested Swerve on several scenarios involving more than 20 cars and drones following independent plans in realistic environments with hundreds of static obstacles. We are able to show that Swerve is able to efficiently perform online collision detection.

The results from these works imply the following potential research directions.

For the SceneChecker work, we can investigate other strategies for eliminating spurious counterexamples and returning valid ones in unsafe cases. In addition, it would be essential to address other sources of uncertainty in

scenario verification, such as moving obstacles or interactive agents.

For the SWERVE work, it would be interesting to explore how the sequence of handling verification requests can influence the performance of SWERVE and to further boost the verification speed of the server. In addition, it would be beneficial if we could deploy the SWERVE on some popular cloud computing services such as Amazon Web Service (AWS) so it can easily be accessed by a wide range of people.

For both SceneChecker and SWERVE, it would be beneficial to explore other types of symmetries such as permutation and time scaling.

Last but not least, it would be interesting to apply SceneChecker and SWERVE to scenarios simulated in photo-realistic simulators such as Gazebo [99], CARLA [78], CommonRoad [121], and Scenic [31] or even scenarios running with real-world agents such as F1/10th [122] or PiDrone [123].

APPENDIX A

NN-CONTROLLED QUADROTOR CASE STUDY

In this appendix, 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 [65] along with its trained NN controller and an RRT planner to construct its reference trajectories, independent of its dynamics.

The dynamics of the quadrotor are as follows:

$$\dot{q} := \begin{bmatrix} \dot{p}_x^q \\ \dot{p}_y^q \\ \dot{p}_z^q \\ \dot{v}_x^q \\ \dot{v}_y^q \\ \dot{v}_z^q \end{bmatrix} = \begin{bmatrix} \dot{v}_x^q \\ \dot{v}_y^q \\ \dot{v}_z^q \\ g\tan\theta \\ -g\tan\phi \\ \tau - g \end{bmatrix}, \dot{w} := \begin{bmatrix} \dot{p}_x^w \\ \dot{p}_y^w \\ \dot{p}_z^w \\ \dot{v}_x^w \\ \dot{v}_y^w \\ \dot{v}_z^w \end{bmatrix} = \begin{bmatrix} b_x \\ b_y \\ b_z \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{A.1}$$

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. An 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 to the quadrotor dynamics.

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

1. $X = \mathbb{R}^6$, 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,

2. $\langle \Theta, s_{init} \rangle := \langle [K, [-0.5, 0.5]^3], s_{init} \rangle$, 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,

3. $E := \{(r_i, r_{i+1}) \mid r_i, r_{i+1} \in \mathbf{R},\ r.dest = r_{i+1}.src\}$,

4. $guard((r_i, r_{i+1}))$ 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,

5. $reset(q, (r_i, r_{i+1})) = q$, meaning that there is no change in the quadrotor state after it starts following a new segment, and

6. $f(q, r) = g(q, h(q, r_i))$, where $g : X \times \mathbf{U} \to X$ is the right-hand side of the differential equation of q in equation (A.1) 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.

REFERENCES

[1] K. Korosec, "Waymo's driverless taxi service can now be accessed on google maps," Jun 2021. [Online]. Available: https://techcrunch.com/2021/06/03/waymos-driverless-taxi-service-can-now-be-accessed-on-google-maps/

[2] "Utah department of public safety." [Online]. Available: https://highwaysafety.utah.gov/other-focus-areas/autonomous-vehicles/

[3] P. Koopman and M. Wagner, "Toward a framework for highly automated vehicle safety validation," in *WCX World Congress Experience*. SAE International, apr 2018. [Online]. Available: https://doi.org/10.4271/2018-01-1071

[4] S. Riedmaier, T. Ponn, D. Ludwig, B. Schick, and F. Diermeyer, "Survey on scenario-based safety assessment of automated vehicles," *IEEE Access*, vol. 8, pp. 87 456–87 477, 2020.

[5] J. E. Stellet, M. R. Zofka, J. Schumacher, T. Schamm, F. Niewels, and J. M. Zöllner, "Testing of advanced driver assistance towards automated driving: A survey and taxonomy on existing approaches and open questions," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, 2015, pp. 1455–1462.

[6] W. Huang, K. Wang, Y. Lv, and F. Zhu, "Autonomous vehicles testing methods review," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE Press, 2016. [Online]. Available: https://doi.org/10.1109/ITSC.2016.7795548 p. 163–168.

[7] P. Junietz, W. Wachenfeld, K. Klonecki, and H. Winner, "Evaluation of different approaches to address safety validation of automated driving," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, 2018, pp. 491–496.

[8] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Comput. Surv.*, vol. 52, no. 5, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3342355

[9] Y. Selvaraj, W. Ahrendt, and M. Fabian, "Verification of decision making software in an autonomous vehicle: An industrial case study," in *Formal Methods for Industrial Critical Systems*, K. G. Larsen and T. Willemse, Eds. Cham: Springer International Publishing, 2019, pp. 143–159.

[10] W. Xiang, P. Musau, A. A. Wild, D. M. Lopez, N. Hamilton, X. Yang, J. Rosenfeld, and T. T. Johnson, "Verification for machine learning, autonomy, and neural networks survey," 2018.

[11] A. J. Hawkins, "Waymo and cruise dominated autonomous testing in california in the first year of the pandemic," Feb 2021. [Online]. Available: https://www.theverge.com/2021/2/11/22276851/california-self-driving-autonomous-cars-miles-waymo-cruise-2020

[12] WAYMO, "Waymo safety report," Sep 2020. [Online]. Available: https://waymo.com/safety/safety-report

[13] D. J. Fremont, J. Chiu, D. D. Margineantu, D. Osipychev, and S. A. Seshia, "Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI," in *32nd International Conference on Computer Aided Verification (CAV)*, July 2020.

[14] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "On a formal model of safe and scalable self-driving cars," 2018.

[15] J. B. Michael, D. Drusinsky, and D. Wijesekera, "Formal verification of cyberphysical systems," *Computer*, vol. 54, no. 9, pp. 15–24, 2021.

[16] C. Rouff, A. Vanderbilt, W. Truskowski, J. Rash, and M. Hinchey, "Verification of NASA emergent systems," in *Proceedings. Ninth IEEE International Conference on Engineering of Complex Computer Systems*, 2004, pp. 231–238.

[17] C. Pecheur, "Verification and validation of autonomy software at NASA," 02 2001.

[18] G. Brat and A. Jonsson, "Challenges in verification and validation of autonomous systems for space exploration," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 5, 2005, pp. 2909–2914.

[19] G. Brat, E. Denney, D. Giannakopoulou, J. Frank, and A. Jonsson, "Verification of autonomous systems for space applications," in *2006 IEEE Aerospace Conference*, 2006, p. 11.

[20] D. Bhatt, G. Madl, D. Oglesby, and K. Schloegel, "Towards scalable verification of commercial avionics software," in *AIAA Infotech@ Aerospace 2010*, 2010, p. 3452.

[21] O. Laurent, "Using formal methods and testability concepts in the avionics systems validation and verification process," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 1–10.

[22] J. Barnat, J. Beran, L. Brim, T. Kratochvíla, and P. Ročkai, "Tool chain to support automated formal verification of avionics simulink designs," in *Formal Methods for Industrial Critical Systems*, M. Stoelinga and R. Pinger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 78–92.

[23] T. Akazaki, S. Liu, Y. Yamagata, Y. Duan, and J. Hao, "Falsification of cyber-physical systems using deep reinforcement learning," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 456–465.

[24] Y. Annpureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-taliro: A tool for temporal logic falsification for hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 254–257.

[25] J. L. Eddeland, A. Donzé, S. Miremadi, and K. Åkesson, "Industrial temporal logic specifications for falsification of cyber-physical systems," in *ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20), Berlin, Germany, July 12, 2020*, ser. EPiC Series in Computing, vol. 74. EasyChair, 2020. [Online]. Available: https://doi.org/10.29007/r74f pp. 267–274.

[26] G. Ernst, S. Sedwards, Z. Zhang, and I. Hasuo, "Falsification of hybrid systems using adaptive probabilistic search," *ACM Trans. Model. Comput. Simul.*, vol. 31, no. 3, July 2021. [Online]. Available: https://doi.org/10.1145/3459605

[27] C. Menghi, S. Nejati, L. Briand, and Y. I. Parache, "Approximation-refinement testing of compute-intensive cyber-physical models: An approach based on system identification," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3377811.3380370 p. 372–384.

[28] M. Waga, "Falsification of cyber-physical systems with robustness-guided black-box checking," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3365365.3382193

[29] S. Yaghoubi and G. Fainekos, "Gray-box adversarial testing for control systems with machine learning components," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302504.3311814 p. 179–184.

[30] Z. Zhang, G. Ernst, S. Sedwards, P. Arcaini, and I. Hasuo, "Two-layered falsification of hybrid systems guided by monte carlo tree search," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2894–2905, 2018.

[31] D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia, "Scenic: A language for scenario specification and scene generation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3314221.3314633 p. 63–78.

[32] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, "VerifAI: A toolkit for the formal design and analysis of artificial intelligence-based systems," in *31st International Conference on Computer Aided Verification (CAV)*, July 2019.

[33] Q. Thibeault, J. Anderson, A. Chandratre, G. Pedrielli, and G. Fainekos, "PSY-TaLiRo: A Python toolbox for search-based test generation for cyber-physical systems," in *Formal Methods for Industrial Critical Systems*, A. Lluch Lafuente and A. Mavridou, Eds. Cham: Springer International Publishing, 2021, pp. 223–231.

[34] T. Dreossi, A. Donze, and S. A. Seshia, "Compositional falsification of cyber-physical systems with machine learning components," in *Proceedings of the NASA Formal Methods Conference (NFM)*, May 2017.

[35] T. Dreossi, A. Donzé, and S. A. Seshia, "Compositional falsification of cyber-physical systems with machine learning components," *Journal of Automated Reasoning*, vol. 63, no. 4, pp. 1031–1053, 2019.

[36] T. Dreossi, S. Ghosh, X. Yue, K. Keutzer, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Counterexample-guided data augmentation," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018. [Online]. Available: https://doi.org/10.24963/ijcai.2018/286 pp. 2071–2078.

[37] T. Dreossi, S. Jha, and S. A. Seshia, "Semantic adversarial deep learning," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 3–26.

[38] S. A. Seshia, A. Desai, T. Dreossi, D. Fremont, S. Ghosh, E. Kim, S. Shivakumar, M. Vazquez-Chanlatte, and X. Yue, "Formal specification for deep neural networks," in *Proceedings of the International Symposium on Automated Technology for Verification and Analysis (ATVA)*, October 2018, pp. 20–34.

[39] S. Ghosh, F. Berkenkamp, G. Ranade, S. Qadeer, and A. Kapoor, "Verifying controllers against adversarial examples with Bayesian optimization," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018. [Online]. Available: http://dx.doi.org/10.1109/ICRA.2018.8460635

[40] N. Kochdumper, F. Gruber, B. Schürmann, V. Gaßmann, M. Klischat, and M. Althoff, "AROC: A toolbox for automated reachset optimal controller synthesis," in *Proc. of the 24th International Conference on Hybrid Systems: Computation and Control*, 2021.

[41] V. Gaßmann and M. Althoff, "Verified polynomial controller synthesis for disturbed nonlinear systems," *IFAC-PapersOnLine*, vol. 54, no. 5, pp. 85–90, 2021, 7th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896321012544

[42] C. Fan, K. Miller, and S. Mitra, "Fast and guaranteed safe controller synthesis for nonlinear vehicle models," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 629–652.

[43] C. Fan, U. Mathur, S. Mitra, and M. Viswanathan, "Controller synthesis made real: Reach-avoid specifications and linear dynamics," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 347–366.

[44] Z. Huang, Y. Wang, S. Mitra, and G. Dullerud, "Controller synthesis for linear dynamical systems with adversaries," in *Proceedings of the Symposium and Bootcamp on the Science of Security*, ser. HotSos '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2898375.2898378 p. 53–62.

[45] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *CAV*, 2011, pp. 379–395.

[46] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, and R. Grosu, "Xspeed: Accelerating reachability analysis on multi-core processors," in *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, ser. Lecture Notes in Computer Science, N. Piterman, Ed., vol. 9434. Springer, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-26287-1_1 pp. 3–18.

[47] S. Bak and P. S. Duggirala, "HyLAA: A tool for computing simulation-equivalent reachability for linear systems," in *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3049797.3049808 p. 173–178.

[48] C. Fan, B. Qi, S. Mitra, M. Viswanathan, and P. S. Duggirala, "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*, 2016, pp. 531–538.

[49] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DryVR: Data-driven verification and compositional reasoning for automotive systems," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 441–461.

[50] X. Chen, S. Sankaranarayanan, and E. Abraham, "Flow* 1.2: More effective to play with hybrid systems," in *ARCH14-15. 1st and 2nd International Workshop on Applied Verification for Continuous and Hybrid Systems*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 34. EasyChair, 2015. [Online]. Available: https://easychair.org/publications/paper/QrVj pp. 152–159.

[51] M. Althoff, "An introduction to CORA 2015," in *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

[52] S. Kong, S. Gao, W. Chen, and E. Clarke, "dReach: δ-reachability analysis for hybrid systems," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 200–205.

[53] S. Bogomolov, M. Forets, G. Frehse, K. Potomkin, and C. Schilling, "Juliareach: A toolbox for set-based reachability," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019, pp. 39–44.

[54] S. Bogomolov, G. Frehse, A. Gurung, D. Li, G. Martius, and R. Ray, "Falsification of hybrid systems with symbolic reachability analysis and trajectory splicing," *Nonlinear Analysis: Hybrid Systems*, vol. 42, p. 101093, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1751570X21000832

[55] S. Bogomolov, G. Frehse, A. Gurung, D. Li, G. Martius, and R. Ray, "Falsification of hybrid systems using symbolic reachability and trajectory splicing," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302504.3311813 p. 1–10.

[56] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 167–170.

[57] S. B. Liu, B. Schürmann, and M. Althoff, "Reachability-based identification, analysis, and control synthesis of robot systems," 2021.

[58] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham: Springer International Publishing, 2017, pp. 97–117.

[59] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/wang-shiqi pp. 1599–1614.

[60] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai 2: Safety and robustness certification of neural networks with abstract interpretation," in *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[61] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018. [Online]. Available: https://proceedings.mlr.press/v80/mirman18b.html pp. 3578–3586.

[62] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/f2f446980d8e971ef3da97af089481c3-Paper.pdf

[63] H.-D. Tran, S. Bak, W. Xiang, and T. T. Johnson, "Verification of deep convolutional neural networks using imagestars," in *32nd International Conference on Computer-Aided Verification (CAV)*. Springer, July 2020.

[64] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290354

[65] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "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*, 2019, pp. 169–178.

[66] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari, "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*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302504.3313351 p. 262–263.

[67] C. Hsieh, K. Joshi, S. Misailovic, and S. Mitra, "Verifying controllers with convolutional neural network-based perception: A case for intelligible, safe, and precise abstractions," 2021.

[68] Z. Wang, W. Ren, and Q. Qiu, "Lanenet: Real-time lane detection networks for autonomous driving," *ArXiv*, vol. abs/1807.01726, 2018.

[69] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3377813.3381347 p. 11–20.

[70] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Case study: Verifying the safety of an autonomous racing car with a neural network controller," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3365365.3382216

[71] T. Jaunet, G. Bono, R. Vuillemot, and C. Wolf, "SIM2REALVIZ: Visualizing the sim2real gap in robot ego-pose estimation," *CoRR*, vol. abs/2109.11801, 2021. [Online]. Available: https://arxiv.org/abs/2109.11801

[72] A. Kadian, J. Truong, A. Gokaslan, A. Clegg, E. Wijmans, S. Lee, M. Savva, S. Chernova, and D. Batra, "Are we making real progress in simulated environments? measuring the sim2real gap in embodied visual navigation," 2019.

[73] B. Nemire, "Training in Nvidia Isaac Sim closes the sim2real gap," Oct 2021. [Online]. Available: https://developer.nvidia.com/blog/training-in-nvidia-isaac-sim-closes-the-sim2real-gap/

[74] E. Heiden, "Closing the sim2real gap using invertible simulators."

[75] J.-B. Weibel, T. Patten, and M. Vincze, "Addressing the sim2real gap in robotic 3-d object classification," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 407–413, 2020.

[76] J. Hietala, D. Blanco-Mulero, G. Alcan, and V. Kyrki, "Closing the sim2real gap in dynamic cloth manipulation," *arXiv preprint arXiv:2109.04771*, 2021.

[77] H. Sibai, N. Mokhlesi, C. Fan, and S. Mitra, "Multi-agent safety verification using symmetry transformations," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 173–190.

[78] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[79] H. Sibai, Y. Li, and S. Mitra, "Scenechecker: Boosting scenario verification using symmetry abstractions," 2021.

[80] Y. Li, H. Sibai, and S. Mitra, "Swerve: Efficient runtime verification of multi-agent systems using dynamical symmetries and cloud computing," Nov 2021. [Online]. Available: https://figshare.com/articles/software/Swerve_Efficient_Runtime_Verification_of_Multi-Agent_Systems_Using_Dynamical_Symmetries_and_Cloud_Computing/16910812/2

[81] H. Sibai and S. Mitra, "Symmetry abstractions for hybrid systems and their applications," 2020.

[82] H. Sibai, N. Mokhlesi, and S. Mitra, "Using symmetry transformations in equivariant dynamical systems for their safety verification," in *Automated Technology for Verification and Analysis*, Y.-F. Chen, C.-H. Cheng, and J. Esparza, Eds. Cham: Springer International Publishing, 2019, pp. 98–114.

[83] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager, "Adding symmetry reduction to uppaal," 2004.

[84] C. N. Ip and D. L. Dill, "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*, ser. CHDL '93. Amsterdam, The Netherlands, The Netherlands: North-Holland Publishing Co., 1993, pp. 97–111.

[85] S. Bak, Z. Huang, F. A. T. Abad, and M. Caccamo, "Safety and progress for distributed cyber-physical systems with unreliable communication," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 4, Sep. 2015. [Online]. Available: https://doi.org/10.1145/2739046

[86] M. Bujorianu and J.-P. Katoen, "Symmetry reduction for stochastic hybrid systems," in *2008 47th IEEE Conference on Decision and Control : CDC ; Cancun, Mexico, 9 - 11 December 2008. - T. 1.* Piscataway, NJ: IEEE, 2008, nebent.: Proceedings of the 47th IEEE Conference on Decision and Control. [Online]. Available: https://publications.rwth-aachen.de/record/100535 pp. 233–238.

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

[88] M. Z. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking," in *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, 2006, pp. 234–248.

[89] L. R. Antuña, D. Araiza-Illan, S. Campos, and K. Eder, "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*, 2015, pp. 26–37.

[90] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," in *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, 1993, pp. 463–478.

[91] E. M. Clarke and S. Jha, "Symmetry and induction in model checking," in *Computer Science Today: Recent Trends and Developments*, 1995, pp. 455–470.

[92] S. Jacobs and R. Bloem, "Parameterized synthesis," *Logical Methods in Computer Science [electronic only]*, vol. 10, 01 2014.

[93] M. Mann and C. Barrett, "Partial order reduction for deep bug finding in synchronous hardware," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 367–386.

[94] M. Pandey and R. E. Bryant, "Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 7, pp. 918–935, 1999.

[95] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetries to speed up SAT-based Boolean matching for logic synthesis of FPGAs," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1751–1760, 2008. [Online]. Available: https://doi.org/10.1109/TCAD.2008.2003272

[96] J. Maidens and M. Arcak, "Exploiting symmetry for discrete-time reachability computations," *IEEE Control Systems Letters*, vol. 2, no. 2, pp. 213–217, 2018.

[97] A. Majumdar and R. Tedrake, "Funnel libraries for real-time robust feedback motion planning," *The International Journal of Robotics Research*, vol. 36, no. 8, pp. 947–982, 2017. [Online]. Available: https://doi.org/10.1177/0278364917712421

[98] H. Sibai and S. Mitra, "Symmetry abstractions for hybrid systems and their applications," 2020.

[99] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, Sep. 2004, pp. 2149–2154 vol.3.

[100] Nvidia, "Ros bridge¶." [Online]. Available: https://docs.nvidia.com/isaac/isaac/packages/ros_bridge/doc/ros_bridge.html

[101] N. V. Patel, "NASA's next lunar rover will run open-source software," Apr 2021. [Online]. Available: https://www.technologyreview.com/2021/04/12/1022420/{NASA}-lunar-rover-viper-open-source-software

[102] ApolloAuto, "Apolloauto/apollo: An open autonomous driving platform." [Online]. Available: https://github.com/ApolloAuto/apollo

[103] Blue Robotics, "Bluerov2," *Datasheet, June*, 2016.

[104] E. Zapridou, E. Bartocci, and P. Katsaros, "Runtime verification of autonomous driving systems in CARLA," in *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, ser. Lecture Notes in Computer Science, J. Deshmukh and D. Nickovic, Eds., vol. 12399. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-60508-7_9 pp. 172–183.

[105] S. Jaksic, E. Bartocci, R. Grosu, and D. Nickovic, "An algebraic framework for runtime verification," 2018.

[106] H.-D. Tran, L. V. Nguyen, P. Musau, W. Xiang, and T. T. Johnson, "Decentralized real-time safety verification for distributed cyber-physical systems," in *Formal Techniques for Distributed Objects, Components, and Systems (FORTE'19)*, J. A. Pérez and N. Yoshida, Eds. Cham: Springer International Publishing, June 2019, pp. 261–277.

[107] K. Y. Rozier and J. Schumann, "R2U2: Tool overview," in *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, ser. Kalpa Publications in Computing, G. Reger and K. Havelund, Eds., vol. 3. EasyChair, 2017. [Online]. Available: https://easychair.org/publications/paper/Vncw pp. 138–156.

[108] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications," pp. 135–175, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-75632-5_5

[109] X. Zheng, C. Julien, R. Podorozhny, and F. Cassez, "Braceassertion: Runtime verification of cyber-physical systems," in *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*, 2015, pp. 298–306.

[110] S. Kang, I. Chun, and H.-S. Kim, "Design and implementation of runtime verification framework for cyber-physical production systems," *Journal of Engineering*, vol. 2019, p. 2875236, 2019. [Online]. Available: https://doi.org/10.1155/2019/2875236

[111] A. Majumdar and R. Tedrake, "Funnel libraries for real-time robust feedback motion planning," 2017.

[112] M. Khaled and M. Zamani, "Cloud-ready acceleration of formal method techniques for cyber–physical systems," *IEEE Design Test*, vol. 38, no. 5, pp. 25–34, 2021.

[113] M. Pourmehrab, L. Elefteriadou, and S. Ranka, "Smart intersection control algorithms for automated vehicles," in *2017 Tenth International Conference on Contemporary Computing (IC3)*. IEEE, 2017, pp. 1–6.

[114] O. Barzilai, N. Voloch, A. Hasgall, O. L. Steiner, and N. Ahituv, "Traffic control in a smart intersection by an algorithm with social priorities," *Contemporary Engineering Sciences*, vol. 11, no. 31, pp. 1499–1511, 2018.

[115] T. Niels, N. Mitrovic, K. Bogenberger, A. Stevanovic, and R. L. Bertini, "Smart intersection management for connected and automated vehicles and pedestrians," in *2019 6th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*. IEEE, 2019, pp. 1–10.

[116] S. Bharadwaj, T. Wongpiromsarn, N. Neogi, J. Muffoletto, and U. Topcu, "Minimum-violation traffic management for urban air mobility," in *NASA Formal Methods*, A. Dutle, M. M. Moscato, L. Titolo, C. A. Muñoz, and I. Perez, Eds. Cham: Springer International Publishing, 2021, pp. 37–52.

[117] C. Hsieh, H. Sibai, H. Taylor, and S. Mitra, "Unmanned air-traffic management (UTM): Formalization, a prototype implementation, and performance evaluation," *ArXiv*, vol. abs/2009.04655, 2020.

[118] S. Mitra, *Verifying Cyber-Physical Systems a Path to Safe Autonomy*. The MIT Press, 2021.

[119] H. Ledoux, K. Arroyo Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis, "CityJSON: A compact and easy-to-use encoding of the CityGML data model," *Open Geospatial Data, Software and Standards*, vol. 4, no. 1, p. 4, Jun 2019. [Online]. Available: https://doi.org/10.1186/s40965-019-0064-0

[120] B. Paden, M. Cap, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," 2016.

[121] M. Althoff, M. Koschi, and S. Manzinger, "Commonroad: Composable benchmarks for motion planning on roads," in *Proc. of the IEEE Intelligent Vehicles Symposium*, 2017.

[122] M. O'Kelly, H. Zheng, D. Karthik, and R. Mangharam, "F1TENTH: An open-source evaluation environment for continuous control and reinforcement learning," in *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, ser. Proceedings of Machine Learning Research, H. J. Escalante and R. Hadsell, Eds., vol. 123. PMLR, 08–14 Dec 2020. [Online]. Available: https://proceedings.mlr.press/v123/o-kelly20a.html pp. 77–89.

[123] L. Eller, T. Guérin, B. Huang, G. Warren, S. Yang, J. Roy, and S. Tellex, "Advanced autonomy on a low-cost educational drone platform," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 1032–1039.