ACCELERATING VERIFICATION OF CYBER-PHYSICAL SYSTEMS USING SYMMETRY

BY

HUSSEIN SIBAI

DISSERTATION

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

Urbana, Illinois

Doctoral Committee:

Professor Sayan Mitra, Chair
Professor Daniel Liberzon
Professor David Forsyth
Professor Geir Dullerud
Assistant Professor Sasa Misailovic

Abstract

Autonomous systems are increasingly being deployed in safety-critical applications such as transportation and medicine. Numerous approaches to analyze their safety have been considered including testing, falsification, and formal verification. The major challenge for all of these approaches is scalability to large and complex models. To address this challenge, we propose to use the symmetry naturally present in the dynamics of many of these systems.

Reachability-based safety analysis simulates the dynamical models of the autonomous systems, such as differential equations or hybrid automata, and checks if any of their reachable states is unsafe. Symmetries in dynamical systems are maps that transform any of their trajectories to other trajectories. In this thesis, we show how to use known symmetries of autonomous systems to cache their reachable states and abstract their dynamical models to accelerate their safety analysis.

The main contributions of this thesis are as follows: 1. Augmenting a state-of-the-art data-driven safety verification algorithm with a cache to reuse computed sets of reachable states. The proposed algorithm uses symmetries of the model under verification to increase the cache hit rate. 2. Augmenting traditional hybrid automata safety verification algorithms with a cache to reuse computed sets of reachable states. The proposed algorithm uses symmetries to share computed reachable sets between different modes and automata being verified. 3. Abstracting hybrid automata by combining modes with symmetric dynamics in the same abstract modes. 4. Designing a symmetry-based counter-example guided abstraction-refinement (CEGAR) algorithm for hybrid automata with symmetric continuous dynamics to accelerate their safety verification. 5. Finally, designing an efficient testing algorithm for autonomous systems that uses a cache to share symmetric trajectories among the test cases of a test suite, avoiding repetition of high-fidelity simulations.

The algorithmic contributions of this thesis come with theoretical guarantees that ensure their soundness and completeness. The algorithms presented build on top of state-of-the-art reachability analysis and verification algorithms. They

accelerate their computations, without affecting their soundness and completeness guarantees. Finally, we present software implementations and empirical analyses of the different algorithms presented, showing up to orders of magnitude speedup in verification and testing time of different dynamical models including a car, fixed-wing aircraft, a neural network-controlled quadrotor, and a Gazebo-based Hector quadrotor.

To my parents, Mohamad Ali and Rima, for their love and support.

Acknowledgments

First, I sincerely thank my advisor Sayan Mitra for all his guidance and support throughout the past six years. Sayan's mentorship has shaped my personality as a researcher. He has provided all the technical and personal advice, the environment, and the resources needed for a first-class well-rounded Ph.D. experience. He has provided me with the freedom of research exploration that is needed for innovation as much as with the constructive criticism and advice that are required for growth and progress. He created an environment that fostered my collaborations as well as my independent thinking. He valued my personal wellbeing, celebrated my successes, helped in the pitfalls, and encouraged when in doubt. For these, and many others, I will always be grateful for Sayan.

I deeply thank my committee members: Daniel Liberzon, David Forsyth, Geir Dullerud, and Sasa Misailovic. Their feedback on this thesis and our collaborations on different projects have widened my perspective and sharpened my skills. I will always cherish their friendship and mentorship. I also collaborated with and learned from several other faculty members, from whom I want to particularly thank Nikita Borisov, Matus Telgarsky, and Ruoyu Sun.

I have been part of an amazing research group with which I was lucky to share this journey. I thank Chuchu Fan, Ritwika Ghosh, Chiao Hsieh, Nicole Chan, Navid Mokhlesi, Mathew Potok, Yangge Li, Kristina Miller, Sung Woo, Minghao Jiang, and Dawei Sun. I particularly thank Chuchu for being a great friend and a role model, Ritwika and Chiao for our valuable discussions about research and life, and Nicole for all her support.

My stay in Urbana-Champaign was an unmatched home-like experience because of a large close-knit family of friends: Lina ElFarouk, Nabil Ramlawi, Ali Kanj, Mohammad Nouriddine, Ahmed Fawaz, Amir Ayman, Hania Taha, Hussein Hazimeh, Ali Kotb, Pamela Tannous, Hussein Darir, Hassan Dbouk, Pual Gharzouzi, Rabel Rizkalla, Tarek Gebrael, Tarek Sakakini, Farah Hariri, and many others from the Lebanese and Arab community. For them, I will always be grateful. I want to

particularly thank Lina for being an amazing friend, for her constant support, for spreading positive energy, for the rehearsals and proofreads of my presentations and papers, and for being there. I also want to thank Nabil for all the discussions on life, politics, and research, for his amazing food, and all his help; Ali for his great insight, his informative discussions, and him working with me during our trips; Mohammad and Ahmed for being my big brothers away from home and making my move to Illinois smooth and for the food during Ramadan; Amir and Hania for all the trips where Hania is planning the next trip and Amir is doing the actual work; Hussein for all the nights we spent the first year working together; Ali for spreading joy; Hussein for our random meetings discussing research and future plans; Hassan, Paul, and Rabel for being amazing roommates.

I thank my undergraduate advisors Louay Bazzi and Fadi Zaraket for all their advice and support that lead me to joining graduate school. I thank my competitive programming coach at AUB Nagi Nahas for his unmatched guidance throughout my undergraduate years and my application process to graduate schools.

Words cannot express my gratitude to my parents Mohamad Ali and Rima. I don't know how to thank a lifetime of sacrifices. Foremost, after the grace of God, they are the reason I reached where I am today. I thank my brothers Ali ElHadi, Ahmad, and Jamal, and my sister-in-law Zahraa for their friendship, for their love, and all their support. I thank my beautiful niece Fatima and nephew Mohamad Ali for warming my heart with their photos and videos.

Finally, to many others who paved the way for me to be here, who made my stay joyful, and who enriched my experience during these years, thank you.

Contents

Chapter 1

Introduction: Scalability Challenges and the Potential of Symmetry

Safety as a fundamental concern Motivated by the tremendous advances in hardware, machine learning, and robotics, autonomous systems are being increasingly deployed, even in safety-critical applications. For example, more than two million drones are expected to be deployed in the United States airspace by 2024 [1]. Compared to more established fields such as hardware and software engineering, rigorous and efficient methods for the testing, verification, and validation of autonomous systems are in their infancy. Without such methods, guaranteeing acceptable levels of human and property safety, meeting safety laws and regulations, and gaining public trust would not be feasible.

Scalability as a major challenge in safety assessment Several safety analysis approaches for autonomous systems have been proposed and used in the past, including high-fidelity simulations [2, 3, 4, 5, 6], real-life or field testing [7, 8, 9], falsification [10, 11, 12, 13], and formal verification [14, 15, 16, 17, 18, 19, 20, 21]. Non-formal approaches, although useful to ensure proper behavior of systems in normal conditions or selected scenarios, fail to provide formal guarantees on safety. Once more scenarios have to be tested or higher-fidelity simulations are needed, non-formal approaches require significant computation, human effort, and time. This problem is only further exacerbated in formal methods. Although the latter provide a more comprehensive coverage of operating conditions than non-formal ones, they are far from being scalable to models at the sizes of systems currently deployed.

Structure as a scalability booster Despite the potential complexity of autonomous systems, they usually posses structural properties that aid their design and analysis. Decomposability to smaller interconnected subsystems, bounded sensitivity, stability, and linearity or nonlinearity, are examples of structural properties. Such properties constrain the behaviors of the systems, and consequently, aid their

analysis [22, 23, 16, 24, 21]. Such properties can be the key for efficient testing, verification, and synthesis, of autonomous systems. In this thesis, we investigate the benefits of utilizing symmetry towards accelerating safety verification and testing. Symmetry is an essential structural property of many natural dynamical systems.

1.1 Background: symmetry and verification

Symmetry utilization has been broadly used in variety of domains.

Symmetry in dynamical systems Symmetry in a dynamical system $\dot{x} = f(x,p)$, with parameter p, is a map γ that transforms solutions or *trajectories* of the system to other trajectories. For example, consider a trajectory $\xi_0(t) = \xi(x_0,p,t)$ for $t \in \mathbb{R}^+$, of a vehicle, starting from x_0 and following waypoint p. When ξ_0 is *shifted* by a, the result $\gamma_a(\xi_0)$ is just the trajectory of the vehicle starting from $\gamma_a(x_0)$ following p', $\xi_0' = \xi(\gamma_a(x_0),p',t)$. Here p' is p shifted by a. That is, the symmetry γ_a relates different trajectories of the system.

> "*It is only slightly overstating the case to say that physics is the study of symmetry* [25]."
>
> (Philip Anderson, Nobel laureate, 1977)

Symmetry in science and engineering Symmetry plays an important role in analysis of physical processes by summarizing the laws of nature independent of specific dynamics [26, 27] and deriving conservation laws [28] using Noether's theorem [29]. Symmetry related concepts have been used to explain and suppress unstable oscillations in feedback connected systems [30], show existence of passive gaits under changing ground slopes [31], design control inputs for synchronization of neural networks [32, 33], and decrease the sample complexity of deep learning [34, 35, 36, 37, 38, 39], computer vision [40, 41], reinforcement learning [42, 43], dynamics prediction [44], and inference attacks on neural networks [45]. Metamorphic testing is a paradigm of software and autonomous systems testing based on symmetry [46, 47, 48, 49, 50, 51]. In metamorphic testing, the system is executed from symmetric inputs and its outputs are checked if they satisfy known symmetric properties of the system. Metamorphic testing is usually

followed to solve the *oracle* problem: having to know the correct output for each test. This problem is especially prevalent in testing autonomous systems. It is circumvented by comparing the outputs of the system given symmetric inputs.

Symmetry in model checking Symmetry has also played an important role in handling the state space explosion in model checking computational processes. The idea of *symmetry reduction* is to reduce the state space by considering two global states to be equivalent (*bisimilar*), if the states are identical, including permuting the identities of participating components [52, 53]. Equivalently, symmetry can reduce the number of behaviors to be explored for verification when one behavior can be seen as a permutation, or a more general transformation, of another. Symmetry reduction was incorporated in early explicit state model checkers like Murϕ [54] and Uppaal [55]. Symmetry has been exploited in a number of contexts to accelerate verification such as probabilistic models [56, 57], automata [52, 53], distributed architectures [58], and hardware [59, 60]. Translating the idea of symmetry utilization into improved performance of model checking has proven to be both fruitful and nontrivial as witnessed by the sustained attention that this area has received over the past three decades.

Reachability analysis as a cornerstone for verification The set of states that can be reached by a dynamical system over a given time horizon is called the reachable set. Computing or approximating the reachable set is known as *reachability analysis*. As the cornerstone for safety verification of dynamical systems, and notwithstanding the undecidability results [61] and the curse of dimensionality plaguing the early algorithms, over the last decade, the formal methods research community has pushed the reach of reachability analysis tools from small, academic, linear models to realistic nonlinear and hybrid, continuous and discrete, models [17, 62, 63, 64, 65, 66, 67, 16, 68, 69, 70], linear models with millions of dimensions [67, 71], and deep neural networks (DNN) and DNN-controlled systems (NNCS) [72, 73, 19]. Reachability analysis has delivered automatic analysis of automotive, aerospace, and medical applications [74, 75, 76, 69, 64, 17]. Despite these successes, computing reachable sets is still computationally expensive as it requires non-trivial optimization problems and integrating non-linear functions [16, 68, 69, 17, 70]. Compared with computation, transforming reachable sets is much cheaper.

Previous applications of symmetry in reachability analysis Symmetry was used to accelerate the safety verification of dynamical systems achieving promising results, in some cases by orders of magnitude speedups [77]. In [77], Maidens et al. used the Cartan moving frame method for symmetric nonlinear discrete-time dynamical systems to move from absolute representation of states to relative ones. That resulted in orders of magnitude speed up in verifying that two Dubin's vehicles in a given scenario would not collide. A similar method was used by Irfan et al. in [78] to verify closed-loop safety properties of the Airborne Collision Avoidance System for small unmanned aircraft (ACAS sXu). Bak et al. [79] suggested using symmetry maps, called *reachability reduction transformations*, to transform reachsets to symmetric reachsets for continuous dynamical systems modeling non-interacting vehicles. Majumdar et al. [80] proposed a safe motion planning algorithm that computes a family of reachsets offline and composes them online using symmetry. Bujorianu et al. [81] 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.

1.2 The thesis

In this thesis, we propose to use symmetries of dynamical systems to accelerate their testing, reachability analysis, and safety verification. It follows two approaches: *caching* and *abstraction*. In the first approach, it provides data structures and algorithms that use symmetry to efficiently cache reachsets and high-fidelity simulations. In the second approach, it provides abstraction-refinement algorithms that reduce hybrid automata by combining their symmetric states. The designed algorithms cover a wide variety of dynamical models including continuous-time dynamical systems, hybrid systems, multi-agent systems, and NN-controlled closed-loop systems. The algorithms are accompanied with theoretical guarantees on their soundness and completeness. This thesis presents three new Python tools that implement the contributed algorithms and use existing reachability analysis and verification tools as sub-routines. The thesis shows experimental results showing orders of magnitude speedup in testing and safety verification time of scenarios involving NN-controlled cars and quadrotors following predefined plans with hundreds of segments in environments with hundreds of obstacles. In the rest of the introduction, we discuss the different contributions in more depth.

1.2.1 Symmetry and caching in data-driven verification

One prominent approach for reachability analysis is based on generalizing individual behaviors or simulations to cover a whole set of behaviors. The idea was pioneered by Donze and Maler [82] and implemented in Breach [18] with sound generalization guarantees for linear models based on *sensitivity analysis*. Subsequently, the idea has been significantly extended to cover nonlinear, hybrid, and black-box models and it has been implemented in tools like C2E2 and DryVR [83, 65, 76, 69]. It has been denoted later by *data-driven reachability analysis*.

In data-driven reachability analysis, a single behavior ξ of the system from an initial state, is generalized to a *compact set* of *neighboring* behaviors that contains all the behaviors starting from a small neighborhood around the initial state of ξ. Thus, the computed neighboring set of behaviors always contains ξ and its size is determined by the algorithms for sensitivity analysis. This thesis pursue an additional type of generalization that uses *symmetry transforms* on the state space. Given a group Γ of operators on the state space, and a single behavior ξ, ξ can be generalized to $\gamma(\xi)$, for each $\gamma \in \Gamma$. Symmetry transformations can be applied to sets of behaviors symbolically. Not only can this type of generalization work in conjunction with sensitivity analysis, it captures structural properties of the system that make behaviors similar in a way that is not covered by sensitivity analysis.

In data-driven reachability analysis, the generalization of a behavior to its neighboring behaviors might contain ones that cannot be realized by the system. In that case, the computed reachable set over-approximates the actual one of the system. The cause for such over-approximation error is conservative sensitivity analysis. The latter becomes more accurate with smaller initial sets [16, 15].

In a safety verification problem, a set of possible initial states, a time bound, and an unsafe set of states are specified. The problem is to check if any behavior of the system starting from the specified initial set intersects the unsafe set within the time bound. When the problem is being solved using data-driven verification, and the computed reachable set intersects the unsafe set, but not the simulated behavior, the initial set is partitioned. Then, reachable sets are computed from each of the initial set parts for a more accurate reachability analysis.

In Chapter 3, and its published version [84], we present an algorithm that accelerates the state-of-the-art data-driven safety verification method, implemented in DryVR [15] and C2E2 [85], by augmenting it with a cache for reachable sets.

The cache is tree-shaped that stores the reachsets computed at each partition level of the initial set. Given symmetries of the system under verification, the algorithm transforms reachable sets stored in the cache for parts of the initial set to new reachable sets of the system starting from other parts of the initial set. In certain cases, the algorithm is able to use symmetries to reduce the initial set before reachability analysis, further saving the cache-access time. The algorithm is implemented on top of DryVR and evaluated . Experimental results for verifying the non-collision of two dubin's cars using the new algorithm show orders of magnitude savings in verification time.

1.2.2 Symmetry and caching in multi-agent systems

Cyber-physical systems (CPSs) consist of physical and digital components. The state of a CPS is a concatenation of a state that evolves continuously in time representing a physical process and a state that evolves discretely in time representing the digital system. Cyber-physical systems are essential parts of our everyday lives. Examples include vehicles, heating and cooling systems, and medical devices.

In this thesis, we investigate how symmetry principles could benefit the analysis of CPSs. Not surprisingly, the verification problem for CPS inherits the state space explosion problem. Autonomous CPS commonly work in multi-agent environments, e.g., a car in an urban setting—where even the number of scenarios to consider explodes combinatorially with the number of agents. This has been identified as an important challenge for testing and verification [86].

Cyber-physical systems are usually mathematically modeled using *hybrid automata* [87, 88]. A hybrid automaton represents the evolution of the physical component of a CPS using ordinary differential equations (ODEs) and its digital component using finite state machines (FSMs). The states of the FSM are called the *modes* and the states of the ODEs are called the *states* of the hybrid automaton. In each mode, the state evolves continuously in time according to an ODE specific to that mode. The automaton transition from a mode to another over an edge in the FSM when the state satisfies a condition called the *guard* of that edge. After a mode transition, the state might discretely transition to a new value according to a function called the *reset* of that edge.

In reachability analysis-based safety verification of hybrid automata, the reachable sets of the modes are computed sequentially starting from the initial mode

and ending with the last mode visited with the given time bound [17, 85, 15]. In Chapter 4, and its published version [89], we present an algorithm that accelerates reachability analysis of hybrid automata by augmenting with a cache of reachable sets. The algorithm utilizes symmetries to transform reachable sets of a certain mode stored in the cache to new reachable sets starting from symmetric initial sets and following symmetric modes. The algorithm utilizes the concept of virtual system to efficiently cache reachable sets. The virtual system is an automaton with a single representative mode. The behaviors of a virtual system can be transformed using symmetries to obtains behaviors for any other mode in the original automaton.

For example, an aircraft following a predefined sequence of waypoints can be modeled as a hybrid automaton, where the waypoints are the modes, and the state represents the position, speed, and heading angle of the aircraft. The reachable set of the aircraft following one waypoint can be rotated and translated to get its reachable set following a rotated and translated waypoint.

In a non-interacting multi-agent scenario, each agent can be modeled as a hybrid automaton. If the agents have the same continuous dynamics, the algorithm of Chapter 4 uses the concept of virtual system to transform reachable sets of one agents following a certain mode to a another agent following a symmetric mode starting from a symmetric initial set.

The algorithm is implemented in a new Python tool named CacheReach. It uses existing reachability analysis tools such as DryVR [15] and Flow* [17] as subroutines to compute reachable sets for different modes. The algorithm shows 64% speedup in verification time on scenarios with linear and nonlinear vehicles following different sequences of waypoints.

1.2.3 CEGAR algorithm for verifying hybrid automata using symmetry

The computation time for verifying a hybrid automaton depends heavily on the number of its modes and the complexity of its continuous and discrete dynamics [16, 17, 15]. Abstraction-refinement verification algorithms construct simpler, or *abstract*, automata for the original, or *concrete*, ones, before verification [90, 91, 22, 92]. There are different approaches to construct simpler automata including: over-approximating non-linear ODEs with linear differential inclusions [92], relaxing

the constraints on transitions in the hybrid automata [91], and combining different modes in a representative mode with continuous dynamics specified as a linear differential inclusion of all the ODEs of the concrete modes [22]. They have been proven to scale to larger and more complex automata beyond the reach of traditional verification algorithms. The behaviors, or *executions*, of a concrete automaton can be mapped to corresponding executions of its abstract one. Such a relation from concrete to abstract executions is called a *forward simulation relation* [93, 94, 95]. However, some executions of the abstract automaton might not correspond to executions of the concrete one. Consequently, if the abstract automaton is verified to be safe, then the concrete automaton is safe, but not necessarily the other way around.

When a safety verification algorithm returns *unsafe*, it usually provides an execution that violates the safety property, i.e., enters the unsafe set. Such an execution is called a *counter-example*. When a counter-example for the abstract automaton does not correspond to a counter-example for the concrete one, we call it a *spurious* counter-example. Otherwise, we call it a *valid* one. Counter-example guided abstraction-refinement (CEGAR) algorithms use a spurious counter-example as a guide to refine the abstract automaton to capture more accurately the dynamics of the concrete one. In general, a CEGAR algorithm consists of five steps: (1) construct an abstract automaton for the concrete automaton need to be verified, (2) verify the abstract automaton, (3) if *safe*, return *safe*, if *unsafe*, check if the counter-example is spurious, (4) refine the abstract automaton to a more accurate one that eliminates the spurious counter-example, and (5) repeat, if necessary.

In Chapter 5, we present a novel CEGAR algorithm based on symmetry. The abstract automaton is constructed by representing every class of modes with symmetric continuous dynamics in the concrete automaton in a single abstract mode. The edges, guards, and resets, are then grouped accordingly. The unsafe set specified for the concrete automaton is transformed as well using symmetry to an unsafe set for the abstract one. The CEGAR algorithm then uses any of the existing safety verification algorithms and tools to verify the abstract automaton. The CEGAR algorithm checks if a counter-example of the abstract automaton, returned by the existing verification subroutine, is spurious, using a depth-first-search (DFS) algorithm over the concrete automaton. If it turned out to not be spurious, the algorithm returns a set of concrete counter-examples. Otherwise, the CEGAR algorithm calls a novel refinement algorithm to eliminate the spurious counter-example. The refinement algorithm selectively and recursively splits abstract modes to more

modes representing fewer concrete modes. It ensures that the resulting abstract automaton after refinement does not have the input spurious counter-example as a valid execution that intersects the unsafe set.

In Chapter 6, we present a novel tool SceneChecker that implements an abstraction-refinement safety verification algorithm based on symmetry. It uses the same abstraction as the CEGAR algorithm described in the previous paragraphs. However, it does not check for spurious counter-examples and its refinement algorithm does not ensure that a spurious counter-example is eliminated in a single call. SceneChecker uses existing reachability analysis tools as subroutines to computer per-mode reachable sets. It checks for fixed-point when computing the reachable set of the abstract automaton to avoid repeating computations. It does not cache reachable sets, in contrast to the work presented in Chapters 3 and 4, and thus avoiding cache space and cache access time overhead. Experimental results show an average of 14$\times$ speedup in verification time of scenarios with NN-controlled quadrotors and cars navigating predefined plans with hundreds of waypoints in 2D and 3D environments with hundreds of polytopic obstacles.

1.2.4 Symmetry for faster high-fidelity testing

Testing has been the default and the more scalable alternative, compared to formal verification, in analyzing the safety of autonomous systems. Directly testing the actual system, e.g., an autonomous car, in the physical environment is expensive and dangerous. For instance, billions of miles are estimated to be needed to be driven by autonomous car to ensure the acceptable level of safety of less than one catastrophic failure per hour [96]. Instead, simulation engines have been developed to efficiently and safely test autonomous systems under design [2, 6, 3]. Different simulation engines have different fidelity levels. The higher the fidelity of the simulation engine, the more its simulations resemble the real-life behaviors of the system. The degree of accuracy of simulations is a topic of recent research interest, under the name of Sim2Real [8]. On the other hand, the higher the fidelity of the simulation engine, the less scalable and more computationally expensive it is. Even in simulation, driving billions of miles after each update of the system is prohibitive. By 2020, Waymo, a leading autonomous cars company with enormous computational resources, was able to simulate a total of 15 billion miles, and drive 20 million real-world miles [7].

A test suite specifies a set of tests that are executed to check if the system satisfies a given property, such as safety. For autonomous systems, a test would specify the number of agents in the scenario, the initial states of the different agents, the plans that the agents will follow, and the environmental parameters and conditions. Although different tests in the test suite might aim to cover different scenarios under which the autonomous systems might operate, the corresponding behaviors of the systems in these tests might be similar and symmetric. Further, regressive test suites, those that are executed after each system update or code change, might share the same behaviors with previous runs, especially when not all components of the system are altered. A behavior of the system in one scenario generated using a high-fidelity simulator, can be transformed to obtain the behavior of the system in a symmetric scenario for a different test. Consequently, parts of the tests in a test suite might not need a high-fidelity simulator to execute, and can reuse previously simulated behaviors instead.

For example, the Airborne Collision Avoidance System for smaller Unmanned Aircraft Systems (ACAS sXu), is a software system that has been developed by the Federal Aviation Administration (FAA) to provide Detection and Avoid (DAA) capabilities for small unmanned aircraft systems (UASs) [97]. The software system would be deployed on the UASs or on the cloud. Given the relative positions, relative heading angles, and speeds of the different aircraft within close proximity of each other, ACAS sXu provides advisories, such as Weak Left, Strong right, and climb at rate 300 ft/minute, for the agents to follow to avoid collision. It is essential to thoroughly test ACAS sXu in closed-loop with any aircraft dynamics it will deployed on. Moreover, this testing should be repeated whenever the ACAS sXu software is updated. A test suite in this context would consist of test cases specifying the initial states of different aircraft in an encounter, and the reference trajectories they are following. The ACAS sXu system runs at 1 Hz providing advisories, if necessary, every second. An advisory for an aircraft would adjust the reference trajectory it is following. In case the aircraft are quadrotors, their dynamics during the one-second intervals depend only the desired velocity vectors specified by their reference trajectories, transformed to their body coordinates. The desired velocity vector in body coordinates will determine the needed acceleration, and consequently the needed thrust and rotor angular velocities, for a particular quadrotor. Different quadrotors in the same or different test cases might, approximately, share the same one-second behaviors in body coordinates. We call such one-second behaviors *trajectory segments*. Similarly, the

output of ACAS sXu only depends on parts of the states of the aircraft represented in relative coordinates, independent of the world coordinates. Moreover, the ACAS sXu outputs are expected to be symmetric for symmetric inputs, e.g. advises Weak Left when on a reflected state it advised Weak Right. These symmetries in the software system complement their counterparts in the dynamics.

In Chapter 7, we present an algorithm that uses symmetry in the dynamics of autonomous systems to efficiently cache their high-fidelity simulated behaviors to accelerate multi-agent testing. We consider systems where agents communicate periodically, similar to the ACAS system discussed in the previous paragraph. We present two versions of the algorithm: symTest and absSymTest. Given a test suite for multi-agent testing, the algorithm generates the trajectories of the agents. The first version, symTest, executes the tests of the test suite sequentially while caching any newly simulated representative trajectory segment of any agent. Before launching the high-fidelity simulator to execute a particular test case, symTest tries to retrieve their representative trajectory segments from the cache and transform them using symmetry to the desired ones. The agents have the same dynamics and thus share the same cache. If at a certain time instant, the trajectory segment for the next period of any of the agents in the test case is not stored in the cache, the algorithm launches the high fidelity simulator to generate the trajectories of the agents for the rest of the test, and saves the resulting trajectory segments in the cache. The second version of the algorithm, absSymTest, executes all the tests of the test suite in parallel, sequentially in time, using a single simulation of a single *representative* agent. At each period, the representative agent simulates a trajectory segment symmetric to the one needed by one of the agents in one of the test cases, and caches the result. A preliminary Python implementation of symTest to accelerate closed-loop testing of ACAS sXu deployed on Hector Quadrotors [98] simulated using Gazebo [2] and the Robot Operating System (ROS) [99], is shown. Also, we present a Python implementation of absSymTest to accelerate closed-loop testing of NN-version of ACAS Xu deployed on six-dimensional NN-controlled quadrotors. Finally, preliminary experimental results showing savings in trajectories simulated and computation time using different symmetries are shown and discussed.

1.2.5 Organization of the thesis

In summary, the thesis is organized as follows:

1. in Chapter 2, we present the definitions of two dynamical models, symmetry of dynamical systems, reachability analysis, and data-driven safety verification. We also present example systems and symmetries that we use in the rest of the thesis,

2. in Chapter 3, we present an algorithm that augments an existing data-driven safety verification of continuous-time dynamical systems algorithm with symmetry-based caching and abstraction of reachsets,

3. in Chapter 4, we extend the contributions of Chapter 3 to multi-agent hybrid systems by augmenting an existing verification algorithm with symmetry-based caching that allows sharing computed reachsets between multiple modes and agents,

4. in Chapter 5, we present an alternative approach for symmetry-utilization in the verification of hybrid systems to that of Chapter 4 based on abstractions. We present an algorithm that uses symmetries to simplify the model before verification. The algorithm iteratively verifies and refines the simplified, or abstract, model to ensure accurate results. It is a symmetry-based CEGAR algorithm to accelerate verification of hybrid systems,

5. in Chapter 6, we present SceneChecker, a Python tool that implements a simplified version of the algorithm Chapter 5 for efficient scenario verification,

6. in Chapter 7, we change course from formal verification and address the scalability challenge of high-fidelity simulations of autonomous systems. We present a symmetry-based efficient testing algorithm for multi-agent autonomous systems, following the same ideas of caching and abstraction of the previous chapters, and

7. finally, in Chapter 8, we conclude and suggest several promising directions to build on the results of this thesis.

Chapter 2

Preliminaries: Systems, Symmetries, and Safety

In this chapter, we describe the notations, definitions, theorems, and algorithms that will be used throughout the dissertation. Experienced readers may skip to the next chapter after a quick glance at the examples in Section 2.2.3 and the verification algorithm in Section 2.4. We start with few notations for lists, sets, and functions in Section 2.1. We then describe mathematical frameworks for modeling hybrid and dynamical systems in Section 2.2. In the same section, we present the formal definitions of symmetries of dynamical systems along with several examples. In section 2.3, we define the bounded safety verification problem for dynamical systems and relate it to the notion of reachability analysis. In Section 2.4, we discuss data-driven verification, a state-of-the-art method for reachability analysis and bounded safety verification. Finally, in Section 2.5, we discuss the different metrics according to which we evaluate the proposed algorithms in the dissertation.

2.1 Notations for sets, vectors, and functions

We denote by $\mathbb{N}$, $\mathbb{R}$, and $\mathbb{R}^+$ the sets of natural numbers, real numbers and non-negative reals, respectively. Given a positive integer N, we denote by $[N]$ the set of integers $\{1,\ldots,N\}$. Given a finite set S, we denote its cardinality by $|S|$. Given two sets S_1 and $S_2 \subset \mathbb{R}^n$, $S_1 \oplus S_2 = \{s_1 + s_2 \mid s_1 \in S_1 \text{ and } s_2 \in S_2\}$ is the Minkowski sum of S_1 and S_2. Given a vector $v \in \mathbb{R}^n$, we denote by v_i the i^{th} component of v. Given another vector $u \in \mathbb{R}^m$, we define $[v,u]$ to be the vector of length $n+m$ that results from appending u to v. We denote $diag(v)$ to be the diagonal matrix with diagonal v.

We define an n-dimensional hyper-rectangle by a 2d-array specifying its bottom-left and upper-right corners. For any $\delta > 0$, $grid(H,\delta)$, is a collection of 2δ-separated points along axis parallel planes such that the δ-balls around these points cover H. Given $\varepsilon \in (\mathbb{R}^+)^n$, $B(v,\varepsilon)$ is the n-dimensional hyper-rectangle centered

at v with i^{th} dimension sides having length $\varepsilon[i]$, for all $i \in [n]$.

Given a set of indices $L \subseteq [n]$, the projection of vector v on the set of dimensions L is denoted by $v[L]$. Similarly, the projection of a hyperrectangle H on the set of dimensions L is denoted by $H[L]$. We denote by the set $S\downarrow_L = \{v' : \exists v \in S, \forall\, i \in L, v_i = v'_i \text{ and } v'_i = 0, \text{otherwise}\}$.

A continuous function $\beta : \mathbb{R}^+ \to \mathbb{R}^+$ is said to be a class-$\mathscr{K}$ function if it is strictly increasing and $\beta(0) = 0$. Given a function $\gamma : \mathbb{R}^n \to \mathbb{R}^n$ and a set $S \subseteq \mathbb{R}^n$, as usual, we lift the function to subsets of $\mathbb{R}^n$, and define $\gamma(S) = \{\gamma(x) \mid x \in S\}$. We also lift it to vectors of $\mathbb{R}^n$ and define $\gamma(v) = [\gamma(v_1), \gamma(v_2), \ldots, \gamma(v_k)]$, for any $v \in \mathbb{R}^{n\times k}$. Similarly, we lift it to sets of the form $S \in 2^{\mathbb{R}^k} \times \mathbb{R}^+$ and define $\gamma(S) = \{(\gamma(X), t) \mid (X,t) \in S\}$. Fix any time interval $[t_i, t_f]$ and any set S. For any function $\xi : S \times [t_i, t_f] \to S$ and $s \in S$, we define $\xi(s,\cdot) : [t_i, t_f] \to S$ to be the function of time that results from setting the first argument of ξ to s. We define $\arctan_2(y,x)$ to be the phase of the complex number $x + jy$.

The length of a finite sequence *seq* is denoted by *seq.len* and its elements between i and j, inclusive, by $seq[i : j]$.

2.2 Models and symmetries of dynamical systems

In this dissertation, we consider two mathematical models for dynamical systems: continuous-time and hybrid systems. Each of the following chapters will tackle systems of one of these types of models.

2.2.1 Continuous-time dynamical systems

Given a state space $X \subseteq \mathbb{R}^n$ and mode space $\mathscr{P} \subseteq \mathbb{R}^m$, a *dynamical system* is defined by a function $f : X \times \mathscr{P} \to X$ that is locally Lipschitz in the first argument. We call f a *dynamic* function. The state and the mode of the system will evolve continuously and discretely in time, respectively. The mode will only change when f is defining the continuous dynamics of a hybrid automaton. See Section 2.2.4 for more details.

We define $\xi : X \times \mathscr{P} \times [0,\infty) \to X$ to be the function that generates *trajectories* of the system. For any $x_0 \in X$ and $p \in \mathscr{P}$, $\xi(x_0, p, \cdot)$ is the trajectory that starts from x_0 and follows mode p. It should satisfy $\xi(x_0, p, 0) = x_0$ and that for all

$t \in [0, \infty)$,

$$\frac{d\xi}{dt}(x_0, p, t) = f(\xi(x_0, p, t), p). \tag{2.1}$$

We say that $\xi(x_0, p, t)$ is the state of the system at time t when it starts from x_0 in mode p. For any time-bounded trajectory ξ, i.e., defined over a finite interval in the third argument, $dur(\xi)$ is its last time point. The first and last state in such a trajectory are denoted by $\xi.fstate$ and $\xi.lstate$, respectively. Given an initial state $x_0 \in X$ and mode p, the trajectory $\xi(x_0, p, \cdot)$ is the unique solution of the ordinary differential equation (ODE) (2.1) since f is Lipschitz [100]. In this thesis, we mainly work with bounded-time solutions of ODEs. For any time bound T for which we want to verify system (2.1), we assume that all its trajectories exist for all $t \in [0, T]$. When p is *not* a constant, to emphasize its influence on f we will call the system *a parameterized dynamical system*. Otherwise, we call it *an autonomous dynamical system* and drop p from the definitions of the trajectories and dynamic functions.

2.2.2 Symmetry in dynamical systems

Symmetry takes a central place in analysis of dynamical systems [101]. The research line pertinent to our work develops the conditions under which one can get a solution of a continuous-time dynamical system by transforming another solution [101, 102, 103]. Our work in this dissertation uses these results and extends them to hybrid automata by defining symmetry-based abstractions in Chapter 5 and to control systems by defining symmetric controllers in Section A.2.

Symmetries of a dynamical system are modeled as groups of operators on the system's state space. First, recall the definition of a group.

Definition 2.1. *A group $(\Gamma, \circ)$ is a set Γ and an operator $\circ : \Gamma \times \Gamma \to \Gamma$ satisfying the following properties:*

- *Closure: For any $\gamma_1, \gamma_2 \in \Gamma$, $\gamma_1 \circ \gamma_2 \in \Gamma$,*
- *Associativity: For any γ_1, γ_2, and $\gamma_3 \in \Gamma$, $\gamma_1 \circ (\gamma_2 \circ \gamma_3) = (\gamma_1 \circ \gamma_2) \circ \gamma_3$,*
- *Identity: There is an element $e \in \Gamma$ such that $e \circ \gamma = \gamma \circ e = \gamma$, and*
- *Inverse: For every element $\gamma \in \Gamma$, there exists a unique element $\gamma^{-1} \in \Gamma$, called its inverse, such that $\gamma \circ \gamma^{-1} = \gamma^{-1} \circ \gamma = e$.*

We now define symmetry in dynamical systems.

Definition 2.2 (Definition 2 in [26]). *Let Γ be a group of maps acting on X. We say that $\gamma \in \Gamma$ is a symmetry of system (2.1) if for any solution $\xi(x_0, p, \cdot)$, $\gamma(\xi(x_0, p, \cdot))$ is a solution as well.*

Thus, if γ is a symmetry of (2.1), then a new solutions can be obtained by just applying γ to existing solutions. Herein lies the opportunity of exploiting symmetries in verification. As we shall see in Section 2.4, verification algorithms simulate the system from different initial states to over-approximate the set of states that the system might reach. Such simulations as well as the reachable set computations are expensive, while transforming simulations via application of γ is computationally inexpensive. In Chapter 3, we will discuss how symmetry can be used to transform simulations and reachable sets in verification algorithms.

How can we know that γ is a symmetry for (2.1)? It turns out that, a sufficient condition exists that can be checked syntactically without computing the solutions. This sufficient condition only requires us to check commutativity of γ with the dynamic function f. Systems that meet this criterion are called *equivariant*.

Definition 2.3 (Definition 3 in [26]). *Let Γ be a group of operators acting on $\mathbb{R}^n$. The dynamic function $f : X \times \mathscr{P} \to X$ is said to be Γ-equivariant if for any $\gamma \in \Gamma$, there exists $\rho : \mathscr{P} \to \mathscr{P}$ such that,*

$$\forall\, x \in X, \forall\, p \in \mathscr{P},\ \frac{\partial \gamma}{\partial x} f(x, p) = f(\gamma(x), \rho(p)). \tag{2.2}$$

The following theorem shows that for equivariant systems, solutions are symmetric.

Theorem 2.1 (part of Theorem 10 of [26], [27]). *If f is Γ-equivariant, then all maps in Γ are symmetries of (2.1). Moreover, for any $\gamma \in \Gamma$, map $\rho : \mathscr{P} \to \mathscr{P}$ that satisfies equation (2.2), $x_0 \in X$, and $p \in \mathscr{P}$, $\gamma(\xi(x_0, p, \cdot)) = \xi(\gamma(x_0), \rho(p), \cdot)$.*

Remark 2.1. *If γ in Theorem 2.1 is linear, i.e. $\gamma(x) = Ax$, then the condition in equation (2.2) for γ to be a symmetry becomes $\gamma(f(x, p)) = f(\gamma(x), \rho(p))$.*

For black-box models, Definition 2.2 can be checked using sampling methods.

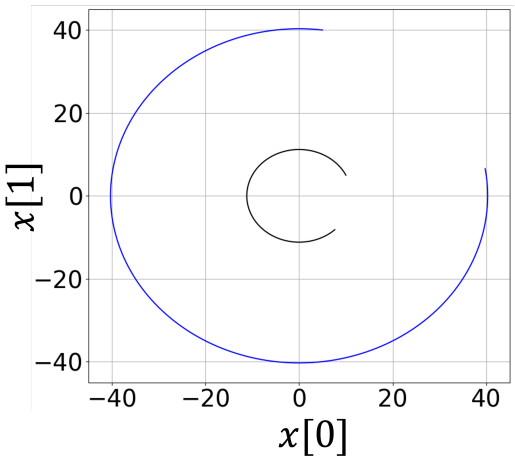


Figure 2.1: Simple oscillator trajectories. Black: trajectory starts from $[10,5]$ and runs for 5 seconds. Blue trajectory of the same oscillator starts from $[5,40]$ and is generated by transforming the black trajectory by the symmetry transformation map $\gamma := Bx$, where B is as defined in equation (2.4) with $a = 2$ and $b = 3$.

2.2.3 Example symmetries for dynamical systems

Equivariant systems are ubiquitous in nature and in relevant models of cyber-physical systems. Below are few examples of simple equivariant systems with respect to different symmetries. The first five examples correspond to autonomous dynamical systems, while the sixth describes a parameterized one. We start with a simple 2-dimensional linear oscillator system.

Example 2.1. Consider the *oscillator* system

$$\dot{x}[0] = -x[1], \dot{x}[1] = x[0]. \tag{2.3}$$

where $x \in \mathbb{R}^2$. Let Γ be the set of maps of the form Bx, where

$$B = \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \tag{2.4}$$

and $a, b \in \mathbb{R}$ and B is not the zero matrix. Let $\circ$ be the matrix multiplication operator, then system (2.3) is Γ-equivariant. The maps in Γ scale and rotate the states of the system. See Figure 2.1 for an example.

In general, for any linear system of the form $\dot{x} = Ax$, any map $\gamma : X \to X$ such that $\gamma(x) = Bx$ and $BA = AB$, is a symmetry of the system.

Example 2.2. The *Lorenz attractor* models the two-dimensional motion of a fluid in a container [104]. Its dynamics can be modeled using the following ODEs:

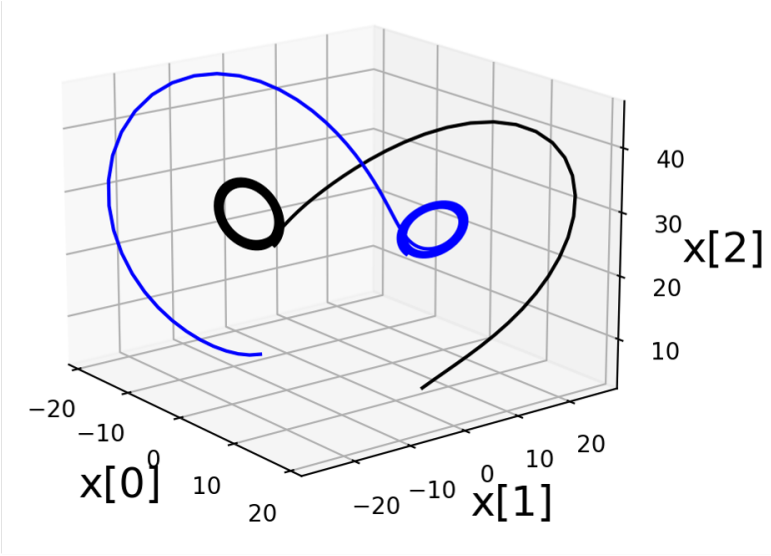


Figure 2.2: Black: trajectory of the Lorenz attractor starting from $[10,5,3]$ running for 3 seconds. Blue: trajectory of the Lorenz attractor starting from $[-10,-5,3]$ generated by transforming the black trajectory using γ shown in Example 2.2.

$$\begin{aligned}
\dot{x}[0] &= -px[0] + px[1], \\
\dot{x}[1] &= -x[0]x[2] + rx[0] - x[1], \text{ and} \\
\dot{x}[2] &= x[0]x[1] - bx[2].
\end{aligned} \tag{2.5}$$

where p, r, and b are parameters and $x \in \mathbb{R}^3$. Let Γ be the group that contains γ: $x \to [-x[0], -x[1], x[2]]$ and the identity map. Then, system (2.5) is Γ-equivariant[1]. The map γ reflects the state on the $x[0] = x[1] = 0$ line. An example is shown in Figure 2.2.

Example 2.3. A vehicle model is usually equivariant to the group of all translations of its position. The *kinematic bicycle model* of a car is described with the following ODEs [105]:

$$\begin{aligned}
&\dot{x}[0] = x[3]\cos x[4],\ \dot{x}[1] = x[3]\sin x[4],\ \dot{x}[2] = u, \\
&\dot{x}[3] = a,\ \dot{x}[4] = \frac{x[3]}{L}\tan(x[2]).
\end{aligned} \tag{2.6}$$

where u and a can be any control signals and $x \in \mathbb{R}^5$ representing the position

[1] `http://www.scholarpedia.org/article/Equivariant_dynamical_systems`

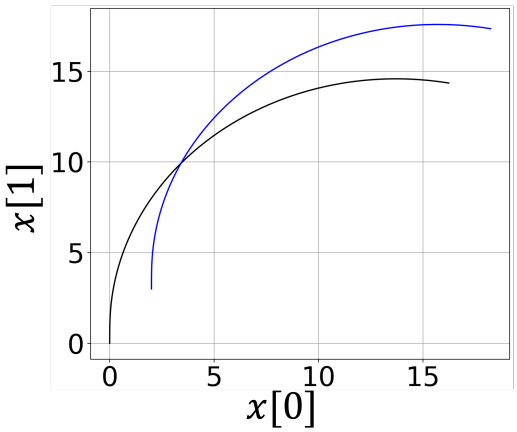


Figure 2.3: Black: trajectory of the car turning right ($u = 0.2$ and $a = 0$) starting from $[0,0,0,1,0]$ running for 3 seconds. Blue: trajectory of the car starting from $[2,3,0,1,0]$ generated by translating the position coordinates of the black trajectory using γ shown in Example 2.3 with $\delta = [2,3]$.

in the plane, the steering angle, the speed, and the heading angle of the car, respectively. Let Γ be the set of translations of the form $\gamma : x \to [x[0] + \delta[0], x[1] + \delta[1], x[2], x[3], x[4]]$, for all $\delta \in \mathbb{R}^2$. Then, system (2.6) is Γ-equivariant. See Figure 2.3 for an example.

Example 2.4. Consider the system consisting of two identical non-interacting cars with having the same model as that described in Example 2.3. The state of the system consists of the states x_1 and x_2 stacked together. Let Γ be the set containing the operator $\gamma : (x_1, x_2) \to (x_2, x_1)$ and the identity operator. Moreover, assume that u and a are the same for both cars. Then, the system is Γ-equivariant.

Given the closure property of a group, composition of symmetries are also symmetries as shown in the following example.

Example 2.5. Let Γ be the group generated by the set of transformations of the form $\gamma : (x_1, x_2) \to ((x_2[0] + \delta_2[0], x_2[1] + \delta_2[1], \dots), (x_1[0] + \delta_1[0], x_1[1] + \delta_1[1], \dots))$, where δ_1 and $\delta_2 \in \mathbb{R}^2$. Then, the system is Γ-equivariant. Hence, it is equivariant to translations in the positions and permutation of both cars.

Example 2.6. We introduce a *fixed-wing aircraft* following a waypoint. The state space is $X = \mathbb{R}^4$, parameter space $\mathscr{P} = \mathbb{R}^4$, and $f : X \times \mathscr{P} \to X$ defined as follows:

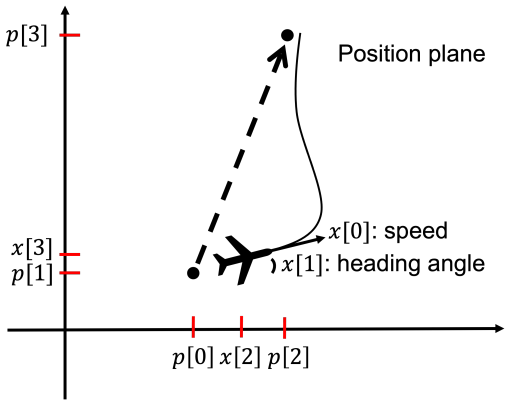


Figure 2.4: The state and parameter of the fixed-wing aircraft described in Example 2.6.

for any $x \in X$ and $p \in \mathscr{P}$,

$$\begin{aligned}
\dot{x}[0] &= \frac{T_c - c_{d1}x[0]^2}{m}, \\
\dot{x}[1] &= \frac{g}{x[0]} \sin\phi, \\
\dot{x}[2] &= x[0]\cos x[1], \text{ and} \\
\dot{x}[3] &= x[0]\sin x[1],
\end{aligned}$$

where $T_c = k_1 m(v_c - x[0])$, $\phi = k_2 \frac{v_c}{g}(\psi_c - x[1])$, $\psi_c = \arctan_2(\frac{x[2]-p[2]}{x[3]-p[3]})$, and k_1, k_2, m, g, c_{d1}, and v_c are positive constants. This models a fixed-wing aircraft starting from a waypoint and following another in the 2D plane: $x[0]$ is its speed, $x[1]$ is its heading angle, $(x[2], x[3])$ is its position in the plane, $(p[0], p[1])$ is the position of the source waypoint, and $(p[2], p[3])$ is the position of the destination one. Note that the source waypoint does not affect the dynamics, but will be useful later in the dissertation.

Let Γ be the group of maps that are each defined by a $goal \in \mathbb{R}^2$ and $\theta \in \mathbb{R}$ as follows: let $\gamma : X \to X$ and $\rho : \mathscr{P} \to \mathscr{P}$ be defined as:

$$\begin{aligned}
\gamma(x) = [x[0], x[1] - \theta, (x[2] - goal[0])\cos(\theta) + (x[3] - goal[1])\sin(\theta), \\
- (x[2] - goal[0])\sin(\theta) + (x[3] - goal[1])\cos(\theta)] \text{ and} \qquad (2.7)
\end{aligned}$$

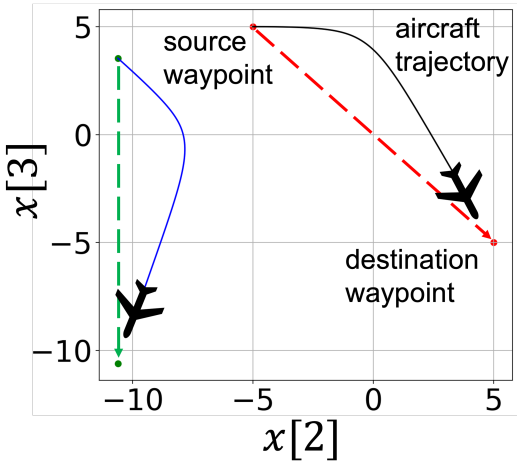


Figure 2.5: Black: trajectory of the fixed-wing aircraft following the waypoint $[5,-5]$, which is shown in red, starting from $[-5.0,5.0,0.0,5.0]$ running for 15 seconds. Blue: trajectory of the aircraft starting from $[-10.61,3.54,-0.79,5.0]$ following the waypoint $[-10.61,-10.61]$, which is shown in green. The blue trajectory and the green waypoints were generated by transforming the black trajectory using γ and the red waypoints using ρ, presented in Example 2.6 with $goal = (5,10)$ and $\theta = \pi/4$.

$$\begin{aligned}\rho(p) = [&(p[0]-goal[0])\cos(\theta)+(p[1]-goal[1])\sin(\theta),\\ &-(p[0]-goal[0])\sin(\theta)+(p[1]-goal[1])\cos(\theta),\\ &(p[2]-goal[0])\cos(\theta)+(p[3]-goal[1])\sin(\theta),\\ &-(p[2]-goal[0])\sin(\theta)+(p[3]-goal[1])\cos(\theta)].\end{aligned} \tag{2.8}$$

Then, for all $x \in X$ and $p \in \mathscr{P}$, $\gamma(f(x,p)) = f(\gamma(x),\rho(p))$. The transformation γ would change the origin of X from $[0,0,0,0]$ to $[0,0,goal[0],goal[1]]$. Then, it would rotate the third and four axes clockwise by the angle θ. Moreover, ρ would translate the origin of the parameter space $\mathscr{P}$ to $[goal[0],goal[1],goal[0],goal[1]]$ and rotate both pairs of the first and second axes and the third and fourth ones counter-clockwise by θ. For the aircraft, this means translating and rotating the plane where the aircraft and the waypoint positions reside. See Figure 2.5 for an example.

2.2.4 Hybrid dynamical systems

In this dissertation, we will need to model cyber-physical systems with both physical components that evolve continuously in time and software components that are naturally modeled as discrete-time state machines. To model such systems,

we follow the hybrid automaton framework [106, 93, 94, 87, 107, 61]. A hybrid automaton combines a collection of dynamical systems indexed by the modes, and rules for transitioning between the modes.

Definition 2.4. *A* hybrid automaton *is a tuple* $\mathscr{A} = \langle X, \mathscr{P}, X_{init}, p_{init}, E, guard, reset, f \rangle$, *where*

(a) $X \subseteq \mathbb{R}^n$ *is the continuous state space and* $\mathscr{P}$ *is a set of discrete states. Continuous states are simply called* states *and the discrete ones are called* modes *or* parameters.

(b) $X_{init} \subseteq X$ *is a set of possible initial states and* $p_{init} \in \mathscr{P}$ *is the initial mode,*

(c) $E \subseteq \mathscr{P} \times \mathscr{P}$ *is the set of directed edges over modes that define mode transitions,*

(d) $guard : E \to 2^X$ *gives the set of states from which an edge transition is enabled,*

(e) $reset : X \times E \to 2^X$ *gives the updated (post) state after a transition is taken, and*

(f) $f : X \times \mathscr{P} \to X$ *is a* dynamic function *that defines the continuous evolution. It is Lipschitz continuous in the first argument.*

For any mode $p \in \mathscr{P}$, the continuous variables evolve according to the ODE specified by $f(\cdot, p)$ and the semantics defined in Section 2.2.1. The edge set E, the *guard*, and the *reset* together define the discrete transitions. For an edge $e = (p, p') \in E$, we denote its source mode p by $e.src$ and its destination mode p' by $e.dest$. A sequence of modes $p_0, p_1, \ldots$, where for all $i \geq 0$, $(p_i, p_{i+1}) \in E$ is called a *path*. Moreover, we abuse notation and denote $guard((p, p'))$ by $guard(p, p')$. Then, $guard(p, p')$ is the set from which a transition from mode p to mode p' is possible. From a state $x \in guard(p, p')$, the post-state x' after the transition has to be in $reset(x, (p, p'))$. Such state-mode pairs $((x, p), (x', p'))$ define the transitions of $\mathscr{A}$ and we write $(x, p) \to (x', p')$. Note that there are no urgent transitions here, and guards may be ignored.

In the current formulation only one initial mode is specified. Also, there can be at most one edge connecting a pair of modes. The results in the dissertation can be easily generalized to automata with multiple initial modes and multiple edges connecting mode pairs.

The semantics of a hybrid automaton is defined by executions which are sequences of trajectories and transitions. An *execution* of $\mathscr{A}$ is a sequence of pairs of trajectories and modes $\sigma = (\xi_0, p_0), (\xi_1, p_1), \ldots$, where (a) each ξ_i is a trajectory of $\mathscr{A}$ following mode p_i, (b) each $(\xi_i.lstate, p_i) \to (\xi_{i+1}.fstate, p_{i+1})$ is a transition as defined above, and (c) $\xi_0.fstate \in X_{init}$ and $p_0 = p_{init}$. A *finite* and *time-bounded* execution has a finite number of discrete transitions and all of its trajectories are time-bounded. The duration of a finite and time-bounded execution $\sigma = (\xi_0, p_0), (\xi_1, p_1) \ldots (\xi_k, p_k)$ is $dur(\sigma) = \sum_i dur(\xi_i)$, its length $\sigma.len = k$, its last state is $\sigma.lstate = \xi_k.lstate$, and its last mode is $\sigma.lmode = p_k$.

Finally, fix $J \in \mathbb{N}$; $exec_{\mathscr{A}}(J)$ is the set of all executions of $\mathscr{A}$ with at most J transitions; When the transitions are unbounded, it is denoted by $exec_{\mathscr{A}}$.

Later in Chapter 5, we will discuss how an equivariance property of the dynamic function of a hybrid automaton can be used to produce an automaton with fewer modes and symmetric executions to those of the original automaton. For now, we conclude this section with an example of a hybrid automaton. A language for specifying hybrid automata succinctly can be found in [85, 87], but to minimize the notational overhead, in this thesis we present the automaton using standard mathematical notations.

Example 2.7 (Robot following waypoints)**.** Consider a robot following a sequence of waypoints $\{w_i \in \mathbb{R}^2 \mid i \in [4]\}$ on the plane connected with directed roads $\{r_i \in \mathbb{R}^4 \mid i \in [5]\}$ forming an axis-aligned rectangle centered at the origin (see Figure 2.6a). For any road $r_i \in \mathbb{R}^4$, we define $r_i.src$ to be its first two coordinates $r_i[0:1]$ representing its start waypoint and $r_i.dest$ to be its last two coordinates $r_i[2:3]$ specifying its end waypoint. The robot starts from an arbitrary point in some initial set $X_{init} \subset \mathbb{R}^3$. We fix three possible rectangles' dimensions: ε_0, ε_1, and $\varepsilon_2 \in (\mathbb{R}^+)^2$. We say that the robot reached w_0 following road r_0 if it is located in the rectangle $B(w_0, \varepsilon_0)$. If it was following road r_4 instead, we say it reached w_0 if it is located in the smaller rectangle $B(w_0, \varepsilon_1)$. Moreover, for any $i \in \{1, 2, 3\}$, we say that it reached w_i following r_i if it is located in the rectangle $B(w_i, \varepsilon_1)$.

To formalize the dynamics of the robot in this scenario, we construct a corresponding hybrid automaton. We use the five roads as five modes of the automaton, i.e. $\mathscr{P} = \{r_0, r_1, r_2, r_3, r_4\}$. In each mode, the robot would follow the destination waypoint of the corresponding road. We define the resulting automaton as follows:

$$R := \langle X, \mathscr{P}, X_{init}, p_{init}, E, guard, reset, f \rangle$$

is shown in Figure 2.6b, where

(a) $X \subseteq \mathbb{R}^3$, representing the position and orientation with respect to the $x[0]$-axis, and $\mathscr{P} := \{p_i := r_i \mid i \in [5]\}$, the set of roads in Figure 2.6a,

(b) $X_{init} := B(r_0.src, \varepsilon_2) \times [0, \pi/2]$, where the angle range is arbitrarily chosen, and $p_{init} := p_0 = r_0$,

(c) $E := \{e_0 := (p_0, p_1), e_1 := (p_1, p_2), e_2 := (p_2, p_3), e_3 := (p_3, p_4), e_4 := (p_4, p_1)\}$,

(d)

$$guard(e_i) := \begin{cases} B(w_i, \varepsilon_0) \times \mathbb{R}, \text{ if } i = 0, \\ B(w_i, \varepsilon_1) \times \mathbb{R}, \text{ if } i = \{1,2,3,4\}. \end{cases}$$

(e) $\forall x \in X, e \in E$,

$$reset(x, e) := \{x\},$$

is the identity map, and

(f) $\forall x \in X, \forall p \in \mathscr{P}$,

$$f(x,p) := \frac{dx}{dt} = \begin{bmatrix} v\cos(x[2]) \\ v\sin(x[2]) \\ 2v\sin(\alpha)/L \end{bmatrix}, \text{where} \tag{2.9}$$

$\alpha = \arctan_2(p[3] - x[1], p[2] - x[0]) - x[2]$ and v and L are the fixed speed and length of the robot, respectively [108].

2.3 Safety and reachability

This section presents several definition on the safety verification problem and reachability analysis.

2.3.1 The bounded-time safety verification problem

The *bounded-time safety verification problem* consists of several components: the model of the system, the uncertain initial state represented by a compact set of

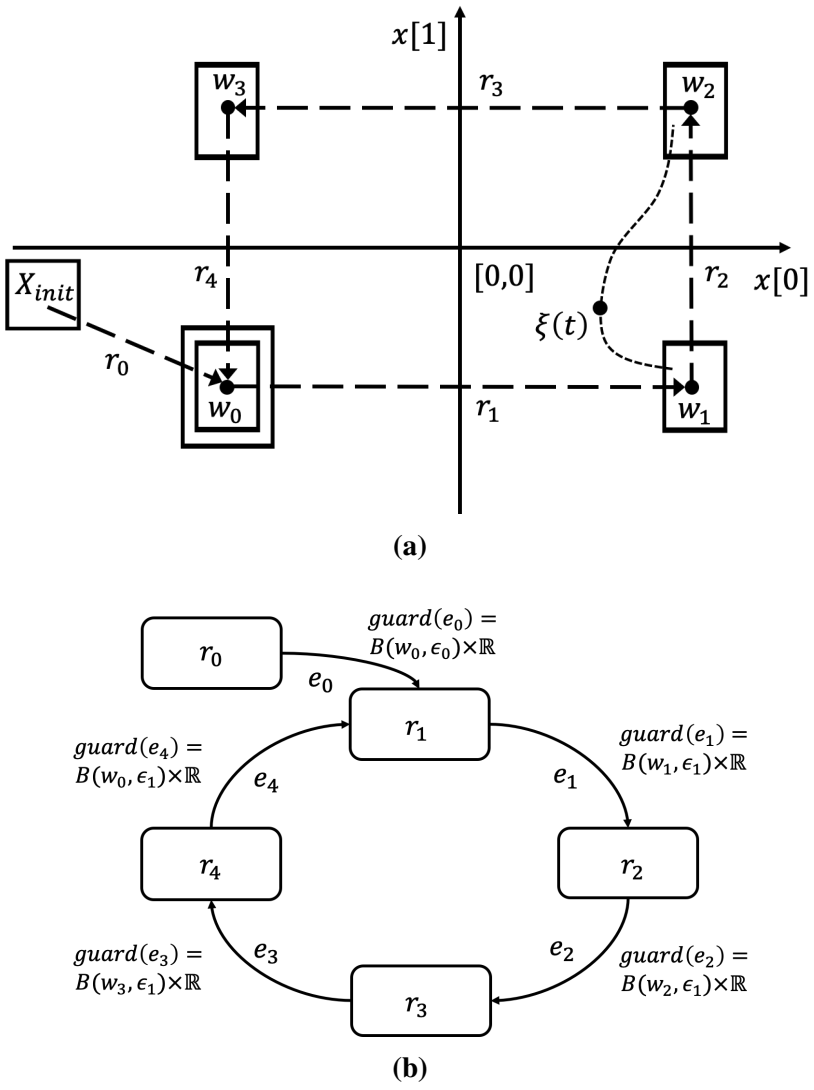


Figure 2.6: (2.6a) A robot, with a state containing its position and orientation, following a sequence of 2D waypoints forming a rectangle starting from its initial set X_{init}. It reaches a waypoint if it reaches the rectangle centered at it. It has to reach the larger rectangle centered at w_0 when starting from X_{init}. (2.6b) The state machine representing the discrete transitions of the hybrid automaton R describing the scenario in Figure 2.6a with the roads being the modes. The resets are omitted since they are just the identity map for all the modes.

possible initial states, the time bound for which the safety property has to be verified, and the *unsafe set* of states. See Figure 2.7 for an example. The unsafe set represents states that the system should not reach during its operation. Fixed and dynamic obstacles, off-lane positions, speed limits, and maximum rotation angles are all examples of unsafe sets. Given a model of the system, an unsafe set

of states, and a time bound, the bounded safety verification problem requires us to check whether there exists a trajectory or execution of the system that intersects the unsafe set within the specified time bound.

A *safety verification algorithm* is a procedure that decides the result of the problem. Such an algorithm is *sound* iff its result is always correct. More concretely, if it returns *safe*, then there is no trajectory or execution of the system that starts from the specified initial set of states and intersects the unsafe set within the specified time bound. If it returns *unsafe* instead, then there exists a trajectory or execution of the system that starts from the specified initial set of states and intersects the unsafe set within the specified time bound. Such an execution is called a *counter-example* of the safety property.

The algorithm is *complete* if it terminates, with a *safe* or *unsafe* decision for any given input. A famous result by Henzinger et al. [109] shows that the safety verification problem is *undecidable* for hybrid automata, even those with constant dynamics. That means that there is no safety verification algorithm that is both sound and complete. Approximate guarantees are thus needed.

The algorithm is *relatively completely* if it terminates with a *safe* or *unsafe* decision for *robustly safe* and *robustly unsafe* systems. A system is robustly safe if there exists an $\varepsilon \in (\mathbb{R}^+)^n$, such that for any $t \in [0,T]$ and any trajectory ξ (or execution σ), $B(\xi(t),\varepsilon)$ (or $B(\sigma(t),\varepsilon)$), does not intersect the unsafe set. A system is robustly unsafe if there exists a trajectory ξ (or execution σ), $\varepsilon \in (\mathbb{R}^+)^n$, and time $t \in [0,T]$ and $\tau > 0$, such that $B(\xi(s),\varepsilon)$ (or $B(\sigma(s),\varepsilon)$) is contained in the unsafe set for all $s \in [t,t+\tau]$.

2.3.2 Reachsets and reachtubes

The standard method for solving the bounded safety verification problem is to compute or approximate the reachable states of the system. The set of *reachable states* of a continuous-time dynamical system of equation (2.1) between times t_1 and t_2, starting from initial set $X_0 \subset X$ following mode $p \in \mathscr{P}$ at time $t_0 = 0$ is defined as

$$Reach(X_0, p, [t_1, t_2]) = \{x \in X \mid \exists\, x_0 \in X_0,\ t \in [t_1, t_2] \text{ s.t. } \xi(x_0, p, t) = x\}.$$

Thus, given a time bound $T \geq 0$ and an unsafe set $O \subseteq X$, computing or over-approximating $Reach(X_0, p, [0,T])$ and checking $Reach(X_0, p, [0,T]) \cap O = \emptyset$ is

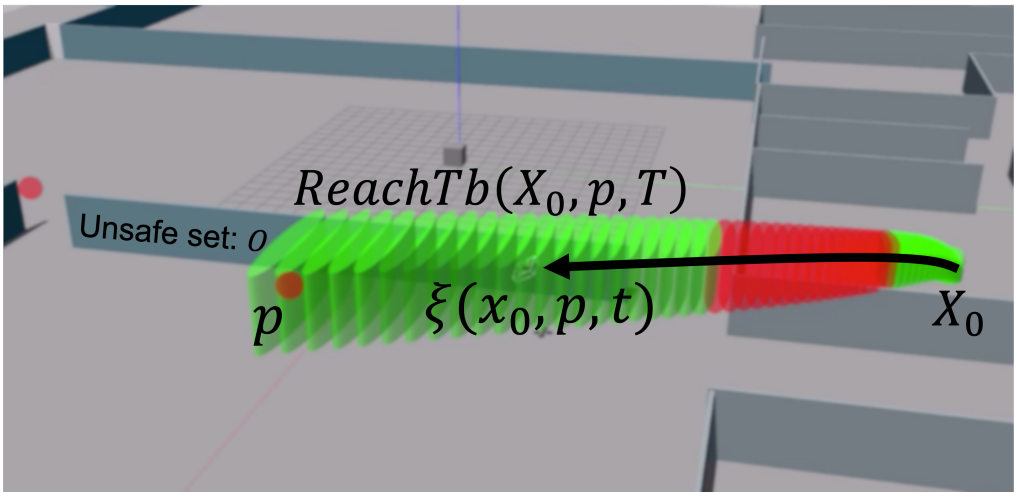


Figure 2.7: An example trajectory, reachtube, and unsafe set for a Hector quadrotor [98], all projected to the 3D physical space. The quadrotor is following the red waypoints in a Gazebo [2] environment modeling the first floor of the Electrical and Computer Engineering building at the University of Illinois Urbana-Champaign. The unsafe set of states O, when projected to physical space, model the walls in the environment. The unsafe set O, when projected to any of the other dimensions, the heading angle and speed, becomes the range $(-\infty, \infty)$. That means that it is unsafe for the quadrotor to collide with the wall at any speed or heading angle. The elements of the reachtube that intersect with the unsafe set are colored red, and the rest are colored green.

sufficent for verifying bounded safety. Instead of $Reach(X_0, p, [t,t])$ we write $Reach(X_0, p, t)$ in short. In the autonomous system case, i.e. when p is constant, we drop the p argument from *Reach*'s definition.

Sometimes we find it convenient to preserve the time information of reaching states. This leads to the notion of reachtubes. Given a time bound $T > 0$, we define *reachtube* $ReachTb(X_0, p, T) = \{(X_i, t_i)\}_{i=0}^{J-1}$ to be a sequence of time-stamped sets such that for each i, $X_i = Reach(X_0, p, [t_{i-1}, t_i])$, $t_0 = 0$, $t_{-1} = 0$, and $t_{J-1} = T$. For a given $(X_0, p, [ftime, etime])$-reachtube rtb, we denote its parameters by $rtb.X_0$, $rtb.p$, $rtb.ftime$, and $rtb.etime$, respectively, its cardinality J by $rtb.len$, and its last pair (X_{J-1}, t_{J-1}) by $rt.last$.

For a hybrid automaton $\mathscr{A}$, we define the set of *reachable states* for a time bound $T \geq 0$ as:

$$Reach_{\mathscr{A}}(T) = \{x \in X \mid \exists\, \sigma \in exec_{\mathscr{A}}, dur(\sigma) \leq T \text{ and } \sigma.lstate = x\}. \quad (2.10)$$

We denote the unbounded-time reachable set by $Reach^{\infty}_{\mathscr{A}}$. The restriction of $Reach_{\mathscr{A}}(T)$ to states reached in mode $p \in \mathscr{P}$ is denoted by $Reach_{\mathscr{A}}(p, T)$. Note

that T here is the bound on the duration of the whole execution including the time spent in other modes than p. We call such reachsets *reachset segments*. Given a $T \geq 0$ and an *unsafe map* $O : \mathscr{P} \to 2^X$, the *hybrid system bounded-time safety verification* problem requires us to check whether $\forall p \in \mathscr{P}, Reach_{\mathscr{A}}(p,T) \cap O(p) = \emptyset$.

In the rest of the dissertation, for simplicity of presentation, we use the same notation of reachable sets and reachtubes introduced in this section, for their over-approximations. We call these over-approximations reachable sets and reachtubes, as well.

2.3.3 Software tools for reachability analysis

A numerical simulation of system (2.1) is a reachtube with X_0 being a singleton state $x_0 \in X$. It is a discrete-time representation of $\xi(x_0, p, \cdot)$. Several numerical solvers provide such representation of trajectories such as VNODE-LP [110] and CAPD Dyn-Sys library [111].

Computing reachsets exactly is theoretically hard [133]. There are many reachability analysis tools that can compute bounded-time over-approximations of the reachsets such as SpaceEx [113], Hylaa [67], Pirk [114], C2E2 [85], DryVR [115], Flow* [17], Cora [116], and JuliaReach [117]. It is an active area of research that is expanding to new models such as NNs and control systems with ML components with new tools appearing every year, such as Verisig [20], nnenum [118], and NNV [19]. The Applied Verification for Continuous and Hybrid Systems (ARCH) workshop [119] is an annual one that discusses and evaluates the performance of such tools. It is part of the Cyber-physical systems Week (CPS week), the major venue for CPS research. Generally speaking, given an initial set X_{init} for a set of ODEs, these tools can return a sequence of sets that contain the exact reachset over small time intervals. In the last section of this chapter, we present the method implemented in the DryVR tool [115] for generating reachset approximations from simulations and sensitivity analysis.

2.3.4 Operations on reachsets and reachtubes

In this dissertation, we will find it useful to transform solutions and reachtubes using maps of the form $\gamma : X \to X$, which will be the symmetries of the system.

Given a solution ξ and a reachtube $ReachTb(X_0, p, T)$, we define the γ-transformed solution $\gamma \cdot \xi$ and reachtube $\gamma \cdot ReachTb(X_0, p, T)$ as follows:

$$\forall\, t, (\gamma \cdot \xi)(x_0, p, t) = \gamma(\xi(x_0, p, t)) \text{ and } \gamma \cdot ReachTb(X_0, p, T) = \{(\gamma(X_i), t_i)\}_{i=0}^{J-1}.$$

Notice that this transformation does not alter the time-stamps. In Theorems 3.1 and 4.1, we show that if γ is a symmetry of the system, then the transformation result is also a reachtube starting from the transformed initial set.

We define union, truncation, concatenate, and time-shift operators on reachtubes. Without loss of generality, we assume equal separation between the time points, i.e. $\exists\ \tau_s > 0, \forall i \in [J-1],\ t_i - t_{i-1} = \tau_s$. Fix $rtb_1 = \{(X_{i,1}, t_{i-1,1})\}_{i=0}^{J_1-1}$ and $rtb_2 = \{(X_{i,2}, t_{i,2})\}_{i=0}^{J_2-1}$ to be two reachtubes, where $J_1 = rtb_1.len$ and $J_2 = rtb_2.len$. If $t_{i,1} = t_{i,2}$ for all $i \in [0 : \min(J_1, J_2) - 1]$, we say they are *time-aligned*. Without loss of generality, assume that the two reachtubes are time-aligned and that $j_1 \leq j_2$. The operators are defined as follows:

- $timeShift(rtb_1, t_s) = \{(X_{i,1}, t_s + t_{i,1})\}_{i=0}^{J_1-1}$,
- *union*: $rtb_1 \cup rtb_2 = \{(X_{i,1} \cup X_{i,2}, t_{i,1})\}_{i=0}^{J_1-1} \cup \{(X_{i,2}, t_{i,2})\}_{i=J_1}^{J_2-1}$,
- *concatenation*: $rtb_1 \frown rtb_2 = rtb_1 \cup \{(X_{i,2}, t_{J_1-1,1} + t_{i,2}))\}_{i=0}^{J_2-1}$, and
- $truncate(rtb_1, t_c) = \{(X_{i,1}, t_{i,1})\}_{i=0}^{k}$, where $t_{k,1} \geq t_c$ and $t_{k-1,1} < t_c$.

It can be shown that, under proper conditions, the result of all of these operations are reachtubes of the system as well. The result of $timeShift(rtb_1, t_s)$ is a reachtube starting from the initial set $X_{i,0}$ at time t_s, as we assume that the dynamics are time-invariant, throughout this dissertation. If $J_1 = J_2$, then the union $rtb_1 \cup rtb_2$ results in the reachtube of the system starting from the union of the initial sets of rtb_1 and rtb_2 and having a time duration of $(J_1 - 1)\tau_s$. If $X_{J_1-1,1} \subseteq X_{0,2}$, then $rtb_1 \frown rtb_2$ results in the reachtube of the system starting from the initial set of rtb_1 and having a time bound equal to the sum of the time bounds of the two reachtubes, again because the dynamics are time-invariant. The fourth operation, $truncate(rtb_1, t_c)$, results in the reachtube starting from the initial set of rtb_1, but with a smaller time bound that is equal to the truncation time. These operations are mainly used in Chapter 4.

Finally, rtb_1 is a subset of rtb_2, or $rtb_1 \subseteq rtb_2$, iff $X_{i,1} \subseteq X_{i,2}$, for all $i \in [0 : J_1 - 1]$.

2.4 Data-driven verification

For completeness of the dissertation, we describe in this section one of the prominent classes of algorithms for safety verification, namely *data-driven verification*. Data-driven verification can be used for both dynamical and hybrid models. For conciseness we present an algorithm for autonomous dynamical systems. For hybrid models we refer the reader to Chapter 11 of [87] and for systems with inputs we refer to [69].

Data-driven verification algorithms answer the bounded safety verification question using numerical data generated from running the system or the model. Such data can be obtained from numerical simulations, tests, and field experiments. As we consider only the autonomous case the modes are dropped from the notations for trajectories and reachsets. The key idea is to generalize an individual simulation of a trajectory $\xi(x_0,[0,T])$ to over-approximate the reachtube $ReachTb(B(x_0,\delta),T)$, for some $\delta > 0$. This generalization covers a δ-ball $B(x_0,\delta)$ of the initial set X_0, and several simulations can then cover all of X_0 and over-approximate $ReachTb(X_0,T)$, which in turn could prove safety. If the over-approximations turn out to be too conservative and safety cannot be concluded, then δ has to be reduced, and more precise over-approximations of $ReachTb(X_0,T)$ have to be computed with smaller generalization radius δ and more simulation data.

The generalization strategy discussed above is entirely based on computing sensitivity of the solution $\xi(x_0,t)$ to the initial condition x_0. The precise notion of sensitivity needed for the verification algorithm to guarantee soundness and relative completeness is formalized as *discrepancy function* [83] (see Figure 2.8).

Definition 2.5. *A discrepancy function of system (2.1) with initial set of states $K \subseteq \mathbb{R}^n$ is a class-$\mathscr{K}$ function in the first argument $\beta : \mathbb{R}^+ \times \mathbb{R}^+ \to \mathbb{R}^+$ that satisfies the following conditions: (1) $\forall\, x,x' \in X_0, t \geq 0$, $\|\xi(x,t) - \xi(x',t)\| \leq \beta(\|x-x'\|,t)$, (2) $\beta(\|\xi(x,t) - \xi(x',t)\|,t) \to 0$ as $\|x-x'\| \to 0$.*

The first condition in Definition 2.5 says that β upper-bounds the distance between two trajectories as a function of the distance between their initial states. This ensures that generalizations are sound. The second condition makes the bound shrink as the initial states get closer. As we shall see, this is necessary for achieving precision of over-approximation and in turn completeness.

Algorithms have been developed for computing discrepancy for linear, nonlinear, and hybrid dynamical models [18, 69, 65] as well as for estimating it for black-

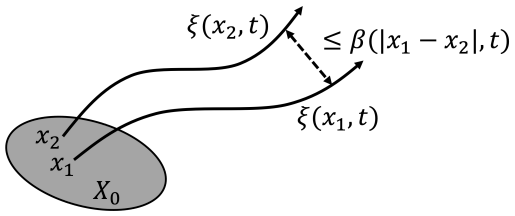


Figure 2.8: A discrepancy function upper-bounds the distance between any two trajectories at any time instant given the distance between their initial states.

box systems [15]. The resulting software tools, C2E2 [85], STRONG [120], and DryVR [115], have been successfully applied to verify automotive, aerospace, and medical embedded systems [76, 121, 122].

Algorithm 2.1 describes data-driven verification for the continuous-time dynamical system (2.1). We refer to this algorithm as ddVer in this dissertation, especially in Chapter 3. Figure 2.9 shows an example execution of such an algorithm. Given the compact initial set of states $X_0 \subseteq X$, a time bound $T > 0$, and an unsafe set $O \subseteq X$, ddVer answers the safety verification question. It initializes a stack called *coverstack* with a cover of X_0. Then, it checks the safety from each element in the cover. For a given $B(x_0, \delta)$ in *coverstack*, ddVer simulates system (2.1) from x_0 and bloats to compute an over-approximation of $ReachTb(B(x_0, \delta), T)$. Formally, the set $sim \oplus \beta$ is a Minkowski sum. This can be computed by increasing the radius in each dimension of *sim* at a time instant t by $\beta(\delta, t)$. The first condition on β ensures that this set is indeed an over-approximation of $ReachTb(B(x_0, \delta), T)$. If this over-approximation is disjoint from O then it is safe and is removed from *coverstack*. If instead, the over-approximation intersects with O then that is inconclusive and $B(x_0, \delta)$ is partitioned into smaller sets and added to *coverstack*. The second condition on β ensures that this *refinement* leads to a more precise over-approximation of $ReachTb(B(x_0, \delta), T)$. On the other hand, if the simulation hits O, that serves as a counterexample and ddVer returns Unsafe. Finally, if *coverstack* becomes empty, that implies that the algorithm reached a partition of X_0 from which all the over-approximated reachtubes are disjoint from O, and then ddVer returns Safe.

In different parts of the dissertation, we use DryVR [15], a safety verification and synthesis tool for hybrid automata with black-box dynamics. DryVr implements a generalized version of ddVer for hybrid systems. DryVR learns discrepancy from simulations as it is designed to work with unknown dynamical models. This discrepancy learning functionality is unnecessary when using DryVR in all

Algorithm 2.1 ddVer: Data-driven safety verification algorithm

input: X_0, T, O, Γ, β
coverstack $\leftarrow$ finite cover $\cup_i B(x_i, \delta) \supseteq X_0$
while *coverstack* $\neq \emptyset$ **do**
    $B(x_0, \delta) = coverstack.pop()$
    sim $\leftarrow$simulate $\xi(x_0, \cdot)$ up to time T
    $rt \leftarrow sim \oplus \beta$
    if *sim* intersects with O **then**
        return: Unsafe
    else if rt intersects with O **then**
        Refine cover and add to the *coverstack*
return: Safe

of our experiments in this dissertation, as checking equivariance requires some knowledge of the model. For convenience, we use DryVR's discrepancy learning instead of deriving discrepancy functions by hand. That said, some symmetries can be checked without complete knowledge of the model. For example, we know that dynamics of vehicles do not depend on their absolute position even without knowledge of precise dynamics.

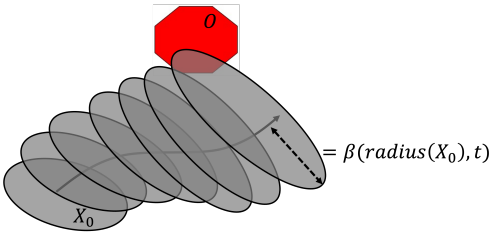
2.5 Evaluation metrics for safety verification algorithms

The focus of this dissertation is accelerating verification using symmetry. The evaluation criteria according to which the different proposed algorithms are compared, are the following:

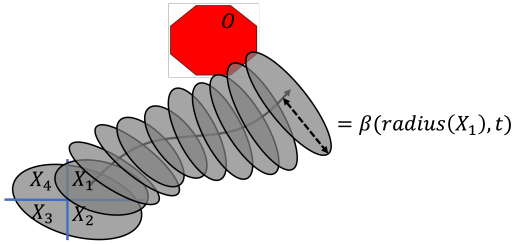
- *Running time*: the faster a verification algorithm is, the more usable it is. This is the main factor preventing formal methods from being deployed more widely in industry.

- *Space*: the smaller memory space the verification algorithm is, the more usable it is. This factor becomes especially important if the algorithm would be deployed on the edge, on devices with limited storage and memory space.

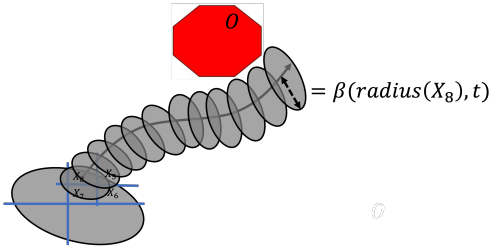
The running time of an algorithm in an experiment usually depends on factors related to the computation power of the machine used, instead of depending solely on the parameters of the problem being solved. For a fairer comparison, sometimes this dissertation evaluates the algorithms based on more controllable



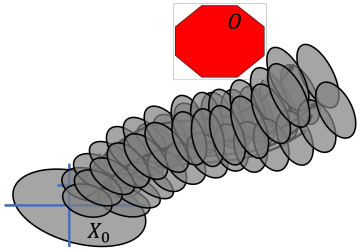
(a) A reachtube computed using a simulation and the discrepancy function β.



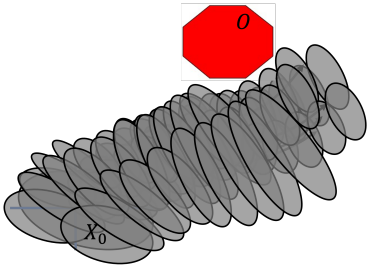
(b) Partitioning the initial set decreases the over-approximation error of the computed reachtube, since β approaches zero as its first parameter approaches zero.



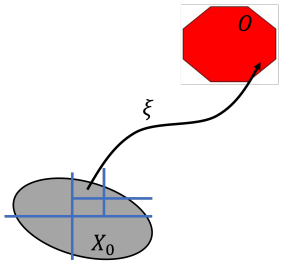
(c) A data-driven verification algorithm recursively partitions the initial set until a reachtube disjoint from the unsafe set can be computed, or a counter-example is found.



(d) A data-driven verification algorithm computes the reachtubes starting from all the parts of the initial set, until all are proven safe or a counter-example is found.



(e) An example reachtube that would be computed if the system is safe.



(f) An example counter-example that may be found after further partitioning the initial set if the system is unsafe.

Figure 2.9: A sketch illustrating the different operations of a data-driven verification algorithm.

metrics: the number of computed simulations, reachsets, or reachtubes and the number of transformed ones. Computing any of these objects is in general more computationally expensive that transforming them. Hence, an algorithm, with the same sum of the numbers of computed and transformed reachtubes, but with fewer computed ones than another algorithm, is, in general, a faster one.

This dissertation discusses how the evaluation metrics scale with the input parameters of the proposed algorithms. Such parameters include the number of agents, number of modes, number of obstacles, complexity of the dynamics represented by the linearity or nonlinearity type and the number of dimensions of the state, and the symmetries used.

Finally, for each proposed algorithm, this dissertation discusses its soundness and relative completeness guarantees. It discusses the sources of over-approximation errors and how they get reduced by the algorithm.

Chapter 3

Symmetry for Faster Data-driven Verification

In this chapter, we investigate how additional knowledge about symmetry transformations of equivariant dynamical systems can reduce the computation effort for their verification. It is based on a conference paper presented in the International Symposium on Automated Technology for Verification and Analysis (ATVA) in 2019 in Taipei, Taiwan [84][1]. The bounded safety verification problem in Section 2.3 relies on computing reachsets, potentially several of them, which is computationally expensive. Here comes the idea of caching computed reachsets for faster verification. However, computing reachsets from the same initial sets for which reachsets have been previously computed is rarely needed. Symmetry allows transforming computed trajectories of a system to new ones starting from *different* initial states. Consequently, symmetry can be used to *efficiently* transform cached reachset to new ones starting from different initial sets of states.

We augment the standard data-driven verification algorithm with a new procedure that attempts to verify the safety of the system starting from a new initial set of states by transforming previously computed reachtubes using symmetry. This new algorithm required the creation of a new cache-tree data structure for multi-resolution reachtubes. We implemented our algorithm on top of DryVR. Our experimental results on several benchmarks show significant improvements in verification time.

3.1 Overview

In this chapter, we demonstrate how efficient caching of reachsets using the symmetries of dynamical systems can accelerate their data-driven verification. The research program on data-driven verification and falsification has recently been met with some successes [18, 83, 65, 123]. The core idea is to use simulation, together

[1]This work is in collaboration with Navid Mokhlesi. It was nominated for the best paper award

with model-based sensitivity analysis or property-specific robustness margins, to provide *coverage guarantees* or expedite the discovery of counter-examples. Software tools implementing these approaches have been used to verify embedded medical devices, automotive, and aerospace systems [76, 65, 123, 18]. In this chapter, we examine the question: *how can we reduce the number of simulations and reachsets needed in the data-driven verification of a dynamical system, by utilizing more information about the model in the form of its symmetries?*

The chapter builds-up on the foundational results in symmetry transformations for dynamical systems [27, 124, 26] to provide results that allow us to compute the reachtubes of an autonomous continuous-time dynamical system from a given initial set X'_{init}, by transforming previously computed reachtubes from a *different* initial set X_{init}. Since the computation of a reachtube from scratch is usually more expensive than applying a transformation to a set, this reduces the number of reachtube computations, and therefore, the number of simulations.

Then, we present a verification algorithm symCacheTree based on transforming cached reachtubes using a given symmetry transformation γ of the system instead of computing new ones. We augment the standard data-driven safety verification algorithm shown in Section 2.4 with symCacheTree to reduce the number of reachtubes that need to be computed from scratch. We do that by caching reachtubes as they are computed by the main algorithm in a tree structure representing refinements. Before any new reachtube is computed from a given refinement of the initial set, symCacheTree is asked if it can determine the safety of the system based on the cached reachtubes. It will then do a breadth-first search (BFS) over the tree to find suitable cached reachtubes that are useful under the transformation, γ. It either returns a decision on safety or says it cannot determine that. In that case, the main algorithm computes the reachtube from scratch. We prove that the symmetry assisted algorithm is sound and complete. We further generalize symCacheTree to use a set of symmetry transformations instead of one. We call the new algorithm symGroupCacheTree. It allows us to compute multiple reachtubes, instead of just one, from any single cached reachtube. This will prove to be essential to get significant savings computation time and to overcome the caching overhead, as shown in Section 3.4.

Finally, we implemented the algorithms on top of the DryVR tool [15]. We augmented DryVR with symGroupCacheTree. We tested our approach on several linear and nonlinear examples with different symmetry transformations. We showed that in certain cases, by using symmetry, one can eliminate several

dimensions of the system from the computation of reachtubes, which resulted in significant speedups (more than $1000\times$ in some cases).

The chapter starts with the main theorems for transforming reachtubes in Section 3.2. In Section 3.3, we present symCacheTree and symGroupCacheTree along with the key guarantees. The results of experiments are in Section 3.4 and conclusions and future directions are in Section 3.5.

3.2 Transforming reachsets using symmetry

In this section, we present new results that use symmetry ideas of Section 2.2.2 towards safety verification. We show how symmetry operators can be used to get new reachtubes by transforming existing ones. This is important for data-driven verification because computation of new reachtubes is in general much more expensive than transforming reachtubes.

For convenience, we will fix a set of initial states $X_0 \subseteq X$, a time bound $T > 0$, a group Γ of operators on X, and an operator $\gamma \in \Gamma$ throughout this section. The following theorem formalizes transformation of reachtubes based on symmetry. It follows from Theorem 2.1.

Theorem 3.1. *If system (2.1) is Γ-equivariant, then*

$$\forall \gamma \in \Gamma, \gamma(\mathit{ReachTb}(X_0, T)) = \mathit{ReachTb}(\gamma(X_0), T).$$

Proof. By Theorem 2.1, given any solution $\xi(x_0, \cdot)$ of system (2.1), where $x_0 \in X_0$, $\gamma(\xi(x_0, \cdot))$ is its solution starting from $\gamma(x_0)$, i.e. $\gamma(\xi(x_0, \cdot)) = \xi(\gamma(x_0), \cdot)$.

Proof that $\gamma(\mathit{ReachTb}(X_0, T)) \subseteq \mathit{ReachTb}(\gamma(X_0), T)$: Fix any pair $(X_i, t_i) \in \mathit{ReachTb}(X_0, T)$ and fix an $x \in X_i$. Then, there exists $x_0 \in X_0$ such that $\xi(x_0, t) = x$ for some $t \in [t_{i-1}, t_i]$. Hence, by Theorem 2.1, $\xi(\gamma(x_0), t) = \gamma(x)$. Therefore, $\gamma(x) \in \mathit{ReachTb}(\gamma(X_0), T)$. Since x is arbitrary, $\gamma(\mathit{ReachTb}(X_0, T)) \subseteq \mathit{ReachTb}(\gamma(X_0), T)$.

Proof that $\mathit{ReachTb}(\gamma(X_0), T) \subseteq \gamma(\mathit{ReachTb}(X_0, T))$: Fix any pair $(X_i, t_i) \in \mathit{ReachTb}(\gamma(X_0), T)$ and fix an $x \in X_i$. Then, there exists $x_0 \in \gamma(X_0)$ such that $\xi(x_0, t) = x$ for some $t \in [t_{i-1}, t_i]$. Since $x_0 \in \gamma(X_0)$, there exists $x_0' \in X_0$ s.t. $\gamma(x_0') = x_0$. By Theorem 2.1, $\gamma(\xi(x_0', t)) = x$. Hence, $x \in \gamma(\mathit{ReachTb}(X_0, T))$. Again, since x is arbitrary, $\mathit{ReachTb}(\gamma(X_0), T) \subseteq \gamma(\mathit{ReachTb}(X_0, T))$.

□

Corollary 3.1 shows how a new reachtube from a set of initial states $X_0' \subseteq X$ can be computed by γ-transforming an existing $ReachTb(X_0, T)$.

Corollary 3.1. *If system (2.1) is Γ-equivariant, and $X_0' \subseteq X$, then if there exists $\gamma \in \Gamma$ such that $X_0' \subseteq \gamma(X_0)$, then $ReachTb(X_0', T) \subseteq \gamma(ReachTb(X_0, T))$.*

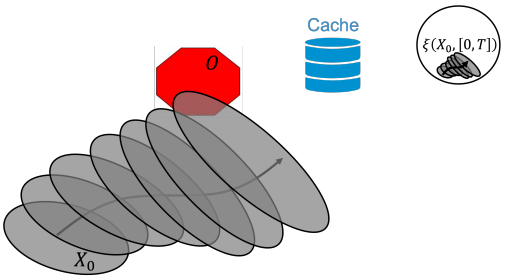
Remark 3.1. Corollary 3.1 remains true if instead of $ReachTb(X_0, T)$, we have a tube that over-approximates it. Moreover, Theorem 3.1 and Corollary 3.1 are also true if we replace the reachtubes with reachsets.

3.3 Caching and symmetry in data-driven verification

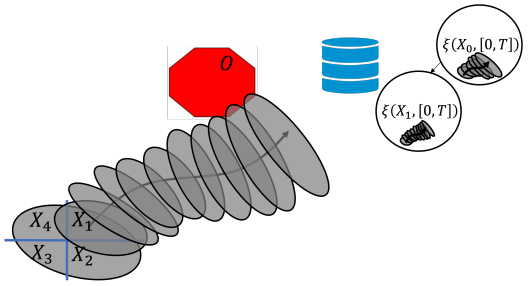
Algorithm 3.1 ddSymVer safety verification algorithm

```
1:  input: X_0, T, O, Γ, β
2:  coverstack ← finite cover ∪_i B(x_i, δ) ⊇ X_0
3:  cachetree ← ∅
4:  while coverstack ≠ ∅ do
5:      B(x_0, δ), parentnode = coverstack.pop()
6:      ans ← symGroupCacheTree(O, Γ, cachetree, B(x_0, δ))
7:      if ans = Unsafe then return: ans
8:      else if ans = Safe then continue
9:      else
10:         sim ← simulate ξ(x_0, ·) for a duration of T time units
11:         rt ← sim ⊕ β
12:         cachetree.insert(node(B(x_0, δ), rt, sim, parentnode))
13:         if sim intersects with O then
14:             return: Unsafe
15:         else if rt intersects with O then
16:             Refine cover and add to the coverstack.
17:             The parent node of the refined initial sets is the current node.
18: return: Safe
```

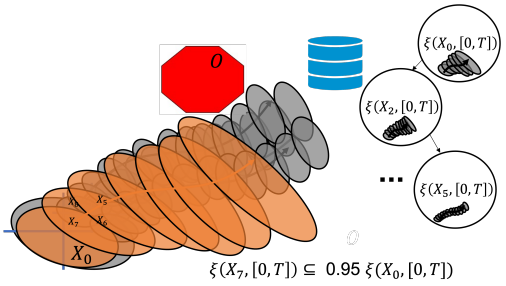
In this section, we add to the ddVer procedure of Section 2.4 a new procedure called symCacheTree for caching and transforming reachtubes using a new data structure *cachetree*. The result is the new ddSymVer algorithm. The symCacheTree procedure, shown in Algorithm 3.1, uses symmetry to save ddVer from computing fresh reachtubes in line 11 in case they can be transformed from already computed and cached reachtubes. Later in Section 3.3.5, we will replace symCacheTree with the more general symGroupCacheTree procedure. An example run of ddSymVer is shown in Figure 3.1.



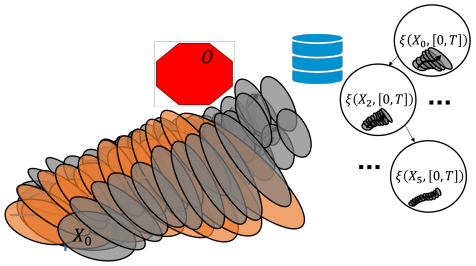
(a) At the beginning ddSymVer, the cache *cachetree* is empty and the reachtube starting from X_0 is computed from scratch, as that in Figure 2.9a. The computed trajectory and reachtube are then saved in the root node of *cachetree*.



(b) Since the reachtube starting from X_0 intersects O, similar to ddVer, ddSymVer partitions the initial set. Since $X_1 \nsubseteq \gamma(X_0)$ and $X_1 \nsubseteq X_0$, ddVer has to compute the reachtube starting from X_1 from scratch as well. The result is saved in a child node of that of X_0 in *cachetree*.



(c) Since $X_7 \subseteq \gamma(X_0)$, ddSymVer checks the safety of the system starting from X_7 by calling symCacheTree. The latter computes the reachtube by transforming the saved reachtube starting from X_0 that is saved in *cachetree* using γ. Then, it checks if the resulting reachtube intersects the unsafe set, and returns the result to ddSymVer. The transformed reachtube is shown in orange.



(d) The algorithm ddSymVer, as ddVer, terminates when the system has been proven to be safe when starting from all parts of the input initial set or a counterexample has been found. Some of the parts would be proven by symCacheTree (or symGroupCacheTree) and the others by computation from scratch, using discrepancy functions, for example.

Figure 3.1: An illustration of a run of ddSymVer on the oscillator system and the corresponding symmetry described in Example 2.1. The parameters of the symmetry of equation (2.4) are chosen to be $a = 0.95$ and $b = 0$, resulting in scaling both dimensions by a 0.95 factor.

3.3.1 The *cachetree* data structure

The data structure *cachetree* is a binary tree which stores reachtubes. Each node *node* in *cachetree* stores an initial set *initset*, a simulation *sim* of duration T from the center of *initset*, and an over-approximation *rt* of $\mathit{ReachTb}(\mathit{initset}, T)$. Moreover, *node* stores pointers *node.left* and *node.right* to the two nodes storing the reachtubes starting from two initial sets partitioning *initset*, if they exist. Otherwise, the two pointers will be set to *Null*. Similarly, *node* stores a pointer *node.parent* to its parent node. The root node of *cachetree* is denoted by *root* and *root.parent* is *Null*.

The *cachetree.insert* function takes as input a *node* object that contains the initial set X_{init}, computed simulation *sim*, the reachtube *rtb*, and the pointer *parentnode* to which the *node* will be added as a child. The pointers to the children nodes in *node* are initialized to *Null*. Then, *cachetree.insert* would assign one of the children pointers of *parentnode* to *node*.

The following proposition states the key invariants of *cachetree*.

Proposition 3.1. *For non-Null nodes:*

$$\mathit{root.initset} = X_0, \tag{3.1}$$

$$\forall\ \mathit{node}, \mathit{node.left.initset} \subseteq \mathit{node.initset}, \tag{3.2}$$

$$\forall\ \mathit{node}, \mathit{node.right.initset} \subseteq \mathit{node.initset}, \tag{3.3}$$

$$\forall\ \mathit{node}, \mathit{node.left.initset} \cap \mathit{node.right.initset} = \emptyset, \text{ and} \tag{3.4}$$

$$\forall\ \mathit{node}, \mathit{node.left.initset} \cup \mathit{node.right.initset} = \mathit{node.initset}. \tag{3.5}$$

That is, the *initset* of the root node is equal to X_0, the given initial set of the system in ddSymVer; each child's *initset* is contained in the *initset* of the parent; the *initset*s of the children partition the *initset* of the parent. Hence, by property (2) of the discrepancy function β (Definition 2.5) it follows that the union of the reachtubes of the children is a tighter over-approximation of the reachtube of the parent, for the same initial set. Since the refinement of the initial set in ddSymVer is done depth-first, *cachetree* is also constructed in the same way.

3.3.2 The ddSymVer procedure

The idea behind both symCacheTree and symGroupCacheTree, which are shown in Algorithms 3.2 and 3.3), respectively, is as follows: given a *cachetree* storing reachtubes as they are computed, an initial set of states $initset_n$ that ddSymVer needs to check the safety for, a symmetry operator γ (or a group of them Γ) for system (2.1) and the unsafe set O, symCacheTree (or symGroupCacheTree) traverses *cachetree* to check if the safety of the system starting from $initset_n$ can be decided by transforming reachtubes stored in *cachetree*.

Before getting into symCacheTree and symGroupCacheTree, we note the simple additions to ddVer that lead to ddSymVer. The new lines in ddSymVer compared to ddVer are 3, 6-9, and 12. First, *cachetree* is initialized to an empty tree (line 3). Then, symCacheTree (or symGroupCacheTree) is used for the safety check (lines 6-9) and fresh reachtube computation is performed only if the check returns inconclusive answer (lines 10-11). In the last case, a fresh reachtube *rt* gets computed in line 11 and inserted as a new node in *cachetree* in line 12.

In brief, symCacheTree (symGroupCacheTree) uses symmetry to save ddSymVer from computing the reachtube $ReachTb(initset_n, T)$ afresh in line 6 from initial set $initset_n$ in the case that safety of $ReachTb(initset_n, T)$ can be inferred by transforming an existing reachtube in *cachetree*. That is, given an unsafe set O, a tree *cachetree* storing reachtubes (previously computed), and a symmetry operator γ (a group of symmetries Γ) for system (2.1), symCacheTree (Algorithm 3.2) or symGroupCacheTree (Algorithm 3.3) checks if the safety of the system when it starts from $initset_n$ can be decided by transforming and combining the reachtubes in *cachetree*.

3.3.3 The symCacheTree procedure

The core of the symCacheTree algorithm shown in Algorithm 3.2 is to answer queries of the form: *can safety be decided from a given initial set* $initset_n$*, by transforming and combining the reachtubes in cachetree?*

They are answered by performing a *breadth-first traversal (BFS)* of *cachetree*. symCacheTree first checks if the γ-transformed *initset* of *root* contains $initset_n$. If not, the transformation of the union of all *initset*s of all nodes in *cachetree* would not contain $initset_n$. In this case we cannot use Corollary 3.1 to get an over-approximation of $ReachTb(initset_n, T)$ and symCacheTree returns Compute

(line 4). If the γ-transformed *initset* of the root does contain $initset_n$, we have at least one tube that over-approximates it which is $\gamma(root.rt)$ by Corollary 3.1. Then, the *root* is inserted to the queue *traversalQueue* that stores the nodes that need to be visited in the BFS.

Then, the algorithm proceeds similar to ddVer. There are two differences: first, it does not compute new reachtubes, it just uses the transformations of the reachtubes in *cachetree*. Second, it refines in BFS manner instead of DFS. In more detail, at each iteration, a node is dequeued from *traversalQueue*. If its transformed initial set $initset_c$ using γ does not intersect with $initset_n$, that means that $\gamma(ReachTb(initset_c, T))$ and $ReachTb(initset_n, T)$ do not intersect. Hence, the node is not useful for this initial set. Also, if the transformed reachtube $\gamma(node.rt)$ does not intersects O, the part of $initset_n$ that is covered by $\gamma(node.initset)$ is safe and no need to refine it more. In both cases, the loop proceeds for the next node (line 10). If the transformed simulation of the node starts from $initset_n$ and hits O, then we have a counter example by Theorem 2.1. Hence, ddSymVer returns Unsafe (line 12). If the transformed reachtube $\gamma(node.rt)$ intersects O, symCacheTree cannot decide if that is because of the overapproximation error, or because of a trajectory that does not start from $initset_n$ or because of one that does. Hence, it needs to refine more. Before refining, it checks if the union of the transformed *initset*s of the children of the current node covers the part of $initset_n$ that was covered by their parent. If that is NOT the case, then part of $initset_n$ cannot be covered by a node with a tighter reachtube. That is because γ is invertible and nodes at the same level of the tree are disjoint. Hence, no node at the same level can cover the missing part. Thus, symCacheTree returns Compute, asking ddSymVer to compute the over-approximation from scratch (line 15). Otherwise, symCacheTree enqueue all the children nodes in *traversalQueue* (line 14).

If *traversalQueue* becomes empty, then symCacheTree has an over-approximation of the reachtube starting from $initset_n$ that does not intersect with O. Hence, it returns Safe in line 16.

3.3.4 Correctness guarantees of symCacheTree

The following two theorems show the correctness guarantees of symCacheTree. Theorem 3.2 shows that if *cachetree* has reachtubes that can prove that the system is safe using γ, it will return Safe. If it has a trajectory that can prove that the

Algorithm 3.2 symCacheTree

1: **input:** $O, \gamma, cachetree, initset_n$
2: $initset_c := cachetree.root.initset$
3: **if** $initset_n \not\subseteq \gamma(initset_c)$ **then**
4: **return:** Compute
5: $traversalQueue := \{cachetree.root\}$
6: **while** $traversalQueue \neq \emptyset$ **do**
7: $node \leftarrow traversalQueue.dequeue()$
8: $initset_c := node.initset$; $\{(R_i, t_i)_{i=0}^{k}\} = node.sim$
9: **if** $\gamma(initset_c) \cap initset_n = \emptyset$ or $\gamma(node.rt) \cap O = \emptyset$ **then**
10: continue
11: **if** $\exists j \mid \gamma(R_j) \cap O \neq \emptyset$ and $\gamma(R_0) \in initset_n$ **then**
12: Return Unsafe
13: **else if** $\gamma(node.initset) \cap initset_n \subseteq \bigcup_i \gamma(node.children[i].initset)$ **then**
14: $traversalQueue.enqueue(\{node.left, node.right\})$
15: **else return:** Compute
16: **return:** Safe

system is unsafe using γ, it will either ask ddSymVer to compute the reachtube from scratch or will return Unsafe. Theorem 3.3 shows that if symCacheTree returns Safe, then the reachtube of the system starting from $initset_n$ does not intersect O. Moreover, if it returns Unsafe, then there exists a trajectory that starts from $initset_n$ and intersects O.

Theorem 3.2 (Completeness)**.** *If there exists a set of nodes S in cachetree with*

$$initset_n \subseteq \cup_{s \in S} \gamma(s.initset) \text{ and } \cup_{s \in S} \gamma(s.rt) \cap O = \emptyset,$$

symCacheTree *will return* Safe*. Also, if there exists a node s in cachetree where* $\gamma(s.sim) \cap O \neq \emptyset$ *and starts from* $initset_n$*, then* symCacheTree *will return* Unsafe *or* Compute*.*

Proof. We will prove the Safe and Unsafe cases separately. But first, note that symCacheTree traverses *cachetree* in a BFS manner and *cachetree* is of finite size. Hence, symCacheTree terminates.

Safe First, the *initset* of the root contains all the *initset*s in the tree. Hence, if the union of the transformation of the *initset*s of the nodes in S under γ contains $initset_n$, then so does $\gamma(root.initset)$. Then, the condition at line 3 will not be satisfied and symCacheTree will not return Compute in line 4.

Second, we prove that symCacheTree will not return Unsafe. Assume that the condition at line 11 was satisfied at a given iteration where some node $node_p$ is the one being visited. Then, $\gamma(node_p.sim)$ starts from $initset_n$ and intersects O. By Theorem 3.1, this is a simulation of the system that starts from $\gamma(node_p.sim.R_0)$. However, by the assumption of the current theorem, there exists a $node_c \in S$ with $node.initset_c$ that contains $\gamma(node_p.sim.R_0)$. Moreover, from Definition 2.5 and ddSymVer, we know that simulation belongs to $node_c.rt$. But, $node_c.rt \cap O = \emptyset$, contradiction. Hence, the condition at line 13 could not be satisfied under the assumptions of the theorem.

Third, we will prove that symCacheTree will not return Compute in line 15. Assume for the sake of contradiction that line 15 was executed at a given iteration and $node_p$ was the node being visited. Then, all the conditions in the while loop were not satisfied. Hence, there exists subset $\gamma(node_p.initset) \cap initset_n$ that is not covered by the transformed initial sets of the children of $node_p$. Since γ is inveritble and the initial sets of the children are disjoint, then that part would not be covered by other nodes in the tree other than $node_p$ and its ancestors. However, all of these parent nodes have *rt* that intersect O for them to got refined in the first place. Hence, none of them belong to S. Therefore, there is no node in S with *initset* that cover that part. That contradicts our assumption that the union of *initset*s of nodes in S contains $initset_n$. Therefore, symCacheTree does not return Compute under the assumptions of the theorem.

Finally, since the algorithm terminates and it would neither return Compute nor Unsafe, it will return Safe.

Unsafe Let $node_p$ be a node in *cachetree* with $\gamma(node.sim)$ that starts from $initset_n$ and intersects O. Then, in the BFS traversal of *cachetree*, the condition at line 9 will be true for none of its ancestors since $node.sim \subseteq node.rt$. Hence, symCacheTree will either return Compute or Unsafe before visiting $node_p$ or it will visit it and line 11 will be true. In that case, symCacheTree will return Unsafe. □

Theorem 3.3 (Soundness)**.** symCacheTree *is sound: if it returns* Safe*, then the reachtube* $ReachTb(initset_n, T)$ *does not intersect O and if it returns* Unsafe*, then there exists a trajectory starting from* $initset_n$ *that enters the unsafe set.*

Proof. We will again separate the proofs for the Safe and Unsafe cases.

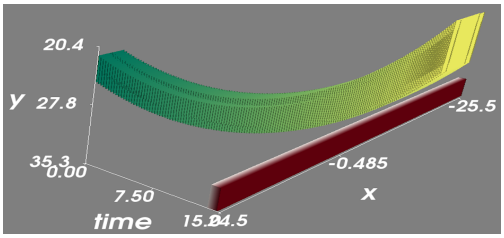
Safe If symCacheTree returns Safe, it means that the condition in line 3 is not true. Moreover, in every iteration of the while loop, either the condition in line 9 or the one in line 13 is satisfied. When the one in line 9 is satisfied, it means either that the node is useless for computing the reachtube starting from $initset_n$ or the reachtube does not intersect O. If the condition at line 13 was satisfied, it adds children with union of the transformed *initset*s cover the part of $initset_n$ that was covered by the parent *initset*. Hence, none of the states in $initset_n$ will not be checked the safety for. Thus, every state in $initset_n$ will belong to some $node.initset$ for which line 9 will be true. Therefore, the system starting from $initset_n$ is safe.

Unsafe Assume that the node visited in the iteration at which symCacheTree returned Unsafe be $node_p$. Then, $\gamma(node_p.sim)$ starts from $initset_n$ and intersects O. Moreover, by Theorem 2.1, it is the simulation of the system starting from $node_p.sim.R_0$. Hence, it is a counter example and the system is indeed unsafe. □

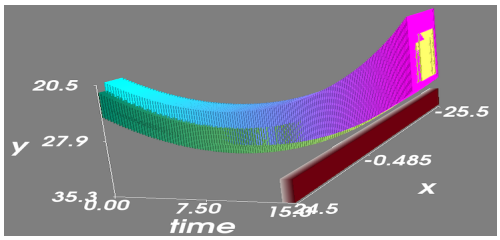
In summary, symCacheTree shows that a single symmetry γ could decrease the number of fresh reachtube computations needed for verification. Next, we revisit Example 2.3 to illustrate the need for multiple symmetry maps.

Circular orbits and scaling symmetry The linear system in Example 2.3 has circular orbits. Consider the initial set $X_0 = [[21.5, 21.5], [24.5, 24.5]]$, the unsafe set $x_2 \geq 32$ after $t = 1.4s$, and the time bound $T = 1.5s$. Any matrix B that commutes with A, the RHS of the differential equation, is a symmetry transformation. However, once this matrix is fixed, we do not change it as per symCacheTree. Any diagonal matrix that commutes with A has equal diagonal elements. Such a matrix would scale x_1 and x_2 by the same factor. Hence, applying B to any axis aligned box would either scale the box up or down on the diagonal. That means applying B to X_0 wouldn't contain the upper left or bottom right partitions, but only possibly the bottom left corner. With $B = [[0.95, 0], [0, 0.95]]$, only one out of 7 reachtubes is obtained via transformation (first row of Table 3.1).

That is because we are using a single transform which leaves symCacheTree useless in most of the input cases. Figure 3.2a shows the reachtube (colored green to yellow) computed using ddVer, unsafe set (brown). Figure 3.2b shows the reachtube computed using ddSymVer. The part of the reachtube that was computed using symmetry is colored between blue and violet. The other part is still between yellow and green. Only the upper left corner has been transformed



(a) Reachtube computed using DryVR, a tool that implements ddVer.



(b) Reachtube computed using ddSymVer implemented on top of DryVR.

Figure 3.2: Reachtubes of the oscillator system of Example 2.1. Tube shown in Green-to-Yellow is the reachtube computed from scratch in line 11 of ddSymVer. Tube shown in Blue-to-Violet is the reachtube retrieved from *cachetree* by the call to symCacheTree in line 6 of ddSymVer. Brown bar is the unsafe set O. Seven reachtubes had to be computed from different parts of the initial set to verify the safety of the system. ddSymVer was able to retrieve one of the seven reachtubes from *cachetree* using γ that scales the state with the factor 0.95. The other six reachtubes had to be computed from scratch.

instead if being computed. Next, we present symGroupCacheTree, a generalization of symCacheTree that uses a group of symmetries aiming for a bigger ratio of transformed to computed reachtubes.

3.3.5 The symGroupCacheTree procedure

Procedure symGroupCacheTree shown in Algorithm 3.3 is a generalization of the symCacheTree algorithm that uses a group of symmetries instead of a single one. The symGroupCacheTree procedure still does BFS over *cachetree*, keeps track of the parts of the input initial set $initset_n$ that are not proven safe (line 11); returns Unsafe with the same logic (line 13), and return Compute in case there are parts of $initset_n$ that are not proven safe nor have refinements in *cachetree* (line 17).

The key difference from symCacheTree is that different transformations may be useful at different nodes. This leads to the possibility of multiple nodes in *cachetree*, that are not ancestors or descendants of each other, covering the same parts of $initset_n$ under different transformations. Recall that in symCacheTree, only ancestors cover the parts of $initset_n$ that are covered by their descendants since γ is invertible. Hence, it was sufficient to not add the children of a node to *traversalQueue* to know that the part it covers, by transforming its initial set, from $initset_n$ is safe (line 10 of symCacheTree). However, it is not sufficient in symGroupCacheTree since there might be another node that covers the same part

Algorithm 3.3 symGroupCacheTree

1: **input:** O, Γ, *cachetree*, $initset_n$
2: $initset_c := cachetree.root.initset$
3: **if** $initset_n \not\subseteq \cup_{\gamma\in\Gamma}\gamma(initset_c)$ **then**
4: **return:** Compute
5: $leftstates \leftarrow initset_n$
6: $traversalQueue := \{cachetree.root\}$
7: **while** $traversalQueue \neq \emptyset$ and $leftstates \neq \emptyset$ **do**
8: $node \leftarrow traversalQueue.dequeue()$
9: $initset_c := node.initset$; $\{(R_i, t_i)_{i=0}^{k}\} = node.sim$
10: $K = \{x : \exists\gamma \in \Gamma, x \in \gamma(initset_c)$ and $\gamma(node.rt) \cap O = \emptyset\}$
11: $leftstates \leftarrow leftstates \backslash K$
12: **if** $\exists\gamma \in \Gamma, j \mid \gamma(R_j) \cap O \neq \emptyset$ and $\gamma(R_0) \in leftstates$ **then**
13: Return Unsafe
14: **if** $len(node.children) > 0$ **then**
15: $traversalQueue.enqueue(node.children)$
16: **if** $leftstates \neq \emptyset$ **then**
17: **return:** Compute
18: **return:** Safe

of $initset_n$ and has a transformed reachtube that intersects O, hence refining what already has been proven to be safe. The solution is to remove explicitly from $initset_n$ what has been proven to be safe (line 11). The resulting set may not be convex but can be stored as a set of polytopes. Moreover, symGroupCacheTree cannot return Compute when the transformed reachtube of a visited node intersects O and its children initial sets do not contain the part it covers from $initset_n$ as in line 15 of symCacheTree. That is because other nodes may cover that part utilizing the multiple symmetries. Hence, it cannot return Compute unless it traversed the whole tree and parts of $initset_n$ remained not be proven to be safe. We show that symGroupCacheTree has the same guarantees as symCacheTree in the following two theorems.

Theorem 3.4 (Completeness)**.** *If there exists a set of nodes S in cachetree, where each $s \in S$ has a corresponding set of transformations $\Gamma_s \subseteq \Gamma$, such that*

$$initset_n \subseteq \cup_{s\in S, \gamma_s\in\Gamma_s}\gamma_s(s.initset) \text{ and } O \cap \cup_{s\in S,\gamma_s\in\Gamma_s}\gamma_s(s.rt) = \emptyset,$$

symCacheTree *will return* Safe*. Also, if there exists a node s in cachetree and a $\gamma \in \Gamma$, where $\gamma(s.sim)$ intersects U and starts from $initset_n$, then* symGroupCacheTree *will return* Unsafe *or* Compute*.*

Proof. The proof is similar to that of Theorem 3.2. First, it will terminate since it is a BFS traversal of a tree of finite size. We again prove the safe and unsafe cases separately.

Safe symGroupCacheTree will not return Compute in line 4 since the union of the mapped initial set of the root using all $\gamma \in \Gamma$ contains $\cup_{s \in S, \gamma_s \in \Gamma_s} \gamma_s(s.initset)$ and that contains $initset_n$ by the assumption of the theorem. Hence, the condition at line 3 will not be satisfied.

symGroupCacheTree will not return Unsafe either. Assume for the sake of contradiction that there exists a *node* s_1 in *cachetree* and a $\gamma_{s_1} \in \Gamma$ such that the transformed simulation starts from $initset_n$ and intersects O. But, by the assumption of the theorem, there exists a *node* $s_2 \in S$ and a $\gamma_{s_2} \in \Gamma_{s_2}$ such that the transformed reachtube contains the transformed simulation of s_1 and does not intersect the unsafe set. Contradiction.

symGroupCacheTree will not return Compute in line 17, since otherwise it would have exited the while loop as *traversalQueue* is empty. However, a node is not visited in the BFS if and only if *leftstates* got empty before the search reached it. Hence, for any node $s \in S$, either the while loop terminated before reaching it because *leftstates* got empty or it gets visited and it removes $\cup_{\gamma_s \in \Gamma_s} \gamma_s(s.initset)$ from *leftstates* since the corresponding reachtubes do not intersect O. Since the union of this set over all nodes in S contains $initset_n$, *leftstates* will be empty after the exit of the while loop. Hence, symGroupCacheTree will return Safe.

Unsafe If there exists a node $s_1 \in S$ and a $\gamma \in \Gamma_{s_1}$ such that its transformed simulation intersects O, then for any *node* s_2 in *cachetree*, there exists no $\gamma \in \Gamma_{s_2}$ such that the transformed initial set of s_2 contains the transformed initial state of the simulation of s_1 while the transformed reachtube of s_2 does not intersect the unsafe set. Hence, *leftstates* will never be empty. Therefore, symGroupCacheTree will return Compute or Unsafe. □

Theorem 3.5 (Soundness)**.** symGroupCacheTree *is sound: if it returns* Safe*, then the reachtube* $ReachTb(initset_n, T)$ *does not intersect* O *and if it returns* Unsafe*, then there exists a trajectory starting from* $initset_n$ *that enters the unsafe set.*

Proof. If symGroupCacheTree returns Safe, then *leftstates* is empty at line 16. Hence, for every $x \in initset_n$, there exists a $\gamma \in \Gamma$ and a *node* s in *cachetree* such

that $x \in \gamma(s.initset)$ and $\gamma(s.rt) \cap U = \emptyset$. By Remark 3.1, $\gamma(s.rt)$ is an overapproximation of $ReachTb(\gamma(s.initset), T)$. Since $\xi(x, [0,T]) \in ReachTb(\gamma(s.initset), T)$, it is safe. Therefore, the system is safe starting from $initset_n$.

If symGroupCacheTree returns Unsafe, it means there exists a $\gamma \in \Gamma$ and a simulation with mapped initial state under γ belongs to $initset_n$ and mapped reachable state in the simulation that intersect O. By Theorem 2.1, it is the simulation of the system starting from the mapped initial state. Hence, we have a trajectory that starts from $initset_n$ and intersects O. Hence, the system is Unsafe. □

The new challenge in symGroupCacheTree is in computing the union at line 3, computing K in line 10 and in the $\exists$ in line 12. These operations depend on Γ if it is finite or infinite and on how easy is it to search over it. We revisit the arbitrary translation symmetry of the car model from Section 2.2.3 to show that these operations are easy to compute in certain cases.

3.3.6 Revisiting arbitrary translations

Recall that the car model in Example 2.3 in Section 2.2.3 is equivariant to all translations in its position. In this section, we show how to apply symGroupCacheTree not just for that particular example, but to arbitrary systems having the same property that some state variables do not appear on the right-hand-side (RHS) of the system's ODE. Let D be the set of components of the states that do not appear on the RHS of system (2.1) and Γ be the set of all translations of the components in D. To check the *if* condition at line 3, we only have to check if $initset_c$ projected to the dimensions not in D, i.e. $[\dim(X)] \backslash D$, contains $initset_n$ projected to the same components. Since if it is true, $initset_c$ can be translated and scaled arbitrarily in its components in D so that the union over Γ in line 3 contains $initset_n$.

More specifically, given two initial sets X_0 and X_0' and the reachtube starting from X_0', we compute $Y \subseteq X$ such that $X_0' \downarrow_D \oplus Y = X_0 \downarrow_D$. Then, if $X_0 \downarrow_{[\dim(X)] \backslash D} \subseteq X_0' \downarrow_{\dim(X) \backslash D}$, by Corollary 3.1, we can use Y to compute an over-approximation of $ReachTb(X_0, T)$ by computing $ReachTb(X_0', T) \oplus Y$. Then, let Y be defined to satisfy $initset_c \downarrow_D \oplus Y = leftstates \downarrow_D$, in line 10. We set K in line 10 of symGroupCacheTree to be equal to $initset_c \oplus Y$ if $node.rt \oplus Y \cap O = \emptyset$ and to $\emptyset$, otherwise.

Setting K to $\emptyset$ in case the transformed reachtube intersects the unsafe set is different from what is implemented in line 10. Setting K to $\emptyset$ means that either the whole

set of transformations $\gamma \in \Gamma$ that are defined by Y are used or none. This difference in implementation will not affect the soundness and completeness guarantees of symGroupCacheTree, but will only affect its performance. The new implementation will return Compute more often than the one shown in Algorithm 3.3. Another alternative implementation would be refining Y to only cover the part of $initset_n$ where the transformed reachtube using the refined Y does not intersect O. We will leave such an optimization for future work.

To check the $\exists$ operator in line 12, we can follow the same procedure that we described for *node.rt* since a simulation is a special case of a reachtube as described in Section 3.3.5. Basically, we compute Y such that $node.sim.R_0 \downarrow_D \oplus Y = leftstates \downarrow_D$. The new condition in line 12 would be then: if $\exists j, R_j \oplus Y \cap O \neq \emptyset$. Notice that we dropped the predicate $\gamma(R_0) \in leftstates$ from the condition since we know that $R_0 \in leftstates$ and Y is bloating it to the extent it is equal to *leftstates*.

Optimized symGroupCacheTree **for arbitrary translations** The size of $X_0' \downarrow_D$ above does not matter. Even if $X_0' \downarrow_D$ is just a point, one can compute a Y so that $X_0' \downarrow_D \oplus Y$ equals $X_0 \downarrow_D$. Hence, for a given initial set of states $X_0 \subseteq X$, instead of computing $ReachTb(X_0, T)$, we can compute $ReachTb(X_0 \downarrow_D, T)$ and then compute a Y such that $X_0 \downarrow_D \oplus Y = X_0$. That Y will be equal to $X_0[D]$ (recall hyperrectangles projection definition from Section 2.1). The reachtube $ReachTb(X_0, T)$ would be then equal to $ReachTb(X_0 \downarrow_D, T) \oplus Y$. This decreases the number of dimensions that the system need to refine by $|D|$. This is in contrast with what is done in symGroupCacheTree where the reachtubes are computed without changing the given initial set $initset_n$. This new method further accelerates the verification of the car models to the extent that the verification time becomes a 1s when they take an hour on DryVR, as shown in Section 3.4. We call this algorithm TransOptimized and refer to it as version 2 of symGroupCacheTree when applied to arbitrary translation invariance transformations.

3.4 Experimental evaluation

We implemented symCacheTree and symGroupCacheTree in Python 2.7 on top of DryVR [2]. We augmented DryVR and implemented ddSymVer. In our experiments, we only consider continuous-time dynamical systems of the form of system (2.1).

[2] `https://github.com/qibolun/DryVR_0.2`

In this section, we present the experimental results on several examples shown in Section 2.2.3 using symCacheTree and symGroupCacheTree. The transformations used are linear. Two of the systems are linear and one is non-linear. The results of the experiments are shown in Table 3.1. The experiments were ran on a computer with specs shown at the end of this chapter. In the reachtube plots we use the green-to-yellow colors if it was computed from scratch, the blue-to-violet colors if it was computed using symmetry transformations, and the white-to-red colors for the unsafe sets.

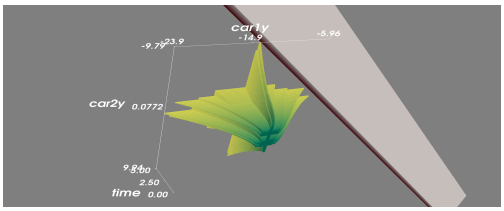
Verifying non-convex initial sets

ddVer assumes that the initial set X_0 of the continuous-time system (2.1) is a single hyperrectangle. However, this assumption hinders the use of some useful transformations such as permutation. For example, consider the two cars system in Example 2.4 moving straight and breaking with the same deceleration, i.e. u is zero and a is the same for both. Recall that this system is equivariant with respect to switching the states of the two cars in the system's state. The system is unsafe if the cars are too close to each other. Assume that the initial set of the two-cars system is a single hyperrectangle: $X_0 = [[l_1, l_2], [u_1, u_2]]$, where l_1, l_2, u_1, and $u_2 \in X$. If the two hyperrectangles $[l_1, u_1]$ and $[l_2, u_2]$ do not intersect, γ would not be useful since for any $Y \subseteq X_0$, $\gamma(Y) \cap X_0 = \emptyset$. Now consider the case where X_0 is the union of two hyperrectangles: $[[l_1, l_2], [u_1, u_2]] \cup [[l'_1, l'_2], [u'_1, u'_2]]$, where $[l'_1, u'_1] \cap [l_2, u_2] \neq \emptyset$ and $[l'_2, u'_2] \cap [l_1, u_1] \neq \emptyset$. Then, the reachtube starting from $[l'_1, u'_1] \cap [l_2, u_2]$ for the first car and $[l'_2, u'_2] \cap [l_1, u_1]$ for the second one can be computed by switching the states of the two cars in the reachtube starting from $[l'_2, u'_2] \cap [l_1, u_1]$ for the first car and $[l'_1, u'_1] \cap [l_2, u_2]$ for the second one.

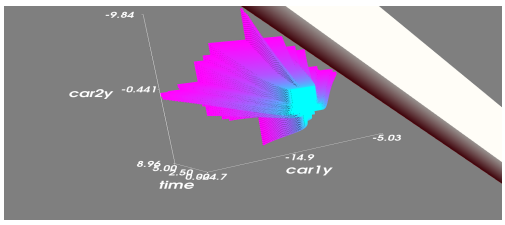
We implemented ddSymVer for initial sets consisting of unions of hyperrectangles by calling ddSymVer sequentially for each hyperrectangle. The computed reachtubes after each call are cached in a *cachetree* and used in later calls.

Cars and permutation invariance

For the two-cars example (Example 2.4), we ran ddSymVer for the initial set having $(x_1[0:1], x_2[0:1]) \in [[0, -2.42, 0, -22.28], [2, 3.93, 0.1, -12.82]]$, a time bound of 5s, and the unsafe set being $|x_1[1] - x_2[1]| < 5$. All computed reachtubes were cached in a *cachetree* which we saved on the hard-drive. It returned Safe. Then, we used it in symCacheTree to verify the system starting from



(a) Reachtube computed using ddSymVer implemented on top of DryVR. The resulting *cachetree* from the reachtube computation was saved on hard-drive. We used it verify the system from a different initial set of states shown on the right.



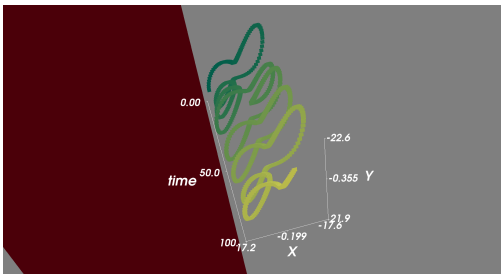
(b) Reachtube computed using ddSymVer implemented on top of DryVR. The whole reachtube was computed by retrieving several reachtubes from the saved *cachetree* and then permuting the states of the two cars in the reachtubes' hyperrectangles, i.e. applying the permutation symmetry to the retrieved reachtubes.

Figure 3.3: Reachtubes for the two-cars system of Example 2.4 when using states permutation symmetry. The figures show the projection of the reachtube to the longitudinal positions of the two cars on the road and to time. Tubes shown in Green-to-Yellow are the reachtube computed from scratch in line 11 of ddSymVer. Tubes shown in Blue-to-Violet are the reachtube retrieved from *cachetree* by the call to symCacheTree in line 6 of ddSymVer. Brown bar is the unsafe set O.

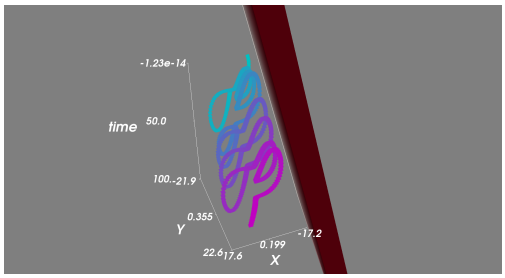
$[[0,-22.28,0,-2.42],[0.1,-12.82,2,3.93]]$. The resulting *cachetree* was around 20 GB, and traversing it while transforming the stored reachtubes takes much longer than computing the reachtube directly. We halted it manually and tried a smaller initial set: $[[0.01,-14.2,0.01,1.4],[0.1,-13.9,2,3.9]]$ using the same *cachetree* which returned Safe from the first run after 93s; the output is shown in Figures 3.3a and 3.3b. Figure 3.3a shows the tube when computed by ddVer and Figure 3.3b when computed by ddSymVer. Figure 3.3b has only blue-to-violet colors since it was all computed using a symmetry transformation: the permutation of the cars states.

Lorenz attractor and oscillator revisited

Recall from Section 2.2.3 that the Lorenz attractor's symmetry map is $x \rightarrow (-x[0],-x[1],x[2])$. So for any given hyper-rectangle initial set $X_0 = [[l_0,l_1,l_2],[u_0,u_1,u_2]]$ and a corresponding over-approximation of the reachtube, we automatically get an over-approximation of the reachtube with the initial set $[[-u_0,-u_1,l_2],[-l_0,-l_1,u_2]]$. We called ddSymVer to verify that the system starting from the



(a) Reachtube computed using ddSymVer implemented on top of DryVR. The resulting *cachetree* from the reachtube computation was saved on hard-drive. We used it verify the system from a different initial set of states shown on the right.

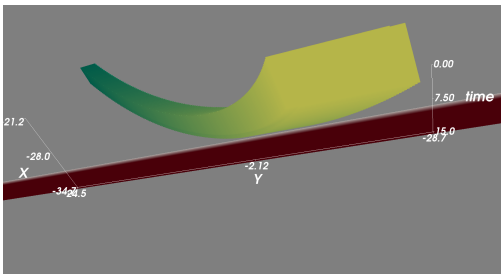


(b) Reachtube computed using ddSymVer implemented on top of DryVR. The whole reachtube was computed by retrieving several reachtubes from the saved *cachetree* and then reflecting the states of the Lorenz attractor in the reachtubes' hyperrectangles, i.e. applying the reflection symmetry to the retrieved reachtubes.

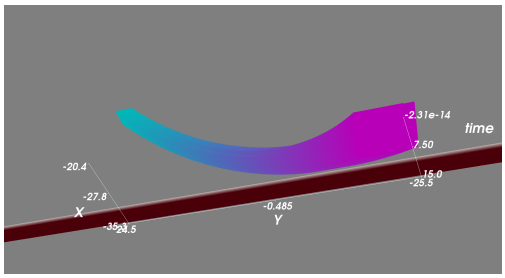
Figure 3.4: Reachtubes for Lorenz attractor of Example 2.2. The figures show the projection of the reachtube to the first two dimensions of the system's state and to time. Tube shown in Green-to-Yellow is the reachtube computed from scratch in line 11 of ddSymVer. Tube shown in Blue-to-Violet is the reachtube retrieved from *cachetree* by the call to symCacheTree in line 6 of ddSymVer. Brown bar is the unsafe set O.

initial set $[[14.9, 14.9, 35.9], [15.1, 15.1, 36.1]]$ and evolving for $T = 10s$ will not intersect the unsafe set $x[0] \geq 20$: ddSymVer returned Safe. We used the resulting *cachetree* in symCacheTree to verify that the system starting from the new initial set $[[-15.09, -15.09, 35.91], [-14.91, -14.91, 36.09]]$ will not intersect the same unsafe set within the same time bound. The resulting statistics on the number of reachtubes computed and total verification time are shown in Table 3.1. Lorenz1 is the one corresponding to the first initial set and Lorenz2 to the for which we use permutation symmetry. The resulting reachtubes are shown in Figure 3.4a for Lorenz1 and Figure 3.4b for Lorenz2.

We revisit the oscillator example from Section 3.3.3 and test the performance of symCacheTree with the transformation being: $\gamma : x \rightarrow [-x[1], x[0]]$ instead of the scaling one. Then, we compute the reachtube starting from the same initial set as that in Section 3.3.3 and saved its *cachetree*. After that, we used ddSymVer with symCacheTree to verify the system starting from the new initial set $[[-24.49, 21.51], [-21.51, 24.49]]$ with a time bound of 1.5s. The statistics are shown in Table 3.1. The resulting reachtubes are shown in Figures 3.5a and 3.5b. Again, the whole tube is blue-to-violet in Figure 3.5b since it is computed fully by transforming parts of *cachetree*.



(a) Reachtube computed using ddSymVer implemented on top of DryVR. The resulting *cachetree* from the reachtube computation was saved on hard-drive. We used it verify the system from a different initial set of states shown on the right.



(b) Reachtube computed using ddSymVer implemented on top of DryVR. The whole reachtube was computed by retrieving several reachtubes from the saved *cachetree* and then applying the symmetry $\gamma : x \rightarrow [-x[1], x[0]]$ to the retrieved reachtubes.

Figure 3.5: Reachtubes of the oscillator system of Example 2.1. Tube shown in Green-to-Yellow is the reachtube computed from scratch in line 11 of ddSymVer. Tube shown in Blue-to-Violet is the reachtube retrieved from *cachetree* by the call to symCacheTree in line 6 of ddSymVer. Brown bar is the unsafe set O.

In all of the previous examples, ddVer was faster than ddSymVer since a single symmetry was used and the refinements are not large enough so that the ratio of transformed reachtubes to computed ones is large enough to account for the overhead added by the checks of symCacheTree. This can be improved by using a group of transformations, i.e., using symGroupCacheTree, storing compressed reachtubes, and optimizing the code.

Cars and general translation

Finally, we ran ddSymVer with the two versions of symGroupCacheTree for translation invariance described in Section 3.3.6 on three different scenarios of the 2-cars Example 2.4: both are braking (*bb*), both are at constant speed (*cc*), and one is breaking and the other at constant speed (*bc*). In all of them, the time bound is $T = 5s$ and the unsafe set is $|y_1 - y_2| < 5$. The first two cases were safe while the third was not. DryVR timed out on the *cc* case as mentioned previously in the permutation case while both versions of translation invariance algorithms were able to terminate in few seconds. The two versions of the algorithm gave the same result as DryVR while being orders of magnitude faster on the *bb* and *bc* cases. Moreover, the second version, where the initial set is a single point in the components in D, is an order of magnitude faster than the first version, where

symGroupCacheTree is used without modifications that reduce the initial set before reachtube computations.

The figures for the two-cars example with arbitrary translation invariance using the two versions of the algorithm are shown in Figures 3.6a-3.6d.

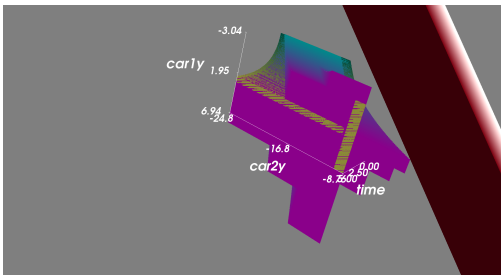
Computer Specifications For Experiment's Runtime Results

The specifications of the desktop used for the experiments are as follows:

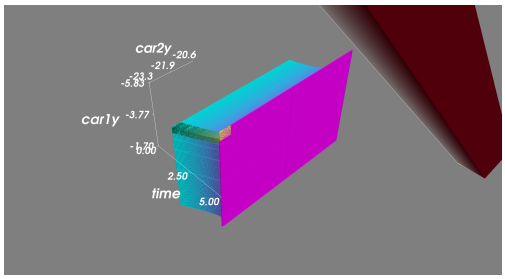
- Processor: Dual Intel Xeon Silver 4110 2.1 GHz, 3.0 GHz Turbo, 8 C, 9.6 GT/s 2 UPI, 11 MB Cache, HT (85 W) DDR4-2400
- Operating System: Ubuntu Linux 16.04
- Graphics Card: NVIDIA® Quadro® P5000 16 GB, 4 DP, DL-DVI-D, (7X20T)
- Memory: 32 GB 4x8 GB DDR4 2666 MHz RDIMM ECC
- Boot Drives (PCIe SSDs): Dell Ultra-Speed Drive Duo PCIe SSD x8 Card, 2 M.2 256 GB PCIe NVMe Class 40 Solid State Drive
- Hard Drive: 3.5" 1 TB 7200 rpm SATA Hard Drive

3.5 Conclusions

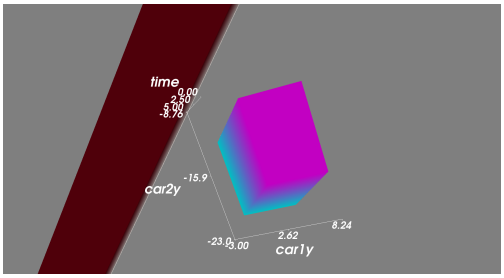
Equiavariant dynamical systems have groups of symmetry transformations that map solutions to other solutions. We use these transformations to map reachtubes to other reachtubes. Based on this, we presented algorithms (symCacheTree and symGroupCacheTree) that use symmetry transformations, to verify the safety of equivariant systems by transforming previously computed reachtubes stored in a tree structure representing refinements. We use these algorithms to augment data-driven verification algorithms to reduce the number of reachtubes need to be computed. We implemented the algorithms and tried them on several examples showing significant improvement in running times. In the next chapter, we discuss an extension of the results of this chapter to hybrid systems.



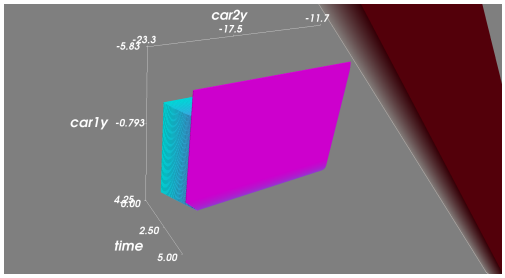
(a) (cc v1) Reachtube computed using ddSymVer implemented on top of DryVR while using symGroupCacheTree instead of symCacheTree.



(b) (bb v1) Reachtube computed using ddSymVer implemented on top of DryVR while using symGroupCacheTree instead of symCacheTree.



(c) (cc v2) Reachtube computed using ddSymVer implemented on top of DryVR while using the optimized version of symGroupCacheTree instead of symCacheTree.



(d) (bb v2) Reachtube computed using ddSymVer implemented on top of DryVR while using the optimized version of symGroupCacheTree instead of symCacheTree.

Figure 3.6: Reachtubes for the two-cars system of Example 2.5 while using only the translation invariance symmetry of both cars' positions. The figures show the projections of the reachtubes to the longitudinal positions of the two cars on the road and to time. Left column shows the reachtubes of the system when both cars are cruising while the right one shows the reachtubes of the system when both cars are braking. First row shows the results when using Version 1 of ddSymVer with symGroupCacheTree while the second one shows the results when using Version 2. Tubes shown in Green-to-Yellow are the reachtube computed from scratch in line 11 of ddSymVer. Tubes shown in Blue-to-Violet are the reachtube retrieved from *cachetree* by the call to symCacheTree in line 6 of ddSymVer. Brown bar is the unsafe set O. Version 2 of the symGroupCacheTree reduces the initial set first using symmetries before computing the reachtube. The ranges of initial positions of the two cars in the initial set of states of the system were reduced to a single point before computing the reachtube. That resulted in a reachtube of the system being a single simulation in the position coordinates. The computed reachtube then gets bloated using symmetries to get the reachtube starting from the original initial set with the intended ranges of cars' initial positions. That is the reason that the second row shows reachtubes in Blue-to-Violet: the reachtube computed from scratch is a single simulation when projected to the cars' longitudinal positions while the transformed tube bloats the computed tube using Y.

Table 3.1: Results. Columns 3-5: number of times symCacheTree (or symGroupCacheTree) returned Compute, Safe, Unsafe, resp. Number of transformed reachtubes used in analysis (SRefs), time (seconds) to verify with DryVR+symmetry (DryVR+sym), total number reachtubes computed by DryVR (NoSRefs), time to verify with DryVR.

Model	γ	Compute	Safe	Unsafe	SRefs	DryVR+sym	NoSRefs	DryVR
oscillator1	$(0.95x_1, 0.95x_2)$	5	1	0	6	1.78	7	0.54
oscillator2	$(-x_2, x_1)$	0	1	0	7	8.23	3	0.21
Lorenz1	$(-x, -y, z)$	N/A	N/A	N/A	N/A	N/A	3	4.67
Lorenz2	$(-x, -y, z)$	0	1	0	1	33.28	1	4.63
bb2	Perm. Inv.	0	1	0	467	88.35	120	34.47
bb (v1)	Trans. Inv.	10	10	0	165	26.28	12621	4034.55
cc(v1)	Trans. Inv.	19	21	0	545	64.36	N/A	OOM
bc(v1)	Trans. Inv.	24	19	1	639	80.48	3428	1027.18
bb (v2)	Trans. Inv.	0	1	0	1	1.16	12620	4034.55
cc (v2)	Trans. Inv.	0	1	0	1	1.16	N/A	OOM
bc (v2)	Trans. Inv.	0	0	1	1	0.39	3428	1027

Chapter 4

Symmetry for Multi-Agent Safety Verification

In this chapter, we show that symmetry transformations and caching can enable scalable, and possibly unbounded, verification of multi-agent systems. It is based on a conference paper presented in the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) in 2020 in Dublin, Ireland [89] [1]. Symmetry transformations map any solution of the system to another solution. We show that this property can be used to transform cached reachsets to compute new reachsets, for hybrid and multi-agent models. We develop a notion of a *virtual system* which defines symmetry transformations for a broad class of agent models that visit waypoint sequences. Using this notion of a virtual system, we present a prototype implementation CacheReach that builds a cache of reachsets, in a way that is agnostic of the representation of the reachsets and the reachability analysis method used. Our experimental evaluation of CacheReach shows up to 64% speedup in safety verification computation time on multi-agent systems with three-dimensional linear and four-dimensional nonlinear fixed-wing aircraft models following sequences of waypoints. These experiments and our theoretical results illustrate the potential benefits of using symmetry-based caching in the safety verification of multi-agent systems.

4.1 Overview

In Chapter 3, we showed how symmetry of autonomous dynamical systems can be used to accelerate their safety verification. In this chapter, we show how symmetry of *parameterized* dynamical systems can be useful to accelerate the safety verification of non-interacting multi-agent systems. An example system is shown in Figure 4.1.

The dynamics of an agent are modeled using a hybrid automaton (Section 2.2.4).

[1] This work is in collaboration with Navid Mokhlesi and Chuchu Fan.

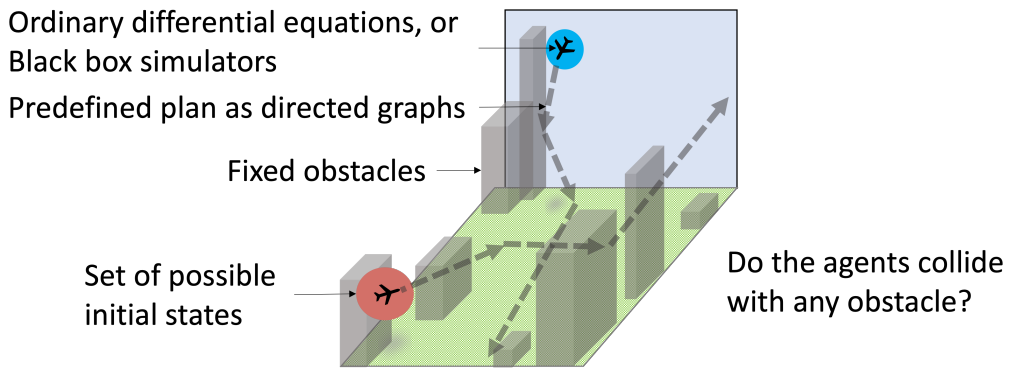


Figure 4.1: Example of a safety verification problem tackled in this chapter: checking if multiple aircraft following segments of waypoints with uncertain initial states may collide with each other or with any other obstacle in the shared environment. The continuous dynamics of each aircraft following one of the segments can be described using the ODE in Example 2.6.

For the broad swathe of dynamical systems modeling agents visiting sequences of waypoints, each segment of waypoints can be modeled as a mode in the automaton. The edges of the automaton model the consecutive segments in the plan.

The challenge in such a safety verification problem is that its dimensionality grows rapidly with the number of agents. Existing reachability analysis-based safety verification methods for hybrid automata would compute the reachtube for each agent separately, if they utilized the non-interacting assumption. Figure 4.2 shows a sketch of reachtubes for the aircraft scenario of Figure 4.1. Otherwise, if the multi-agent system is modeled as a single hybrid automaton, the state space of the system would be the union of the state spaces of all agents. Recall from Section 2.4, that ddVer decomposes the initial set in each dimension to refine the reachtube. Thus, linear increase in the dimension of the state space exponentially increases the number of reachtubes that have to be computed. Even if the reachtube of each agent is computed separately, it has to be generated sequentially, one mode at a time. If the reachtubes of different agents turn out to be intersecting with each other or with the fixed obstacles in the environment, but without a counter-example, their reachtubes would be refined. That requires partitioning to their initial sets and further computations of reachtubes. Thus, adding one agent polynomially increases the number of reachtubes that have to be computed.

However, often agents share the same dynamics. For instance, both fixed-wing aircraft in Figure 4.1 might share the same dynamics, but have different initial states and follow different sequences of waypoints. This commonality has been exploited

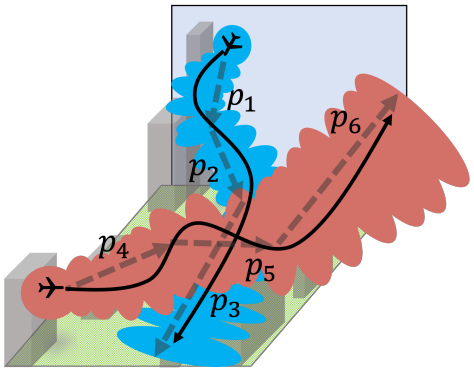


Figure 4.2: Sketch of reachtubes that would be computed for the aircraft in Figure 4.1 using existing tools such DryVR [115], C2E2 [16], and Flow* [17].

in developing specialized proof techniques [125]. For reachability analysis, using symmetry transforms, reachtubes of one agent in one mode can be used to get the reachtubes of other modes and even other agents. In this chapter, we present an algorithm symComputeReachTb (Algorithm 4.1) which caches and reuses per-mode reachtubes, avoiding repeating expensive reachability computations when verifying the safety of multi-agent systems.

Going well beyond the previous theory of Chapter 3, we show how symmetry of parameterized dynamical systems can be used to compute reachtubes of different modes (Theorem 4.1).

We develop a notion of *virtual system* (Section 4.3), which is a unified coordinate system according to which symComputeReachTb saves the computed reachtubes in the cache for efficient retrieval. The virtual system for the system of Figure 4.1 is shown in Figure 4.3. The proposed algorithm symComputeReachTb constructs the virtual system using a user-provided family of symmetry transformations, a pair for each mode for each agent, which we call the *virtual map*. The first transformation in the pair corresponding to a mode in the virtual map would transform the state to a symmetric state. The second transformation would transform the mode to a symmetric, representative, mode. The algorithm symComputeReachTb transforms every per-mode reachtube it computes during the safety verification of the system, as well as its mode, using their corresponding transformations in the virtual map. The representative mode would be the key to the cell in the cache in which symComputeReachTb saves the transformed reachtube. The concept of the virtual system helps the user to select a virtual map that maximizes the number of modes with the same representative mode. We show how this concept can make it possible to verify systems over unbounded time and with infinite number of agents

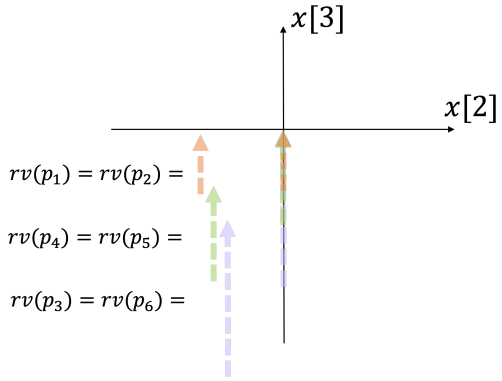


Figure 4.3: Example virtual system for the multi-agent system shown in Figures 4.1 and 4.2 constructed using translational and rotational symmetries. Since p_1 and p_2, p_4 and p_5, and p_3 and p_6 have equal lengths, they can be rotated and translated to three representative segments shown in orange, green and violet. The map rv transforms the segments of the original system of Figure 4.1 to their representative ones in the virtual system. Further details are discussed in Example 4.1.

(Theorem 4.5), provided that no new unproven scenarios appear for the virtual system.

We present a prototype implementation of symComputeReachTb—CacheReach. Our algorithm symComputeReachTb and its implementation CacheReach are agnostic of the representation of the reachtubes and the reachability analysis subroutine. Therefore, any of the ever-improving libraries can be plugged-in for per-mode reachtube computations.

Our experimental evaluation of CacheReach shows safety verification computation time savings of up to 64% on scenarios with multiple agents with three-dimensional linear and four-dimensional nonlinear fixed-wing aircraft model following sequences of waypoints. These savings illustrate the potential benefits of using symmetry and caching in the safety verification of multi-agent systems.

4.2 Symmetry and reachtubes of parameterized dynamical systems

In this section, we define the dynamics model of an agent as a parameterized dynamical system.

Definition 4.1. *The agent dynamics are defined by a tuple $A = \langle X, \mathscr{P}, f \rangle$, where $X \subseteq \mathbb{R}^n$ is its state space, $\mathscr{P} \subseteq \mathbb{R}^m$ is its parameter or mode space, and the dynamic*

function $f : X \times \mathscr{P} \to X$ that is Lipschitz in the first argument. Its trajectories are the same as those defined for parameterized dynamical systems defined in Section 2.2.1.

In the previous chapter, we showed how to get reachtubes of autonomous systems from previously computed ones using symmetry transformations in Theorem 3.1. Here we note that the same can be done for parameterized dynamical systems. This allows different modes of a hybrid system and different agents with similar dynamics to share reachtube computations. Recall from Section 2.2, that in the parameterized setting, associated with each symmetry γ that transforms states to symmetric ones, is a map ρ that transforms the modes to symmetric ones. For example, recall from Figure 2.5, if γ transforms the state of the aircraft by translating and rotating its position to a new coordinate system and translates its heading angle, ρ transforms the segment that the aircraft is following to the new coordinate system in which the position of the aircraft resides. To that end, we fix a group Γ of maps acting on X.

Theorem 4.1. *Assume that the parameterized dynamical system in equation (2.1) is Γ-equivariant. Then for any $\gamma \in \Gamma$ and its corresponding $\rho : \mathscr{P} \to \mathscr{P}$ per Definition 2.3, any $X_0 \subseteq X, p \in \mathscr{P}$, any time interval $[ftime, etime] \subseteq \mathbb{R}^+$, and any $(X_0, p, [ftime, etime])$-reachtube $\{(X_i, [t_{i-1}, t_i])\}_{i=0}^{J-1}$,*

$$\forall i \in [0, \ldots, J-1], Reach(\gamma(X_0), \rho(p), [t_{i-1}, t_i]) = \gamma(Reach(X_0, p, [t_{i-1}, t_i])) \subseteq \gamma(X_i).$$

Proof. (Sketch) First, $Reach(\gamma(X_0), \rho(p), [t_{i-1}, t_i]) = \gamma(Reach(X_0, p, [t_{i-1}, t_i]))$ follows directly from Theorem 2.1. Second, $\gamma(Reach(X_0, p, [t_{i-1}, t_i])) \subseteq \gamma(X_i)$ follows from the reachtube $ReachTb(X_0, p, [t_b, t_e])$ being an over-approximation of the exact reachset during the small time intervals $[t_{i-1}, t_i]$. □

Theorem 4.1 says that we can transform a computed reachtube $\{(X_i, [t_{i-1}, t_i])\}_{i=0}^{J-1}$ to get another reachtube $\{(\gamma(X_i), [t_{i-1}, t_i])\}_{i=0}^{J-1}$, which is an over-approximation of the reachsets starting from $\gamma(X_0)$.

The results of this section subsume the results about transforming reachtubes of autonomous systems-dynamical systems without parameters presented in the previous chapter.

4.3 Virtual system

In order to create the virtual model, a family of symmetries is needed. This is formalized below.

Definition 4.2 (virtual map)**.** *Given a parameterized dynamical system (2.1), a* virtual map *is a set*

$$\Phi = \{(\gamma_p, \rho_p)\}_{p \in \mathscr{P}}, \tag{4.1}$$

where for every $p \in \mathscr{P}$, $\gamma_p : X \to X$, $\rho_p : \mathscr{P} \to \mathscr{P}$, and they satisfy equation (2.2).

Given a parameterized dynamical system and a virtual map Φ, each γ_p is called a *virtual state map* and each ρ_p is called a *virtual mode map*. From Theorem 2.1, it follows that these maps transform trajectories in mode p to trajectories in mode $\rho_p(p)$. Using a virtual map Φ of system (2.1), we define the function $rv : \mathscr{P} \to \mathscr{P}$, where for all $p \in \mathscr{P}$:

$$rv(p) = \rho_p(p). \tag{4.2}$$

In this chapter, without loss of generality, we assume that there is a particular mode $p_v \in \mathscr{P}$ such that $\forall p \in \mathscr{P},\ rv(p) = p_v$, which we call the *virtual* parameter. The contributions of this chapter can be extended in a straight-forward manner to the more general case of multiple virtual parameters. We consider this generalization, among other extensions, in the next chapter. We fix a virtual map Φ for the rest of the chapter, and consequently p_v. Consider the ODE:

$$\frac{d\xi}{dt}(x, p_v, t) = f(\xi(x, p_v, t), p_v). \tag{4.3}$$

Following [26], we call system (4.3) a *virtual system*. Correspondingly, we call system (2.1), the *real system* for the rest of the chapter. The virtual system unifies the behavior of all modes of the real system in one representative mode, the virtual one p_v.

Example 4.1 (Fixed-wing aircraft virtual system)**.** In this example, we describe how the virtual system shown in Figure 4.3 of the multi-agent system of Figure 4.1 can be constructed. Each of the two aircraft in Figure 4.1 has the continuous dynamics described in Example 2.6. Recall that the state of the aircraft x consists of its position in the plane $[x[2], x[3]]$, its heading angle $x[1]$, and its speed $x[0]$. The

parameter p consists of the coordinates of the source and destination waypoints, $[p[0], p[1]]$ and $[p[2], p[3]]$, of the segment that the aircraft is following. Recall, also from Example 2.6, that the dynamics posses rotational and translational symmetry. Given a point $goal \in \mathbb{R}^2$ in the position plane and a rotation angle θ, the γ of Example 2.6 transforms the position of the aircraft to the new coordinate system centered at *goal* and rotated by θ. Similarly, ρ in that example transforms the two waypoints in p to the new coordinate system.

To construct the virtual map that is needed to generate the virtual system of Figure 4.3, for any $p \in \mathscr{P}$, set *goal* in the symmetry defined by Example 2.6 to $[p[2], p[3]]$ and θ to $\arctan_2(p[0] - p[2], p[3] - p[1])$. Name the resulting symmetries γ_p and ρ_p.

Then, for all $p \in \mathscr{P}$, $\rho_p(p)$, and equivalently $rv(p)$, would be a segment of the same length as p, aligned with the vertical axis, and ending at the origin. Since p_1 and p_2, p_4 and p_5, and p_3 and p_6 in Figures 4.1 and 4.2 have equal lengths, the virtual system shown in Figure 4.3 has three segments aligned with the vertical axis and ending at the origin. For the aircraft state and trajectories, γ_p would translate the origin of the position plane to the destination waypoint of p and rotate its axes so that the $x[3]$-axis is aligned with p, the segment between its source and destination waypoints. In the virtual system, the aircraft is always following the vertical axis in the position plane towards the origin. The reachtubes of the different modes of the two agents of Figure 4.2 are shown when transformed to the virtual system in Figure 4.6.

The solutions of the virtual system can be transformed to get solutions of all other modes in $\mathscr{P}$ using the symmetries $\{\gamma_p^{-1}\}_{p \in \mathscr{P}}$. This is shown in the following theorem.

Theorem 4.2. *Given any initial state $x_0 \in X$, and any mode $p \in \mathscr{P}$, $\gamma_p^{-1}(\xi(x_0, p_v, \cdot))$ is a solution of the real system (2.1) with mode p starting from $\gamma_p^{-1}(x_0)$. Similarly, given any $x_0 \in X$, $\gamma_p(\xi(x_0, p, \cdot))$ is the solution of the virtual system (4.3) starting from $\gamma_p(x_0)$.*

Proof. Lets start with the first part of the theorem. Fix $p \in \mathscr{P}$ and let $x_0 = \gamma_p^{-1}(x_0)$. Using Theorem 2.1, $\gamma_p(\xi(x_0, p, \cdot)) = \xi(\gamma_p(x_0), \rho_p(p), \cdot))$ and is the solution of the real system (2.1). Furthermore, $\rho_p(p) = p_v$, by definition, and $\gamma_p(x_0) = \gamma_p(\gamma_p^{-1}(x_0)) = x_0$. Hence, $\gamma_p(\xi(x_0, p, \cdot)) = \xi(x_0, p_v, \cdot)$. Applying γ_p^{-1} on both sides implies the first part of the theorem. The second part is a direct application of Theorem 2.1. □

The following corollary extends the result of Theorem 4.2 to reachtubes. It follows from Theorem 4.1.

Corollary 4.1. *Given an $X_{v,0} \subseteq X$ and a mode $p \in \mathscr{P}$, $\gamma_p^{-1}(ReachTb(X_{v,0}, p_v, [t_b, t_e]))$ is a reachtube of the real system (2.1) with mode p starting from $\gamma_p^{-1}(X_{v,0})$. Similarly, given any initial set $X_0 \subseteq X$, $\gamma_p(ReachTb(X_0, p, [t_b, t_e]))$ is a reachtube of the virtual system (4.3) starting from $\gamma_p(X_0)$.*

Consequently, we get a solution or a reachtube for each mode $p \in \mathscr{P}$ of the real system by simply transforming a single solution or a single reachtube of the virtual system using the symmetries $\{\gamma_p\}_{p \in \mathscr{P}}$ and their inverses. This will be the essential idea behind the savings in computation time of the new symmetry-based reachtube computation algorithm and symmetry-based safety verification algorithms presented next. It will be also the essential idea behind proving safety in the case of unbounded time and infinite number of modes.

Example 4.2 (Fixed-wing aircraft infinite number of reachtubes resulting from transforming a single one)**.** Consider the real system in Example 2.6 and the virtual one in Example 4.1. Fix the initial set, which is represented as a hyper-rectangle, $X_{r,0} = [[1, \frac{\pi}{4}, 3, 1], [2, \frac{\pi}{3}, 4, 2]]$, the real mode $p_r = [2.5, 0.5, 13.3, 5]$, and the time bound 20 seconds. Then, following the method of Example 4.1 to choosing θ and *goal*, we fix $\theta = \arctan_2(2.5 - 13.3, 5 - 0.5) = -1.176$ rad and $goal = [13.3, 5]$. We call the resulting symmetries from fixing these values of θ and *goal* in the transformations defined by Example 2.6, γ_{p_r} and ρ_{p_r}. Let $X_{v,0} = \gamma_{p_r}(X_{r,0})$ and $p_v = \rho_{p_r}(p_r)$. Assume that we have computed the reachtube $rtb_r = ReachTb(X_{r,0}, p_r, T)$ using DryVR, for example. Then, using Corollary 4.1, we can get $rtb_v = ReachTb(X_{v,0}, p_v, T)$ by transforming rtb_r using γ_{p_r}. The benefit of the corollary appears in the following: for any $p \in \mathscr{P} = \mathbb{R}^4$, we can get the corresponding reachtube $ReachTb(\gamma_p^{-1}(X_{v,0}), p, T)$ by transforming rtb_v using γ_p^{-1}.

The projection of $X_{v,0}$ on its last two coordinates $X_{v,0}[2:3]$ represents the possible initial position of the aircraft in the plane relative to the destination waypoint. It would be a rotated square with angle θ. The distance from the center of $X_{v,0}[2:3]$ to the origin of the coordinate system would be equal to the distance from the center of $X_{r,0}[2:3]$ to the destination waypoint of p_r. Moreover, the angle between the $x[1]$-axis and the line connecting the origin with the center of $X_{v,0}[2:3]$ would be equal to the angle from the segment connecting the source and destination waypoints to the line connecting the destination waypoint with the center of $X_{r,0}[2:3]$. On the

other hand, $X_{v,0}[0] = X_{r,0}[0]$ and $X_{v,0}[1] = X_{r,0}[1] - \theta$, i.e. speed remains constant under γ_{p_r} and the heading angle gets translated by $-\theta$.

In summary, the dynamics of the aircraft depend solely on its absolute speed and its relative position and heading angle with respect to the destination waypoint. The virtual system captures what the dynamics actually depend on: it has a unique trajectory for every group of symmetries trajectories of the real system. The source waypoint was included as part of the mode throughout the running example starting from Example 2.6, although the dynamics of the aircraft do not depend on it, to have a reference angle to which we can rotate the coordinate system to. Another alternative to including the source waypoint would have been to use translation symmetry that translates the coordinate system to the destination waypoint.

4.4 Caching and symmetry in multi-agent verification

In this section, we introduce a novel safety verification algorithm, symSafetyVerif, which uses existing reachability subroutines, but exploits symmetry, unlike existing algorithms. In the previous chapter and in [126], we introduced reachtube transformations using symmetry for single mode dynamical systems. Here, we extend the method across modes, introduce the virtual system, and develop the corresponding verification algorithm.

In Section 4.4.1, we define *tubecache*—a data-structure for storing reachtube. In Section 4.4.2, we present the symmetry-based reachtube computation algorithm symComputeReachTb that reuses reachtubes stored in *tubecache*. Finally, in Section 4.4.3, we define the *safetycache* data-structure which stores previously computed safety verification results. These results would be used by the symSafetyVerif algorithm for efficient safety verification of multi-agent systems.

4.4.1 *tubecache*: shared memory for reachtubes

We show how we use the virtual system (4.3) to create a shared cache for the different modes of the real system (2.1) to reuse each others' computed reachtubes. We call this shared memory *tubecache*.

Definition 4.3. *A tubecache is a data structure that stores a set of reachtubes of the virtual system (4.3). It has two methods:* getTube*, for retrieving stored tubes*

and `storeTube`*, for storing a newly computed one.*

The function `getTube` returns a set of reachtubes $\{ReachTb(X_{i,0}, p_v, [0, T_i])\}_{i\in[h]}$, for some $h \in \mathbb{N}$, that are already stored in *tubecache*. Moreover, the union of $X_{i,0}$s is the largest subset of X_0 that can be covered by the initial sets of the reachtubes in *tubecache*. Formally,

$$tubecache.\texttt{getTube}(X_i) = \underset{\{ReachTb(X_{i,0},p_v,[0,T_i])\in tubecache\}_i}{\operatorname{argmax}} \operatorname{Vol}(X_i \cap \cup_i X_{i,0}), \quad (4.4)$$

where $\operatorname{Vol}(\cdot)$ is the Lebesgue measure of the set. Note that for any $X_i \subset X$, a maximizer of (4.4) would be the set of all reachtubes in *tubecache*. However, this is very inefficient and it would be too conservative to be useful for checking safety. Therefore, `getTube` should return the minimum number of reachtubes that maximize (4.4). Note that the reachtubes in *tubecache* may have different time bounds. We will truncate or extend them when used.

4.4.2 Symmetry-based reachtube computation

Given an initial set $X_0 \subseteq X$, a mode $p \in \mathscr{P}$, and time bound T, there are dozens of tools that can return a $ReachTb(X_0, p, [0, T])$. DryVR [115], which we used in the previous chapter, is an example tool. See [16, 127, 18] for other examples. We denote this procedure by $\mathsf{computeReachtube}(X_0, p, [0, T])$.

A reachability analysis-based hybrid automata safety verification algorithm usually computes the reachtubes of the modes sequentially [115, 16]. Sometimes, the algorithm has to compute multiple reachtubes for the same mode starting from different initial sets, for example as a result of refinement (check Section 2.4). We augment such an algorithm with symmetry-utilization capabilities. We do not show the pseudo code for such an algorithm, for simplicity. The contribution of this chapter focuses on how to efficiently obtain the per-mode reachtubes, instead of the order they are requested or how they are combined to obtain the automaton reachtube. Whenever a reachtube is needed by our proposed symmetry-utilizing verification algorithm, instead of directly calling $\mathsf{computeReachtube}$, it calls another procedure, $\mathsf{symComputeReachTb}$, which we introduce in Algorithm 4.1. The latter uses the virtual map to retrieve subsets of the requested reachtube that are already stored in *tubecache*, and computes the rest using $\mathsf{computeReachtube}$. The verification algorithm transforms the initial set of states of the needed reachtube

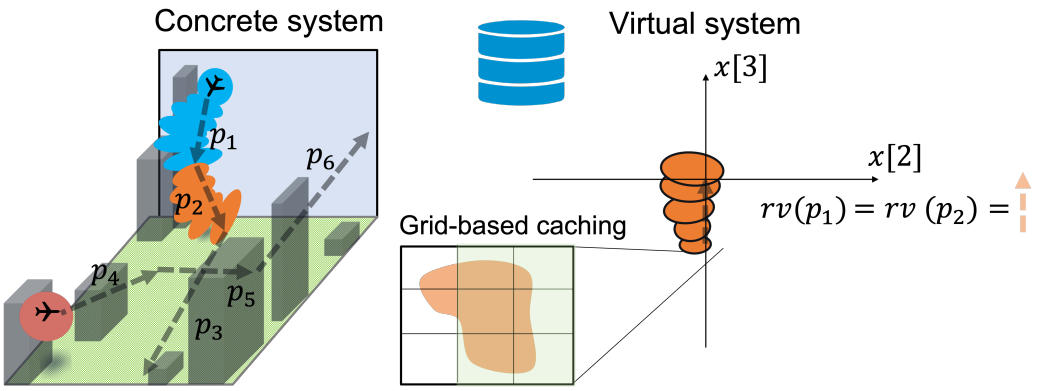


Figure 4.4: A sketch illustrating the safety verification of the system described in Figure 4.1 using symmetry. The *tubecache* implementation we consider in this figure is based on a grid over the state space. As in traditional reachability analysis methods for hybrid automata, the reachtube for each mode of each agent is computed sequentially. To obtain the blue reachtube for mode p_1, the symmetry-utilizing verification algorithm transforms the initial set of states using γ_{p_1}, described in Example 4.1, and then call symComputeReachTb. Then, the latter calls *tubecache*.getTube to check if the reachtube for the transformed initial set is stored in the cache. In turn, *tubecache*.getTube checks if any of the cells of the grid that are intersecting the transformed initial set has a corresponding stored reachtube in the cache. Since the cache is initially empty, symComputeReachTb calls computeReachtube, an abstraction of any existing reachability analysis tool, to compute the reachtube. Finally, symComputeReachTb saves the computed reachtube in the cache (shown in orange on the right). The verification algorithm then transforms the returned reachtube, the orange one, using $\gamma_{p_1}^{-1}$, to obtain the blue reachtube on the right. After the reachtube of p_1 is obtained, the verification algorithm has to obtain the reachtube of p_2. First, it intersects the blue reachtube with the guard of the edge between p_1 and p_2. The guard of the edge is a box in X around the destination waypoint of p_1 that when the aircraft reaches, it can stop following p_1 and start following p_2 instead. The result of the intersection between the reachtube and the guard is the initial set from which the verification algorithm has to generate the reachtube of p_2. Since p_2 has the same representative mode in the virtual system of Figure 4.3 as p_1, they share the same entry in *tubecache*. The verification algorithm transforms the initial set using γ_{p_2}. The result, projected to the last two coordinates, is shown as the orange shape in the grid. The verification algorithm calls symComputeReachTb. The latter calls *tubecache*.getTube which finds that six of the intersecting cells have stored reachtubes (colored by light green). This example is continued in Figure 4.5.

using the virtual map before calling symComputeReachTb, and then transform the returned reachtubes as well. All reachtube computations by symComputeReachTb are done for the virtual system. An example run of the algorithm on the example introduced earlier in Figure 4.1 is shown in Figures 4.4, 4.5, and 4.6. Note that the virtual system shown in the figures has multiple virtual modes, instead of just one. As mentioned earlier, the same theory developed in this chapter generalizes to the multiple virtual modes case.

The algorithm symComputeReachTb takes as input an initial set of states of the virtual system $X_{v,0}$, the time bound T, and the cache *tubecache*. It returns a reachtube of the virtual system starting from $X_{v,0}$ and running for T time units. Hence, to get a reachtube of the real system starting from an initial set X_0 and having a mode p and time bound T, the symmetry-utilizing verification algorithm first transforms X_0 using γ_p to get $X_{v,0}$, calls symComputeReachTb, and transforms the returned reachtube $restube_v$ using γ_p^{-1}.

First, symComputeReachTb initializes $restube_v$ as an empty set to store the result in line 2. It then gets the reachtubes from *tubecache* that corresponds to $X_{v,0}$ using the getTube method in line 3. Now that it has the relevant tubes in *storedtubes*, it adjusts their lengths based on the time bound T. For a retrieved tube with a time bound less than T in line 5, symComputeReachTb extends the tube for the remaining time using computeReachtube in lines 6-7, store the resulting tube in *tubecache* instead of the shorter one in line 8. If the retrieved tube is longer than T (line 9), it trims it in line 10. However, it keeps the long reachtube in *tubecache*. Then, it adds the reachtube with the adjusted length to the result $restube_v$ in line 11.

The union of the initial sets of the tubes retrieved *storedtubes* may not contain all of the initial set $X_{v,0}$. That uncovered part is called X_v' in line 12. The algorithm symComputeReachTb computes the reachtube starting from X_v' from scratch by calling computeReachtube in line 13, stores it in *tubecache* in line 14, and adds it to $restube_v$ in line 15. The resulting reachtube of the virtual system (4.3) is returned in line 16. This reachtube would be transformed by the calling verification algorithm using γ_p^{-1} to get the needed reachtube of the real system (4.3).

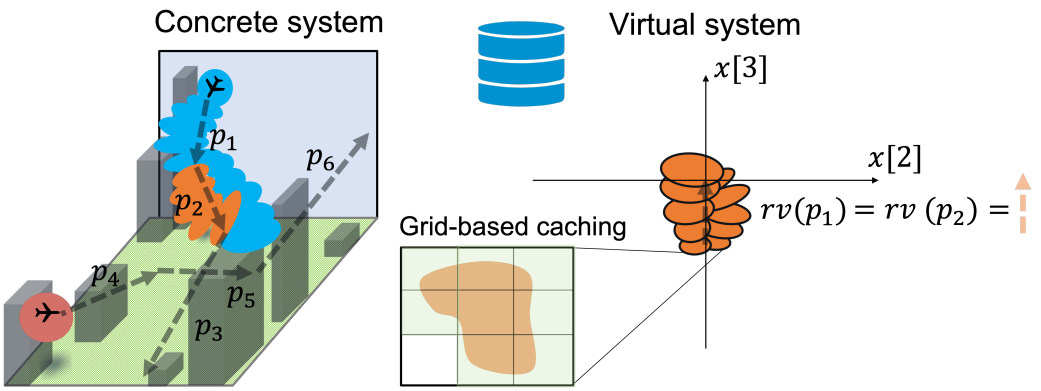


Figure 4.5: The next step of the symmetry-utilizing verification algorithm run discussed in Figure 4.4. The algorithm symComputeReachTb calls computeReachtube to compute the reachtubes starting from the two cells that are intersecting the transformed initial set. Then, the former caches the computed reachtubes by calling *tubecache*.`storeTube`. The two cells are now also covered with light green. The algorithm symComputeReachTb then returns the reachtubes of the eight cells to the verification algorithm. The latter transforms them using $\gamma_{p_2}^{-1}$ to get the reachtube of p_2. The part that was already in the cache is colored with orange on the right, and the part that was computed is colored by blue, instead.

Algorithm 4.1 symComputeReachTb

input: $X_{v,0}, T, tubecache$
$restube_v \leftarrow \emptyset$
$storedtubes \leftarrow tubecache.\texttt{getTube}(X_{v,0})$
for $i \in [|storedtubes|]$ **do**
  if $storedtubes[i].T < T$ **then**
    $(X_{i,0}, [t_i, T_i]) \leftarrow storedtubes[i].end$
    $tube_i \leftarrow storedtubes[i] \frown \text{computeReachtube}(X_{i,0}, p_v, [0, T - t_i])$
    $tubecache.\texttt{storeTube}(tube_i)$
  else if $storedtubes[i].T > T$ **then**
    $tube_i \leftarrow storedtubes[i].truncate(T)$
  $restube_v \leftarrow restube_v \cup tube_i$
$X'_{v,0} \leftarrow X_{v,0} \setminus \cup_i storedtubes[i].X_0$
$tube' = \text{computeReachtube}(X'_{v,0}, p_v, [0, T])$
$tubecache.\texttt{storeTube}(tube')$
$restube_v \leftarrow restube_v \cup tube'$
return: $restube_v$

Theorem 4.3. *The output of Algorithm 4.1 is an over-approximation of the reachtube* $ReachTb(X_{v,0}, p_v, [0, T])$.

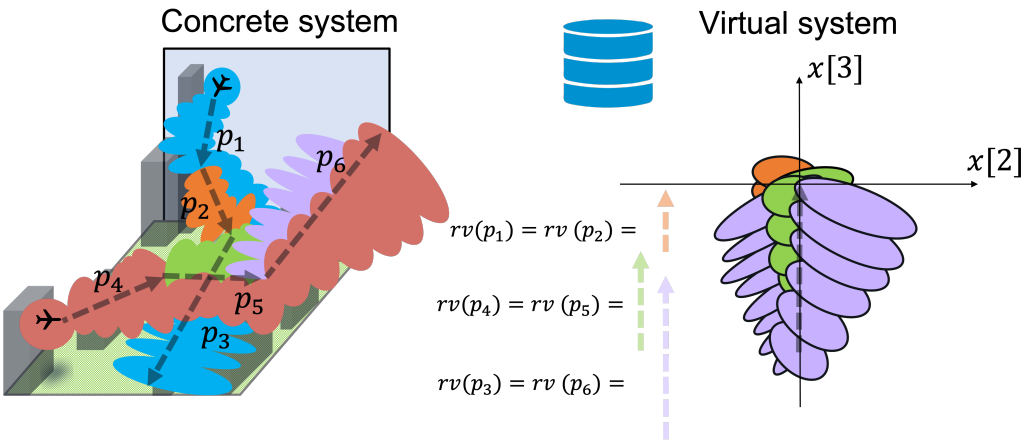


Figure 4.6: The final step of the run described in Figures 4.4 and 4.5. Part of the reachtube computed for p_3 was used to get part of the reachtube of p_6 (shown in violet). Part of the reachtube computed for p_4 was used to compute part of the reachtube of p_5 (shown in green).

Proof. (Sketch) By assumption, the reachability analysis tool being used for reachtube computations by computeReachtube is sound. Thus, computeReachtube returns over-approximations of the requested reachtubes. The set $restube_v$ contains reachtubes that were computed by computeReachtube at some point. There are three types of reachtubes in $restube_v$:

1. When the time bound T_i of the stored reachtube $storedtubes[i]$ is less than T, symComputeReachTb extends $storedtubes[i]$ until time T by concatenating the stored reachtube with computeReachtube$(X_{i,0}, p_v, [0, T - t_i])$, where $(X_{i,0}, [t_i, T_i])$ is the last pair in $storedtubes[i]$. The result is a $(storedtubes[i].X_0, p_v, [0,T])$-reachtube.

2. When the time bound T_i of the stored reachtube $storedtubes[i]$ is more than T, the truncated reachtube is also a valid $(storedtubes[i].X_0, p_v, [0,T])$-reachtube.

3. For $X'_{v,0}$ that is not contained in the union of the initial sets in $storedtubes$, the function computeReachtube will return a valid $(X'_{v,0}, p_v, [0,T])$-reachtube.

The union of the initial sets of the tubes in $storedtubes$ and $X'_{v,0}$ contains $X_{v,0}$, so the union of the reachtubes the algorithm returns a $(X_{v,0}, p_v, [0,T])$-reachtube. □

The importance of symComputeReachTb lies in that if a mode p required a computation of a reachtube and the result is saved in *tubecache*, another mode with

a similar scenario with respect to the virtual system would reuse that tube instead of computing one from scratch. Moreover, reachtubes of the same mode might be reused as well if the scenario was repeated again.

4.4.3 Bounded time safety

In this section, we show how to use *tubecache* and symComputeReachTb of the previous section for bounded and unbounded time safety verification of the real system (2.1). We consider a scenario where the safety verification of multiple modes of the real system (2.1) starting from different initial sets and for different time horizons is needed. We will use the virtual system (4.3) and the symmetries $\{\gamma_p\}_{p\in\mathscr{P}}$ to share safety computations across modes, initial sets, time horizons, and unsafe sets.

We first introduce *safetycache*, a shared memory to store the results of intersecting reachtubes of the virtual system (4.3) with different unsafe sets. It will prevent repeating safety checking computations of different modes under similar scenarios and can be used in finding unbounded time safety properties of the real system (2.1).

Definition 4.4. *A safetycache is a data structure that stores the results of intersecting reachtubes of the virtual system (4.3) with unsafe sets. It has two functions:* `getIntersect`*, for retrieving stored results and* `storeIntersect`*, for storing a newly computed one.*

Given an initial set $X_{v,0}$, a time bound T, and an unsafe set O_v, the reachtube $rtb = ReachTb(X_{v,0}, p_v, [0,T])$ is unsafe if there is another reachtube $rtb' = ReachTb(X'_{v,0}, p_v, [0,T'])$ that is unsafe and is a subset of rtb. Similarly, if rtb is a subset of rtb' and the latter is safe, then rtb is safe as well. Recall the definition of subsets of reachtubes at the end of Section 2.3.2. The `getIntersect` function of *safetycache* returns the truth value of the predicate $ReachTb(X_{v,0}, p_v, [0,T]) \cap O_v \neq \emptyset$ if a subsuming computation is stored, and returns $\perp$, otherwise.

Formally, *safetycache*.`getIntersect`$(X_{v,0}, T, O_v) =$

$$\begin{cases} 0, \text{ if } \exists\, X'_{v,0}, T', O'_v \mid X_{v,0} \supseteq X'_{v,0}, T \geq T', O_v \supseteq O'_v, safetycache(X'_{v,0}, T', O'_v) = 0, \\ 1, \text{ if } \exists\, X'_{v,0}, T', O'_v \mid X_{v,0} \subseteq X'_{v,0}, T \leq T', O_v \subseteq O'_v, safetycache(X'_{v,0}, T', O'_v) = 1, \\ \perp, \text{ otherwise,} \end{cases}$$

where 0 means *unsafe* and 1 means *safe*.

It is equivalent to check the intersection of a reachtube of the real system (2.1) with an unsafe set O and to check the intersection of the corresponding reachtube and unsafe set of the virtual one. This is formalized in the following lemma.

Lemma 4.1. *Consider an unsafe set $O \subseteq X \times \mathbb{R}^+$ and $rtb = ReachTb(X_0, p, [t_1, t_2])$. Then, for any invertible $\gamma : X \to X$, $rtube \cap O \neq \emptyset$ if and only if $\gamma(rtube) \cap \gamma(O) \neq \emptyset$.*

Now that we have established the equivalence of safety checking between the real and virtual systems, we present Algorithm 4.2 denoted by symSafetyVerif. It uses *safetycache*, *tubecache*, and symComputeReachTb in order to share safety verification computations across modes. The method symSafetyVerif would be called several times to check safety of different scenarios and *safetycache* and *tubecache* would be maintained across calls.

The function symSafetyVerif takes as input an initial set X_0, a mode p, a time bound T, an unsafe set O, the symmetry γ_p, and *safetycache* and *tubecache* that resulted from previous runs of the algorithm.

It starts by transforming the initial and unsafe sets X_0 and O to a virtual system initial and unsafe sets $X_{v,0}$ and O_v using γ_p in line 2. It then checks if a subsuming result of the safety check for the tuple $(X_{v,0}, T, O_v)$ exists in *safetycache* using its method `getIntersect` in line 3. If it does exist, it returns it directly in line 8. Otherwise, the approximate reachtube is computed using symComputeReachTb in line 5. The returned tube is intersected with O_v in line 6 and the result of the intersection is stored in *safetycache* in line 7 and returned in line 8.

Algorithm 4.2 symSafetyVerif

1: **input:** $X_0, p, T, O, \gamma_p, safetycache, tubecache$
2: $X_{v,0} \leftarrow \gamma_p(X_0)$, $O_v \leftarrow \gamma_p(O)$
3: $result \leftarrow safetycache.\texttt{getIntersect}(X_{v,0}, T, O_v)$
4: **if** $result = \perp$ **then**
5: $\quad rtb \leftarrow$ symComputeReachTb $(X_{v,0}, T, tubecache)$
6: $\quad result \leftarrow (tube \cap O_v = \emptyset)$
7: $\quad safetycache.\texttt{storeIntersect}(X_{v,0}, T, O_v, result)$
8: **return:** *result*

Theorem 4.4. *If* symSafetyVerif *returns safe, then $ReachTb(X_0, p, [0, T]) \cap O = \emptyset$.*

Proof. From Theorem 4.3, if the result is not stored in *safetycache*, we know that *rtb* in line 5 is an over-approximation of $ReachTb(X_{v,0}, p_v, [0, T])$. Moreover,

we know from Corollary 4.1 that $ReachTb(X_0, p, [0,T]) \subseteq \gamma_p^{-1}(rtb)$. But, from Lemma 4.1, we know that the truth value of the predicate $(rtb \cap O_v = \emptyset)$ is equal to that of $(\gamma_p^{-1}(rtb) \cap O = \emptyset)$ and hence *result* is *safe* if $\gamma_p^{-1}(rtb) \cap O = \emptyset$ and thus it is *safe* if $ReachTb(X_0, p, T) \cap O = \emptyset$. Finally, the stored values in *safetycache* are results from previous runs, and hence have the same property. □

However, if symSafetyVerif returns *unsafe*, it might be that *rtb* in line 5 intersected the unsafe set because of an over-approximation error. There are two sources of such errors: first, the method computeReachtube used by symComputeReachTb can itself result in over-approximation errors. Actually, it will, most of the time [16, 127]. But it may be exact too [67]. Second, the *tubecache*.`getTube` method which would return a list of tubes with the union of their initial sets strictly over-approximating the needed initial set. The first problem can be solved by asking the method computeReachtube to compute tighter reachtubes. Existing methods provide this option at the expense of worse computational complexity [16, 127]. However, we can use symmetry in these tightening computations as well, as we did in the previous chapter and in [126] when we reduced the dimension of the reachtube to be computed using a group of symmetries. We can also replace saved tubes in *tubecache* with newly computed tighter ones or store multi-resolution reachtubes using a *cachetree* as we did in the previous chapter. The second problem can be solved by asking *tubecache*.`getTube` to return only the tubes with initial sets that are fully contained in the asked initial set. This would decrease the savings from transforming cached results, but it would reduce the false-positive error, saying *unsafe* while it is *safe*.

4.4.4 Unbounded time safety

In this section, we show how infinite number of results of safety checks, i.e. results of intersections of reachtubes with unsafe sets, can be deduced from finite ones. The following corollary applies Lemma 4.1 to the symmetries $\{\gamma_p\}_{p \in \mathscr{P}}$ that map the different modes of the real system (2.1) to the unique virtual one (4.3).

Corollary 4.2 (Infinite safety verification results from a single one)**.** *Fix $O \subseteq X$ and $rtube = ReachTb(X_{v,0}, p_v, [0,T])$. If $rtube \cap O = \emptyset$, then $\forall p \in \mathscr{P}$, $\gamma_p^{-1}(rtube) \cap \gamma_p^{-1}(O) = \emptyset$.*

The corollary means that from a single scenario safety check, i.e., an intersection operation between a reachtube $ReachTb(X_0, p_v, [0,T])$ and unsafe set O, we can

deduce the safety of any mode $p \in \mathscr{P}$ starting from $\gamma_p^{-1}(X_0)$ and running for T time units with respect to the corresponding unsafe set $\gamma_p^{-1}(O)$. This would, for example, imply unbounded time safety of a hybrid automaton under the assumption that the unsafe sets of the modes are at the same relative position with respect to the reachtube. But, *safetycache* stores a number of results of such operations. We can infer from each one of them the safety of infinite scenarios. This is formalized in the following theorem which follows directly from Corollary 4.2.

Theorem 4.5 (Infinite safety verification results from finite ones)**.** *For any mode $p \in \mathscr{P}$, initial set $X_0 \subseteq X$, time bound $T \geq 0$, and unsafe set $O \subset X$, such that $X_0 \subseteq \gamma_p^{-1}(X_0')$, $O \subseteq \gamma_p^{-1}(O')$, and safetycache$(X_0', T, O') = 1$, system (2.1) is safe.*

As more results are added to *safetycache*, then we can deduce the safety of more scenarios in all modes. If at a given point of time, we are sure that no new scenarios would appear, we can deduce the safety for unbounded time and unbounded number of agents with the same dynamics having scenarios already covered. This is formalized and solved in a rigorous algorithm in the next chapter using symmetry-based abstractions of hybrid automata. There, the hybrid automata is first reduced to a one with fewer modes and the reachtubes of the latter are computed and checked if they intersect corresponding unsafe sets. That eliminates the need for transforming reachtubes back to the real system and soundly computes the possible initial states for each mode of the automaton.

Example 4.3 (Fixed-wing aircraft infinite number of safety verification results from computing a single one)**.** Consider the initial set X_0, mode p, time bound T, their corresponding virtual ones $X_{v,0}$ and p_v, and the symmetry transformation γ_{p_r} considered in Example 4.2. Let the unsafe set be $O = [[0, -\infty, 11.9, 5.1], [\infty, \infty, 12.9, 6.1]] \times \mathbb{R}^+$ and $O_v = \gamma_{p_r}(O)$. Assume that $rtb_v \cap O_v = \emptyset$ and the result is stored in *safetycache*. Then, for all $p \in \mathscr{P}$, $\gamma_p^{-1}(rtb_v) \cap \gamma_p^{-1}(O_v) = \emptyset$.

For the aircraft, O could represent a mountain. Crashing with the mountain at any speed, heading angle, and time is unsafe. O_v represents the relative position of the mountain with respect to the segment of waypoints. Theorem 4.5 says that for any initial set of states X_0 of the aircraft and time bound T, if the relative positions of the aircraft, unsafe set, and the segment of waypoints are the same or subsumed by those of $X_{v,0}$, O_v, and the origin, we can infer safety irrespective of their absolute positions.

4.5 Experimental evaluation

We implemented a software safety verification tool for multi-agent hybrid systems based on symComputeReachTb using Python 3. We named it CacheReach. By hybrid, we mean systems that transition between different modes under different conditions. We tested it on a linear dynamical system and the aircraft model of Example 2.6, following sequences of waypoints, using DryVR [15] and Flow* [127] as reachability subroutines. Our code is available in a figshare repository [128] and has been tested on an Ubuntu virtual machine available in another figshare repository [129].

4.5.1 CacheReach: multi-agent safety verification tool

Our tool CacheReach takes as input a JSON file specifying a list of N agents of dimension n. It also specifies the python file that contains the dynamics function f of Definition 4.1 and two symmetry-related functions: *symGamma* and *symGammaInv*. Given a $p \in \mathscr{P}$ and a polytope[2] *poly* of dimension n representing a set of states of the agent, *symGamma* returns $\gamma_p(poly)$, where γ_p is the symmetry map to the virtual system. Similarly, *symGammaInv* would return $\gamma_p^{-1}(poly)$. The list of modes that the i^{th} agent transition between sequentially and their corresponding transitions conditions, denoted by guards, are specified as well and denoted by H_i. The guard of the j^{th} mode of the i^{th} agent $H_i[j].guard$ is a hyper-rectangle in the state space which when the agent reaches, it transitions to the $(j+1)^{st}$ mode. The guard $H_i[j]$ has time bound $H_i[j].T$ on how long the agent can stay in the mode. Moreover, it specifies the initial set of states for each agent as a hyper-rectangle. Finally, it specifies the static unsafe set O and the subset of dimensions $L \subseteq [n]$ that is relevant for dynamic safety checking between agents. If the reachtubes of two agents projected on L intersect each other, it would model a collision between the agents. For example, L would be the set $\{2,3\}$ for the aircraft model in Example 2.6 as $(x[2], x[3])$ represents its position.

CacheReach would return *unsafe* if the reachtubes of the agents starting from their initial sets of states and following the sequence of modes intersect a static unsafe set, or when projected to L, intersect each other. It would return *safe*, otherwise. Currently, CacheReach assumes that all agents share the same dynamics but do not interact. Hence, it has a single *tubecache* that is shared by all agents.

[2] `https://github.com/tulip-control/polytope`

CacheReach computes the reachtubes of individual agents iteratively. For each agent i, it would compute the reachtube $mtube_i$ of the j^{th} mode using symComputeReachTb. Then, CacheReach intersects $mtube_i$ with the guard using the function *guardIntersect* to get the initial set $initset_i$ for the next mode. In addition to $initset_i$, *guardIntersect* computes the minimum and maximum times: $mintime_i$ and $maxtime_i$, respectively, at which $mtube_i$ intersects the guard. The value $mintime_i$ is the time at which a trajectory of the next mode may start at and $maxtime_i$ is the maximum such time. These values are used to check safety against time-annotated unsafe sets such as collision between agents.

The computed tube $mtube_i$ gets appended to $atube_i$ storing the full reachtube of the i^{th} agent. The benefit of this method is that now all modes of all agents can be mapped to a single virtual system. They can resuse each others reachtubes using *tubecache* that is getting updated at every call to symComputeReachTb. Moreover, the static safety is done in the usual way.

The collision between agents is done by the function checkDynamicSafety. It takes two full reachtubes of two agents $atube_1$ and $atube_2$ along with two arrays $lookback_1$ and $lookback_2$. For agent i, the array $lookback_i$ consists of pairs of integers $(ind_j, timerange_j)$ specifying the index identifying the beginning of the j^{th} mode tube in $atube_i$ and the uncertainty in the starting time of the trajectories from its initial set. checkDynamicSafety would use this information to time-align parts of $atube_1$ and $atube_2$ so that the intersection check happens only between two sets that may have been reached at the same time by the two agents.

4.5.2 symComputeReachTb implementation

To implement the *tubecache* in CacheReach, we grid the state space with a resolution $\delta \in X$. *tubecache* is then implemented as a Python dictionary. The key of an element of *tubecache* would be the center of a grid cell in X and its value would be the reachtube of the virtual system starting from the cell as its initial set of states.

Given an initial set $X_{v,0}$, symComputeReachTb would compute its bounding box, quantize its bottom-left and upper-right corners with respect to the grid, iterate over all cells and check if that cell intersects with $X_{v,0}$. If it does, we check if *tubecache* has the corresponding tube and add it to the result. Otherwise, we compute it from scratch and store it in *tubecache* with the key being the center of the cell. Then, $X'_{v,0}$ in symComputeReachTb would consist of the cells that it did not find the

reachtube for and *storedtubes* would consists of the reachtubes that were retrieved.

4.5.3 Experimental results

We ran experiments using our tool CacheReach on two models: a 3-dimensional linear dynamical system example and the nonlinear aircraft model described in Example 2.6. The linear model is of the form $\dot{x} = A(x - p[3:5])$, where

$$A = diag([-3, -3, -1]), \tag{4.5}$$

$x \in \mathbb{R}^3$, and $p \in \mathbb{R}^6$. Note that we changed the matrix A from the one we presented in [130], which was:

$$A = \begin{bmatrix} -3 & 1 & 0 \\ 0 & -2 & 1 \\ 0 & 0 & -1 \end{bmatrix}. \tag{4.6}$$

This change is a correction of the error made in [130] in assuming that for the latter choice of A (4.6), the system is equivariant to the rotation of the first two dimensions and the translation of all three dimensions. However it is not, while the new one (4.5), is. We considered scenarios with single, two, and three agents for each model following different sequences of waypoints. The sequences of waypoints for the linear model are translations and rotations of a digital-*S* shaped path. For the aircraft model, the paths are random crossing paths going north-east. In every scenario, all the agents have the same model. In the aircraft scenarios, the agent would switch to the next waypoint once its position is within 0.5 units from the current waypoint in each dimension. The initial set of the aircraft was of size 1 in the position components, 0.1 in the speed, and 0.01 in the heading angle. We used Flow* [127] and DryVR [15] to compute reachtubes from scratch for the linear example. We only used DryVR for the aircraft model since our C++ Flow* wrapper does not handle a model having $\arctan_2$ in the dynamics. We ran all scenarios in CacheReach with and without using *tubecache*. The symmetry used for the aircraft was the one we showed in Example 4.1. For the linear model, the symmetry transformation γ_p that was used to map the state to the virtual system was by coordinate transformation where the new origin is at the next waypoint $p[3:5]$ and rotating the plane of the first two dimensions by the angle between the

Table 4.1: Results.

tool \ agent model		Linear(1,2,and 3 agents)			aircraft(1,2,and 3 agents)		
Sym-DryVR	computed	57	90	90	635.23	1181.38	1550.62
	transformed	42	165	264	20.76	286.62	501.38
	time (min)	0.093	0.163	0.187	3.42	8.2	10.59
Sym-Flow*	computed	39.8	61.14	66.15			
	transformed	19.2	84.85	143.85	NA	NA	NA
	time (min)	0.387	0.62	0.684			
NoSym-DryVR	computed	99	255	354	656	1468	2052
	time (min)	0.062	0.355	0.52	3.71	10.78	15.47
NoSym-Flow*	computed	59	151	210			
	time (min)	0.53	1.328	1.5	NA	NA	NA

previous and the next waypoints $p[0:2]$ and $p[3:5]$ projected to the plane. We compared the computation time with and without symmetry and show the results in Table 4.1. The reachtubes for three nonlinear and three linear agents are shown in Figure 4.7. The different colors represent reachtubes of different agents, the black points represent the waypoints, the black segments connect consecutive waypoints, and the red rectangles represent the unsafe sets. The figures on the top represent the real reachtubes while those on the bottom represent the ones corresponding to the virtual system saved in *tubecache*.

In Table 4.1, we call CacheReach, when ran with DryVR while using *tubecache*, Sym-DryVR, for symmetric DryVR. We call it Sym-Flow* if we are using Flow* instead. If we are not using *tubecache*, we call them NoSym-DryVR and NoSym-Flow*, respectively. Remember in symComputeReachTb, some tubes may be cached but they have shorter time horizons than the needed tube. So, we compute the rest from scratch. Here, we report the fractions of tubes computed from scratch and tubes that were transformed from cached ones. Moreover, we report the execution time till the tubes are computed. In the experiments, we always compute the full tubes even if it was detected to be unsafe earlier to have a fair comparison of running times. Moreover, the execution time does not include dynamic safety checking as the four versions of the experiments are doing the same computations for that purpose. We are using CacheReach in all scenarios with other reachability computation tools to decrease the degrees of freedom and show the benefits of transforming reachtubes over computing them. The Sym versions result in decrease of running time up-to 64% in the linear case with three agents. The ratio of transformed vs. computed tubes increases as the number of agents increase. This means that different agents are sharing reachtubes with each other in

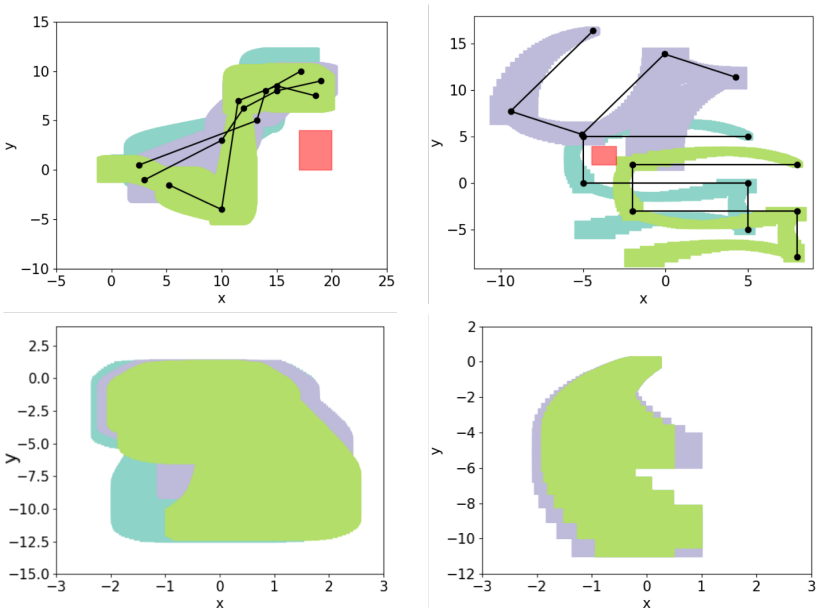


Figure 4.7: Reachtubes for three fixed-wing aircrafts (left) and three linear models (right). Real reachtubes (top) vs. the virtual ones saved in *tubecache* (bottom).

the virtual system. The total number of reachtubes is the same, whether *tubecache* is used or not. This means that the quality of the tubes, i.e., how tight they are, is the same whether we are transforming from *tubecache* or computing from scratch since the initial sets of modes are computed from intersections of reachtubes with guards. The fatter the reachtube is, the larger the initial set gets and the larger the number of reachtubes need to be computed.

4.6 Conclusions

In this chapter, we investigated how symmetry transformations and caching can help achieve scalable, and possibly unbounded, verification of multi-agent systems. We developed a notion of *virtual system* which define symmetry transformations for a broad class of hybrid and dynamical agent models visiting waypoint sequences. Using virtual system, we present a prototype tool called CacheReach that builds a cache of reachtubes for the transformed virtual system, in a way that is agnostic of the representation of the reachsets and the reachability analysis subroutine used. Our experimental evaluation show significant improvement in computation time on simple examples and increased savings as number of agents increase.

Chapter 5

Counter-Example Guided Abstraction-Refinement with Symmetries

In this chapter, we introduce a new type of counter-example guided abstraction-refinement algorithm for hybrid automata based on symmetries. The abstraction combines different modes in a concrete automaton $\mathscr{A}$, whose trajectories are related by symmetries, into a single mode in the abstract automaton $\mathscr{B}$. The abstraction sets the guard and reset of an abstract edge to be the union of the symmetry-transformed guards and resets of the concrete edges. We establish the soundness of the abstraction using a forward simulation relation (FSR). The abstract automaton $\mathscr{B}$ can then be verified using existing reachability techniques. In contrast, the previous chapter, verified the concrete automaton while used the concept of virtual system for efficient caching of reachable sets. We present a depth-first-search (DFS) algorithm that checks if a counter-example of $\mathscr{B}$ represents a concrete counter-example of $\mathscr{A}$. We present a refinement algorithm that generate a more accurate abstraction of $\mathscr{A}$ from $\mathscr{B}$ when provided with a spurious counter-example. Our abstraction-refinement algorithm results in simpler automata, that are easier to verify. Moreover, it can be composed with other abstraction-refinement techniques to achieve further savings. In the next chapter, we present a tool implementing a simpler version of the algorithm achieving 14% average speedup in verification time for different scenarios.

5.1 Overview

In Chapter 3, we utilized symmetry of an autonomous dynamical system to cache and retrieve its reachsets using symmetry transformations. When given a continuous family of symmetry transformations, we were able to reduce the dimensionality of the reachsets that had to be computed, achieving orders of magnitude speedup in verification time. In contrast, in this chapter, we focus on hybrid systems instead of continuous dynamical systems. In Chapter 4, we extended the results of Chapter 3

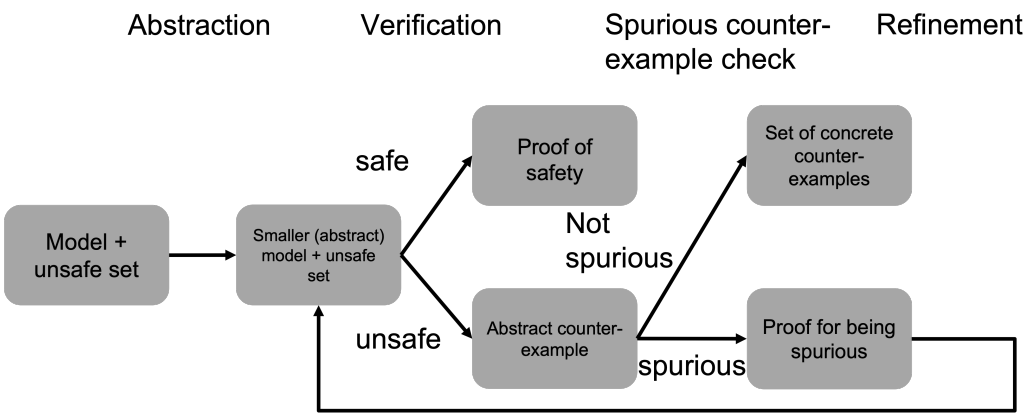


Figure 5.1: The general architecture for a counter-example guided abstraction-refinement algorithm (CEGAR) for safety verification. It has four steps: abstraction, verification, spurious counter-example check, and refinement. Given a model of the system and a property, a safety one in this thesis, to check, the abstraction step results in simpler model and property that are easier to check. The verification step checks if the abstract model is safe, usually using existing verification techniques. If the abstract model is safe, then the original model is safe. This is a property of the abstraction. If it is unsafe instead, the verification step will return a counter-example as a proof of non-safety. The CEGAR algorithm then checks if the counter-example is a spurious one, i.e., a behavior of the abstract model that does not correspond to a behavior of the original model. If it does, then there is a set of concrete counter-examples represented by the abstract one, which the CEGAR algorithm returns as a proof of non-safety of the concrete model. If the abstract counter-example is a spurious one, it is used by the refinement step of the CEGAR algorithm to refine the abstract model to a tighter one. The new abstract model would not have that abstract counter-example as a valid behavior.

to the parameterized dynamical systems and multi-agent settings. That allowed us to tackle hybrid automata as well. We constructed a representative mode p_v of all the modes, using a virtual map which consists of a family of symmetries, and then used p_v as a proxy to share reachsets across different modes using a cache. However, we did not develop the general abstraction-based view of symmetry (Definition 5.1 and Theorem 5.2). Consequently, to verify an automaton, the reachtubes of the modes had to be computed, or retrieved, sequentially. Hence, even if a reachtube of a mode is stored in the cache, the reachtube has to be searched for and retrieved from the cache, which itself might be a computationally expensive operation. In this chapter, we show how this can be avoided through an abstraction-refinement algorithm based on symmetries.

Hybrid system models bring together continuous and discrete behaviors [131, 132, 88], and have proven to be useful in the design and analysis of a wide variety of systems, ranging from automotive, medical, to manufacturing and robotics. Exact algorithmic solutions for many of the synthesis and analysis problems for hybrid systems are known to be computationally intractable [133]. Therefore, one aims to develop approximate solutions, and the main approach is to work with an *abstraction* of the hybrid system model. Roughly, an abstraction of a hybrid automaton $\mathscr{A}$ is a simpler automaton $\mathscr{B}$ that subsumes all behaviors of $\mathscr{A}$. For example, $\mathscr{B}$ may have fewer variables than $\mathscr{A}$, or fewer modes, or it may have linear or rectangular dynamics approximating $\mathscr{A}$'s nonlinear dynamics. $\mathscr{A}$ is called the *concrete* or *real*, and $\mathscr{B}$ the *abstract* or *virtual* automaton. Ideally, $\mathscr{B}$ is simpler and yet a useful over-approximation of $\mathscr{A}$, and the formal analysis and synthesis of $\mathscr{B}$ is tractable.

Given a hybrid automaton $\mathscr{A}$ having a set of discrete states (or *modes*) $\mathscr{P}$ and a virtual map Φ, our abstraction partitions $\mathscr{P}$ to create the abstract automaton $\mathscr{A}_v$. Each *equivalent class* of the partition is represented by a single mode in $\mathscr{A}_v$. Any trajectory ξ of any mode $p \in \mathscr{P}$, can be transformed using Φ to get a trajectory ξ_v of the representative abstract mode p_v, and vice versa. Accordingly, all concrete edges between any two equivalent mode classes would be represented with a single abstract edge. A set of concrete edges represented with the same abstract edge forms an equivalent edge class. The edges of $\mathscr{A}$ are annotated with *guards* and *resets*. These dictate when the discrete transitions over the edges can be taken and how the state would be updated, respectively. The abstraction transforms the guards and resets of all concrete edges using Φ. Then, it unions all of the transformed guards and resets of an edge equivalent class to get the guard and reset

of the corresponding abstract edge. This means that an execution of $\mathscr{A}_v$ would transition over an edge e_v if any of the transformed guards of the edges of $\mathscr{A}$ that e_v represents is satisfied. Moreover, the execution would split into several executions after a reset. Each of these executions start from a transformed version of the state defined by the reset of an edge of $\mathscr{A}$ that is represented by e_v. We establish the soundness of the abstraction using a FSR (Theorem 5.2). We use the robot following a sequence of waypoints scenario described in Example 2.7 as a running example. We show its corresponding abstract hybrid automaton that results from using translation and rotation symmetry maps.

Our abstraction does not change the dimensions of the mode and state spaces, nor does it change the complexity of the dynamics, while resulting in an automaton with fewer modes.

The abstraction can be useful for solving several problems related to tractability of synthesis and verification. We introduce an algorithm that detects *spurious* counter-examples, i.e. executions of the abstract automaton $\mathscr{A}_v$ that violate a specified safety property, but do not correspond to executions of the concrete automaton $\mathscr{A}$. Also, we introduce a refinement algorithm that eliminates detected spurious counter-examples by splitting selected abstract modes. We combine the abstraction, spurious counter-example check, and refinement algorithms in a counter-example abstraction-refinement (CEGAR) verification algorithm based on symmetry. The general architecture of a CEGAR algorithm is shown in Figure 5.1.

We implemented an earlier version of the abstraction-refinement algorithm in a tool called SceneChecker achieving an order of magnitude speedup in verification time of several scenarios [134]. The scenarios included drones and cars navigating cluttered environments with hundreds of obstacles according to predefined plans with hundreds of waypoints. We will discuss SceneChecker in the next chapter. The SceneChecker's version did not guarantee the elimination of a detected spurious counter-example in a single step of refinement. Thus, we expect that the CEGAR algorithm we propose in this chapter to be even faster. Finally, our abstraction method can be composed with other abstraction techniques for achieving further reduction in the size of hybrid automata.

5.1.1 Abstractions for hybrid automata: brief literature review

Simulation relations and abstractions have been an essential part of the formulation of hybrid automata [135, 106]. Several important verification and synthesis techniques for hybrid systems have relied on abstractions. The early decidability results for verification of rectangular hybrid systems were based on creating discrete and finite state abstractions [136, 107, 133]. Decidability results based on abstractions for more general classes were presented in [137, 138]. In a sequence of papers [139, 140, 141], metric-based abstractions were developed for more general hybrid models and shown to be useful for both verification and synthesis. Techniques have been developed for automatically making abstractions more precise based on data and counter-examples [142, 143, 92, 144]. Finally, several practical approaches have been proposed for computing abstractions based on linearization [145], state space partitioning [143, 146], and hybridization [147].

We briefly discuss here some of the existing abstraction-refinement methods for efficient safety verification of hybrid automata. Doyen et al. presented a method that abstracts affine hybrid automata using rectangular hybrid automata [90]. Although it has optimality guarantees, their refinement algorithm does not ensure that a given spurious counter-example is eliminated after a single refinement step. Jha et al. [91] presented an abstraction-refinement algorithm for linear hybrid automata that leverages the power of linear programming for counter-example validation and refinement. Their method does eliminate spurious counter-examples, while not growing the number of continuous variables. Bogomolov et al. [22] defined an abstraction-refinement algorithm that abstracts a given affine hybrid automata in two steps. First, their algorithm overapproximates the dynamics of each mode using a polytope-based differential inclusions. Then, it combines different concrete modes in the same abstract mode. Zutshi et al. [13] designed a CEGAR-based falsification, instead of verification, algorithm for hybrid automata. Their algorithm implicitly constructs tiling-based abstractions while searching for a valid execution of the automaton that violates a given property. It refines the tiling resolution when no valid counter-example is found. A line of research by Prabhakar and Roohi et al. [148, 149, 92] presented a CEGAR-based verification algorithm for nonlinear hybrid automata. Their method abstracts nonlinear hybrid atuomata by partitioning the invariant state space of each mode and over-approximating the dynamics in each part using polytope-based differential inclusions.

An important characteristic of dynamical systems that has *not* been explored

for constructing abstractions in the literature is *symmetry*. That will be the topic of this chapter. Our abstraction method maps the dynamics of the concrete modes exactly to the representative ones of the abstract modes through symmetries instead of over-approximating them using rectangular or polytopic differential inclusions. Thus, our algorithm does not lead to over-approximation errors from continuous dynamics, while the other methods do. Finally, the other abstraction-refinement algorithms are orthogonal to ours and can be composed with it. An abstract automaton generated by our method would have a single representative mode, before refinement, for each set of symmetric concrete modes. Applying one of the existing abstraction-refinement algorithms to the abstract automaton generated by our method further reduces its size by combining its non-symmetric modes.

5.2 Virtual or abstract hybrid automaton

In this section, we present our symmetry-based abstraction of hybrid automata along with the corresponding FSR. Our abstract automata have fewer numbers of modes and edges, than their concrete counterparts.

Our abstraction is an extension of the concept of virtual system for parameterized dynamical systems that we defined in Chapter 4, to hybrid systems. Following the same notation, we use subscript v to denote the components of the abstract, or virtual, automaton and no subscript for those of the concrete, or real, one.

5.2.1 Creating the virtual model

As in Chapter 4, in order to create the virtual model, a virtual map Φ of the real model is needed. The resulting rv map from Φ, as defined in equation 4.2, will be used in Definition 5.1 to map concrete modes to virtual ones. Moreover, its inverse will be used to map virtual modes to the sets of concrete modes, or the equivalent mode classes, that they represent.

In this chapter, we do not assume that there is a single $p_v \in \mathscr{P}$ such that $\forall p \in \mathscr{P}, rv(p) = p_v$, in contrast with the previous chapter. The assumption was made there for simplicity. The results of Chapter 4 can be extended to the general case by creating different caches for each possible value of $rv(p)$, for $p \in \mathscr{P}$.

Definition 5.1 (virtual model)**.** *Given a hybrid automaton $\mathscr{A}$, and a virtual map Φ, the resulting* abstract (virtual) hybrid automaton *is:*

$$\mathscr{A}_v := \langle X_v, \mathscr{P}_v, X_{init,v}, p_{init,v}, E_v, guard_v, reset_v, f_v \rangle, \text{ where}$$

(a) $X_v := X$ *and* $\mathscr{P}_v := \mathscr{P}$

(b) $X_{init,v} := \gamma_{p_{init}}(X_{init})$ *and* $p_{init,v} := rv(p_{init})$,

(c) $E_v := rv(E) = \{(rv(p_1), rv(p_2)) \mid e = (p_1, p_2) \in E\}$

(d) $\forall e_v \in E_v$,

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

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

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

(f) $\forall p_v \in \mathscr{P}_v$, $\forall x_v \in X_v, f_v(x_v, p_v) := f(x_v, p_v)$.

The trajectories and executions of the virtual hybrid automaton $\mathscr{A}_v$ are defined in the same way as those of general hybrid automata in Definition 2.4.

Example 5.1 (Coordinate transformation symmetry)**.** Consider the robot presented in Example 2.7 and the corresponding automaton of the scenario described in Figure 2.6a. Fix a $p^* \in \mathbb{R}^4$. We define $\gamma_{ct} : X \to X$ and $\rho_{ct} : \mathscr{P} \to \mathscr{P}$ to be the maps that transform the coordinate system of the plane where the robot and roads reside. These maps transform the coordinate system so that p^* will be collinear with the $x[0]$-axis and $p^*.dest$ be the origin of the system. Formally, for every $x \in X$ and $p \in \mathscr{P}$,

$$\gamma_{ct}(x) = [\mathbf{R}_\theta(x[0:1] - p^*.dest), x[2] - \theta], \tag{5.1}$$

$$\rho_{ct}(p) = [\mathbf{R}_\theta(p.src - p^*.dest), \mathbf{R}_\theta(p.dest - p^*.dest)], \tag{5.2}$$

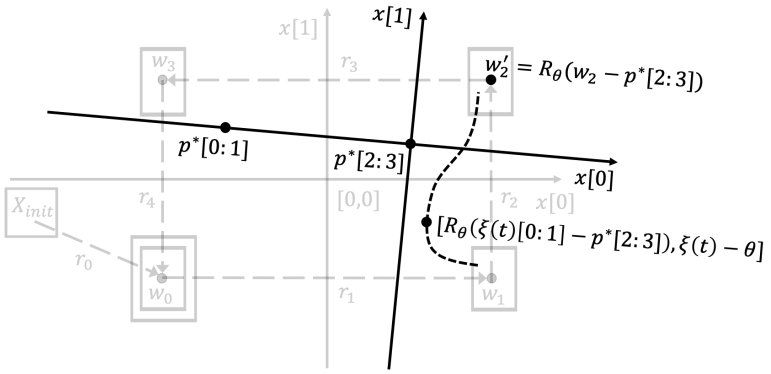


Figure 5.2: Changing the axes of the coordinate system so that: the segment connecting $p^*.src = p^*[0:1]$ to $p^*.dest = p^*[2:3]$ is the new $x[0]$-axis and $p^*.dest$ is the new origin. Such a transformation does not affect the intrinsic behavior of the robot, but only transforms the states in its trajectories to conform with the new coordinate system. Such a coordinate transformation is a valid symmetry.

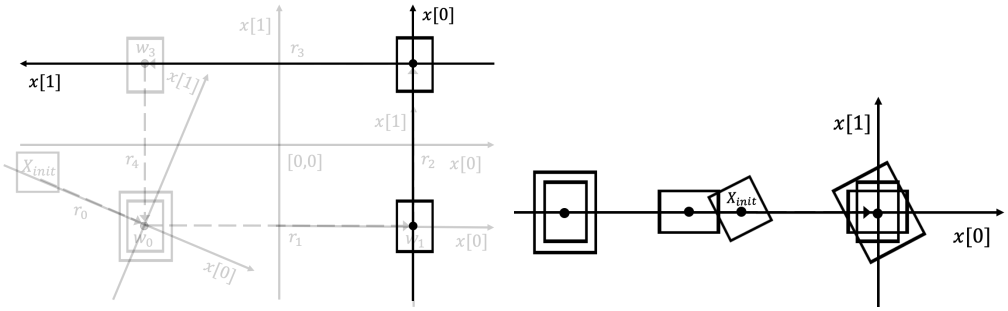
where $\theta = \arctan_2(p^*.dest[1] - p^*.src[1], p^*.dest[0] - p^*.src[0])$ and

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \tag{5.3}$$

is the rotation matrix with angle θ. Then, we can check with simple algebra, that for all $x \in X$ and $p \in \mathscr{P}$, $\frac{\partial \gamma_{ct}}{\partial x} f(x,p) = f(\gamma_{ct}(x), \rho_{ct}(p))$. An example of this coordinate transformation and the resulting new trajectory and waypoint being followed are shown in Figure 5.2.

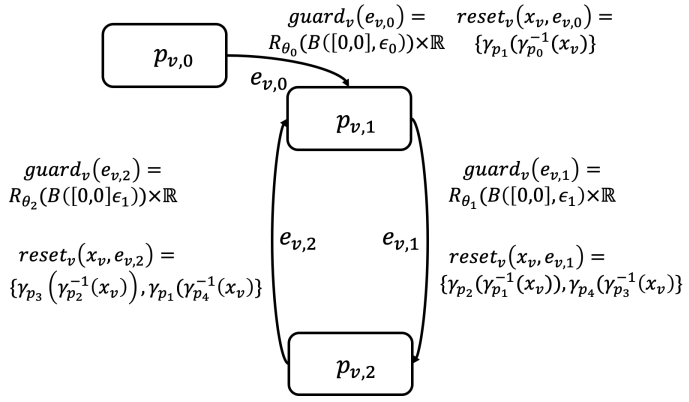
Example 5.2 (Virtual system with a coordinate transformation virtual map)**.** Consider the scenario of the robot traversing a rectangular-shaped path by which we introduced hybrid automata in Chapter 2. We described it in Figure 2.6a and defined its hybrid automaton R in Example 2.7. To construct its virtual automaton, we need a virtual map Φ first. For every mode $p \in \mathscr{P}$, we define γ_p and ρ_p to be the coordinate transformation maps γ_{ct} and ρ_{ct} that we presented in Example 5.1. Recall that we need to fix a mode p^* according to which γ_{ct} and ρ_{ct} transform the coordinate systems of the state and mode spaces. To construct the virtual system, when defining γ_p and ρ_p in Φ, for each mode $p \in \mathscr{P}$, we choose p^* to be equal to p. Figure 5.3a shows a visualization of transforming p_2 using ρ_{p_2}. The concrete mode p_2 gets represented in the virtual automaton by the virtual mode $p_{v,2}$ which is a road on the $x[0]$-axis, as shown in Figures 5.3b and 5.3c.

Similar to the fixed-wing aircraft virtual system in Example 4.1, any map ρ_p in Φ transforms the source waypoint of p to a waypoint on the $x[0]$-axis that preserves



(a) A visualization of transforming p_2 using ρ_{p_2}. The road r_2, which is the concrete mode p_2, becomes the $x[0]$-axis and its destination waypoint $p_2.dest$ becomes the new origin. The original coordinate system and the one resulting from transforming r_0, or equivalently, p_0, using ρ_{p_0} are shaded along with the original scenario.

(b) The abstract (virtual) scenario. It consists of three segments of waypoints, all aligned with the $x[0]$-axis and ending at the origin. The (rotated) rectangles centered at the origin show the guards, projected to the $(x[0],x[1])$-plane, of the abstract edges. The rotated rectangle closest to the origin shows the abstract initial set. The further (rotated) rectangles show the abstract reset applied to the states in the abstract guards, i.e. the potential initial states of the robot following each abstract mode.



(c) The abstract automaton R_v modeling the scenario in Figure 5.3b, where $\forall i \in [0:4], \theta_i = \arctan_2(r_i.dest[1] - r_i.src[1], r_i.dest[0] - r_i.src[0])$.

Figure 5.3: Constructing the abstract automaton described in Example 5.2.

the length of the road. This means that there will be as many virtual modes as there are roads with distinct lengths of roads in the path. Note that the robot in this example, similar to that of the fixed-wing aircraft in Example 4.1, has dynamics that only depend on the destination waypoint, instead of both the segment. Hence, modifying ρ_{ct} to transform both the source and destination waypoints to the origin, i.e. the segment becomes a single point, while preserving γ_{ct} to be the same as that of Example 5.1, would be a valid symmetry of the dynamics. Thus, a family of symmetries that map all roads to the same road with zero length starting and ending at the origin would be a valid virtual map for the robot in this example.

Back to the virtual automaton definition, the set $guard(e_2)$ of $\mathscr{A}$, becomes $B(0,\varepsilon_1^T)\times\mathbb{R}$, after applying γ_{p_2}. This rectangle will be part of the guard of the virtual edge $e_{v,2}$, as specified in Definition 5.1 part (d). Its projection to the first two dimensions is shown as the rectangle $B(0,\varepsilon_1^T)$ centered at the origin in Figure 5.3b.

Figure 5.3a also shows that the rectangle $B(w_1,\varepsilon_1)$, which is $guard(e_1)[0:1]$, becomes $B(w_1-w_2,\varepsilon_1^T)$ a rectangle centered at w_1-w_2 and rotated with $\frac{\pi}{2}$ radians, after applying γ_{p_2}. Recall that the reset of any edge of R is just the identity map. Hence, the rotated rectangle centered at w_1-w_2 represents $\gamma_{p_2}(reset(guard(e_1)),e_1)[0:1]$. This will be part of the set of possible reset states after transitioning over the virtual edge $e_{v,1}$ per part (e) of Definition 5.1. It is shown as the rectangle centered at $p_{v,2}.src$ in Figure 5.3b.

The illustration above for p_2 would be repeated for every $p\in\mathscr{P}$, to construct the virtual system shown in Figure 5.3b.

The resulting virtual automaton would be:

$$R_v=\langle X_v,\mathscr{P}_v,X_{init,v},p_{init,v},E_v,guard_v,reset_v,f_v\rangle,\text{ where}$$

(a) $X_v=X=\mathbb{R}^3$ and $\mathscr{P}_v=\{p_{v,0}=\mathbf{R}_\theta r_0,p_{v,1}=\mathbf{R}_0 r_1,p_{v,2}=\mathbf{R}_{\pi/2}r_2\}$,

(b) $X_{init,v}=\gamma_{p_{init}}(X_{init})$ and $p_{init,v}=rv(p_{init})=p_{v,0}$,

(c) $E_v=\{e_{v,0}=[p_{v,0},p_{v,1}],e_{v,1}=[p_{v,1},p_{v,2}],e_{v,2}=[p_{v,2},p_{v,1}]\}$,

(d)

$$guard_v(e_{v,i})=\begin{cases}\mathbf{R}_\theta B(p_{v,i}.dest,\varepsilon_0)\times\mathbb{R},\text{ if } i=0,\\ \mathbf{R}_0 B(p_{v,i}.dest,\varepsilon_1)\times\mathbb{R},\text{ if } i=1,\\ \mathbf{R}_{\pi/2}B(p_{v,i}.dest,\varepsilon_1)\times\mathbb{R},\text{ if } i=2,\end{cases}$$

where $p_{v,i} = e_{v,i}.src, \forall i \in [3]$,

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

$$reset_v(x_v, e_{v,i}) = \begin{cases} \{\gamma_{p_1}(\gamma_{p_0}^{-1}(x_v))\}, \text{ if } i = 0, \\ \{\gamma_{p_2}(\gamma_{p_1}^{-1}(x_v)), \gamma_{p_4}(\gamma_{p_3}^{-1}(x_v))\}, \text{ if } i = 1, \\ \{\gamma_{p_3}(\gamma_{p_2}^{-1}(x_v)), \gamma_{p_4}(\gamma_{p_1}^{-1}(x_v))\}, \text{ if } i = 2, \end{cases}$$

(f) $\forall x_v \in X_v, \forall p_v \in \mathscr{P}_v, f_v(x_v, p_v) = f(x_v, p_v)$.

The resets of the guards of the three edges in E_v constitute the set of all possible reseted states. They are shown as the rectangles on the $x[0]$-axis, but not at the origin, in Figure 5.3b.

The virtual automaton R_v has three modes and three edges (see Figure 5.3b) versus the five modes and five edges of R. In Figure 5.3b, the guards and reseted guards are overlapping. This suggests that the reach set of R_v has a smaller volume than that of R. That in turn means that the reach sets or reachtubes computations would be generally easier and faster for R_v than that for R.

5.2.2 Forward simulation relation (FSR): from concrete to virtual

In this section, we establish a correspondence from the executions of the concrete system to those of the virtual one through a FSR [93, 94, 95]. A FSR is a standard approach to describe the similarity of behavior of two different hybrid automata.

Definition 5.2 (FSR [93])**.** *A* forward simulation relation *from hybrid automaton $\mathscr{A}_1$ to another one $\mathscr{A}_2$, is a relation $\mathscr{R} \subseteq (X_1 \times \mathscr{P}_1) \times (X_2 \times \mathscr{P}_2)$, such that*

(a) for any initial $x_{0,1} \in X_{init,1}$, there exists a state $x_{0,2} \in X_{init,2}$, such that $(x_{0,1}, p_{init,1})\mathscr{R}(x_{0,2}, p_{init,2})$,

(b) For any discrete transition $(x_1, p_1) \to (x_1', p_1')$ of $\mathscr{A}_1$ and $(x_2, p_2) \in X_2 \times \mathscr{P}_2$, where $(x_1, p_1)\mathscr{R}(x_2, p_2)$, there exists $(x_2', p_2') \in X_2 \times \mathscr{P}_2$ such that $(x_2, p_2) \to (x_2', p_2')$ is a discrete transition of $\mathscr{A}_2$ and $(x_1', p_1')\mathscr{R}(x_2', p_2')$, and

(c) For any solution $\xi_1(x_1, p_1, \cdot)$ of $\mathscr{A}_1$ and pair $(x_2, p_2) \in X_2 \times \mathscr{P}_2$, such that $(x_1, p_1)\mathscr{R}(x_2, p_2)$, there exists a solution $\xi_2(x_2, p_2, \cdot)$, where $dur(\xi_1) = dur(\xi_2)$ and $(\xi_1.lstate, p_1)\mathscr{R}(\xi_2.lstate, p_2)$.

Existence of a FSR implies that for any execution of $\mathscr{A}_1$ there is a corresponding related execution of $\mathscr{A}_2$. The following theorem is an adoption of Corollary 4.23 of [93] into our hybrid modeling framework.

Theorem 5.1 (executions correspondence [93])**.** *If there exists a forward simulation relation $\mathscr{R}$ from $\mathscr{A}_1$ to $\mathscr{A}_2$, then for every execution σ_1 of $\mathscr{A}_1$, there exists a corresponding execution σ_2 of $\mathscr{A}_2$ such that*

(a) $\sigma_1.len = \sigma_2.len$,

(b) $\forall\, i \in [\sigma_1.len]$, $dur(\xi_{1,i}) = dur(\xi_{2,i})$, *and*

(c) $\forall\, i \in [\sigma_1.len]$, $(\xi_{1,i}.lstate, p_{1,i}) \mathscr{R} (\xi_{2,i}.lstate, p_{2,i})$.

Now we introduce a FSR from the concrete hybrid automaton to the virtual one.

Theorem 5.2 (FSR: concrete to virtual)**.** *Consider the relation $\mathscr{R}_{rv} \subseteq (X \times P) \times (X_v \times \mathscr{P}_v)$ defined as $(x,p)\mathscr{R}_{rv}(x_v,p_v)$ iff:*

(a) $x_v = \gamma_p(x)$, *and*

(b) $p_v = rv(p)$.

Then, $\mathscr{R}_{rv}$ is a forward simulation relation from $\mathscr{A}$ to $\mathscr{A}_v$.

Proof. $\mathscr{R}_{rv}$ satisfies Definition 5.2.(a) since: for any $x_0 \in X_{init}$, $\gamma_{p_{init}}(x_0) \in X_{init,v}$, and $p_{init,v} = rv(p_{init})$, by Definition 5.1.(b).

To prove that $\mathscr{R}_{rv}$ satisfies Definition 5.2.(b), fix a discrete transition $(x,p) \to (x',p')$ of $\mathscr{A}$ and $(x_v,p_v) \in X_v \times \mathscr{P}_v$ such that $(x,p,x_v,p_v) \in \mathscr{R}_{rv}$. We will show that if we choose $x'_v = \gamma_{p'}(x_v)$ and $p'_v = rv(p')$, then $(x'_v,p'_v) \in X_v \times \mathscr{P}_v$, $(x',p',x'_v,p'_v) \in \mathscr{R}_{rv}$, and $(x_v,p_v) \to (x'_v,p'_v)$ is a valid discrete transition of $\mathscr{A}_v$.

First, $x'_v \in X_v$ since $x' \in X$, $\gamma_{p'}$ is a map from X to X, and by Definition 5.1.(a), $X = X_v$. Moreover, $p'_v \in \mathscr{P}_v$ by Definition 5.1.(a). Second, $(x',p',x'_v,p'_v) \in \mathscr{R}_{rv}$ since $x'_v = \gamma_{p'}(x')$ and $p'_v = rv(p')$. Third, fix $e = (p,p')$. Then, by the definition of discrete transitions of $\mathscr{A}$, $x \in guard(e)$ and $x' \in reset(x,e)$. Also, from the definition of E_v in Definition 5.1.(c), the edge $e_v = (p_v,p'_v) \in E_v$. Also, by the definition of $\mathscr{R}_{rv}$ and the assumption that x and x_v are related under $\mathscr{R}_{rv}$, $x_v = \gamma_p(x)$. That means that $x_v \in \gamma_p(guard(e))$, since $x \in guard(e)$. But, by Definition 5.1.(d), $\gamma_p(guard(e)) \subseteq guard_v(e_v)$. Then, $x_v \in guard_v(e_v)$. Moreover, since $x' \in reset(x,e))$ and $x = \gamma_p^{-1}(x_v)$, then $x' \in reset(\gamma_p^{-1}(x_v),e)$. Hence, $x'_v = \gamma_{p'}(x') \in \gamma_{p'}(reset(\gamma_p^{-1}(x_v),e))$. Using Definition 5.1.(e), we know that $\gamma_{p'}(reset(\gamma_p^{-1}(x_v),e)) \subseteq$

$reset_v(x_v, e_v)$. We have $x'_v \in reset_v(x_v, e_v)$. Therefore, $(x_v, p_v) \to (x'_v, p'_v)$ is a valid discrete transition of $\mathscr{A}_v$.

To prove that $\mathscr{R}_{rv}$ satisfies Definition 5.2.(c), fix a solution $\xi(x, p, \cdot)$ of $\mathscr{A}$ and a pair $(x_v, p_v) \in X_v \times \mathscr{P}_v$, such that $(x, p, x_v, p_v) \in \mathscr{R}_{rv}$. Then, we will show that $dur(\xi) = dur(\xi_v)$ and $(\xi(x, p, dur(\xi)), p,\ \xi_v(x_v, p_v, dur(\xi)), p_v) \in \mathscr{R}_{rv}$. Since x and x_v are related under $\mathscr{R}_{rv}$, then $x_v = \gamma_p(x)$. Moreover, using Theorem 2.1, $\forall\, t \in dur(\xi)$, $\xi(\gamma_p(x), \rho_p(p), t) = \gamma_p(\xi(x, p, t))$. But, $rv(p) = p_v$ and using Definition 5.1.(f), $\xi(\gamma_p(x), \rho_p(p), \cdot) = \xi_v(\gamma_p(x), p_v, \cdot)$, which is a solution of $\mathscr{A}_v$ starting from $\gamma_p(x) = x_v$. In addition, from the assumption that the guards are optional, we can choose ξ_v that does not transition before $dur(\xi)$. Therefore, $\forall\, t \in dur(\xi)$, $(\xi(x, p, t), p,\ \xi_v(x_v, p_v, t), p_v) \in \mathscr{R}_{rv}$. □

For any execution σ of the concrete automaton $\mathscr{A}$, we denote by $\mathscr{R}(\sigma)$ the corresponding execution of the abstract automaton $\mathscr{A}_v$, per Theorems 5.1. Given a hybrid automaton $\mathscr{A}$, an abstract automaton $\mathscr{A}$, and the corresponding FSR $\mathscr{R}$, we say that $\mathscr{A} \preceq_{\mathscr{R}} \mathscr{A}_v$ to show that all executions of $\mathscr{A}$ correspond to executions in $\mathscr{A}_v$ using the FSR $\mathscr{R}_{rv}$. If there are two abstractions $\mathscr{A}_{v,1}$ and $\mathscr{A}_{v,2}$ of $\mathscr{A}$ with corresponding FSRs $\mathscr{R}_1$ and $\mathscr{R}_2$ which satisfy $\mathscr{A} \preceq_{\mathscr{R}_1} \mathscr{A}_{v,1} \preceq_{\mathscr{R}_2} \mathscr{A}_{v,2}$, then we say that $\mathscr{A}_{v,1}$ is a *tighter* abstraction of $\mathscr{A}$ than $\mathscr{A}_{v,2}$.

The following corollary is also an adoption of Theorem 4.2 of [93] into our hybrid automaton framework. First, let A, B and C be three hybrid automata and Φ_{AB} and Φ_{BC} be two virtual maps of A and B, respectively. Assume that B is the abstract automaton of A resulting from the virtual map Φ_{AB} and C is the abstract automaton of B resulting from the virtual map Φ_{BC}. Denote by $\mathscr{R}_{AB}$ and $\mathscr{R}_{BC}$ the FSRs of the two abstractions, respectively.

Corollary 5.1 (Theorem 4.2 in [93])**.** *C is the virtual automaton of A with FSR $\mathscr{R}_{AB} \circ \mathscr{R}_{BC}$ and virtual map $\Phi_{AC} = \Phi_{AB} \circ \Phi_{BC}$, where $\circ$ is the composition operator.*

Corollary 5.1 shows that we can apply symmetries in sequence to get hierarchical levels of abstractions of $\mathscr{A}$.

It is worth noting that there may not be a forward simulation relation from $\mathscr{A}_v$ to $\mathscr{A}$. The guard and reset of an edge e_v of $\mathscr{A}_v$ are the union of all the transformed versions of the guards and resets of the edges of $\mathscr{A}$ that get mapped to e_v. Hence, some discrete transitions in $\mathscr{A}_v$ may not have corresponding ones in $\mathscr{A}$. For example, consider two edges $e_1 = (p_{11}, p_{12})$ and $e_2 = (p_{21}, p_{22})$ of $\mathscr{A}$ with $rv(e_1) = rv(e_2) = e_v = (p_{v1}, p_{v2})$, an edge of $\mathscr{A}_v$. Then, a transition

over e_v would be allowed in $\mathscr{A}_v$ with reseted state being $\gamma_{p_{22}}(reset(x_v, e_2))$ if $x_v \in \gamma_{p_{11}}(guard(e_1))$. Such a transition may not have a correspondent one in $\mathscr{A}$, since it resembles a transition from p_{11} to p_{22}. Thus, some executions of $\mathscr{A}_v$ may not have corresponding executions in $\mathscr{A}$. This might lead to a scenario where $\mathscr{A}_v$ might not satisfy a property because of an execution that does not correspond to an execution of $\mathscr{A}$. This is discussed further in the next section.

5.3 Counter-example validation

In this section, we present an algorithm that given an execution of the abstract automaton, check if it corresponds to an execution of the concrete automaton.

Recall from Section 2.2.4 that for hybrid automata, we define the unsafe states through a map $O : \mathscr{P} \to 2^X$ that specifies a set of states for each mode. As in Chapter 4, we transform the unsafe sets of states of $\mathscr{A}$ to define the unsafe sets of $\mathscr{A}_v$. More specifically, we define the abstraction $O_v : \mathscr{P}_v \to 2^{X_v}$ of O as follows:

$$\forall p_v \in \mathscr{P}_v, O_v(p_v) := \bigcup_{p \in \mathscr{P} \text{ s.t. } rv(p)=p_v} \gamma_p(O(p)). \tag{5.4}$$

Corollary 5.2. *For any execution σ of the concrete automaton $\mathscr{A}$ such that $\sigma.lstate \in O(\sigma.lmode)$, the execution $\sigma_v := \mathscr{R}_{rv}(\sigma)$ of the abstract automaton $\mathscr{A}_v$ satisfies $\sigma_v.lstate \in O_v(\sigma_v.lmode)$.*

Proof. It follows from Theorems 5.1 and 5.2. □

An execution of a hybrid automaton $\mathscr{A}$ is called a *minimal counter-example* if $\sigma.lstate \in O(\sigma.lmode)$ and $\nexists\, t < dur(\sigma)$ such that $\sigma(t) \in O(\sigma.curp(t))$. Non-minimal counter-examples are executions that reach at least a single unsafe state but not necessarily the last one. In this section, without loss of generality, we consider only minimal counter-examples, and simply denote them by counter-examples.

Definition 5.3. *An execution σ_v of $\mathscr{A}_v$ is a* spurious *counter-example if it is a counter-example of $\mathscr{A}_v$ against O_v but there is no execution σ of $\mathscr{A}$ such that $\mathscr{R}_{rv}(\sigma) = \sigma_v$. Otherwise, we call σ_v an* actual *counter-example.*

It is much easier to distinguish actual counter-examples from spurious ones in our symmetry-based abstraction method than other abstraction techniques in the literature. The reason is that our abstraction does not over-approximate the

continuous dynamics in each mode before combining them, in contrast with all other abstraction frameworks, up to our knowledge.

5.3.1 Description of the counter-example validation procedure validateCE

In this section, we will fix a counter-example execution of the abstract automaton $\mathscr{A}_v$: $\sigma_v = (\xi_{v,0}, p_{v,0}), (\xi_{v,1}, p_{v,1}), \ldots (\xi_{v,k}, p_{v,k})$, where $\xi_{v,k}.lstate \in O_v(p_{v,k})$. Checking if σ_v is spurious can be done using a depth-first-search (DFS) on the concrete mode graph, in case that graph is finite. The pseudocode is shown in Algorithm 5.1, which we name validateCE. The idea is to efficiently check if any of the inverse mappings of σ_v under the FSR in the previous section is a concrete execution that intersects the unsafe sets in O. We define the *inverse set* of σ_v as follows:

$$S := \big\{ s := (\xi_0, p_0), \ldots, (\xi_k, p_k) \mid \forall i \in [0:k], p_i \in rv^{-1}(p_{v,i}) \\ \text{and } \xi_i := \gamma_{p_i}^{-1}(\xi_{v,i}) \big\}. \tag{5.5}$$

Lemma 5.1. *A counter-example σ_v of the abstract hybrid automaton $\mathscr{A}_v$ is an actual counter-example iff there exists $s \in S$ that: (1) is an execution of $\mathscr{A}$, i.e., $s \in Execs_{\mathscr{A}}$, and (2) $s.lstate \in O(s.lmode)$.*

Proof. The elements of S are all the inverse transforms of σ_v under the FSR $\mathscr{R}_{rv}$. □

Checking if an $s = (\xi_0, p_0), \ldots, (\xi_k, p_k) \in S$ belongs to $Execs_{\mathscr{A}}$ can be done efficiently by checking if $\xi_0.fstate \in X_{init}$, $p_0 = p_{init}$, and $\forall i \in [0:k-1]$, $(p_i, p_{i+1}) \in E$, $\xi_i.lstate \in guard((p_i, p_{i+1}))$, and $\xi_{i+1}.fstate \in reset(\xi_i.fstate, (p_i.p_{i+1}))$.

The algorithm validateCE is a recursive procedure that takes as input the abstract counter-example σ_v, the potential concrete counter-example σ constructed so far in previous calls to validateCE, and the index of the element of σ_v being visited, or equivalently, the current depth of the DFS. The algorithm validateCE returns the set of concrete counter-examples Σ represented by σ_v.

If $\gamma_{p_{init}}^{-1}(\xi_{v,0}.fstate) \in X_{init}$, the initial call to validateCE has the following input parameters: (1) σ being set to $\langle(\gamma_{p_{init}}^{-1}(\xi_{v,0}), p_{init})\rangle$ and (2) $i = 0$. Otherwise, validateCE is not called, and σ_v is a spurious counter-example.

Then, validateCE will follow a DFS to construct the set of concrete counter-examples Σ defined in line 3. For any mode $p_i \in rv^{-1}(p_{v,i})$ being visited by the

DFS, validateCE will check which of the concrete modes in $rv^{-1}(p_{v,i+1})$ makes a valid concrete transition from p_i (line 5). That includes checking if (p_i, p_{i+1}) is a concrete edge, if last state of $\gamma_{p_i}^{-1}(\xi_{v,i})$ belongs to its *guard*, and if the first state of $\gamma_{p_{i+1}}^{-1}(\xi_{v,i+1})$ belongs to the *reset* of the last state of the previous trajectory. For each of the valid such modes, validateCE collect all their returned concrete counter-examples in Σ.

In the base case where $i = k$, i.e., i is equal to the length of σ_v, the constructed σ is checked if its last state $\sigma.lstate$ intersects the unsafe set $O(p_k)$ of its last mode p_k. If it does, then it is a concrete counter-example, otherwise it is not. If the returned Σ is non-empty, then σ_v is an actual counter-example. Otherwise, it is a spurious one.

Algorithm 5.1 validateCE(σ_v, σ, i)

1: **if** $i = \sigma_v.len$ **then**
2: **return** $\{\sigma\}$ if $\sigma.lstate \in O(\sigma.lmode)$ and $\emptyset$, otherwise
3: $\Sigma \leftarrow \emptyset$
4: **for** $p_{i+1} \in rv^{-1}(p_{v,i+1})$ **do**
5: **if** $(p_i, p_{i+1}) \in E$ and $\gamma_{p_i}^{-1}(\xi_{v,i}).lstate \in guard(p_i, p_{i+1})$ and $\gamma_{p_{v,i+1}}^{-1}(\xi_{v,i+1}).fstate \in reset(\gamma_{p_i}^{-1}(\xi_{v,i}).lstate, (p_i, p_{i+1}))$ **then**
6: $\Sigma \leftarrow \Sigma \cup$ validateCE$(\sigma_v, \sigma \frown \langle(\gamma_{p_{i+1}}^{-1}(\xi_{v,i+1}), p_{i+1})\rangle, i+1)$
7: **return** Σ

5.3.2 validateCE's correctness

In this section, we show that validateCE is sound and complete. Throughout this section, we fix a counter-example $\sigma_v \in Execs_{\mathscr{A}_v}$ to the bounded safety property of $\mathscr{A}_v$ with the unsafe map O_v. We also fix Σ to be the set returned by validateCE$(\sigma_v, \langle(\gamma_{p_{init}}^{-1}(\xi_{v,0}), p_{init})\rangle, 0)$ if $\gamma_{p_{init}}^{-1}(\xi_{v,0}).fstate \in X_{init}$, and the empty set $\emptyset$, otherwise,

In the following theorem, we show that validateCE is complete.

Theorem 5.3. *If there is a concrete counter-example σ of the bounded safety property of the hybrid automaton $\mathscr{A}$ against the unsafe map O and is represented by the execution σ_v of the abstract automaton $\mathscr{A}_v$, then σ will belong to the output of* validateCE. *That is, if $\exists \sigma \in Execs_{\mathscr{A}}$ such that $\sigma_v = \mathscr{R}_{rv}(\sigma)$ and $\sigma.lstate \in O(\sigma.lmode)$, then $\sigma \in \Sigma$.*

Proof. The proof is by contradiction. Assume that there exists a $\sigma = (\xi_0, p_0), \ldots, (\xi_k, p_k) \in \mathit{Execs}_{\mathscr{A}}$ that satisfy the precondition in the theorem and does not belong to the Σ returned by the call. Then, either (1) $(\xi_0, p_0) \neq (\gamma_{p_{init}}^{-1}(\xi_{v,0}), p_{init})$, (2) the condition in line 2 that $\sigma.lstate \in O(\sigma.lmode)$ was not satisfied, or (3) there exists an $i \in [0 : \sigma.len - 1]$ such that $p_{i+1} \notin rv^{-1}(p_{v,i+1})$, the condition in line (5) was not satisfied, or $\gamma_{p_{i+1}}(\xi_{i+1}) \neq \xi_{v,i+1}$.

By the assumption of the theorem, $\sigma_v = \mathscr{R}_{rv}(\sigma)$. Also, by Theorem 5.2, $(\xi_0, p_0) = (\gamma_{p_{init}}(\sigma.fstate), rv(p_{init}))$. Then, since $\gamma_{p_{init}}$ is invertible, if (1) is true, it leads to a contradiction.

By the assumption of the theorem, $\sigma.lstate \in O(\sigma.lmode)$. Then, if (2) is true, that leads to a contradiction.

By the assumption of the theorem, $\sigma_v = \mathscr{R}_{rv}(\sigma)$ and $\sigma \in \mathit{Execs}_{\mathscr{A}}$. Then, if there is an $i \in [0 : \sigma.len]$ where $p_i \notin rv^{-1}(p_{v,i})$ or $\gamma_{p_i}^{-1}(\xi_{v,i}) \neq \xi_i$, then by Theorems 5.1 and 5.2, $\sigma_v \neq \mathscr{R}_{rv}(\sigma)$, which results in a contradiction. If, instead, one of the predicates in the condition in line (5) is not satisfied, then $(\xi_i.lstate, p_i) \to (\xi_{i+1}.fstate, p_{i+1})$ is not a valid discrete transition of $\mathscr{A}$, and thus $\sigma \notin \mathit{Execs}_{\mathscr{A}}$, which results in a contradiction. □

In the following theorem, we show that validateCE is sound.

Theorem 5.4. *There will be no elements in the output of* validateCE *that are not concrete counter-examples of the bounded safety property of the hybrid automaton $\mathscr{A}$ against the unsafe map O and are represented by the execution σ_v of the abstract automaton $\mathscr{A}_v$. That is, $\forall \sigma \in \Sigma$, $\mathscr{R}_{rv}(\sigma) = \sigma_v$, $\sigma \in \mathit{Execs}_{\mathscr{A}}$, and $\sigma.lstate \in O(\sigma.lmode)$.*

Proof. The proof is by induction on the indices of any sequence of trajectories and modes pairs generated by validateCE. If $\Sigma = \emptyset$, then the theorem is vacuously true. Otherwise, fix a $\sigma = (\xi_0, p_0), \ldots, (\xi_k, p_k) \in \Sigma$. Then, by the assumption on the input parameters for the first call, $\xi_0 = \gamma_{p_{init}}^{-1}(\xi_{v,0})$, $\xi_0.fstate \in X_{init}$, and $p_0 = p_{init}$. Additionally, by Theorem 2.1, $\gamma_{p_{init}}$ being a symmetry, and the assumption that $\xi_{v,0}$ is a trajectory of $\mathscr{A}_v$, we can infer that ξ_0 is a trajectory of $\mathscr{A}$. Moreover, since $\sigma \in \Sigma$, then the condition in line 2 was satisfied and $\sigma.lstate \in O(\sigma.lmode)$. Thus, $\mathscr{R}_{rv}(\sigma[0]) = \sigma_v[0]$, $\sigma[0] \in \mathit{Execs}_{\mathscr{A}}$, and $\sigma.lstate \in O(\sigma.lmode)$.

Fix an $i \in [0, \sigma.len - 1]$ and assume that $\mathscr{R}_{rv}(\sigma[0 : i]) = \sigma_v[0 : i]$ and $\sigma[0 : i] \in \mathit{Execs}_{\mathscr{A}}$. By line 4, $p_{i+1} \in rv^{-1}(p_{v,i+1})$ and by line 6, $\xi_{i+1} = \gamma_{p_{i+1}}^{-1}(\xi_{v,i+1})$. Hence, $\mathscr{R}_{rv}(\sigma[0 : i+1]) = \sigma_v[0 : i+1]$. By line 5 and the induction assumption that

$\xi_i = \gamma_{p_i}^{-1}(\xi_{v,i})$, we can infer that $(p_i, p_{i+1}) \in E$, $\xi_i.lstate \in guard(p_i, p_{i+1})$, and $\xi_{i+1}.fstate \in reset(\xi.lstate, (p_i, p_{i+1}))$. Thus, $(\xi_i.lstate, p_i) \to (\xi_{i+1}.fstate, p_{i+1})$ is a valid discrete transition in $\mathscr{A}$. Moreover, ξ_{i+1} is a trajectory of $\mathscr{A}$ since $\xi_{v,i+1}$ is a trajectory of $\mathscr{A}_v$ and $\gamma_{p_{i+1}}$ is a symmetry. Therefore, $\sigma[0:i+1] \in Execs_{\mathscr{A}}$. □

In summary, validateCE returns all concrete counter-examples of $\mathscr{A}$ represented by the given abstract counter-example of $\mathscr{A}_v$, and nothing else.

5.4 Refinement: eliminating spurious counter-examples

If the counter-example validation procedure validateCE returned an empty Σ, then σ_v is a spurious counter-example and the abstract automaton $\mathscr{A}_v$ should be refined. The aim of the refinement of $\mathscr{A}_v$ is to eliminate the spurious counter-example σ_v.

We refine $\mathscr{A}_v$ by eliminating the possibilities of having an element $s \in S$ defined in equation (5.5) that (1) is not an execution of $\mathscr{A}$ and having $s.lstate \in O(s.lmode)$ or (2) is an execution of $\mathscr{A}$ but having $s.lstate \notin O(s.lmode)$ from being represented by an execution of $\mathscr{A}_v$ that intersects O_v. The pseudocode of our refinement algorithm refine is shown in Algorithm 5.5.

The algorithm refine takes as input the spurious counter-example σ_v, the abstract automaton $\mathscr{A}_v$, the abstract unsafe map O_v, and the map rv from concrete modes of $\mathscr{A}$ to their representative abstract modes of $\mathscr{A}_v$. The map rv was defined in equation (4.2) in Chapter 4. The output of refine is an abstract automaton $\mathscr{A}_{v,ref}$ of $\mathscr{A}$, where $\mathscr{A} \preceq \mathscr{A}_{v,ref} \preceq \mathscr{A}_v$, a corresponding unsafe map $O_{v,ref}$ satisfying equation (5.4), a map rv_{ref} from concrete modes of $\mathscr{A}$ to their representative abstract modes in $\mathscr{A}_{v,ref}$, and a forest-shaped data-structure vv that tracks the abstract modes as they get refined in refine.

The core idea of refine is to recursively split the abstract modes of $\mathscr{A}_v$ to eliminate the illegal discrete transitions that led to the spurious counter-example σ_v. The data-structure vv along with rv_{ref} will keep track of the recursive splits. For each mode $p_v \in \mathscr{P}_v$, $vv(p_v)$ will store a tree with the root being p_v. The leafs of $vv(p_v)$ are the most refined version of p_v. The internal nodes of $vv(p_v)$ store the intermediate modes that resulted from splitting p_v and got further split by refine. We summarize the properties of vv and rv_{ref} in the following proposition.

Proposition 5.1. *The following are invariants of vv and* rv_{ref}*:*

$$\forall\ p_v \in \mathscr{P}_v, vv(p_v).root = p_v, \tag{5.6}$$

$$\forall\ p_v \in \mathscr{P}_v, rv_{ref}^{-1}(p_v) = rv^{-1}(p_v), \tag{5.7}$$

$$\forall\ \textit{non-leaf node} \in vv, \forall\ \textit{child} \in \textit{node.children}, rv_{ref}^{-1}(\textit{child}) \subseteq rv_{ref}^{-1}(\textit{node}), \tag{5.8}$$

$$\forall\ \textit{non-leaf node} \in vv, \bigcap_{\textit{child} \in \textit{node.children}} rv_{ref}^{-1}(\textit{child}) = \emptyset,\ \textit{and} \tag{5.9}$$

$$\forall\ \textit{non-leaf node} \in vv, \bigcup_{\textit{child} \in \textit{node.children}} rv_{ref}^{-1}(\textit{child}) = rv_{ref}^{-1}(\textit{node}). \tag{5.10}$$

For any *node* in *vv*, we denote by $vv^{-1}(node)$ the leafs of the tree with root *node*, i.e. the most refined modes of *node*.

The algorithm refine calls two subroutines refineHelper and splitMode shown in Algorithms 5.4 and 5.2, respectively. The subroutine refineHelper is a recursive procedure that also uses another subroutine splitEdge. The latter's pseudocode is shown in Algorithm 5.3.

splitMode Given a set of concrete modes $rpars \subseteq rv_{ref}^{-1}(node)$, for some $node \in vv$, splitMode creates two children for *node* in *vv*. One of the children *child* would have $rv_{ref}^{-1}(child)$ set to *rpars* and the other child would represent the rest of the concrete modes that were represented by *node*, i.e. $rv_{ref}^{-1}(node) \backslash rpars$. Accordingly, corresponding abstract modes for the newly created children nodes are created for $\mathscr{A}_{v,ref}$ by splitMode. The edges in $\mathscr{A}_{v,ref}$ and the unsafe sets in $O_{v,ref}$ for the new modes are defined according to their definitions in Definition 5.1 and equation (5.4), respectively. In addition to the updated $\mathscr{A}_{v,ref}$, $O_{v,ref}$, rv_{ref}, and *vv*, splitMode returns the first child of *node* that represents *rpars*.

splitEdge Fix three disjoint sets of pairs of concrete modes: $redges_E$, $redges_G$, and $redges_R$ that are subsets of the set of pairs of concrete modes represented by two nodes $node_1$ and $node_2 \in vv$, i.e. $redges_E \cup redges_G \cup redges_R \subseteq (vv^{-1}(node_1), vv^{-1}(node_2))$. Given the three sets, splitEdge would create several children nodes for both $node_1$ and $node_2$ such that for any $child_1 \in node_1.children$ and $child_2 \in node_2.children$, either all $(p_1, p_2) \in (rv_{ref}^{-1}(child_1), rv_{ref}^{-1}(child_2))$ belong to $redges_E$, all belong to $redges_G$, all belong to $redges_R$, or all belong to $(vv^{-1}(node_1), vv^{-1}(node_2)) \backslash (redges_E \cup redges_G \cup redges_R)$. In other words, splitEdge would split the edge $(node_1, node_2)$ to several edges in $\mathscr{A}_{v,ref}$ where each of the new

Algorithm 5.2 splitMode

1: **input:** $p^*_{v,ref}, \mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, rpars$
2: **if** $rpars = \emptyset$ or $p^*_{v,ref}$ is not a leaf node in vv **then return:** $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, \perp, \perp$.
3: Create two new abstract modes $p_{v,1}$ and $p_{v,2}$ and add them to $\mathscr{P}_{v,ref}$.
4: Add $p_{v,1}$ and $p_{v,2}$ as children to $p^*_{v,ref}$ in vv.
5: **for** $p \in rv^{-1}_{ref}(p^*_{v,ref})$ **do**
6: **if** $p \in rpars$ **then** $rv_{ref}(p) \leftarrow p_{v,1}$
7: **else** $rv_{ref}(p) \leftarrow p_{v,2}$
8: $p_{init,v,ref} \leftarrow rv_{ref}(p_{init})$
9: Set $f_{v,ref}(\cdot, p_{v,1}) = f_{v,ref}(\cdot, p_{v,2}) = f_v(\cdot, p_{v,ref})$.
10: $O_{v,ref}(p_{v,1}) \leftarrow \cup_{p \in rv^{-1}_{ref}(p_{v,1})} \gamma_p(O(p))$
11: $O_{v,ref}(p_{v,2}) \leftarrow \cup_{p \in rv^{-1}_{ref}(p_{v,2})} \gamma_p(O(p))$
12: **for** $e_{v,ref} \in E_{v,ref}$ such that $p^*_{v,ref} = e_{v,ref}.dest$ and $e_{v,ref}.src \neq e_{v,ref}.dest$ **do**
13: Create two new edges $(e_{v,ref}.src, p_{v,1})$ and $(e_{v,ref}.src, p_{v,2})$.
14: **for** $e_{v,ref} \in E_{v,ref}$ such that $p^*_{v,ref} = e_{v,ref}.src$ and $e_{v,ref}.src \neq e_{v,ref}.dest$ **do**
15: Create two new edges $(p_{v,1}, e_{v,ref}.dest)$ and $(p_{v,2}, e_{v,ref}.dest)$.
16: **if** $\exists\, e_v \in E_v$ such that $e_v.src = e_v.dest = p^*_{v,ref}$ **then**
17: Create four new edges $(p_{v,1}, p_{v,1})$, $(p_{v,2}, p_{v,2})$, $(p_{v,1}, p_{v,2})$, and $(p_{v,2}, p_{v,1})$.
18: Define the guards and resets of the new edges using rv_{ref} according to Definition 5.1.
19: **return:** $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, p_{v,1}, p_{v,2}$

edges represent purely a subset of one of the four sets that partition the pairs of modes that the edge $(node_1, node_2)$ represents. Note that it is possible that $node_1 = node_2$ and splitEdge would create children nodes for $node_1$ according to the same description above. Finally, splitEdge would define the edges and unsafe sets of the the newly created according to their definitions in Definition 5.1 and equation (5.4), respectively. In addition to the updated $\mathscr{A}_{v,ref}$, $O_{v,ref}$, rv_{ref}, and vv, splitEdge returns the resulting set of all abstract edges of the form $\{(child_1, child_2) | child_1 \in node_1.children, child_2 \in node_2.children\}$.

refineHelper Given any edge $e_{v,ref} \in E_{v,ref}$ from $node_1$ to $node_2 \in vv$ and an index $i \in [0 : \sigma_v.len - 1]$, the subroutine refineHelper, in lines 1- 8, defines:

1. $redges_E$ to be the set of pairs of concrete modes of the form $(p_{i-1}, p_i) \in rv^{-1}_{ref}((node_1, node_2))$ that are not concrete edges, i.e., $(p_{i-1}, p_i) \notin E$,

2. $redges_G$ to be the set of pairs of concrete modes of the form $(p_{i-1}, p_i) \in rv^{-1}_{ref}((node_1, node_2))$ that are concrete edges but having the $\gamma^{-1}_{p_{i-1}}$-transformed $(i-1)^{th}$ trajectory $\xi_{v,i-1}$ of the spurious counter-example σ_v not ending in

Algorithm 5.3 splitEdge

1: **input:** $e^*_{v,ref}, \mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, redges_E, redges_G, redges_R$
2: **if** $redges_E = redges_G = redges_R = \emptyset$ or any pair of the three sets has a non-empty intersection or one of the modes $e^*_{v,ref}$ is a non-leaf node in vv **then return:** $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, \perp$.
3: $(p^*_{v,1}, p^*_{v,2}) \leftarrow e^*_{v,ref}$
4: **for** $p_{v,1} \in vv^{-1}(p^*_{v,1})$ **do**
5: **for** $p_1 \in rv^{-1}_{ref}(p_{v,1})$ **do**
6: **for** $p_{v,2} \in vv^{-1}(p^*_{v,2})$ **do**
7: $rpars \leftarrow \emptyset$
8: **for** $p_2 \in rv^{-1}_{ref}(p_{v,2})$ **do**
9: **if** $(p_1, p_2) \in redges_E$ **then**
10: $rpars \leftarrow rpars \cup \{p_2\}$
11: $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, p_{v,ref,1}, p_{v,ref,2} \leftarrow \mathsf{splitMode}(p_{v,2}, \ldots, rpars)$
12: Repeat lines 6-11 with $redges_E$ replaced with $redges_G$
13: Repeat lines 6-11 with $redges_E$ replaced with $redges_R$
14: Repeat lines 4-13 with $p^*_{v,1}$ and $p^*_{v,2}$, $p_{v,1}$ and $p_{v,2}$, and p_1 and p_2 switched.
15: Set $\{e_{v,ref,j}\}_{j\in[N]}$ to be the resulting edges between the most refined versions of $p^*_{v,1}$ and $p^*_{v,2}$.
16: **return:** $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, \{e_{v,ref,j}\}_{j\in[N]}$

$guard(p_{i-1}, p_i)$, i.e. $(p_{i-1}, p_i) \in E$ and $\gamma^{-1}_{p_{i-1}}(\xi_{v,i-1}).lstate \notin guard(p_{i-1}, p_i)$, and

3. $redges_R$ to be the set of pairs of concrete modes of the form $(p_{i-1}, p_i) \in rv^{-1}_{ref}((node_1, node_2))$ that are concrete edges and having the $\gamma^{-1}_{p_{i-1}}$-transformed $(i-1)^{th}$ trajectory $\xi_{v,i-1}$ of the spurious counter-example σ_v ending in $guard(p_{i-1}, p_i)$, but the $\gamma^{-1}_{p_i}$-transformed i^{th} trajectory $\xi_{v,i}$ of the spurious counter-example σ_v does not start in $reset(\gamma^{-1}_{p_{i-1}}(\xi_{v,i-1}).lstate, (p_{i-1}, p_i))$, i.e., $(p_{i-1}, p_i) \in E$, $\gamma^{-1}_{p_{i-1}}(\xi_{v,i-1}).lstate \in guard(p_{i-1}, p_i)$, and $\gamma^{-1}_{p_i}(\xi_{v,i}).fstate \notin reset(\gamma^{-1}_{p_{i-1}}(\xi_{v,i-1}).lstate, (p_{i-1}, p_i))$.

After that, refineHelper calls splitEdge to split $e_{v,ref}$ into different abstract edges to separate the concrete edges from $redges_E$, $redges_G$, $redges_R$, and $rv^{-1}_{ref}(e_{v,ref}) \setminus redges_E \cup redges_G \cup redges_R$ from being represented together. Then, if $i \geq 1$, refineHelper is called recursively to split all the refined versions of the $(i-1)^{th}$ edge in the spurious counter-example σ_v. To do that, refineHelper iterates over all the edges $\{e'_{v,j}\}_{j\in[N]}$ returned by splitEdge in line 11. In a nested loop in line 12, refineHelper iterates over the most refined versions of the j^{th} returned edge: $vv^{-1}(e'_{v,j})$. In a third nested loop, refineHelper iterates over the most refined

versions of the $(i-1)^{th}$ edge in σ_v: $vv^{-1}(e_{v,i-1})$. If the source node (or, equivalently, mode) of an edge in $vv^{-1}(e'_{v,j})$ is equal to the destination node of an edge in $vv^{-1}(e_{v,i-1})$, refineHelper calls itself recursively to split the latter edge in line 15.

Although refineHelper recurses backwards starting from the last edge σ_v, an edge appearing earlier in σ_v might have been refined by the time its index has been reached by refineHelper since the mode graph of $\mathscr{A}_{v,ref}$ might have cycles. Moreover, after each recursive call to refineHelper in line 15, more nodes might have been added to vv. That is the reason refineHelper iterates of the vv^{-1} of the edges in lines 12 and 13, and not the edges themselves.

If $i = 0$, refineHelper calls splitMode to split the refined versions of the source modes of all the refined versions of the edges of $e_{v,ref}$ returned by splitEdge in line 24. In line 17, refineHelper iterates over the returned edges by splitEdge. For each returned edge $e'_{v,j}$, refineHelper iterates over its refined versions $vv^{-1}(e'_{v,j})$ in line 18. For each of refined edge $e'_{v,c}$, refineHelper iterates in line 20 over the concrete modes represented by its source mode, i.e. $rv^{-1}_{ref}(e'_{v,c}.src)$, to construct the set *rpars*. For any concrete mode $p_0 \in rv^{-1}_{ref}(e'_{v,c}.src)$, refineHelper checks if the $\gamma^{-1}_{p_0}$-transformed first trajectory $\xi_{v,0}$ of σ_v starts from X_{init}, i.e. $\gamma^{-1}_{p_0}(\xi_{v,0}).fstate \in X_{init}$. If that condition is satisfied, p_0 is added to *rpars* in line 24. Finally, splitMode is called in each iteration of the refined edges to split their sources modes according to the constructed *rpars* in line 22.

Back to describing refine. It first initializes the forest vv with root nodes being the modes of $\mathscr{A}_v$ in line 1. Then, it constructs the set *rpars* to split the last mode $p_{v,k}$ visited by the spurious counter-example σ_v in lines 2-5. The aim of splitting $p_{v,k}$ is to eliminate all $s \in S$ of equation 5.5 which do not intersect the unsafe set, i.e. $s.lstate \notin O(s.lmode)$. To do that, refine adds any $p_k \in rv^{-1}(p_{v,k})$ for which the $\gamma^{-1}_{p_k}$-transformed last state $\xi_{v,k}.lstate$ of σ_v does not intersect $O(p_k)$, i.e., $\gamma^{-1}_{p_k}(\xi_{v,k}.lstate) \notin O_{v,ref}(p_k)$, to *rpars* in line 5. Then, refine calls splitMode to split $p_{v,k}$ in line 6. The returned values from that call initialize $\mathscr{A}_{v,ref}$, $O_{v,ref}$, and rv_{ref}, and update vv. The new mode p'_v in $\mathscr{A}_{v,ref}$ that represents the concrete modes in *rpars* is also returned by splitMode. If the length k of σ_v is longer than one, then refine iterates over the refined versions of the edge $e_{v,k-1}$ in line 8, i.e. $vv^{-1}(e_{v,k-1})$. For any $e_{v,ref} \in vv^{-1}(e_{v,k-1})$, if its destination mode belongs to the set of refined versions of p'_v, i.e. to $vv^{-1}(p'_v)$, refine calls refineHelper to recursively split $e_{v,ref}$ in line 10. As mentioned earlier, refine iterates over the refined versions of $e_{v,k}$ since the latter might got refined in earlier calls to refineHelper and the most refined versions are stored in vv.

Algorithm 5.4 refineHelper($\sigma_v, \mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, e_{v,ref}, i$)

$redges_E \leftarrow \emptyset$, $redges_G \leftarrow \emptyset$, $redges_R \leftarrow \emptyset$
for $(p_{i-1}, p_i) \in rv_{ref}^{-1}(e'_v)$ **do**
    if $(p_{i-1}, p_i) \notin E$ **then**
        $redges_E \leftarrow redges_E \cup \{(p_{i-1}, p_i)\}$
    else if $\gamma_{p_{i-1}}^{-1}(\xi_{v,i-1}).lstate \notin guard(p_{i-1}, p_i)$ **then**
        $redges_G \leftarrow redges_G \cup \{(p_{i-1}, p_i)\}$
    else if $\gamma_{p_i}^{-1}(\xi_{v,i}).fstate \notin reset(\gamma_{p_{i-1}}^{-1}(\xi_{v,i-1}).lstate, (p_{i-1}, p_i))$ **then**
        $redges_R \leftarrow redges_R \cup \{(p_{i-1}, p_i)\}$
$\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, \{e'_{v,j}\}_{j\in[N]} \leftarrow$ splitEdge$(e_{v,ref}, \ldots, redges_R)$
if $i \geq 1$ **then**
    for $j \in [N]$ **do**
        for $e'_{v,c} \in vv^{-1}(e'_{v,j})$ **do**
            for $e'_{v,n} \in vv^{-1}(e_{v,i-1})$ **do**
                if $e'_{v,c}.src = e'_{v,n}.dest$ **then**
                    $\mathscr{A}_{v,ref}, O'_v, rv_{ref}, vv \leftarrow$ refineHelper$(\sigma_v, \ldots, e'_{v,n}, i-1)$
else if $i = 0$ **then**
    for $j \in [N]$ **do**
        for $e'_{v,c} \in vv^{-1}(e_{v,j})$ **do**
            $rpars \leftarrow \emptyset$
            for $p_0 \in rv_{ref}^{-1}(e'_{v,c}.src)$ **do**
                if $\gamma_{p_0}^{-1}(\xi_{v,0}).fstate \notin X_{init}$ **then**
                    $rpars \leftarrow rpars \cup \{p_0\}$
            $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, p_{v,ref,1}, p_{v,ref,2} \leftarrow$
                splitMode$(e'_{v,c}.src, \ldots, rpars)$
return $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv$

Finally, refine deletes all modes in $\mathscr{A}_{v,ref}$ that correspond to non-leaf nodes of vv. It deletes as well their corresponding entries in $O_{v,ref}$ and rv_{ref} and all connected edges. This is done to retain the most refined versions of any abstract mode and at the same time ensure that each concrete mode is represented by a single abstract mode in $\mathscr{A}_{v,ref}$.

Algorithm 5.5 refine$(\sigma_v, \mathscr{A}_v, O_v, rv)$

1: $vv \leftarrow \{p_v : p_v \mid p_v \in \mathscr{P}_v\}$
2: $rpars \leftarrow \emptyset$
3: **for** $p_k \in rv^{-1}(p_{v,k})$ **do**
4: **if** $\gamma_{p_k}^{-1}(\xi_{v,k}).lstate \notin O(p_k)$ **then**
5: $rpars \leftarrow rpars \cup \{p_k\}$
6: $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv, p_{v,ref,1}, p_{v,ref,2} \leftarrow$ splitMode$(p_{v,k}, \mathscr{A}_v, O_v, rv, vv, rpars)$
7: **if** $k \geq 1$ **then**
8: **for** $e_{v,ref} \in vv^{-1}(e_{v,k-1})$ **do** ▷ vv gets updated inside the loop
9: **if** $e_{v,ref}.dest \in vv^{-1}(p_{v,ref,1})$ **then**
10: $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv \leftarrow$ refineHelper$(\sigma_v, \ldots, e_{v,ref}, k-1)$
11: Delete all modes in $\mathscr{A}_{v,ref}$ that are non-leaf nodes in vv along with connected edges and unsafe sets entries in $O_{v,ref}$ and rv_{ref}.
12: **return** $\mathscr{A}_{v,ref}, O_{v,ref}, rv_{ref}, vv$

5.4.1 Correctness of refinement subroutine

In this section, we prove that refine is correct: it eliminates a given spurious counter-example and results in a tighter abstract automaton of the concrete one.

Definition 5.4. *A spurious counter-example σ_v of $\mathscr{A}_v$ is eliminated in the refined automaton $\mathscr{A}_{v,ref}$, iff $\nexists\ s \in S$, the set of candidate concrete counter-examples we defined in equation (5.5), that has a corresponding execution of $\mathscr{A}_{v,ref}$ that intersects the unsafe sets in $O_{v,ref}$. More formally, $\forall s \in S$, $\mathscr{R}_{rv_{ref}}(s) \notin Execs_{\mathscr{A}_{v,ref}}$ or $\mathscr{R}_{rv_{ref}}(s).lstate \notin O_{v,ref}(rv_{ref}(s.lmode))$, where $\mathscr{R}_{rv_{ref}}$ is the FSR relating $\mathscr{A}$ to $\mathscr{A}_{v,ref}$.*

Theorem 5.5. *Fix any spurious counter-example $\sigma_v \in Execs_{\mathscr{A}_v}$. Then, the automaton $\mathscr{A}_{v,ref}$ and unsafe map $O_{v,ref}$ generated by* refine$(\sigma_v, \mathscr{A}_v, O_v, rv)$ *are tighter abstractions of $\mathscr{A}$ and O than $\mathscr{A}_v$ and O_v which eliminate the counter-example σ_v.*

Proof. The output automaton $\mathscr{A}_{v,ref}$, the unsafe map $O_{v,ref}$, the map rv_{ref} from concrete to abstract modes of $\mathscr{A}_{v,ref}$, and the forest vv that map each mode of $\mathscr{A}_{v,ref}$

to their refined modes, only get updated in the splitMode and splitEdge calls in line 6 in refine and lines 9 and 24 in refineHelper. Moreover, in line 11, refine delete all modes in $\mathscr{A}_{v,ref}$ and $O_{v,ref}$ that are non-leaf modes in vv. The outputs $\mathscr{A}_{v,ref}$ and $O_{v,ref}$ of any of these two functions, with non-leaf modes in vv deleted, are tighter abstractions of $\mathscr{A}$ and O, as will be proved in the next section. Moreover, in both splitMode and splitEdge, vv and rv_{ref} maintain the maps from any mode to its refined versions and from $\mathscr{P}$ to $\mathscr{P}_{v,ref}$, respectively. Thus, the outputs $\mathscr{A}_{v,ref}$, $O_{v,ref}$, rv_{ref}, and vv of refine are a tighter abstraction of $\mathscr{A}$ than $\mathscr{A}_v$, a tighter abstraction of O than O_v, the map from concrete modes of $\mathscr{A}$ to abstract modes of $\mathscr{A}_{v,ref}$, and the map from the modes of $\mathscr{A}_{v,ref}$ to their refined versions, respectively.

The second part of the proof is by contradiction. Assume that there exists an element $s^* = (\xi_0^*, p_0^*), \ldots, (\xi_k^*, p_k^*) \in S$, defined in equation (5.5) for σ_v, such that $\mathscr{R}_{rv_{ref}}(s^*) = \sigma_{v,ref}$ and $\sigma_{v,ref}.lstate \in O_{v,ref}(\sigma_{v,ref}.lmode)$, for some $\sigma_{v,ref} \in Execs_{\mathscr{A}_{v,ref}}$. By Lemma 5.1 and the assumption that σ_v is a spurious counter-example, we know that $s^* \notin Execs_{\mathscr{A}}$ or $s^*.lstate \notin O(s^*.lmode)$.

If $s^*.lstate \notin O(s^*.lmode)$, then $s^*.lmode$ will be added to *rpars* in line 5 of refine. Thus, after the call to splitMode in line 6 in refine, $s^*.lmode$ will be represented, along with the other modes in *rpars*, by a separate mode in $\mathscr{A}_{v,ref}$ than those not in *rpars*. Lets call that mode $p^*_{v,ref}$. Any mode $p_k \in rpars$ has the property that $\gamma_{p_k}^{-1}(\xi_{v,k}).lstate \notin O(p_k)$. Consequently, $\xi_{v,k}.lstate \notin \cup_{p_k \in rpars} \gamma_{p_k}(O(p_k))$. The right-hand-side is equal to $O_{v,ref}(p_{v,ref})$, according to equation (5.4). However, since $\mathscr{A} \preceq_{\mathscr{R}_{rv_{ref}}} \mathscr{A}_{v,ref} \preceq_{\mathscr{R}_{rv}} \mathscr{A}_v$, then $\sigma_{v,ref}.lstate = \xi_{v,k}.lstate$. Therefore, $\mathscr{R}_{rv_{ref}}(s).lstate \notin O_{v,ref}(p_{v,ref})$. All splits of $p^*_{v,ref}$ after line 6 will preserve this property, by replacing $O_{v,ref}(p_{v,ref})$ by the union of the unsafe sets of the resulting modes after splits, i.e. $\cup_{p_{v,ref} \in vv^{-1}(p^*_{v,ref})}(O_{v,ref}(p_{v,ref}))$. In conclusion, if $s^*.lstate \notin O(s^*.lmode)$, then $\sigma_{v,ref}.lstate \notin O_{v,ref}(\sigma_{v,ref}.lmode)$, which is a contradiction.

From the previous paragraph, we can conclude that $s^* \notin Execs_{\mathscr{A}}$, for the assumption that $\sigma_{v,ref}$ is a counter-example of $\mathscr{A}_{v,ref}$ to hold. Therefore, either (1) there exists an $i \in [\sigma_v.len]$ such that the $(i-1)^{th}$ discrete transition in s^* from mode p^*_{i-1} to mode p^*_i is not allowed in $\mathscr{A}$, i.e., $(p^*_{i-1}, p^*_i) \notin E$, $\gamma^{-1}_{p^*_{i-1}}(\xi_{v,i-1}).lstate \notin guard(p^*_{i-1}, p^*_i)$, or $\gamma^{-1}_{p^*_i}(\xi_{v,i}).fstate \notin reset(\gamma^{-1}_{p^*_{i-1}}(\xi_{v,i-1}).lstate, (p^*_{i-1}, p^*_i))$, or (2) $s^*.fstate \notin X_{init}$.

If (1) is True, then (p^*_{i-1}, p^*_i) will be added to $redges_E$ in line 4, $redges_G$ in line 6, or $redges_R$ in line 8 in refineHelper. Thus, if (p^*_{i-1}, p^*_i) was added to $redges_E$, after

the call to splitEdge in line 9 in refineHelper, (p^*_{i-1}, p^*_i) will be represented, along with some of the other edges in $redges_E$, by a separate edge in $\mathscr{A}_{v,ref}$ than those not in $redges_E$, and possibly other edges in $redges_E$ as well. The same would happen if (p^*_{i-1}, p^*_i) was added to $redges_G$ or $redges_R$ instead. Lets call the edge that represents (p^*_{i-1}, p^*_i) in $\mathscr{A}_{v,ref}$ by $e^*_{v,ref}$. Any pair of modes in $redges_E$ has the property that it is not an edge in $\mathscr{A}$. Consequently, the edges in $\mathscr{A}_{v,ref}$ that represent pairs of modes in $redges_E$ will have empty guards, since they do not represent concrete edges in $\mathscr{A}$, and can be removed from $\mathscr{A}_{v,ref}$ without any alteration of its semantics.

Similarly, any edge (p_{i-1}, p_i) in $redges_G$ has the property that $\gamma^{-1}_{p_{i-1}}(\xi_{v,i-1}).lstate \notin guard(p_{i-1}, p_i)$. Hence, $\xi_{v,i-1}.lstate \notin \cup_{(p_{i-1},p_i)\in redges_G} \gamma_{p_{i-1}}(guard(p_{i-1}, p_i))$. The latter is equal to $guard_{v,ref}(e^*_{v,ref})$, according to Definition 5.1.(d). However, since $\mathscr{A} \preceq_{\mathscr{R}_{rv_{ref}}} \mathscr{A}_{v,ref} \preceq_{\mathscr{R}_{rv}} \mathscr{A}_v$, $\mathscr{R}_{rv_{ref}}(\xi^*_{i-1}).lstate = \xi_{v,i-1}.lstate$, where ξ^*_{i-1} is the $(i-1)^{th}$ trajectory in s^*. Therefore, $\mathscr{R}_{rv_{ref}}(\xi^*_{i-1}).lstate \notin guard_{v,ref}(e^*_{v,ref})$. All splits of $e^*_{v,ref}$ after line 9 will preserve this property, by replacing $guard_{v,ref}(e^*_{v,ref})$ by the union of the guards of the resulting edges after splits: $\cup_{e_{v,ref}\in vv^{-1}(e^*_{v,ref})} guard_{v,ref}(e_{v,ref})$. A similar analogy can be constructed for the case where $(p^*_{i-1}, p^*_i) \in redges_R$. That means that the i^{th} discrete transition in $\sigma_{v,ref}$ is not allowed in $\mathscr{A}_{v,ref}$ and thus $\sigma_{v,ref} \notin Execs_{\mathscr{A}_{v,ref}}$, which is a contradiction.

Finally, if (2) is True, then p^*_0 will be added to *rpars* in line 22 of refineHelper. Consequently, p^*_0 will be represented, along with other p_0 that satisfy $\gamma^{-1}_{p_0}(\xi_{v,0}).fstate \notin X_{init}$, in the same mode $p^*_{v,ref}$ in the call to splitMode in line 24. Consequently, $\xi_{v,0}.fstate \notin \cup_{p_0\in vv^{-1}(p^*_{v,ref})} \gamma_{p_0}(X_{init})$. Thus, $p_{init} \notin vv^{-1}(p^*_{v,ref})$, since $\sigma_v \in Execs_{\mathscr{A}_v}$ and $\xi_{v,0}.fstate \in X_{init,v} = \gamma_{p_{init}}(X_{init}) = X_{init,v,ref}$, according to Definition 5.1. However, since $\mathscr{A} \preceq_{\mathscr{R}_{rv_{ref}}} \mathscr{A}_{v,ref} \preceq_{\mathscr{R}_{rv}} \mathscr{A}_v$, then $\sigma_{v,ref}.fstate = \xi_{v,0}.fstate$. Therefore, $\sigma_{v,ref}.fstate \notin X_{init,v,ref}$. All splits of $p^*_{v,ref}$ in different iterations of the loop in line 18 will preserve this property. In conclusion, if $s^*.fstate \notin X_{init}$, then $\sigma_{v,ref}.fstate \notin X_{init,v,ref}$, which is a contradiction.

We conclude that there is no $s \in S$ that satisfies $\mathscr{R}_{rv_{ref}}(s) \in Execs_{\mathscr{A}_{v,ref}}$ and $s.lstate \in O_{v,ref}(s.lmode)$. In other words, refine eliminates the spurious counter-example and result in a tighter automaton and unsafe set than the given ones. □

5.4.2 Discussion of splitMode and splitEdge

In this section, we more thoroughly describe splitMode and splitEdge, the algorithms used by refine and refineHelper to split an abstract mode $p^*_{v,ref}$ and an abstract edge $e^*_{v,ref}$. Then, we prove that their resulting $\mathscr{A}_{v,ref}$s are abstractions of $\mathscr{A}$ and that $\mathscr{A}_v$ is an abstraction of the $\mathscr{A}_{v,ref}$s, using two forward simulation relations.

splitMode The procedure splitMode takes as input the mode to be split $p^*_{v,ref}$, the automaton $\mathscr{A}_{v,ref}$, the unsafe map $O_{v,ref}$, the map rv_{ref} from concrete modes to modes of $\mathscr{A}_{v,ref}$, the map vv that maps modes of $\mathscr{A}_{v,ref}$ to their refined versions, and a set of concrete modes $rpars \subseteq rv^{-1}_{ref}(p^*_{v,ref})$. It outputs the updated refined automaton $\mathscr{A}_{v,ref}$, the updated refined unsafe map $O_{v,ref}$, the updated map rv_{ref} from concrete modes to refined abstract modes, the updated forest vv, the new mode $p_{v,1}$ it created in $\mathscr{A}_{v,ref}$ to represent the concrete modes in $rpars$, and the new mode $p_{v,2}$ it created in $\mathscr{A}_{v,ref}$ to represent the concrete modes in $rv^{-1}_{ref}(p^*_{v,ref})\backslash rpars$.

It starts by checking if $rpars$ is the empty set or the mode $p^*_{v,ref}$ to be refined is not a leaf node in vv. If that was true, it does not refine and simply returns its input in line 2. Otherwise, it creates two new abstract modes $p_{v,1}$ and $p_{v,2}$ in $\mathscr{P}_{v,ref}$ in line 3. It adds these two modes as two child nodes of $p^*_{v,ref}$ in vv in line 4. After that, it updates the map rv_{ref} to map modes in $rpars$ to $p_{v,1}$ and the rest of the modes in $rv^{-1}_{ref}(p^*_{v,ref})$ to $p_{v,2}$ in lines 5-7. It then updates the initial mode of $\mathscr{A}_{v,ref}$ in case the split mode was the root in line 8. It sets the dynamics of the two new modes to be the same as that of $p^*_{v,ref}$ in line 9. It sets the unsafe sets for the two modes in lines 10 and 11 according to the definition of abstract unsafe map in equation (5.4) Now that it created the new modes, it proceeds into updating the edges of $\mathscr{A}_{v,ref}$ and their $guard_{v,ref}$ and $reset_{v,ref}$ annotations in lines 12-18. Finally, it returns the updated variables in addition to the two new modes.

Correctness guarantees of splitMode In this section, without loss of generality, we assume that the input to splitMode is the abstract automaton $\mathscr{A}_v$ of the concrete automaton $\mathscr{A}$ and its output is $\mathscr{A}_{v,ref}$. We show that $\mathscr{A}_{v,ref}$, with modes corresponding to non-leaf nodes in vv and connected edges removed from $\mathscr{P}_{v,ref}$ and $E_{v,ref}$, is an abstraction of $\mathscr{A}$, and is tighter than $\mathscr{A}_v$. We show that by proving that $\mathscr{A}_v$ is an abstraction of $\mathscr{A}_{v,ref}$.

Consider $\mathscr{R}_{rv_{ref}}$, the same relation as $\mathscr{R}_{rv}$ defined in Theorem 5.2, but using

rv_{ref}, with entries corresponding to non-leaf nodes in vv removed, instead of rv. Formally, $\mathscr{R}_{rv_{ref}} \subseteq (X \times \mathscr{P}) \times (X_{v,ref} \times \mathscr{P}_{v,ref})$, defined as $(x,p)\mathscr{R}_{rv_{ref}}(x_{v,ref}, p_{v,ref})$ iff:

(a) $x_{v,ref} = \gamma_p(x)$, and

(b) $p_{v,ref} = rv_{ref}(p)$, or equivalently, the leaf node in vv representing p.

Let us refer to $\mathscr{R}_{rv_{ref}}$ by $\mathscr{R}_{rv,1}$ and let $\mathscr{R}_{rv,2} \subseteq (X_{v,ref} \times \mathscr{P}_{v,ref}) \times (X_v \times \mathscr{P}_v)$ be defined as: $(x_{v,ref}, p_{v,ref})\mathscr{R}_{rv,2}(x_v, p_v)$ iff:

(a) $x_v = x_{v,ref}$, and

(b)

$$p_v = \begin{cases} p_{v,ref}, \text{ if } p_{v,ref} \notin \{p_{v,1}, p_{v,2}\}, \\ p^*_{v,ref}, \text{ otherwise.} \end{cases} \tag{5.11}$$

The following theorem shows that these two relations are forward simulation relations from $\mathscr{A}$ to $\mathscr{A}_{v,ref}$ and from $\mathscr{A}_{v,ref}$ to $\mathscr{A}_v$, respectively.

Theorem 5.6. *The relations $\mathscr{R}_{rv,1}$ and $\mathscr{R}_{rv,2}$ are FSRs from $\mathscr{A}$ to $\mathscr{A}_{v,ref}$ and $\mathscr{A}_{v,ref}$ to $\mathscr{A}_v$, respectively, and $\mathscr{A} \preceq_{\mathscr{R}_{rv,1}} \mathscr{A}_{v,ref} \preceq_{\mathscr{R}_{rv,2}} \mathscr{A}_v$.*

Proof. Let us prove the first half first: that $\mathscr{R}_{rv,1}$ is a FSR from $\mathscr{A}$ to $\mathscr{A}_{v,ref}$. We do that by showing that $\mathscr{A}_{v,ref}$ is the result of following Definition 5.1 to create an abstraction of $\mathscr{A}$ using a slightly modified version Φ' of the virtual map Φ, where Φ' itself is another virtual map for $\mathscr{A}$.

By definition, $\forall p \in \mathscr{P}, rv(p) = \rho_p(p)$, where $(\gamma_p, \rho_p) \in \Phi$. Let Φ' be equal to Φ for all $p \notin rv^{-1}_{ref}(p_{v,ref})$. For any $p \in rv^{-1}_{ref}(p_{v,ref})$, let $\rho'_p(p) = p_{v,1}$, if $p \in rpars$, and $\rho'_p(p) = p_{v,2}$, otherwise. Moreover, as in line 9, define the continuous dynamics $f_{v,ref}$ to be equal to f_v, for all $p_{v,ref} \in \mathscr{P}_{v,ref} \backslash \{p_{v,1}, p_{v,2}\}$, and to be equal to $f_v(\cdot, p^*_{v,ref})$, otherwise. Then, Φ' is a virtual map of $f_{v,ref}$ since any $(\gamma'_p, \rho'_p) \in \Phi'$ satisfies equation (2.2) for f'_v, because the corresponding $(\gamma_p, \rho_p) \in \Phi$ satisfies it for f_v. The map rv_{ref} is just the result of Φ' as rv is the result of Φ.

The edges created in splitMode for $p_{v,1}$ and $p_{v,2}$ in $\mathscr{A}_{v,ref}$ are a decomposition of the edges connected to $p^*_{v,ref}$ in $\mathscr{A}_v$, including self edges. Hence, the output of splitMode $\mathscr{A}_{v,ref}$ is indeed the result of following Definition 5.1 to construct an abstraction of $\mathscr{A}$ using rv_{ref}. It follows from Theorem 5.2, that $\mathscr{A}_{v,ref}$ is an abstraction of $\mathscr{A}$ and $\mathscr{R}_{rv,1} = \mathscr{R}_{rv_{ref}}$ is a corresponding FSR.

Now we prove the second half of the theorem: that $\mathscr{R}_{rv,2}$ is a FSR from $\mathscr{A}_{v,ref}$ to $\mathscr{A}_v$. We follow similar steps of the proof of the first half in defining a new map, which we name Φ_2, and prove that it is a virtual map of $\mathscr{A}_{v,ref}$. Let $\Phi_2 = \{(\gamma_{p_{v,ref}}, \rho_{p_{v,ref}})\}_{p_{v,ref} \in \mathscr{P}_{v,ref}}$, where $\gamma_{p_{v,ref}}(x_{v,ref}) = x_{v,ref}$ is the identity map and $\rho_{p_{v,ref}}(p_{v,ref}) = p_{v,ref}$, if $p_{v,ref} \notin \{p_{v,1}, p_{v,2}\}$, and $\rho_{p_{v,ref}}(p_{v,ref}) = p^*_{v,ref}$, otherwise. Because of line 9, $(\gamma_{p_{v,ref}}, \rho_{p_{v,ref}})$ satisfy equation (2.2) with the RHS dynamic function being $f_{v,ref}$ of f. Finally, notice that $\mathscr{A}_v$ can be retrieved from $\mathscr{A}_{v,ref}$ by following Definition 5.1 using Φ_2. It follows from Theorem 5.2, that $\mathscr{R}_{rv,2}$ is a FSR from $\mathscr{A}_{v,ref}$ to $\mathscr{A}_v$. Thus, $\mathscr{A} \preceq_{\mathscr{R}_{rv,1}} \mathscr{A}_{v,ref} \preceq_{\mathscr{R}_{rv,2}} \mathscr{A}_v$. □

The following theorem shows that given a set of concrete modes *rpars* represented by an abstract mode $p^*_{v,ref}$, splitMode creates two new abstract modes one representing only those in *rpars* and the other representing the rest of the concrete modes originally represented by $p^*_{v,ref}$. It will be used later in the proof of Theorem 5.8 that shows the correctness of splitEdge.

Theorem 5.7. *The output $p_{v,1}$ of* splitMode *represents only the concrete modes in the input set rpars, i.e. $rv^{-1}_{ref}(p_{v,1}) = rpars$.*

Proof. Follows from lines 5-7 in the pseudocode. □

splitEdge The procedure starts by checking if the input is valid, i.e., the three input sets of edges represented by $e^*_{v,ref}$ are non-empty and the edge is between leaf nodes in *vv*. If the input is not valid, splitEdge returns its input without any splits in line 2. Then, for each concrete mode represented by the source mode $p^*_{v,1}$ of $E^*_{v,ref}$, splitEdge iteratively splits the destination mode $p^*_{v,2}$ of $E^*_{v,ref}$ using splitMode. The aim is to separate the concrete edges, excluding those with the same destination modes, belonging to the four edge sets $redges_E$, $redges_G$, $redges_R$, and the set of the rest of the edges in $rv^{-1}_{ref}(e^*_{v,ref})$, into different abstract edges. This happens in lines 4-13. After that, splitEdge repeats the same procedure by splitting the source mode to separate concrete edges that share the same destination mode in line 14. That means, for each concrete mode represented by the destination mode $p^*_{v,2}$ of $E^*_{v,ref}$, splitEdge will split the refined versions of the source mode $p^*_{v,1}$ of $E^*_{v,ref}$ to separate the edges of the four sets with the same destination mode into separate abstract edges.

Correctness guarantees of splitEdge In this section, we show that splitEdge indeed splits the source and destination modes of the given edge $E^*_{v,ref}$ so that

the resulting abstract edges represent purely one of the four given sets of pairs of concrete modes, while resulting in a tighter abstract automaton of $\mathscr{A}$ than the given one.

Theorem 5.8. *The outputs $\mathscr{A}'_{v,ref}$ and $O'_{v,ref}$ of* splitEdge *are tighter abstractions of $\mathscr{A}$ and O that its inputs $\mathscr{A}_{v,ref}$ and $O_{v,ref}$. Moreover, for any $j \in [N]$, the set E_j of concrete edges represented by the j^{th} of the edges in $\{e_{v,ref,j}\}_{j\in[N]}$, i.e. $E_j = rv_{ref}^{-1}(E^*_{v,ref})$, satisfies: $E_j \subseteq redges_E$, $E_j \subseteq redges_G$, $E_j \subseteq redges_R$, or $E_j \subseteq rv_{ref}^{-1}(E^*_{v,ref}) \backslash redges_E \cup redges_G \cup redges_R$.*

Proof. We start with the first part of the theorem. The only statement in splitEdge that changes the automaton and unsafe map is the call to splitMode in line 11. From Theorem 5.6, we know that the output automaton and unsafe map of splitMode are tighter abstractions of $\mathscr{A}$ and O than its inputs. Therefore, repetitive splitting of modes in $\mathscr{A}_{v,ref}$ using splitMode result in tighter abstractions and the output of splitEdge is a tighter abstraction for $\mathscr{A}$ and O than its input.

Now we prove the second part of the theorem. Fix any two refined modes $p_{v,1}$ of $p^*_{v,1}$ and $p_{v,2}$ of $p^*_{v,2}$ and any concrete mode p_1 represented by $p_{v,1}$. Then, all edges of the form $(p_1, p_2) \in redges_E$, where p_2 is represented by $p_{v,2}$, will be represented by the edge $(p_{v,1}, p_{v,ref,1})$ after the call to splitMode in line 11. At the same time, all other edges represented by the edge $(p_{v,1}, p_{v,2})$ and not in $redges_E$, will be represented by the edge $(p_{v,1}, p_{v,ref,2})$. This is a direct result of Theorems 5.6 and 5.7. The same holds if we replace $redges_E$ with $redges_G$ or $redges_R$ because of lines 12 and 13. Also, the same holds if we switched the 1 subscripts with the 2 ones because of line 14. Therefore, there are no two concrete edges, represented by the same abstract edge in $\mathscr{A}_{v,ref}$ between modes corresponding to leaf nodes in vv at the end of splitEdge, and yet belong to two different sets from $redges_E$, $redges_G$, $redges_R$, or $rv_{ref}^{-1}(e^*_{v,ref}) \backslash redges_E \cup redges_G \cup redges_R$.

□

5.5 Conclusions

We presented the first symmetry-based abstractions of hybrid automata. Our abstractions create automata with fewer number of modes and edges than the concrete ones by aggregating sets of modes into individual modes. Symmetry maps transform trajectories of a concrete mode to trajectories of its corresponding

abstract one, and vice versa. We showed a forward simulation relation that proves the soundness of our abstraction (Theorem 5.2). We showed a method to check whether an abstract counter-example is spurious. Given a spurious counter-example, we showed a method to refine the abstract automaton to eliminate it (Theorem 5.5). In the next chapter, we will discuss an implementation of a slightly simpler version of this procedure and show how that leads to significant savings in verification time. That implementation uses the same abstraction shown in this chapter, but a simpler refinement algorithm that does not necessarily eliminates a spurious counter-example in a single refinement step.

Chapter 6

SceneChecker: Boosting Scenario Verification using Symmetry Abstractions

We present SceneChecker, a tool for verifying scenarios involving vehicles executing complex plans in large cluttered workspaces. It is based on a conference paper presented in the International Conference on Computer Aided Verification (CAV) in 2021 [134][1]. SceneChecker converts the scenario verification problem to a standard hybrid system verification problem, and solves it effectively by exploiting structural properties in the plan and the vehicle dynamics. SceneChecker uses symmetry abstractions, a novel refinement algorithm, and importantly, is built to boost the performance of any existing reachability analysis tool as a plug-in subroutine. We evaluated SceneChecker on several scenarios involving ground and aerial vehicles with nonlinear dynamics and neural network controllers, employing different kinds of symmetries, using different reachability subroutines, and following plans with hundreds of waypoints in complex workspaces. Compared to two leading tools, DryVR [115] and Flow* [17], SceneChecker shows 14× average speedup in verification time, even while using those very tools for reachability subroutines[2].

6.1 Overview

As discussed throughout the thesis, and especially in the introduction, remarkable progress has been made in safety verification of hybrid and cyber-physical systems in the last decade [71, 67, 17, 64, 115, 73, 19, 72]. The methods and tools developed have been applied to check safety of aerospace, medical, and autonomous vehicle control systems [17, 64, 62, 65]. The next barrier in making these techniques

[1] This work is in collaboration with Yangge Li.

[2] SceneChecker can be found on figshare: `https://figshare.com/articles/software/CAV2021_reduce_v6_ova/14504352`, github: `https://github.com/SceneChecker-Development-Team/SceneChecker-Tool.git`, and its website: `https://publish.illinois.edu/scenechecker/`.

usable for more complex applications is to deal with what is colloquially called the *scenario verification problem*. A key part of the scenario verification problem is to check that a vehicle or an agent can execute a plan through a complex environment. A planning algorithm (e.g., probabilistic roadmaps [150] and rapidly-exploring random trees (RRTs) [151]) generates a set of possible paths avoiding obstacles, but only considering the geometry of the scenario, not the dynamics. The verification task has to ensure that the plan can indeed be safely executed by the vehicle with all the dynamic constraints and the state estimation uncertainties. Indeed, one can view a scenario as a hybrid automaton with the modes defined by the segments of the planner, but this leads to massive models. We show an example of a simple scenario with its corresponding automaton in Figure 6.1. Encoding such automata in existing tools presents some practical hurdles. More importantly, analyzing such models is challenging as the over-approximation errors and the analysis times grow rapidly with the number of transitions. At the same time, such large hybrid verification problems also have lots of repetitions and symmetries, which suggest new opportunities.

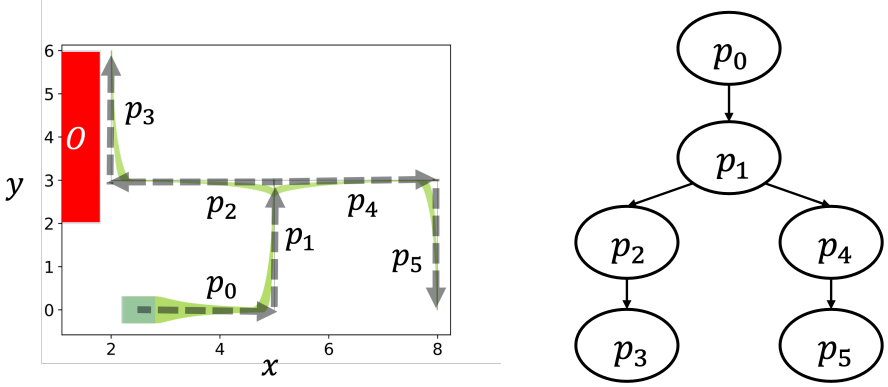


Figure 6.1: A simple scenario with a car following a plan with six segments (left). The blue square on the bottom left represents the initial set of states projected to the position coordinates. The gray arrows represent the segments of waypoints that the car can follow. The green shapes represent the reachtube of the car following the plan and projected to the position coordinates. It is computed using Flow*. The red rectangle represents the unsafe set O projected to the position coordinates. The hybrid automaton modeling the scenario has one mode per segment (right).

We present SceneChecker, a tool that implements a simplified version of the symmetry-based counter-example guided abstraction-refinement (CEGAR) algorithm of Chapter 5 for efficient scenario verification. As shown in the previous chapter, symmetry abstractions significantly reduce the number of modes and edges of an automaton $\mathscr{A}$ by grouping all modes that share symmetric continuous

dynamics. SceneChecker implements a simpler refinement algorithm for symmetry abstractions than that of Chapter 5. The refinement algorithm in SceneChecker does not necessarily eliminates a spurious counter-example in a single refinement step. However, it will do a single mode split in each call, and thus might lead to refined abstract automata with fewer modes than that of Chapter 5. Additionally, SceneChecker is able to use any existing reachability analysis tool as a subroutine. Our current implementation comes with plug-ins for using Flow* [17] and DryVR [115]. SceneChecker's verification algorithm is sound, i.e., if it returns *safe*, then the reachset of $\mathscr{A}$ indeed does not intersect the unsafe set. The algorithm is lossless in the sense that if one can prove safety without using as bstraction, then SceneChecker can also prove safety via abstraction-refinement, and typically a lot faster. However, because SceneChecker uses external reachability analysis tools as subroutines to compute per-mode reachsets, it cannot decide that a system is unsafe. That would require ability to generate counter-examples for the abstract automaton and spurious counter-example checking, which SceneChecker still lacks.

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 $\mathscr{A}$ where modes are defined by the plan's segments. It uses the symmetry maps provided by the user to construct an abstract automaton $\mathscr{A}_v$. Automaton $\mathscr{A}_v$ represents another scenario with fewer segments, each segment representing an equivalence class of symmetric segments in $\mathscr{A}$. A side effect of the abstraction is that upon reaching waypoints in $\mathscr{A}_v$, the agent's state resets nondeterministically to a set of possible states. For example, in the case of rotation and translation invariance, the abstract scenario would have a single segment for any set of segments with a unique length in the original scenario. SceneChecker refines $\mathscr{A}_v$ by splitting one of its modes to two modes. That corresponds to representing a set of symmetric segments with one more segment in the abstract scenario, capturing more accurately the original scenario. Figure 6.2 shows the architecture of SceneChecker.

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

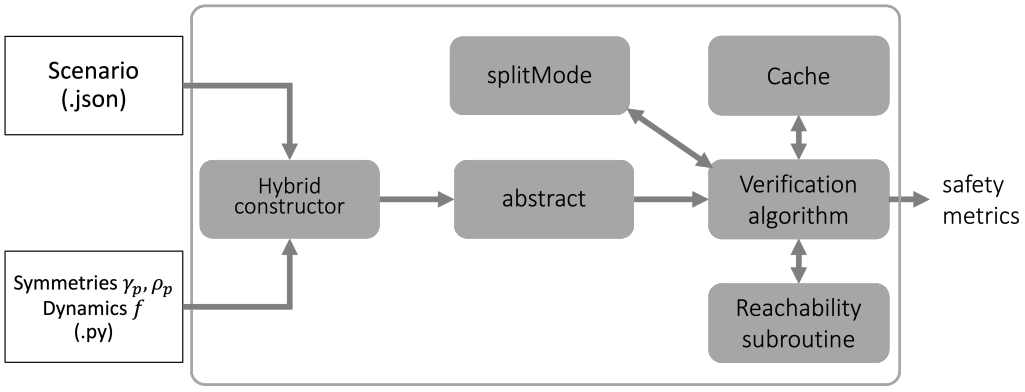


Figure 6.2: SceneChecker's architecture. Its inputs are the Scenario file and the Dynamics file. It outputs the safety results and other performance metrics. The Hybrid constructor component constructs a hybrid automaton from the given scenario. The abstract component constructs the symmetry abstraction. The Verification algorithm component implements Algorithms 6.1 and 6.2. The splitMode component refines the abstract automaton by splitting a given abstract mode. It implements Algorithm 5.2 that is presented in Chapter 5. The Cache component stores the per-mode initial sets from which reachsets have been computed to avoid repeating computations and for fixed point checking. The Reachability subroutine component is the reachability tool being used for computing reachsets called by computeReachset in the Dynamics file.

verification time of SceneChecker with DryVR and Flow* as reachability subroutines against Flow* and DryVR as standalone tools. SceneChecker is faster than both tools in all scenarios considered, achieving an average of 14× speedup in verification time (Table 6.1). 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 6.7).

In Chapter 3, we present a modified version of DryVR that utilizes symmetry to cache reachsets aiming to accelerate simulation-based safety verification of continuous dynamical systems. In Chapter 4, we present the tool CacheReach that implements a hybrid system verification algorithm that uses symmetry to accelerate reachability analysis. CacheReach caches and shares computed reachsets between different modes of non-interacting agents using symmetry.

In contrast, SceneChecker implements a simpler version of the CEGAR algorithm of hybrid automata of Chapter 5. SceneChecker does not cache reachsets and compute the reachset of the abstract automaton directly. Thus SceneChecker (1) saves concrete automaton mode graph traversal, cache-access, and reachset-

transformation computation times and (2) does not incur over-approximation errors due to caching that CacheReach suffers from [89]. 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 6.1 in Section 6.7).

6.2 Specifying scenarios in SceneChecker

A scenario verification problem is specified by a set of fixed obstacles, a plan, and an agent that is supposed to execute the plan without running into the obstacles (e.g., see Figure 6.1). For ground and air vehicles, for example, the agent moves in a subset of the 2D or the 3D Euclidean space called the *workspace*. A *plan* is a directed graph $G = \langle V, \mathscr{P} \rangle$ with vertices V in the workspace called *waypoints* and edges $\mathscr{P}$ called *segments*. We introduce this redundant nomenclature because later we will reserve the term edges to talk about mode transitions in hybrid automata. We use waypoints instead of vertices as a more natural term for points that vehicles have to follow. A general graph allows for nondeterministic and contingency planning.

An *agent* is a control system that can follow waypoints. Let the state space of the agent be X and $X_{init} \subseteq X$ be the uncertain initial set. Let p_{init} be the initial segment in G that the agent has to follow. From any state $x \in X$, the agent follows a segment $p \in \mathscr{P}$ by moving along a *trajectory*. A trajectory is a function $\xi : X \times \mathscr{P} \times \mathbb{R}^+ \to X$ that meets certain dynamical constraints of the vehicle. Dynamics are either specified by ordinary differential equations (ODE) or by a black-box simulator. Note that the trajectories only depend on the segment the agent is following (and not on the full plan G).

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

Scenario JSON file is the first of the two user inputs. It specifies the scenario: X_{init} as a hyperrectangle; $\mathscr{P}$ as a list of lists each representing two waypoints; *guard* as a list of hyperrectangles; *tbound* as a list of floats; and O as a list of polytopes.

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

6.3 Transforming scenarios to hybrid automata

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

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

Each segment $p \in \mathscr{P}$ of the plan G defines a *mode* of $\mathscr{A}$ (e.g., see Figure 6.1). The set of edges $E \subseteq \mathscr{P} \times \mathscr{P}$ of $\mathscr{A}$ is defined as pairs of consecutive segments in G. For an edge $e \in E$, $guard(e)$ is the same as that of G. The *reset* map of $\mathscr{A}$ is the identity map. As we already saw in Chapter 5, the abstract automata will have nontrivial reset maps. We will again see this in the scenario verification context in Section 6.5.

Once the given scenario is modeled as a hybrid automaton, the scenario verification problem becomes the standard bounded safety verification problem of hybrid automata described in Section 2.

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

The only difference is that, for convenience, in SceneChecker, we restrict the durations of the trajectories of any given execution to be bounded by the assigned time bounds of their corresponding modes in *tbound*. More formally, for any execution $\sigma := (\xi_0, p_0), \ldots, (\xi_k, p_k)$, for any $i \in \{0, \ldots, k\}$, $dur(\xi_i) \leq tbound(p_i)$. The reason of this restriction is that neither DryVR nor Flow* do any fixed-point checking when computing the reachsets of hybrid automata. Fixed-point checking is essential for the performance of our abstraction-refinement algorithm: the reachset of the abstract automaton will be smaller in volume and its computation is expected to reach a fixed-point faster than that of the concrete automaton. To solve this issue, we implement fixed-point checking in SceneChecker and use the reachability analysis tools as subroutines to compute per-mode reachsets instead of the full automaton reachset. To compute per-mode reachsets using these tools, a time bound needs to be specified, and that is where *tbound* would be useful. Alternatively, SceneChecker's algorithm can be modified to keep track of a global time bound, and subtract from it the duration of each per-mode reachset being computed. However, that is left as a potential future improvement.

6.4 Specifying symmetry maps in SceneChecker

The hybrid automaton representing a scenario, as constructed by the Hybrid constructor, is transformed into an abstract automaton. The abstraction is constructed by the abstract function (line 1 of Algorithm 6.1) according to Definition 5.1 described in the previous chapter. The virtual map $\Phi = \{(\gamma_p : X \to X, \rho_p : \mathscr{P} \to \mathscr{P})\}_{p \in \mathscr{P}}$ used by abstract is provided by the user. Currently, SceneChecker does not check if the maps specified by the user are indeed symmetries of the dynamics, i.e. satisfy Definition 2.2. In realistic settings, dynamics might not be exactly symmetric due to unmodeled uncertainties. In the future, we plan to account for such uncertainties as part of the reachability analysis.

In scenario verification, a given workspace would have a coordinate system according to which the plan (waypoints) and the agent's state (position, velocity, heading angle, etc.) are represented. In a 2D workspace, for any segment $p \in \mathscr{P}$, an example symmetry ρ_p would transform the two waypoints of p to a new coordinate system where the second waypoint is the origin and p is aligned with the negative side of the horizontal axis (see Figure 6.3). The corresponding γ_p would transform the agent's state to this new coordinate system (e.g., by rotating its position and

velocity vectors and shifting the heading angle). For such a pair (γ_p, ρ_p) to satisfy Definition 2.2), the agent's dynamics have to be invariant to such a coordinate transformation and Definition 2.2 merely formalizes this requirement. Such an invariance property is expected from vehicles' dynamics–rotating or translating the lane should not change how an autonomous car behaves.

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

polyVir(X', p): returns $\gamma_p(X')$ for any polytope $X' \subset X$ and segment $p \in \mathscr{P}$.

modeVir(p): returns $\rho_p(p)$ for any given segment $p \in \mathscr{P}$.

virPoly(X', p): returns $\gamma_p^{-1}(X')$, implementing the inverse of polyVir.

computeReachset$(initset, p, T)$: returns a list of hyperrectangles over-approximating the agent's reachset starting from *initset* following segment p for T time units, for any set of states $initset \subset X$, segment $p \in \mathscr{P}$, and $T \geq 0$.

6.5 Symmetry abstraction of the scenario's automaton

In this section, we describe how the abstract function in Algorithm 6.1 uses the functions in the Dynamics file to construct an abstraction of the scenario's hybrid automaton provided by the Hybrid constructor. Given the symmetry maps of Φ, the symmetry abstraction of $\mathscr{A}$ is another hybrid automaton $\mathscr{A}_v$ that aggregates many symmetric modes (segments) of $\mathscr{A}$ into a single mode of $\mathscr{A}_v$.

Modes and Transitions Any segment $p \in \mathscr{P}$ of $\mathscr{A}$ is mapped to the segment $\rho_p(p)$ in $\mathscr{A}_v$ using modeVir. The set of modes $\mathscr{P}_v$ of $\mathscr{A}_v$ is the set of segments $\{\rho_p(p)\}_{p\in\mathscr{P}}$. For any p_v, $tbound_v(p_v) = \max_{p\in\mathscr{P}, p_v=\rho_p(p)} tbound(p)$. In the example of Section 6.4 (Figure 6.3), the segments in $\mathscr{A}_v$ are aligned with the horizontal axis and ending at the origin. The number of segments in $\mathscr{A}_v$ would be the number of segments in G with unique lengths. The agent would always be moving towards the origin of the workspace in the abstract scenario. Any edge $e = (p, p') \in E$ of $\mathscr{A}$ is mapped to the edge $e_v = (\rho_p(p), \rho_{p'}(p'))$ in $\mathscr{A}_v$. The $guard(e)$ is mapped to $\gamma_p(guard(e))$ using polyVir which becomes part of $guard_v(e_v)$ in $\mathscr{A}_v$. For any $x \in X$, $reset(x, e)$, which is equal to x, is mapped to $\gamma_{p'}(\gamma_p^{-1}(x))$ and becomes part

of $reset_v(x, e_v)$ in $\mathscr{A}_v$. In our example in Section 6.4, the $\gamma_p^{-1}(x)$ would represent x in the absolute coordinate system assuming it was represented in the coordinate system defined by segment p. The $\gamma_{p'}(\gamma_p^{-1}(x))$ would represent $\gamma_p^{-1}(x)$ in the new coordinate system defined by segment p'. The $guard_v(e_v)$ would be the union of rotated hyperrectangles centered at the origin that result from translating and rotating the guards of the edges represented by e_v. The initial set X_{init} of $\mathscr{A}$ is mapped to $X_{init,v} = \gamma_{p_{init}}(X_{init})$, the initial set of $\mathscr{A}_v$.

The unsafe map O is mapped to O_v, where $\forall p_v \in \mathscr{P}_v$, $O_v(p_v) := \cup\{\gamma_p(O(p)) \mid p \in \mathscr{P}, \rho_p(p) = p_v\}$. That means the obstacles near any segment $p \in \mathscr{P}$ in the environment will be mapped to be near its representative segment $\rho_p(p)$ in $\mathscr{A}_v$.

From the forward simulation relation in the previous chapter, we know that if $\mathscr{A}_v$ is safe with respect to O_v, then $\mathscr{A}$ is safe with respect to O. More formally, if $\forall p_v \in \mathscr{P}_v, Reach_{\mathscr{A}_v}(p_v) \cap O_v(p_v) = \emptyset$, then $\forall p \in \mathscr{P}, Reach_{\mathscr{A}}(p) \cap O(p) = \emptyset$.

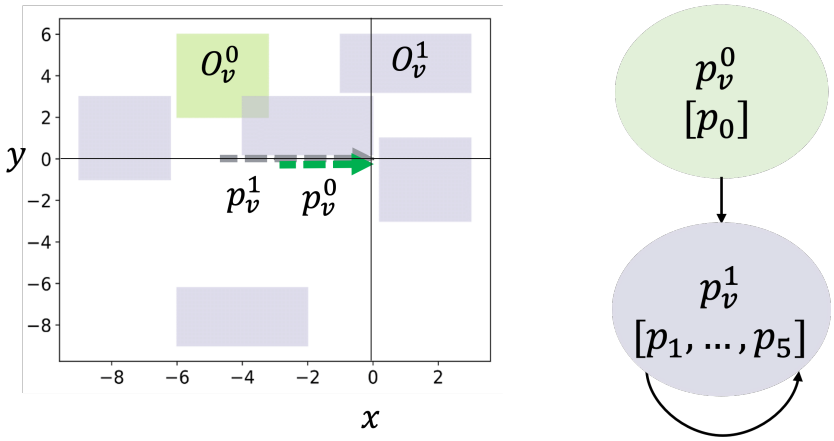


Figure 6.3: Translation and rotation symmetries are used to abstract the scenario in Figure 6.1 resulting in the scenario shown on the left of this figure. The abstraction translates and rotates each segment of the original scenario to a segment aligned with the x-axis and ends at the origin. Since all but the first segment in the original scenario have the same length, they are represented by the same segment p_v^1 in the abstract scenario. The first segment is represented by a separate segment p_v^0. Given the transformation that mapped each segment in the original scenario of Figure 6.1 to its representative segment in the abstract scenario, the same transformation is applied to O (and then a bounding box is computed) to obtain the representative unsafe set. Since $p_{v,1}$ represents five segments, it has an unsafe set (colored in grayish violet) consisting of five rectangles. Each of these rectangles is the relative position of the unsafe set to one of the segments represented by p_v^1. Similarly, p_v^0 has an unsafe set with a single rectangle representing the relative position of O to p_0. The corresponding automaton for the abstract scenario is shown on the right.

6.6 Overview of SceneChecker algorithm

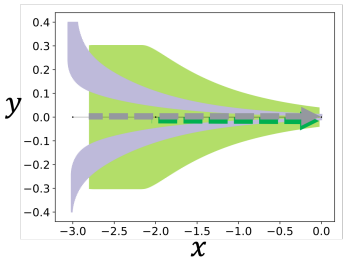
A sketch of the core abstraction-refinement algorithm implemented in SceneChecker is shown in Algorithm 6.1. It constructs a symmetry abstraction $\mathscr{A}_v$ of the concrete automaton $\mathscr{A}$ resulting from the Hybrid constructor. SceneChecker attempts to verify the safety of $\mathscr{A}_v$ using traditional reachability analysis. SceneChecker uses a *Cache* to store per-mode initial sets from which reachsets have been computed and thus avoids repeating computations. An example run is shown in Figures 6.1, 6.3, and 6.4.

Algorithm 6.1 SceneChecker$(\Phi = \{(\gamma_p, \rho_p)\}_{p \in \mathscr{P}}, \mathscr{A}, O)$

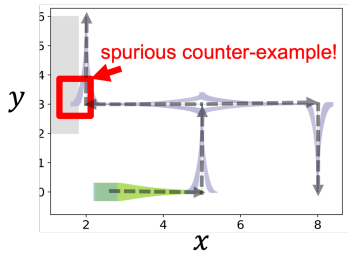
1: $\mathscr{A}_v, O_v \leftarrow \mathsf{abstract}(\mathscr{A}, O, \Phi)$
2: $\forall p \in \mathscr{P}, rv(p) \leftarrow \rho_p(p)$
3: **while** *True* **do**
4: $\quad Cache \leftarrow \{p_v \mapsto \emptyset \mid p_v \in \mathscr{P}_v\}$
5: $\quad result, p_v^* \leftarrow \mathsf{verify}(rv(p_{init}), X_v, Cache, rv, \mathscr{A}_v, O_v)$
6: $\quad$ **if** *result* $=$ *safe* or *unknown* **then return:** *result*
7: $\quad$ **else** $rv, \mathscr{A}_v, O_v \leftarrow \mathsf{splitMode}(p_v^*, rv, \mathscr{A}_v, O_v, \mathscr{A}, O)$

The core algorithm verify shown in Algorithm 6.2 is called iteratively. If verify returns $(safe, \perp)$ or $(unknown, \perp)$, SceneChecker returns the same result. If verify instead results in (refine, p_v^*), splitMode is called to refine $\mathscr{A}_v$ by splitting p_v^* into two modes p_v^1 and p_v^2. Check Algorithm 5.2 in the previous chapter for formal definition of splitMode. Each of the two modes would represent part of the set of the segments of $\mathscr{P}$ that were originally mapped to p_v in rv. Then the edges, guards, resets, and the unsafe sets related to p_v are split according to their definitions.

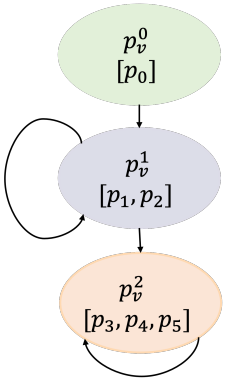
The function verify executes a *depth-first search* (DFS) over the mode graph of $\mathscr{A}_v$. For any mode p_v being visited, computeReachset computes R_v, an over-approximation of the agent's reachset starting from *initset* following segment p_v for time $tbound_v(p_v)$. If $R_v \cap O_v(p_v) = \emptyset$, verify recursively calls p_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 $Cache[p_v]$ (line 12) and verify returns *safe*. Most importantly, if verify returns (refine, p_v^*) for any of p_v's children, it directly returns (refine, p_v^*) for p_v as well (line 7). If any child returns *unknown* or R_v intersects $O_v(p_v)$, verify will need to split p_v. In that case, it checks if $rv^{-1}(p_v)$ is not a singleton set and thus amenable to splitting (line 10). If p_v can be split, verify returns (refine, p_v). Otherwise, verify



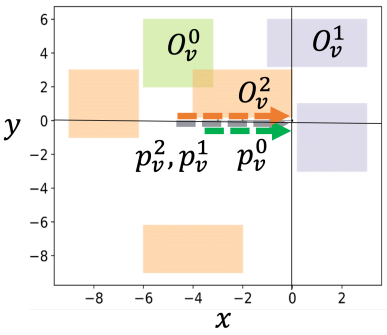
(a) Reachset of the abstract scenario computed using SceneChecker. Per-mode reachsets are computed using DryVR.



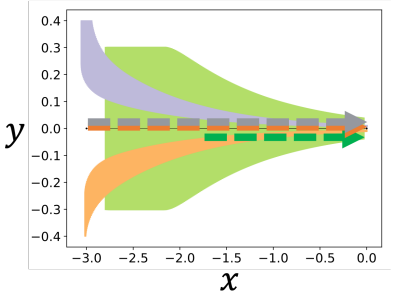
(b) Reachset of the abstract scenario transformed to the original scenario to illustrate the intersection with unsafe set.



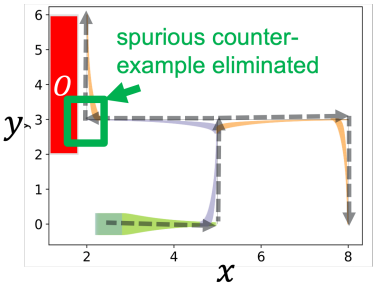
(c) SceneChecker refines the abstract automaton by splitting the abstract mode, that has its reachset intersecting its unsafe set, into two abstract modes. Each new mode represents some of the concrete modes that were represented by the split mode. Then, p_v^1 is split into p_v^1 and p_v^2.



(d) The refined automaton represents a scenario with three segments, two of which are overlapping. However, they have different unsafe sets (orange and grayish violet), guards, and resets. It captures more accurately the original scenario.



(e) Reachset of the refined automaton computed using SceneChecker.



(f) Reachset of the refined automaton transformed, for visualization purposes, to the concrete scenario. After refinement, the reachset is not anymore intersecting the unsafe set.

Figure 6.4: SceneChecker computes the reachset of the car in the scenario of Figure 6.1 through abstraction and refinement.

returns $(unknown, \perp)$ implicitly asking one of p_v's ancestors to be split instead.

Correctness SceneChecker ensures that all the refined automata $\mathscr{A}_v$'s are abstractions of the original hybrid automaton $\mathscr{A}$. This is a direct result of the proof that splitMode results in refined automata shown in the previous chapter (Theorem 5.6). For any mode with a reachset intersecting the unsafe set, SceneChecker keeps splitting that mode and its ancestors until safety can be proven or $\mathscr{A}_v$ becomes $\mathscr{A}$.

Theorem 6.1 (Soundness)**.** *If* SceneChecker *returns safe, then* $\mathscr{A}$ *is safe.*

Algorithm 6.2 verify$(p_v, initset, Cache, rv, \mathscr{A}_v, O_v)$

```
1:  R_v ← computeReachset(initset, p_v)
2:  if R_v ∩ O_v(p_v) = ∅ then
3:      for p'_v ∈ children(p_v) do
4:          initset' ← reset_v(guard_v((p_v, p'_v)) ∩ R_v) \ Cache[p'_v]
5:          if initset' ≠ ∅ then
6:              result, p*_v ← verify(p'_v, initset', Cache, rv, A_v, O_v)
7:              if result = refine then return: refine, p*_v
8:              else if result = unknown then break
9:  if R_v ∩ O_v(p_v) ≠ ∅ or result is unknown then
10:     if |rv^{-1}(p_v)| > 1 then return: refine, p_v
11:     else return: unknown, ⊥
12: Cache[p_v] ← Cache[p_v] ∪ initset
13: return: safe, ⊥
```

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

Counter-examples SceneChecker currently does not find counter-examples to show that the scenario is *unsafe*. There are several sources of over-approximation errors, namely, computeReachset and guard intersections. Even after all the over-approximation errors from symmetry abstractions are eliminated, as refinement does, it still cannot infer unsafe executions or counter-examples because of the other errors. This can be addressed through a hybrid systems verification algorithm

with systematic simulation such as DryVR and C2E2 that can generate counter-examples. Additionally, the verification algorithm should check, similar to ours, to achieve the savings offered by our algorithm. Moreover, the algorithms validateCE and refine to check for spurious counter-examples and eliminating them through refinement from the previous chapter should be implemented to get concrete counter-examples and to prove non-safety.

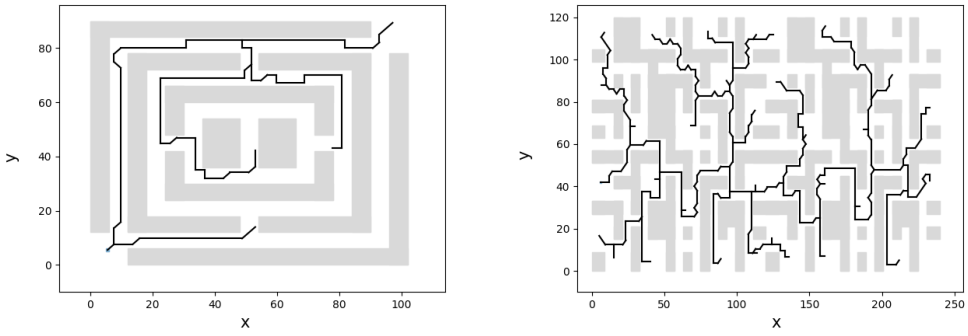
6.7 Experimental evaluation

Machine specifications In our experiments, we used a desktop with the AMD Ryzen 7 5800X CPU @ 3.8 GHz x 8 processor and with a 32 GB memory.

Agents and controllers In our experiments, we consider two types of nonlinear agent models: a standard 3-dimensional car (C) with bicycle dynamics and 2 inputs, and a 6-dimensional quadrotor (Q) with 3 inputs. For each of these agents, we developed a PD controller and a NN controller for tracking segments. The NN controller for the quadrotor is from Verisig's paper [72] but modified to be rotation symmetric. Similarly, the NN controller for the car is also rotation symmetric. Check Appendices A.2 and A.4 for more details on the NN controllers of the quadrotor and the car. Both NN controllers are translation symmetric as they take as input the difference between the agent's state and the segment being followed. The PD controllers are translation and rotation symmetric by design.

Symmetries We experimented with two different collections of symmetry maps Φs: 1) translation symmetry (T), where for any segment p in G, γ_p maps the states so that the coordinate system is translated by a vector that makes its origin at the end waypoint of p, and 2) rotation and translation symmetry (TR), where instead of just translating the origin, Φ rotates the xy-plane so that p is aligned with the x-axis, which we described in Section 6.4. For each agent and one of its controllers, we manually verified that the equivariance condition in Definition 2.3 is satisfied for each of the two Φs.

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 [152] after specifying a number of goal sets that should



(a) S2 **(b)** S3

Figure 6.5: The figures shows the scenarios S2 and S3 and their corresponding plans that we experiment SceneChecker with. S5 is the same as S2 but with waypoints with varying altitude. Grey boxes and polytopes represent unsafe sets, such as building, white represents open space, and black segments represent predefined plans, generated using RRT, for the agents to follow.

be reached. We modified S4 to have more obstacles but still have the same plan and named the new version S4.b and the original one S4.a. When the quadrotor was considered, the waypoints of the 2D scenarios (S1-4) were converted to 3D representation by setting the altitude for each waypoint to 0. Scenario S5 is the same as S2 but S5's waypoints have varying altitudes. The scenarios have different complexities ranging from few segments and obstacles to hundreds of them. All scenarios are safe when traversed by any of the two agents. Scenarios S2-4 are shown in Figures 6.5 and 6.6. Scenario S1 is shown in Figure 6.1.

We verify these scenarios using SceneChecker and CacheReach, each with two instances, one with DryVR and the other with Flow*, implementing the subroutine computeReachset. We also use DryVR and Flow* as independent tools to verify the same scenarios. The results of experiments with tools that involve DryVR (i.e., SceneChecker+DryVR, CacheReach+DryVR, and DryVR) are stochastic and change between runs. The reason is that each time DryVR is called, it randomly samples traces of the system from which it computes the requested reachset. We fix the random seed for repeatable results in this section. We show close averaging-based results on SceneChecker's website.

SceneChecker is able to verify all scenarios with PD controllers. The results are shown in Table 6.1. Figure 6.7 presents the reachsets of the concrete and abstract automata for different scenarios. The C-S1 scenario verified using SceneChecker+Flow* is shown in Figure 6.4.

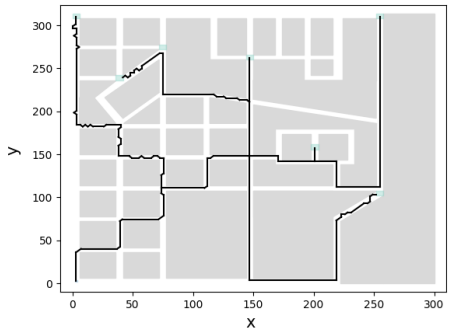


Figure 6.6: The figure shows the scenario S4 and its corresponding plan that we experiment SceneChecker with. Grey boxes and polytopes represent unsafe sets, such as building, white represents open space, and black segments represent predefined plans, generated using RRT, for the agents to follow.

Observation 1: SceneChecker **offers fast scenario verification and boosts existing reachability tools** Looking at the two total time (Tt) columns for the two instances of SceneChecker with the corresponding columns for Flow* and DryVR, it becomes clear that symmetry abstractions can boost the verification performance of reachability engines. For example, in C-S4.a, SceneChecker+DR was around 20× faster than DryVR. In C-S3, SceneChecker with Flow* was around 16× faster than Flow*. In scenario Q-S5, SceneChecker timed out at least in part because a computeReachset call to Flow* timed out. Even when many refinements are required and thus causing several repetitions of the verification process in Algorithm 6.1, SceneChecker is still faster than DryVR and Flow* (C-S4.b). SceneChecker is 14× faster than DryVR and Flow* on average, while using these tools themselves as subroutines. All three tools resulted in *safe* for all scenarios when completed executions.

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

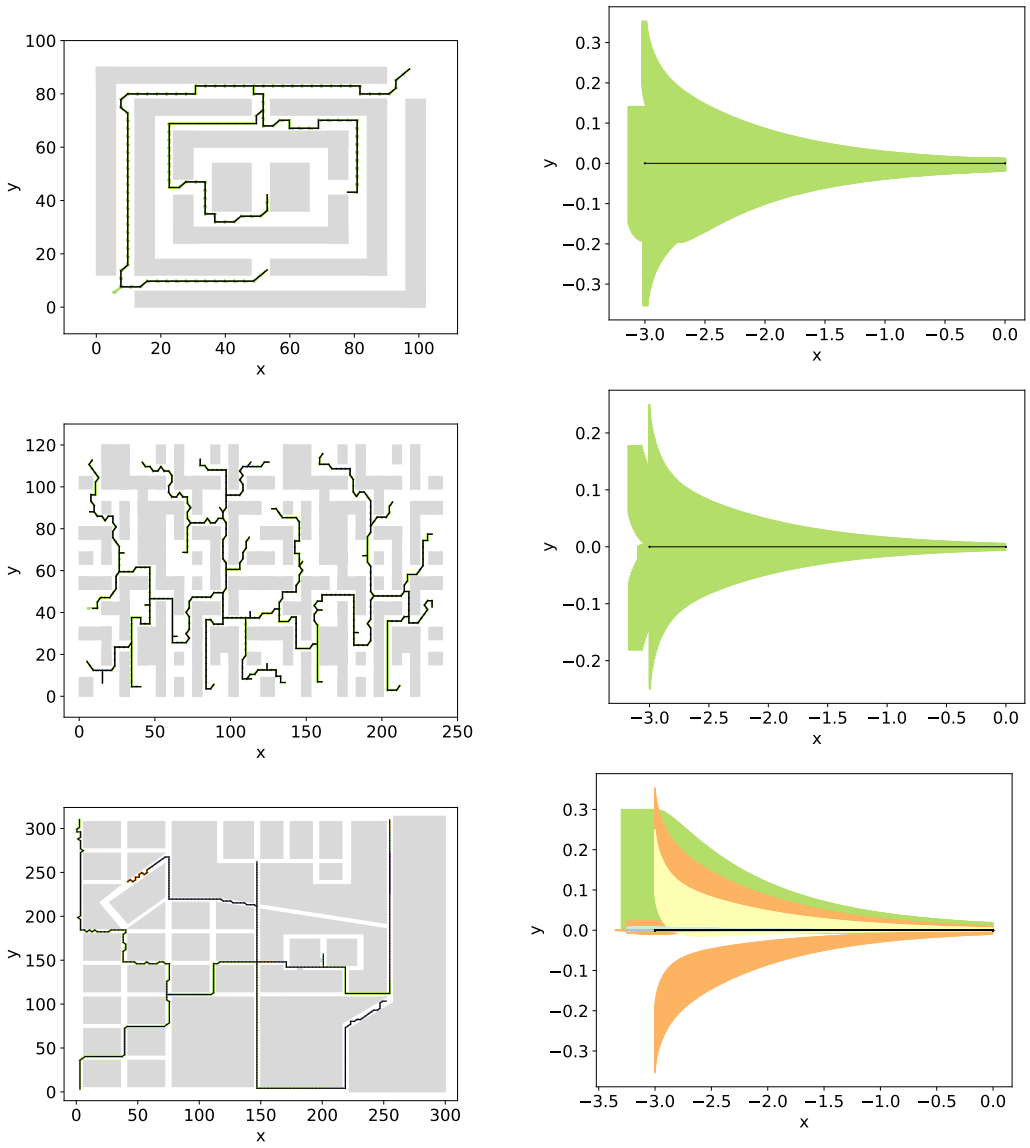


Figure 6.7: Reachsets of the car with the PD-controller in scenarios S2 (first row), S3 (second row), and S4.b (third row). The left column represents the concrete scenario with the car reachsets. The right column represents the reachsets of the abstract automaton. The different colors correspond to different abstract segments (defined in Section 6.5). All abstract segments have the same waypoints but represent different segments of the concrete scenario. That means they lead to different reset of the agent's state after mode transitions.

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. Check Appendix A.3 for a discussion about our attempts for using other verification tools for NN-controlled systems as reachability subroutines. We tried different Φs. The results are shown in Table 6.2. When not using abstraction-refinement, SceneChecker took 10.5, 130.95, and 74.15 minutes for the QNN-S2, QNN-S3, and QNN-S4 scenarios, while DryVR took 5.22, 52.56, and 61.31 minutes for the same scenarios, respectively. Comparing these results with those in Table 6.2 shows that the speedup in verification time of SceneChecker is caused by the abstraction-refinement algorithm, achieving more than $13\times$ in certain scenarios (QNN-S4 using $\Phi = \text{T}$). SceneChecker+DR was more than $10\times$ faster than DryVR in the same scenario.

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 $\mathscr{A}_v$ but larger over-approximation error causing more mode splits. On the other hand, using T leads to a larger $\mathscr{A}_v$ but less over-approximation error and thus fewer refinements. This trade-off can be seen in Table 6.2. For example, QNN-S4 with $\Phi =$T resulted in zero mode splits leading to $|\mathscr{P}_v|^i = |\mathscr{P}_v|^f = 7$, while $\Phi =$ TR resulted in 4 mode splits, starting with $|\mathscr{P}_v|^i = 1$ modes and ending with $|\mathscr{P}_v|^f = 5$, and longer verification time because of refinements. On the other hand, in QNN-S3, $\Phi =$ TR resulted in Nref$= 5$, $|\mathscr{P}_v|^f = 6$, and Tt$= 12.7$ min while $\Phi =$T resulted in Nref$= 4$, $|\mathscr{P}_v|^f = 11$, and Tt$= 16.88$ min.

Observation 5: Complicated dynamics require more verification time Different vehicle dynamics affect the number of refinements performed and consequently the verification time (e.g. QNN-S2, QNN-S4, CNN-S2, and CNN-S4). The car appears to be less stable than the quadrotor leading to longer verification time for the same scenarios. This can also be seen by comparing the results of Tables 6.1 and 6.2. The PD controllers lead to more stable dynamics than the

Table 6.1: Comparison between SceneChecker, DryVR (DR), Flow* (F*), and CacheReach (CacheR). Both SceneChecker and CacheReach use reachability tools as subroutines. The subroutines used are specified after the '+' sign. Φ is TR. The table shows the number of mode-splits performed (Nrefs), the total number of calls to computeReachset (Rc), the total time spent in reachset computations (Rt), and the total computation time in minutes (Tt). In scenarios where a tool ran over 120 minutes, we marked the Tt column as 'Timed out' (TO) and the other ones as 'Not Available' (NA).

		SceneChecker+DR				CacheR+DR		DR	SceneChecker+F*				CacheR+F*		F*
Sc.	$\lvert\mathscr{P}\rvert$	Nrefs	Rc	Rt	Tt	Rc	Tt	Tt	NRefs	Rc	Rt	Tt	Rc	Tt	Tt
C-S1	6	1	4	0.14	0.15	46	1.73	1.28	1	4	0.51	0.52	52	8.20	2.11
C-S2	140	0	1	0.04	0.65	424	19.92	10.57	0	1	0.18	0.79	192	30.95	17.52
C-S3	458	0	1	0.04	4.24	502	19.33	71.41	0	1	0.11	4.34	176	28.64	73.06
C-S4.a	520	2	7	0.26	4.37	404	15.84	94.62	2	7	0.80	4.96	160	25.98	61.53
C-S4.b	520	10	39	1.43	8.69	404	16.06	96.02	10	39	2.83	31.73	160	26.07	60.67
Q-S1	6	1	4	0.04	0.05	NA	TO	0.25	1	4	13.85	14.13	NA	TO	30.17
Q-S2	140	0	1	0.04	0.88	NA	TO	4.97	0	1	3.38	12.62	NA	TO	TO
Q-S3	458	0	1	0.06	5.9	NA	TO	46.34	0	1	4.98	62.66	NA	TO	TO
Q-S4.a	520	0	1	0.06	3.17	NA	TO	56.19	0	1	4.8	34.89	NA	TO	TO
Q-S5	188	0	36	0.85	3.04	NA	TO	8.03	NA	NA	NA	TO	NA	TO	TO

NN controllers requiring less total computation time for both agents. More stable dynamics lead to tighter reachsets and fewer refinements.

Table 6.2: Comparison between Φs. In addition to the statisitics of Table 6.1, this table reports the number of modes and edges in the initial and final (after refinement) abstractions ($|\mathscr{P}_v|^i$, $|E_v|^i$; $|\mathscr{P}_v|^f$, and $|E_v|^f$, respectively)

Sc.	NRef	Φ	$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{P}_v\rvert^i$	$\lvert E_v\rvert^i$	$\lvert\mathscr{P}_v\rvert^f$	$\lvert E_v\rvert^f$	Rc	Rt	Tt
CNN-S2	6	TR	140	1	1	7	17	19	1.51	3.05
CNN-S4	9	TR	520	1	1	10	28	47	3.77	11.25
QNN-S2	3	TR	140	1	1	4	9	9	0.61	3.55
QNN-S3	5	TR	458	1	1	6	16	15	1.51	12.7
QNN-S4	4	TR	520	1	1	5	13	11	1.11	7.43
QNN-S2	0	T	140	7	19	7	19	8	0.53	1.38
QNN-S3	4	T	458	7	30	11	58	29	2.92	16.88
QNN-S4	0	T	520	7	30	7	30	13	1.32	5.34

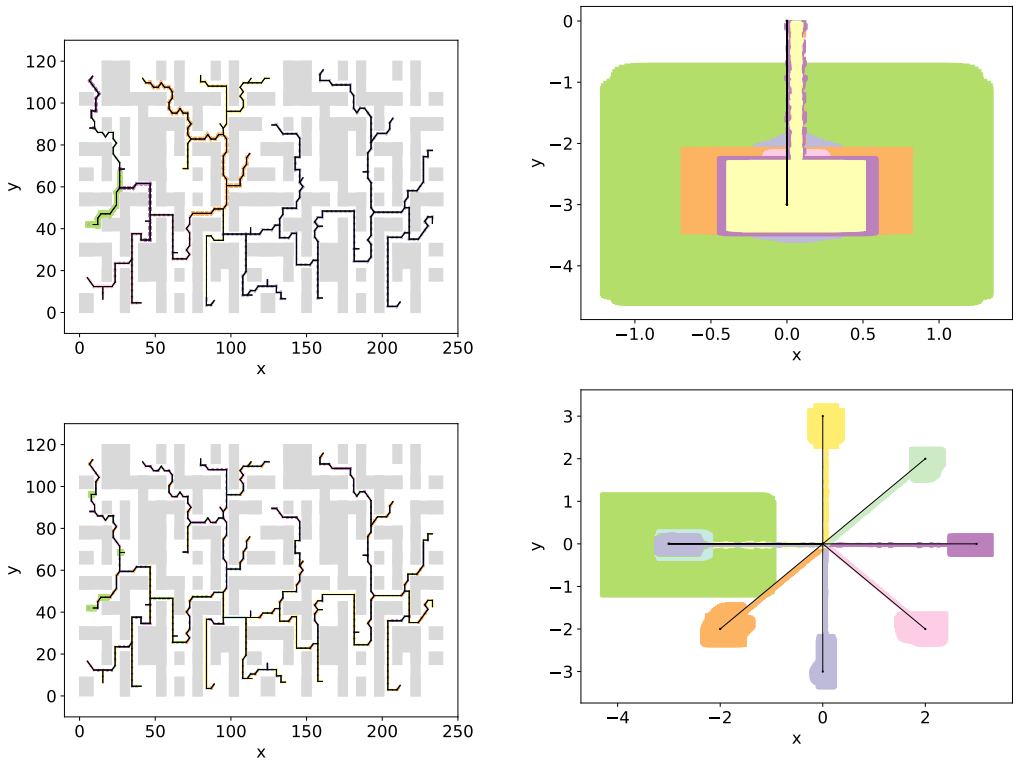


Figure 6.8: Reachsets of the quadrotor with the NN-controller while using different Φs. The first row is the quadrotor's reachset when using $\Phi = \text{TR}$. The second row is when using $\Phi = \text{T}$. The left column represents the concrete scenario with the computed reachsets. The right column represents the reachsets of the abstract automaton.

6.8 Limitations and discussions

SceneChecker allows the choice of modes to be changed from segments to waypoints or sequences of segments as well. The waypoint-defined modes eliminate the need for segments of G to have few unique lengths, but in turn, would only allow using translation symmetry, i.e. $\Phi = \mathrm{T}$. SceneChecker splits only one mode per refinement and then repeats the computation from scratch. It has to refine many times in unsafe scenarios until reaching the result *unknown*. We plan to implement the CEGAR algorithm in Chapter 5 to be able to generate concrete counter-examples without the need for refining $\mathscr{A}_v$ back to $\mathscr{A}$. In the future, it will be important to address other sources of uncertainty in scene verification such as moving obstacles, interactive agents, and other types of symmetries such as permutation and time scaling. Finally, it will be useful to connect a translator to generate scene files from common road simulation frameworks such as CARLA [153], commonroad [154], and Scenic [155].

Chapter 7

Efficient Testing of Autonomous Systems by Exploiting Symmetries: A Case Study on Airborne Collision Avoidance Systems

In this chapter, we switch gears from formal approaches to non-formal ones for assessing the safety of autonomous systems. Testing has been the default approach in industry for checking software, hardware, and cyber-physical systems. It is easier to implement and more efficient to perform than formal verification techniques, at the cost of non-formal guarantees. High-fidelity simulators have been emerging in the past few years for reliable end-to-end testing of autonomous systems. Unfortunately, such simulators are computationally expensive. We address the scalability problem of high-fidelity testing of autonomous systems using symmetry[1]. Instead of transforming reachtubes and reachsets, in this chapter, we go back to the definition of symmetry (Definition 2.2) and use it to transform trajectories generated by high-fidelity simulators to new ones needed for other agents or other test cases. The key idea here is that by dividing the trajectories of the autonomous systems into segments, these segments can be cached and reused in new test cases. We propose an algorithm that implements this idea and uses symmetries for efficient testing. We present two versions of the algorithm: one that simulates the tests sequentially and caches the results, and the other simulates all the tests in parallel using a single agent. We present a setup for closed-loop testing of Airborne Collision Avoidance Systems (ACASs), on which we evaluate the efficiency of the two versions of the proposed algorithm. The ACASs are systems that provide vertical and horizontal advisories for aircraft for avoiding collisions. Our preliminary evaluation results demonstrate a significant reduction in simulation time, while preserving the quality of the generated trajectories.

[1]This work is in collaboration with Yangge Li and Reyhan Jabbarvand Behrouz.

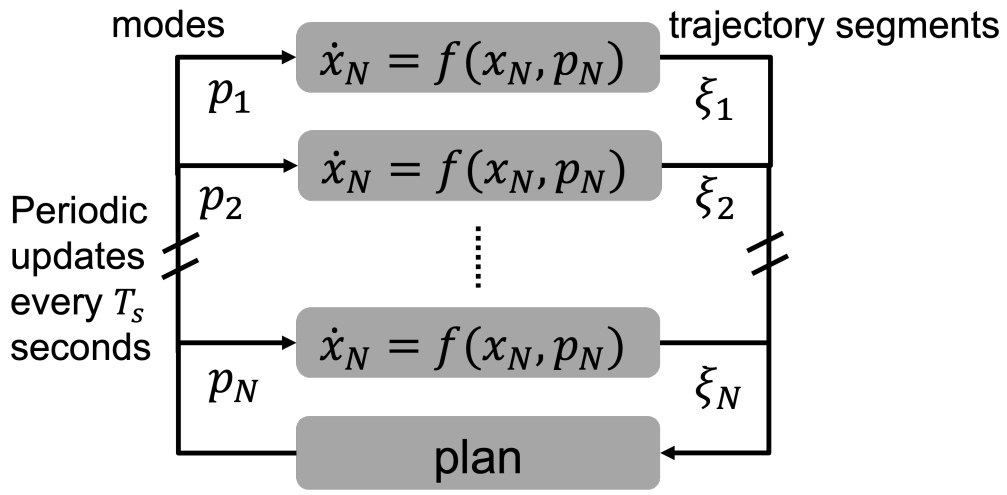


Figure 7.1: Multi-agent systems that symTest and absSymTest are built to test. Each agent has continuous (or discrete) dynamics. The knowledge of the model is not necessary by the testing algorithm, but only its symmetries. Without loss of generality, we assume that all agents share the same dynamics. The agents coordinate through a planning module plan periodically, with period T_s. The module plan generates the modes or parameters $(p_1, \ldots, p_N)$, that the agents will follow in the next period. The agents then send their continuous (or discrete) trajectory segments to plan. An analogous diagram that is specific for our setup for closed-loop testing of ACAS is shown in Figure 7.2.

7.1 Overview

As discussed in the introduction of the thesis in Section 1.2.4, there is an urgent need for scalable high-fidelity closed-loop testing framework of autonomous systems. In this chapter, we use symmetry to tackle the scalability problem in high-fidelity simulations of multi-agent autonomous systems. Specifically, we present an efficient testing algorithm that uses symmetry to avoid re-simulating trajectory segments. The multi-agent systems we consider are of the form shown in Figure 7.1.

Recall from Definition 2.2 that a symmetry of a dynamical system with state space X is a function that acts on X. When given a sequence of states representing one of the system's trajectories, a symmetry applied to each of the states in the sequence results in a new trajectory of the system. For example, aircraft dynamics are translation symmetric: translating the 3D position part of the states of a trajectory ξ of an aircraft by a 3D vector results in another trajectory ξ' of the aircraft starting from the translated initial state. High-fidelity simulations are computation-

ally expensive. Symmetries allow efficient generation of several trajectories by transforming a single trajectory generated from a high-fidelity simulator. Testing multi-agent systems requires many simulations. Our testing algorithm uses symmetry to share computed trajectories between different tests and agents, avoiding expensive simulations, whenever possible.

The first version of our testing algorithm we present in this chapter is symTest. Given a predefined test suite *tests*, symTest runs the tests of *tests* sequentially. It maintains a cache that stores trajectory segments of the agents as they get simulated while executing the different tests. Recall from Section 1.2.4 that we denote the part of the trajectory of an agent over a period between two mode updates by plan by *trajectory segment*. symTest maps each class of symmetric trajectory segments to a unique entry in the cache. For each test, symTest generates the trajectories of the different agents in T_s seconds time steps—the rate at which plan updates the modes. At each time step, for each agent in the encounter, symTest checks if the entry in the cache corresponding to its current state and plan is non-empty. If it is indeed non-empty, symTest transforms the stored trajectory segment using a symmetry transformation to obtain the needed trajectory. Otherwise, if the cache misses any of the queries of any of the agents at a certain period, symTest calls the simulator to generate the rest of the simulation. Hence, symTest only calls the simulator when a new behavior of an agent has not been previously simulated.

We provide another improved version of symTest, which we call absSymTest. Using a carefully chosen simulation of a single *representative* agent, absSymTest generates the trajectories of all agents in all test cases in *tests*. The simulation of the representative agent might not correspond to any of the agents in any of the tests in *tests*. At each period, absSymTest simulates the representative agent to generate a trajectory segment symmetric to one of the agents in one of the test cases, or transits to a state where that is possible. In addition to the cache of trajectories used by symTest, absSymTest keeps track of a set *readyset* of pairs of states and plans to be simulated. At each period, absSymTest checks if the current state of the agent x is symmetric to any of the states x' in *readyset*. The entry in the cache corresponding to the state x', along with its paired plan p', should be empty. If such a pair exists in *readyset*, absSymTest simulates the agent for the next time step as it follows a plan symmetric to p'. The resulting trajectory is saved in the cache after being transformed using symmetry to its representative format. If there is no such pair in *readyset*, absSymTest directs the agent to move to a state symmetric to a state in *readyset* either using a controller or by a direct resetting of its state

using the simulator.

Both versions of our algorithm symTest and absSymTest are useful for running regression tests, which at small cost as loss of fidelity, gains significant speedup in testing time. If the coordination software, such as ACAS, is the only component under test while the dynamics of the agents are kept untouched, the cache of trajectories that the algorithm builds during the run of a test suite *tests*, can be used in future regression runs of *tests*. The cache would represent a simpler model of the dynamics of the agents. This simpler model can easily and efficiently construct trajectories of the agents in different encounters while, to high-extent, preserving their high-fidelity aspect.

We implemented symTest and absSymTest in Python. We compare the algorithm's performance when using different symmetries including 3D position translation, 3D position translation and 2D rotation, and 3D position and velocity translation and 3D rotation. We built a testing setup for closed-loop ACAS system using the Robot Operating System (ROS) [99] and Gazebo [2]. We use the most recent version of ACAS sXu (described in the next section) specifications implemented in Julia executables as our planning module plan. We used Hector Quadrotors [98] as our agents. We also built another testing setup that connects Python-based six-dimensional NN-controlled quadrotor [20] with NN-version of ACAS Xu [156]. We perform *grid-based* and *metamorphic* testing of both systems. Grid-based testing generates test suites by griding the input space of the system under test, and creating a test for each cell in the grid. Metamorphic testing creates test suites by creating tests with related outputs, possibly using symmetric relations [157, 47, 51, 158]. Our experimental results using symTest to test ACAS sXu advising Hector Quadrotors and using absSymTest to test ACAS Xu advising NN-controlled quadrotors show up to 27% and 65% savings in simulation steps, respectively.

Relationship to previous chapters The problem we address and the solution we suggest in this chapter draw many parallels to the previous chapters. In this chapter, we consider the non-formal testing problem rather than the formal verification problem, in contrast with all previous chapters of this thesis. In this chapter, as in Chapter 4, we consider agents with dynamics that are parameterized with modes. Unlike Chapter 4, instead of assuming that the agents do not interact during operation, in this chapter, we consider that the agents coordinate with each other periodically through a planning module called plan. Given the current states, or

trajectories, of the different agents and their predefined plans (e.g. waypoints) they have to follow, plan decides the modes, or equivalently, parameters, they should follow for the next period or time step. The plan specifies these modes for the agents to avoid collision or to reach their destinations, for example. In this chapter, instead of assuming that there is a reachability analysis subroutine that computes the reachsets or reachtubes for the different agents, we assume that there exists a simulator, a high-fidelity and computationally expensive one, that can simulate one or more agents following different plans while coordinating through plan. Moreover, instead of verifying a single scenario, in the experiments and the problem formulation of this chapter, we consider a test suite defining several encounters that has to be simulated. Finally, in contrast with Chapters 4-6, instead of assuming that the maps (γ_p, ρ_p) in the virtual map Φ are parameterized by the mode, in this chapter, we assume that the pairs are parameterized by the state instead, i.e., of the form (γ_x, ρ_x). That said, the results of the previous chapters can be generalized to verify scenarios similar to those we consider in this chapter.

7.2 Testing of autonomous systems: simulators, test suites, and requirements

In this chapter, we consider the problem of efficiently executing a high-fidelity test suite for multi-agent autonomous systems using symmetry.

Simulators, planners, and agents dynamics As in Definition 4.1 in Chapter 4, we assume in this chapter that an agent is described by a tuple $A = \langle X, \mathscr{P}, f \rangle$, where X is the state space, $\mathscr{P}$ is the parameter space, and $f : X \times \mathscr{P} \to X$ defines the right-hand-side of the differential or difference equation modeling its dynamics. Instead of knowing f, we assume that there exists a simulator that is able to generate the trajectories of the agents in different scenarios. The simulator can simulate a single agent following a given mode for a given time step as well as multiple agents in a scenario interacting through the planning module plan. We denote these two operating modes of the simulator by simulateAgent and simulateScenario, respectively. We describe them as follows:

simulateAgent**:** given an initial state of an agent $x_0 \in X$, mode $p \in \mathscr{P}$, and a time bound $T_s \geq 0$, the simulator generates a time-annotated list of states *seg*

representing the discrete-time trajectory of the agent starting from x_0 and following mode p for T_s seconds. The time steps in the generated trajectory need not be equally separated.

simulateScenario**:** the simulator simulates multiple agents in a given scenario while they interact through plan. Given a list of initial states of a set of agents, their default plans (e.g., sequences of waypoints), the planning module plan, and a time bound T, the simulator generates a list of time-annotated trajectories of the agents starting from their initial states and following the modes that are periodically (with period T_s) updated by plan, up till time T.

Test cases and executions A *test case* defines a scenario or an encounter with multiple agents. A test case specifies the initial states and the default plans of the agents to reach their destinations in a certain scenario or encounter. A *test suite* is a set of test cases. An *execution* of the system corresponding to a test case consists of the trajectories of the different agents in the corresponding encounter starting from the defined initial states and following the modes decided by plan.

Requirements to test Given the executions of the test suite, a wide variety of requirements of the planning module plan, the agents dynamics, and their composition, can be checked. Such properties include safety, liveness, and efficiency.

We can check if plan is capable of preventing collision between different agents in different encounters by checking if the position coordinates of their executions intersect at the same time instant in any of the test cases. The main purpose of ACASs is to prevent collision between aircraft. Whether they is capable of that should be tested after each update of their software implementations or the agents dynamics.

Additionally, we can check if plan is capable of guiding the agents to their intended destinations by checking if their executions across different test cases end at the predefined waypoints of the agents.

Also, we can check if plan is efficient by checking how fast can it drive the agents to their destinations by checking the durations of the executions of the different test cases. In ACAS design, an efficiency property of interest is minimizing the number of advisories given for smoother trajectories of aircraft. However, when checking the executions of different test cases, knowing the minimal number of advisories for a certain scenario is not a straightforward task. This is an instance

of the *oracle problem* in testing: the problem of specifying the correct output for each test case. The oracle problem is especially present in testing autonomous systems. *Metamorphic testing* is known for mitigating the oracle problem [159, 157, 160, 158, 47, 161]. In metamorphic testing, instead of specifying the *correct* output for a test case, the outputs of related test cases are compared and checked if they are related as expected. Such relations are called *metamorphic properties*. For example, consider two test cases E1 and E2 for a closed-loop ACAS system, each with two aircraft. Assume that the two test cases only differ by the initial distance between the two aircraft: E1's initial distance is larger than E2's initial distance. Then, an example metamorphic property for ACAS is that the number of non clear-of-conflict resolution advisories given for the aircraft in E1 is smaller than that of E2. Using the executions of *metamorphic* test cases in the test suite, one can check if plan satisfy an expected metamorphic property.

Problem definition Given a test suite *tests*, where all agents share the same dynamics, and a corresponding virtual map $\Phi = \{(\gamma_x, p_x)\}_{x \in X}$, *efficiently* generate the executions of *tests*.

The standard approach to generating executions for a test suite is to call a simulateScenario to generate the executions, without utilizing Φ. In this chapter, we show how using Φ, we can significantly reduce the testing computation time.

7.3 Efficient testing using symmetry

Now we proceed to describe the main contribution of this chapter: an efficient testing algorithm that uses symmetry to avoid running redundant tests. We present two version of the algorithm: symTest and absSymTest (Algorithms 7.1 and 7.2). While symTest is easier to implement, absSymTest achieves significantly better savings. Drawing parallels with the previous chapters, symTest is similar to Chapter 4 in testing the provided scenarios and caching the results, while absSymTest is similar to Chapters 5 and 6 in creating a new reduced scenario from the provided one, that is faster to test. Here are the common elements to both versions:

Inputs and outputs: they take as input a test suite *tests*, a list of lists of states, i.e List[List[X]]. Each element of the list represents a test specifying an encounter of two or more agents. The test consists of the initial states of the agents (List[X]), the

paths they will follow in the encounter, the period at which the agents communicate, and the duration of the encounter. Without loss of generality, we assume that all tests have the same period T_s and duration T. The duration T should not be confused with the input to ACAS T described later in Figure 7.3, which specifies the time to collision. In our experiments, we specify the duration to be longer than the time to collision. This is necessary to check that the aircraft will not collide at a later time. Also, we assume that the paths are implicitly specified by the index of the test in the list.

They also take as input a virtual map $\Phi = \{(\gamma_x, \rho_x)\}_{x \in X}$. In contrast with the one in Definition 4.2, the symmetries in the virtual maps we consider in this chapter are parameterized by the state instead of the mode. This will help us defining symmetries for aircraft in our case study based on body coordinates (Section 7.4.3).

The two versions symTest and absSymTest output *Execs*, a list of lists of lists of states, i.e. List[List[List[X]]]. Each list represents the execution of the encounter of the test with the same index in the test suite *tests*. Each simulation consists of the trajectories, or corresponding approximations, of the agents in the encounter. Each trajectory is a list of states.

Cache for trajectory segments: they maintain a cache that store trajectory segments of an agent with the same dynamics as those of the agents in the test suite. Without loss of generality, we assume that all agents have the same dynamics. Otherwise, they would maintain different caches for different dynamics. The cache would be a dictionary mapping keys to values: {key: $X \times \mathscr{P}$, value: List[X]}. The key is a pair of a state and mode, and the value is a list of states representing a trajectory segment of duration T_s.

Cache key lookup function: they use the function *getKey* to obtain the key to the entry in *Cache* corresponding to a given pair of a state x and a mode p. The function *getKey*(x, p) first uses the input virtual map to transform the pair (x, p) to its representative one $(\gamma_x(x), \rho_x(p))$. It is improbable for two naturally occurring pairs of states and modes to have *exactly* matching transformed pairs of states and modes. Assuming low sensitivity of the dynamics to small perturbations in the initial states and modes, which can be obtained using reachability analysis as described in previous chapters, *getKey* quantizes the transformed pair according to a grid with a predefined resolution.

7.3.1 symTest algorithm

The algorithm symTest iterates over the tests in *tests* in line 5 of Algorithm 7.1. In each iteration, it first tries to retrieve the longest prefix of the simulation for the corresponding test from *Cache*, that stores the trajectory segments. Then, it simulates the test for the rest of its duration. Finally, it caches the resulting trajectories of the agents in *Cache*. Symmetry is used for efficient caching and retrieval of trajectories from the *Cache*.

More specifically, in each test in *tests*, symTest iterates over its time horizon $[0, T]$, T_s seconds at a time in line 6. At each period, symTest calls plan to get the mode for each of the agents for the next period in line 7. Then, symTest iterates over the agents in the scenario in line 8. For each agent with id *agentid* in the encounter, symTest uses *getKey* to get *key*, the key to the entry at which its representative trajectory segment would be stored in *Cache*. If the *Cache* is non-empty, symTest retrieves and saves it in the list *Execs* corresponding to the agent in the particular test case in line 11. If it is not in *Cache*, symTest stops the loop of line 14, deletes all the segments that were added for the other agents in the current iteration of the loop in line 6, and calls simulateScenario to generate the rest of the simulation in line 16. If the segments of the trajectories of all the agents were found in the cache for that period, symTest transforms these segments using symmetries in the input virtual map and stores the results in *Execs* in line 11.

The last part of the algorithm stores the newly simulated trajectory segments of the agents in *Cache*. To do that, it loops over the rest of the periods for the rest of the duration (line 17). Another nested loop iterates over the agents (line 18). The trajectory segment of the agent at that period is transformed using the virtual map and saved in *Cache*. The algorithm finally returns *Execs*.

We evaluate this version of our testing algorithm in a set of preliminary experiments shown in Section 7.5. In the following section, we present the alternative algorithm absSymTest.

7.3.2 absSymTest algorithm

Instead of running the test cases of *tests* one at a time and simulating multiple agents in each test, absSymTest simulates a single representative agent in a single execution. Using the trajectory of the representative agent, absSymTest generates the trajectories of all agents in all test cases of *tests*.

Algorithm 7.1 symTest

1: **input:** *tests*: List[List[X]]; T: $\mathbb{R}^+$; T_s: $\mathbb{R}^+$
2: **output:** *Execs* : List[List[List[X]]
3: **variables:** *Cache* : {key: $X \times \mathscr{P}$, value: List[X]}; *pa*: List[$\mathscr{P}$]; *testid*: $\mathbb{N}$; *agentid*: $\mathbb{N}$; x_0: X; p: $\mathscr{P}$; *seglist*: List[List[List[X]]]; *plist*: List[List[$\mathscr{P}$]]
4: **external functions:** simulateScenario: $\mathbb{N} \times$ List[X] $\times \mathbb{R}^+ \to$ List[List[List[X]]] $\times$ List[List[$\mathscr{P}$]], plan : $\mathbb{N} \times$ List[X] $\to$ List[$\mathscr{P}$], $\Phi : \{(\gamma_x, \rho_x)\}_{x \in X}$, where $\forall x \in X$, $\gamma_x : X \to X$ and $\rho_x : \mathscr{P} \to \mathscr{P}$
5: **for** *testid* $\in [0 : |tests|]$ **do**
6: **for** $i \in [0 : \lfloor T/T_s \rfloor]$ **do**
7: $pa \leftarrow \mathsf{plan}(\langle Execs[testid][0].lstate, Execs[testid][1].lstate, \dots \rangle)$
8: **for** *agentid* $\in [0 : |tests[testid]|]$ **do**
9: $key \leftarrow getKey(Execs[testid][agentid].lstate, pa[agentid])$
10: **if** $Cache[key] \neq \perp$ **then**
11: $Execs[testid][agentid]$.append$(\gamma^{-1}_{Execs[testid][agentid].lstate}(Cache[key]))$
12: **else**
13: Erase trajectories added to *Execs* in the current iteration in line 6
14: **break**
15: $seglist, plist \leftarrow$ simulateScenario$(testid, \langle Execs[testid][0].lstate,$
16: $Execs[testid][1].lstate, \dots \rangle, T - iT_s)$
17: **for** $i' = i, i+1, \dots, T-1$ **do**
18: **for** *agentid* $\in [0 : |tests[testid]|]$ **do**
19: $key \leftarrow getKey(seglist[i'][agentid].fstate, plist[i'][agentid])$
20: $Cache[key] \leftarrow \gamma_{seglist[i'][agentid].fstate}(seglist[i'][agentid])$
21: **return:** *Execs*

In addition to *Cache*, absSymTest keeps track of two sets *readyset* and *waitset*. The set *readyset* stores four-dimensional tuples of test ids, agent ids, states, and modes. Each such tuple represents a query for a trajectory segment of duration T_s. The set *waitset* stores for each test case a set of states for some of its agents.

The algorithm absSymTest starts by calling plan on the initial states of the agents in each test case in *tests* to get the modes they should follow for the next T_s time, in line 6 of Algorithm 7.2. Then, absSymTest initializes *readyset* with the tuples resulting from annotating each initial state of an agent in a test case in *tests* with its mode, agent id, and test id, in line 8.

Then, absSymTest pops the tuples in *readyset* one at a time. For each such tuple $(testid, agentid, x_0, p)$, absSymTest first computes the key for the entry in *Cache* corresponding to (x_0, p) using *getKey* in line 11. The fact that *getKey* takes only the state and mode as input, leaving the agent and test ids, implicitly assumes that the dynamics are the same across different agents and tests. If the entry in the cache is empty, absSymTest asks the agent to drive (or fly) to a state where there exists a mode p' such that the pair of the resulting state of the agent and p' are mapped to the same entry in *Cache* as the pair (x_0, p). Such a task might be fulfilled by resetting the state of the agent, if the simulator allows it, to x_0. Another approach is to use the controller to drive the such a state. Such a task might be challenging for the controller. For example, the task might be to drive an aircraft to a certain pitch and roll. In our experiments with absSymTest, we avoid tackling this challenge by using the virtual map that transform states and modes to body coordinates. Thus, there is always a mode that satisfies the condition in line 13 and the representative agent does not have to be driven to a new state. Once the representative agent reaches that state, absSymTest directs it to follow mode p' for T_s seconds, by calling the simulator in line 14. absSymTest caches the resulting trajectory segment after transforming it to the virtual coordinates in line 15. One can implement the *pop* function to choose the tuple with state x with $\gamma_x(x)$ closest to $\gamma_{reftraj.lstate}(reftraj.lstate)$, the representative state of the current state of the representative agent.

Now that the needed virtual trajectory segment is guaranteed to be in *Cache*, absSymTest retrieves it, transforms it to the original coordinates, and appends it to the corresponding agent trajectory in *Execs* in line 16.

Finally, absSymTest checks if the trajectories of all agents for the test case *testid* reached the same time step. That allows plan to be called to decide the modes that the agents have to follow in the next T_s seconds. While some of the agents are still

one T_s time step behind, *waitset*[*testid*] stores the states of the agents that have their trajectory segments generated. Once the trajectory segments of all agents of *testid* are generated, absSymTest calls plan with the states in *waitset*[*testid*] as input to obtain the modes for the next time step in line 19 and resets *waitset*[*testid*] to an empty list in line 22. Once the modes are known for the next period, in line 21, absSymTest adds the resulting tuples to *readyset* for the corresponding trajectory segments to be later simulated or retrieved from *Cache*.

In the following section, we present the case study on which we evaluate the performance of symTest and absSymTest.

Algorithm 7.2 absSymTest

1: **input:** *tests*: List[List[X]], T: $\mathbb{R}^+$, T_s: $\mathbb{R}^+$
2: **output:** *Execs* : List[List[List[X]]
3: **variables:** *Cache* : {key: $X \times \mathscr{P}$, value: List[X]}, *pa*: List[$\mathscr{P}$], *testid*: $\mathbb{N}$, *agentid*: $\mathbb{N}$, x_0: X, p: $\mathscr{P}$, *reftraj*: List[X], *readyset*: List[$\mathbb{N} \times \mathbb{N} \times X \times \mathscr{P}$], *waitset*: List[List[$X$]]
4: **external functions:** simulateAgent: $X \times \mathscr{P} \times \mathbb{R}^+ \to$ List[X], plan: $\mathbb{N} \times$ List[X] $\to$ List[$\mathscr{P}$], $\Phi : \{(\gamma_x, \rho_x)\}_{x \in X}$, where $\forall x \in X$, $\gamma_x : X \to X$ and $\rho_x : \mathscr{P} \to \mathscr{P}$
5: **for** $testid \in [0 : |tests|]$ **do**
6: $\quad pa \leftarrow \mathsf{plan}(testid, \langle tests[testid][0], tests[testid][1], \dots \rangle)$
7: $\quad$ **for** $agentid \in [0 : |tests[testid]|]$ **do**
8: $\quad\quad readyset.push((testid, agentid, tests[testid][agentid], pa[agentid]))$
9: **while** $readyset \neq \langle\rangle$ **do**
10: $\quad testid, agentid, x_0, p \leftarrow readyset.pop()$
11: $\quad key \leftarrow getKey(x_0, p)$
12: $\quad$ **if** $Cache[key] = \perp$ **then**
13: $\quad\quad$ Drive (or fly) to a state where $\exists\ p'$, $getKey(reftraj.lstate, p') = getKey(x_0, p)$
14: $\quad\quad seg \leftarrow \mathsf{simulateAgent}(reftraj.lstate, p', T_s)$
15: $\quad\quad Cache[key] \leftarrow \gamma_{reftraj.lstate}(seg)$
16: $\quad Execs[testid][agentid].append(\gamma_{x_0}^{-1}(Cache[key]))$
17: $\quad$ **if** $\forall k \in agents[testid] \setminus \{agentid\}$, $|waitset[testid][k]| \neq 0$, and
18: $\quad\quad\quad Execs[testid][k].dur < T$ **then**
19: $\quad\quad pa \leftarrow \mathsf{plan}(testid, waitset[testid])$
20: $\quad\quad$ **for** $k \in agents[testid]$ **do**
21: $\quad\quad\quad readyset.push((testid, k, waitset[testid][k], pa[k]))$
22: $\quad\quad waitset[testid] \leftarrow \langle\rangle$
23: $\quad$ **else**
24: $\quad\quad waitset[testid][agentid].push(\gamma_{x_0}^{-1}(Cache[key]).lstate)$
25: **return:** *Execs*

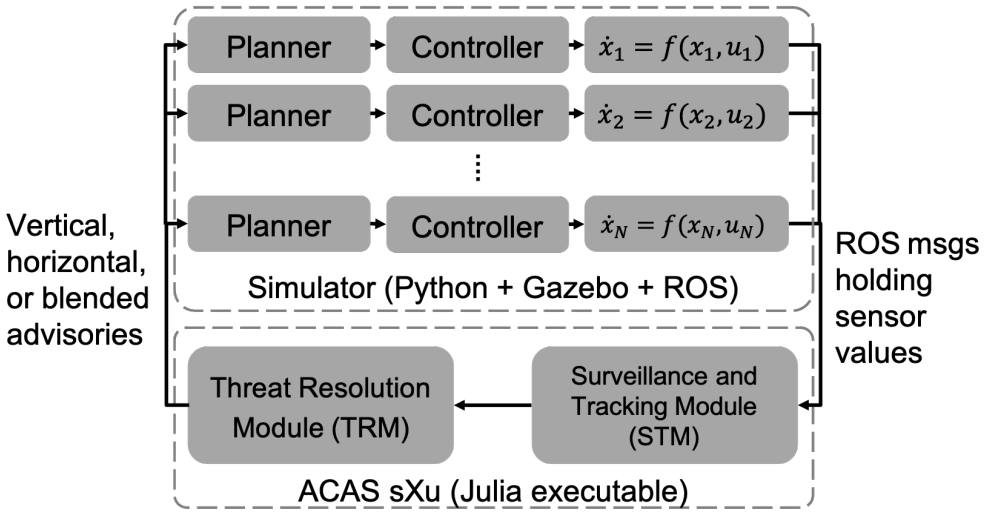


Figure 7.2: Closed-loop integration of ACAS sXu with aircraft. The Planner of each aircraft takes as input the aircraft's state and the ACAS advisory. It has a predefined set of waypoints to follow in the absence of an ACAS advisory. It outputs the corresponding 3D *reference velocity* vector of an ACAS advisory or predefined waypoint. The Controller takes as input the output of the Planner and the aircraft's state and outputs corresponding commands for the aircraft's actuators. The dynamics of the aircraft would determine its next state given its current state and the Controller output. We implemented the Planner in Python and used existing aircraft models to simulate in ROS and Gazebo. The ACAS sXu shown is the Julia executable representing the specification of Version 3 of ACAS sXu released in 2021 [162]. It updates its advisories at 1 Hz frequency. The STM module takes as input a stream of the aircraft positions as ROS messages, estimates the states of the aircraft using Kalman filters, and provide the TRM module with the variables shown in Figure 7.3. The TRM generates the advisories for the aircraft in the encounter.

7.4 Case study: closed-loop testing of ACAS

In this section, we describe the history of ACAS and our testing setup for evaluating its closed-loop behavior.

7.4.1 Overview of Airborne Collision Avoidance Systems

Airborne Collision Avoidance System for smaller Unmanned Aircraft Systems (ACAS sXu) is a software system being designed by the Federal Aviation Administration (FAA) to provide Detection and Avoid (DAA) capabilities for small (below 55 pounds) unmanned aircraft systems (UASs) [97]. ACAS sXu is planned to be deployed either on the aircraft hardware or as a remote service that the aircraft connect to [162]. In an encounter where one or more aircraft are at risk of collision

with each other, with the ground, or with fixed obstacles, ACAS sXu, or simply ACAS, provides the aircraft with advisories to avoid collision. A wrong output of an ACAS might lead to an avoidable aircraft collision. In this chapter, our experiments test the ACAS sXu Version 3 (ACAS_ADS_21_001_V3R0a, V3R0) that was released on March, 2021 [162]. The release comes with a Julia executable specifications that determine the correct ACAS sXu output for a given input. This executable specification is supposed to be used by Aircraft manufacturers when implementing their own ACAS sXu software.

Hundreds of mid-air collision between manned, both civilian and military, aircraft have happened in the past leaving many fatalities [163]. An earlier version of ACAS, the Traffic Collision Avoidance System (TCAS) is believed to have reduced such collisions significantly [164]. However, a mid-air collision between two civilian aircraft in 2002 leaving 71 human fatalities was at least partially attributed to a wrong TCAS advisory [165]. Several versions of TCAS have been developed over the past three decades. By 2023, two to three million of UASs are expected by the FAA to be deployed in the national airspace [1]. This only increases the possibility of mid-air collisions and the chances of edge cases, if any, in the ACASs to be encountered. A proper and thorough testing and validation of such collision avoidance systems is thus necessary.

Several ACAS versions including ACAS Xu (ACAS version for large UASs) and ACAS sXu have been tested by the FAA both in open-loop and closed-loop settings [166, 9]. A line of academic research has also focused effort on verifying neural network-based implementations of both ACAS Xu and ACAS sXu, both in open-loop and closed-loop settings [167, 19, 118, 168, 169, 170]. With the absence of clear component-level specifications, open-loop testing faces the *Oracle* problem: specifying the correct output for each test input. Current research methods verify or test few open-loop properties based on intuition, e.g., if the intruder is near and coming from the right the advisory should be turn left [167, 19]. For closed-loop verification or testing, aircraft dynamical models or simulators are needed. In the literature, low-dimensional differential equations have been used for the existing verification tools to be able to verify safety in reasonable computation time [170, 169]. The FAA and other academic and industrial partners have tested ACAS sXu Version 1 in the field on real aircraft for a number of scenarios in 2019 [166]. However, such field experiments, although necessary, require significant resources and effort. Field experiments are very expensive if done after each software update and if need to cover many scenarios. An efficient simulation environment is thus

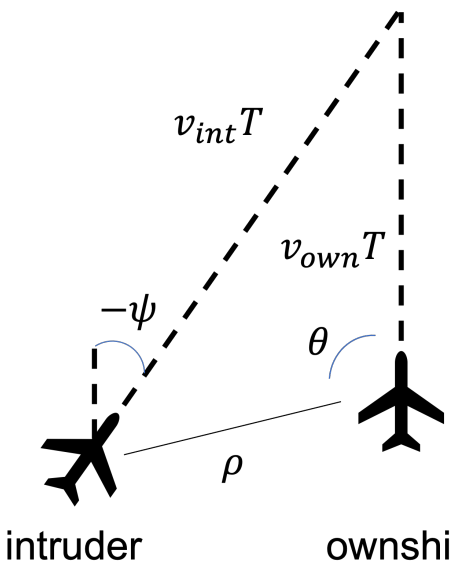


Figure 7.3: Encounter example (Left gaining). The inputs to the module in ACAS Xu that provides horizontal advisories: v_{own} and v_{int} are the speeds of the ownship and the intruder, respectively. ρ is the initial distance between the two aircraft. ψ is the heading angle of the intruder with respect to the ownship heading. θ is the angle from the ownship heading to the intruder. T is the time to collision. τ (not shown in the figure), is the time until the loss of vertical separation. a_{prev} is the previous advisory. Figure inspired by Lopez et al. [169].

required for regression tests of ACAS. These regression tests will help select few critical scenarios to test on real aircraft.

An ACAS takes as input the relative states of the aircraft that are at risk of collision and outputs horizontal, vertical, or blended advisories for the aircraft to follow to avoid collision. In a closed-loop system as the one shown in Figure 7.2, the values of different sensors are sent to the Surveillance and Tracking Module (STM) of ACAS sXu. The set of sensors that ACAS sXu depends on can be found in [162]. The STM module has state estimators such as Kalman filters to estimate the aircraft's states that are needed by the Threat Resolution Module (TRM) of ACAS sXu. The TRM takes as input the relative positions and heading angles, the absolute speeds, the time to loss of vertical separation, the time to loss of horizontal separation, and the previous advisory. The variables are shown in Figure 7.3. The ACAS advisories specify the recommended turn direction (left, right, climb, descend, or maintain) and turn rate in ft/s. An ACAS sXu runs at a 1 Hz frequency updating its advisories to the aircraft each second. In the absence of an ACAS advisory, the agents follow their original plans such as delivering a package or putting out a fire. Such plans are expected to be managed by protocols such as

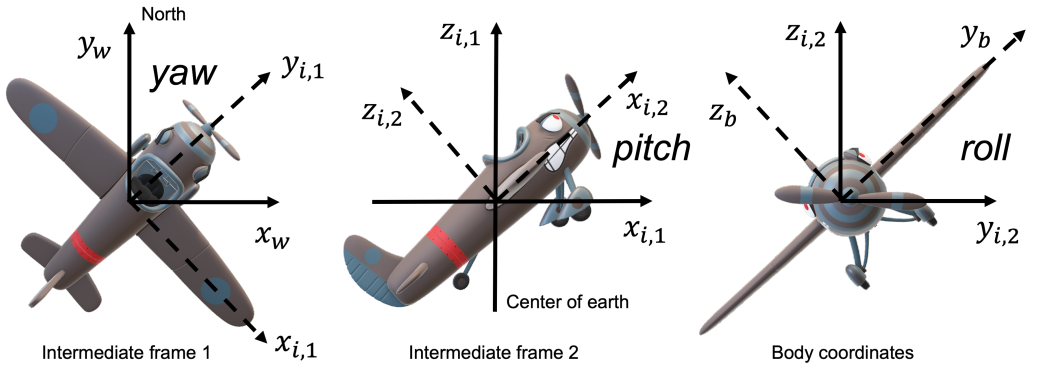


Figure 7.4: Transforming the world coordinates to body ones. First, the *xy*-plane is rotated with the heading (yaw) angle, resulting in the first intermediate frame (left figure). Then, the *xz*-plane of the first intermediate frame is rotated with the pitch angle, resulting in the second intermediate frame (middle figure). Third, the *yz*-plane of the second intermediate frame is rotated with bank (roll) angle, resulting in the body coordinates (right figure). Although the world coordinates shown have the origin at the center of mass of the aircraft, it can be fixed at any point independent from the aircraft. In that case, the first step would translate the origin to the center of mass of the aircraft, then follow the steps in the figure. The body coordinates always have their origin at the center of mass of the aircraft. Figure inspired from the book "Unmanned Small Aircraft" by Beard and McLain [172].

the Unmanned Traffic Management (UTM) protocol for strategic de-conflicting of UASs' paths [171][2].

7.4.2 States, planners, and controllers of aircraft

The kinematic state x of an aircraft can be simplified to be in $\mathbb{R}^9$, where $x[0:2]$ is its position in the 3D physical space, $x[3:5]$ is its 3D linear velocity, and $x[6]$, $x[7]$, and $x[8]$ are its pitch, roll, and yaw angles, respectively (see Figure 7.4). That is, the aircraft's state space is $X \subseteq \mathbb{R}^9$.

An aircraft has a default plan to execute such as search, delivery, or firefighting. Given such a plan or the advisories of an ACAS, and the current state of the aircraft, the planning module plan determines the *current* reference velocity vector $v_r := \langle v_{x,r}, v_{y,r}, v_{z,r} \rangle \in \mathbb{R}^3$ that the controller should drive the aircraft to in the next time step. The vector v_r represents the *mode* of the aircraft in that time step.

Given the state x and the output of plan, the aircraft controller sends commands to its actuators determining its thrust and angular velocities with the aim to control

[2]UTM is also currently under development by the FAA and industrial partners to manage the exponentially increasing deployment of UASs in the national airspace.

the aircraft's linear acceleration. The output of the controller is a function of the current aircraft's velocity subtracted from the reference velocity: $v_r - x[3:5]$. The controller updates its output continuously in time or discretely, but at a higher frequency than that of plan. An example of an ordinary differential equation model of a quadrotor and its corresponding controller can be found in [173].

7.4.3 Reference coordinate frames: world, relative, and body

In this section, we discuss the different coordinate frames in which we represent the states of the aircraft. This discussion will be essential in defining the symmetries of the aircraft dynamics and explaining our efficient testing algorithm in later sections.

Typically, when a vector is defined, it represents the coefficients to different basis in a vector space. A state is a vector. Hence, when we define the state of an aircraft, we define it according to a coordinate system. For example, one coordinate system for the position part of the state would be the Global Positioning System (GPS) for x and y and above sea level altitude for z, all in feet or meters. In this chapter, we consider three types of frames: *world*, *relative*, and *body* coordinates. See Figures 7.3 and 7.4.

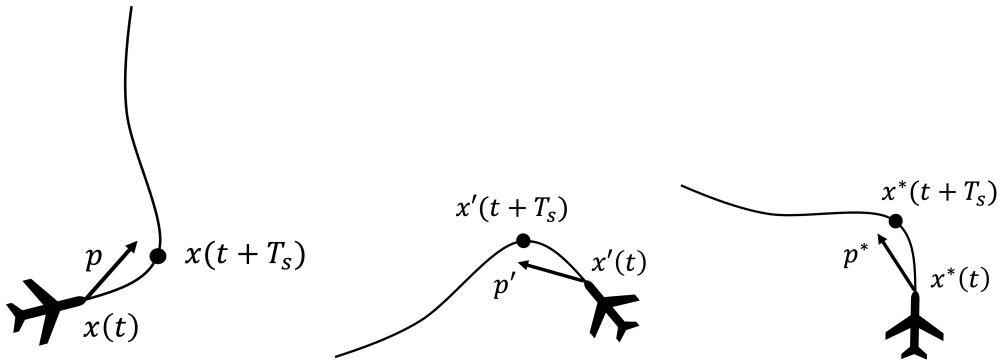
***World* (or *absolute*) coordinates** These coordinates are the typical coordinate systems where the origin and direction of the different axes are chosen independent from the aircraft state. The coordinate system based on GPS and sea level are examples. According to such a coordinate system, we specify the initial states in test cases in our case study.

***Relative* coordinates** These coordinates are based on the state of one aircraft according to which the state of another aircraft is represented. This is the coordinate system according to which an ACAS TRM takes its input. In fact, ACAS TRM takes subset of the relative states as input, ignoring for example the pitch and roll angles of all aircraft. The state of the ownship would be considered the reference state according to which the intruder's state is represented (see Figure 7.3).

***Body* coordinates** These coordinates are based on the state of the aircraft. The controller usually represents the reference velocity vector (or equivalently, parameter or mode) v_r with respect to the current state of the aircraft. The controller translates and rotates the vector v_r by the velocity vector of the aircraft and its

rotation angles, respectively. Then, it would compute from the result the needed actuators input. Because of such representation, the dynamics of an aircraft are expected to be invariant to rotating its yaw angle and rotating v_r accordingly. The same holds for translating the position of the aircraft without changing v_r. If the planning module plan outputs a waypoint instead of a velocity vector, then the closed-loop dynamics would be invariant to translating the aircraft position if the waypoint is translated by the same vector. However, without loss of generality, we assume in this chapter that plan outputs reference velocity vectors. Depending on the dynamics of the aircraft, the controller might be able to adjust the thrust according to the roll and pitch of the aircraft in a way that cancels the effect of gravity and makes the aircraft trajectories symmetric to rotations in the pitch, roll, and yaw, with v_r rotated accordingly. This is for example expected from a quadrotor that is capable of vertical ascent, but might not be expected from a fixed-wing aircraft which will not be capable of flying backwards, for example. Finally, the dynamics might also only depend on the *difference* between the reference and current velocity of the aircraft. In that case, the trajectories of aircraft with the same difference between reference velocity and initial one can be computed from each other through simple transformations.

Symmetries and the reference frames A virtual map $\Phi = \{(\gamma_x, \rho_x)\}_{x \in X}$ of an aircraft would formalize the symmetries that its dynamics satisfy. The transformations in Φ are parameterized with the state x instead of the mode p, in contrast with previous chapters. This allows for transforming a trajectory of an aircraft and the mode it is following to its body coordinates defined by its initial state. For example, in the case of invariance of the dynamics to rotations of its heading and translations of its 3D position, for any x and $x' \in X$, $\gamma_x(x')$ would transform the state x' to the first intermediate frame in Figure 7.4 defined by the state x, i.e. (1) translate the position in x', i.e. $x'[0:2]$, by the position in x, i.e. $x[0:2]$, (2) translate yaw in x', i.e. $x'[8]$ by the yaw in x, i.e. $x[8]$, and (3) rotate the translated position in x' to the new coordinate system where the position horizontal plane is rotated by the yaw in x. The corresponding ρ_x, for any $p \in \mathscr{P}$, would rotate the first two coordinates of p by the yaw in x. Given a trajectory ξ of the aircraft following mode p, one can choose its initial state $\xi.fstate$ to be the reference state for the coordinate system. A *representative* trajectory of ξ in that case would be $\gamma_{\xi.fstate}(\xi)$ following mode $\rho_{\xi.fstate}(p)$, where all the states of ξ would be represented with respect to the first intermediate frame, second intermediate frame, or the body coordinates defined by



(a) Example trajectory segment of an aircraft following the reference velocity vector, or mode, p for T_s seconds, starting from state $x(t)$.

(b) Example trajectory segment of an aircraft following mode p', a rotated version of mode p, starting from state $x'(t)$, a rotated version of state $x(t)$.

(c) Both trajectory segments in Figures 7.5a and 7.5b would be transformed using the virtual map to a representative trajectory segment, the one between $x^*(t)$ and $x^*(t+T_s)$, following p^*.

Figure 7.5: Example application of the translation and rotation-based virtual map that transforms trajectory segments to representative segments based on the first intermediate frame (defined in Figure 7.4) with respect to their initial states. Here, $\gamma_{x(t)}(x(t)) = \gamma_{x'(t)}(x'(t)) = x^*(t)$ and $\rho_{x(t)}(p) = \rho_{x'(t)}(p') = p^*$. Similarly, $\gamma_{x(t)}(x(t+T_s)) = \gamma_{x'(t)}(x'(t+T_s)) = x^*(t+T_s)$.

its initial state. Trajectory segments of different aircraft or of the same one that are symmetric can be represented with the same trajectory using Φ, from which all of them can be retrieved. See, for example, Figure 7.5.

7.4.4 Test suite generation

In this chapter, we assume that a test suite is provided and does not suggest new test generation methods. This section presents the test suite we used in our case study, which we adopted from Lopez et al. [170]. All test cases in the test suite consist of two agents: the ownship and intruder, as in Figure 7.3. Instead of using the ODEs described in [170], we use the high-fidelity Gazebo-based Hector Quadrotor models [2]. The initial states of both agents in the test cases are generated based on constructing a grid over the input variables of ACAS: the angle to intruder w.r.t ownship heading angle θ, the speed of the intruder v_{int}, and the distance between the intruder and ownship ρ. The discretized values are:

- $\theta \in \{-\frac{3\pi}{4}, -\frac{\pi}{2}, -\frac{3\pi}{8}, -\frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{3}, \frac{3\pi}{4}, \pi\}$ rad,

- $v_{int} \in \{60, 150, 300, 450, 600, 750, 900, 1050, 1145\}$ ft/s, and
- $\rho \in \{10000, 43736, 87472, 120000\}$ ft.

Instead of the discretization of τ, the time before loss-of-vertical separation, we choose three different vertical separations between the ownship and intruder, respectively: $\{-2000, 0, 2000\}$ ft. The rest of the input variables to ACAS, namely v_{out} and ψ, can be obtained using formulas presented in [170]. Since these variables are the input to ACAS, they only specify parts of the states of the agents and in relative coordinates. For example, initial pitch, roll, and vertical velocity are unspecified. We fix the initial position of the ownship to be at the origin and the other unspecified variables to zero. In [170], only ten combinations from the discretized values were considered for reachability analysis. In this paper, we test all combinations and consider the resulting test suite a seed from which we generate further tests using metamorphic relations. For each test case, we choose the predefined plans for both agents to be a single waypoint in the 3D physical space. That waypoint is the point at which they are expected to collide given their initial velocities and heading angles.

It is worth noting that metamorphic testing results in test suites defining symmetric encounters. For such test suites, the algorithms presented in this chapter are expected to achieve better savings. However, the encounters do not have to be symmetric for the algorithms presented to speedup testing. They will utilize symmetry between trajectory segments over periods at which the plan updates the modes. Such symmetry would be present even in non-symmetric encounters. Finally, the executions generated by the testing algorithms of this chapter assume that the dynamics of the agents are indeed symmetric with respect to the user-defined virtual map. Thus, if these executions are compared with the executions generated by standard testing algorithms, the result is an assessment of how symmetric the dynamics are. The latter itself is a form of metamorphic testing, but for the dynamics, rather than for ACAS or plan.

7.5 Experimental results

In this section, we describe the experimental setup and preliminary results on running symTest and absSymTest on the test suite described in Section 7.4.4.

7.5.1 Experimental setup

We implemented both of the presented algorithms in Python. The codes for symTest[3] and absSymTest[4] are available online. We use two different simulators and agent dynamics in testing symTest and absSymTest.

Julia executable specifications with Hector Quadrotor models We integrated the executable specifications of ACAS sXu Version 3 (ACAS_ADS_21_001_V3R0a, V3R0) [162] with the Hector Quadrotor [98] using Robot Operating System (ROS) [99] and Gazebo [2]. The resulting simulation setup was used as the scenario simulator simulateScenario in symTest. The constructed simulateScenario takes as input (1) a set of initial states, consisting of the initial positions, linear velocities, and yaw angles, of a set of Hector quadrotors, (2) a preplanned path of waypoints for each quadrotor to follow, and (3) the duration of the simulation. This simulator is built on top of CyPhyHouse [174, 175] and the SkyTrakx toolkit [176]. The new developments over the mentioned toolkits are (1) integrating ACAS sXu in closed-loop with the quadrotors being simulated and (2) adding the capability of initializing the states of quadrotors with non-zero velocities.

A block diagram showing the integration between the ACAS sXu system and the Gazebo simulator is shown in Figure 7.6. The Julia executable of ACAS sXu consists of a server and a client. The client communicates with the Gazebo simulator as well as with the Python procedure that decides the waypoints that an agent should follow through ROS messages. The server has a Surveillance and Tracking Module (STM) instance for each agent in the simulation. Each such instance estimates the state of the corresponding agent given the sequence of its positions sent by Gazebo through the client over a rostopic. The server sends the ACAS advisories to the agents through the client. Each agent generates a waypoint in the 3D physical space to follow from the ACAS advisory. Such a waypoint is defined in a way that following it is equivalent to following the ACAS advisory. The velocity vector suggested by the advisory will be used to compute a reference velocity vector, from which a waypoint is computed assuming one second duration to reach it.

The Hector Quadrotor [98] comes with a built-in waypoint-following controller. Given a waypoint, the controller computes first a reference velocity vector to

[3] `https://github.com/cyphyhouse/CyPhyHouseExperiments/blob/ACAS/experiments_utm/test_suit_reduction_alt.py`

[4] `https://github.com/lyg1597/ACAS-Testing`

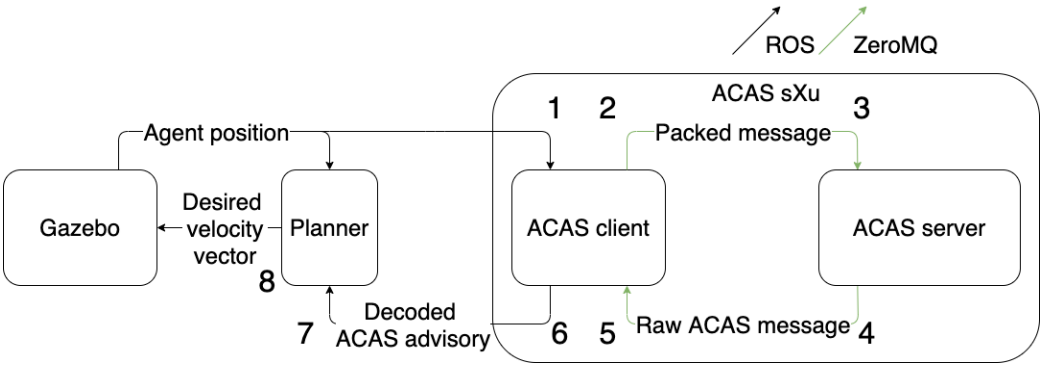


Figure 7.6: Integrating ACAS sXu with Hector Quadrotor simulator using Gazebo and ROS. ZeroMQ is the embedded messaging library used by ACAS to communicate messages between the server and the client. The numbers are used to refer to the messages in Figure 7.7.

follow from which it derives the needed thrust and rotors angular velocities. In our simulation framework, we added ROS publishers that publish these reference velocity vectors.

Using the published ROS messages from Gazebo, our simulation framework for each published reference velocity vector by the controller, the vector itself and the list of states representing the trajectory segment of the quadrotor following it. Such segments get transformed and cached by symTest in *Cache*.

Neural Network-based ACAS Xu with NN-controlled 6-dimensional quadrotors We integrated the NN implementation of ACAS Xu, presented by Julian et al. as a compression of the original lookup tables of ACAS Xu [156], with a Python-based simulator of a NN-controlled six-dimensional quadrotor. The model of the quadrotor was presented by Ivanov et al. in the Verisig verification tool paper [20]. Such a simulator allows to easily generate trajectory segments of the quadrotor following specified reference velocity vectors. Using the high-fidelity simulator in the previous paragraph would create several challenges regarding the synchronization and the timing at which the ROS messages about the sensed states and ACAS resolution advisories are sent and received. We used this simulator as simulateAgent in absSymTest. In the future, we would want to replace this simulator with the high-fidelity one in absSymTest, as the purpose of the chapter is to preserve the high-fidelity aspect while reducing the testing time.

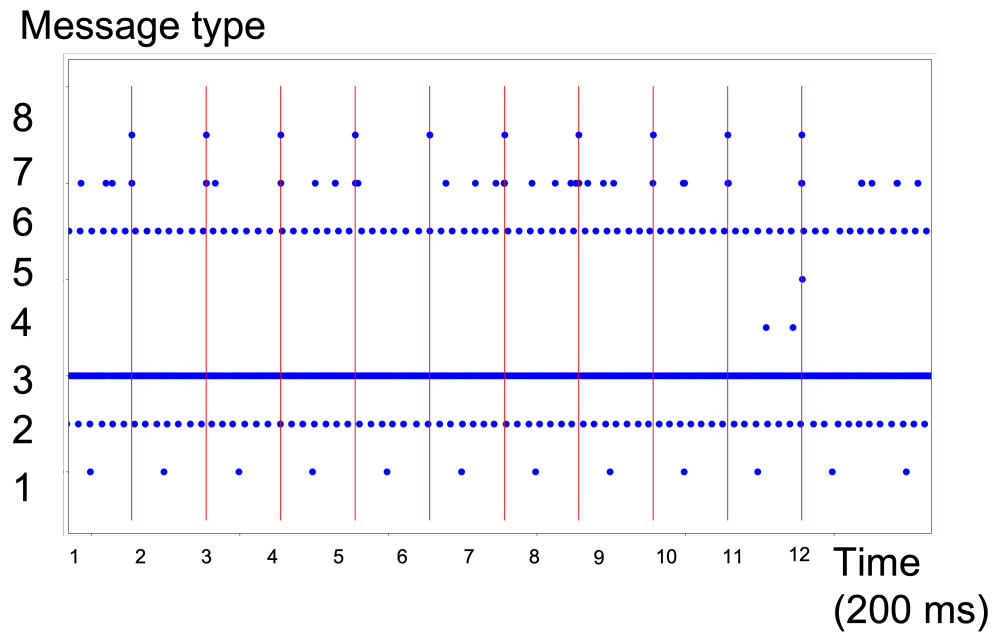


Figure 7.7: The blue dots show the time instants at which messages are sent or received between the different components of Figure 7.6 for a test case. The red vertical lines are the instances at which ACAS generated an advisory. Non-deterministic arrival time of sensing messages might cause non-deterministic simulations: repeating the same test case results in different trajectories of the agents.

Test suites Using the test generation method described in Section 7.4.4, we constructed three test suites. Each test case defines an encounter of two agents starting from different initial states with crossing predefined paths of waypoints. The three test suites are as follows:

$tests_1$: consists of 97 test cases where all agents start from the same altitude, and differ in the other state components.

$tests_2$: consists of 93 test cases. It is divided into thirds: 31 test cases where the two agents starting from equal altitudes, 31 test cases where the ownship starts from 2000 ft lower than the intruder, and 31 test cases where the ownship starts from 2000 ft higher than the intruder.

$tests_3$: consists of 31 test cases. They are the same test cases as the first third of the second test suite $tests_2$, where the agents start from an equal altitude. It is worth noting that even for the cases where agents start from equal altitude, the simulation is still in 3D and ACAS is still able to provide vertical advisories to avoid collision.

Virtual maps Recall from Section 7.3, that given a trajectory segment *seg* of an agent generated by simulateScenario or simulateAgent, symTest and absSymTest store it in their *Cache*. Before storing *seg* in *Cache*, they transform the coordinate system in which every state of *seg* is represented to a new one defined by the initial state *seg.fstate* and the user-defined virtual map.

In our experiments, we considered three virtual maps[5]:

Φ_1, which we denote by *1D rot.*:

- It rotates the heading (yaw) of the quadrotor and translates its position and rotates the mode, or equivalently, the reference velocity vector, accordingly. This map is the one described at the end of Section 7.4.3. Using this map, the initial state of a stored trajectory segment in *Cache* would always have its yaw and its position being zero.
- Key: an eight-dimensional vector including the roll (1D), pitch (1D), the rotated velocity vector (3D), and the rotated mode (3D).
- Resolution of the quantization grid: $[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-2}, 10^{-5}, 10^{-5}, 10^{-2}]$.

Φ_2, which we denote by *3D rot.*:

- It rotates the yaw, pitch, and roll of the quadrotor and translates its position and rotates the mode, accordingly. It transforms the state and the mode to the body coordinates defined by the initial state of the trajectory segment. Using this map, the initial state of a stored trajectory segment in *Cache* would always have its yaw, roll, pitch, and its position being zero.
- Key: a six-dimensional vector including only the rotated velocity vector (3D) and the mode (3D).
- Resolution of the quantization grid: $[10^{-5}, 10^{-5}, 10^{-2}, 10^{-5}, 10^{-5}, 10^{-2}]$.

Φ_3, which we denote by *3D rot. + vel. trans.*:

- It rotates the yaw, pitch, and roll of the quadrotor and translates its position and velocity and rotates and translates the mode, accordingly.

[5]We did not check if the dynamics of the Hector Quadrotor indeed satisfy the symmetries in the used virtual maps. For the low-fidelity one, we show using metamorphic testing in Appendix A.1 that it does not exactly satisfy these symmetries. However, its trajectories are almost symmetric.

Using this map, the initial state of a stored trajectory segment in *Cache* would always be zero, the origin of the state space X.

- Key: a 3-dimensional vector including only the rotated and translated mode (3D).
- Resolution of the quantization grid: $[10^{-5}, 10^{-5}, 10^{-2}]$.

Handling *Cache* misses in symTest In the presented pseudocode of symTest in Algorithm 7.1, simulateScenario is called to generate the rest of the execution of all agents, once the trajectory segment of one of the agents was not found in *Cache*. The other alternative implementation of symTest is to call simulateScenario to generate the execution for the next time step only and then return to checking *Cache* after that. In our results, we call the first implementation "S", for stopping, and the second one "C" for continuing to check *Cache*, even after cache misses. The advantages of the first implementation is that it avoids relaunching the high-fidelity simulator Gazebo, which might be slow, after every cache miss. However, it might miss all the savings from caching as usually the cache misses happen towards the beginning of the simulations. The second implementation utilizes the cache more than the first, but the expense of relaunching of Gazebo after each cache miss might overcome the savings from caching. A solution might be to keep Gazebo, or any high-fidelity simulator being used, running even while the trajectory segments are being retrieved from cache, and then *teleporting* the agent to the new initial state at which a cache miss occurred and from which it has to be simulated. The latter might be useful in implementing absSymTest as well. Specifically, in line 13, absSymTest needs to ask the controller to drive the agent to a new state, just for the sake for it to be simulated from that new state. Instead of waiting to be driven in simulation to the new state, absSymTest can teleport it.

For our preliminary experiments, and to avoid the Gazebo relaunch and agent teleportation problems, we first ran both test suites using standard testing, i.e. running the high-fidelity simulateScenario to generate the executions of all test cases. The executions of $tests_1$ and $tests_2$ with standard testing have a total of 29988 and 69666 one-second trajectories. We replaced simulateScenario in symTest with the generated executions. When a cache miss occurs, symTest retrieves the missed trajectory segment at the same time instant from the pre-generated trajectory.

Evaluation metrics We evaluate the algorithms based on the percentages of trajectory segments of the executions of a test suite that are retrieved and stored in *Cache*. We did not yet define a quantitative metric to measure the accuracy of the constructed executions using symTest and absSymTest with respect to the ones generated using standard testing in Gazebo. It is worth noting that we found that, as expected, the simulations of Gazebo are non-deterministic: simulating the same test case results in different executions in different runs. Examples of such executions are shown in Figure 7.15a. Such non-determinism further complicates the choice of accuracy metric. Other than the non-determinism of high-fidelity simulations, the accuracy of constructed trajectories by symTest and absSymTest depend on how symmetric the dynamics are with respect to the user-defined virtual map and how sensitive their trajectories are to perturbations in their initial states because of the quantizated caching.

7.5.2 Experimental statistics and discussions

We used symTest, the Gazebo simulator for the Hector Quadrotor [98], and the Julia executable for ACAS sXu [162], all components connected through ROS [99], to execute the first two test suites $tests_1$ and $tests_2$. The results for $tests_1$ are shown in Table 7.1 and Figures 7.8, 7.9, 7.10, 7.11, and 7.12 and those for $tests_2$ in Table 7.2 and Figure 7.13. With the same system, we performed metamorphic tests for ACAS sXu and the results are shown in Figures 7.15 and 7.17. We replaced the Hector Quadrotor with the fixed-wing ROSPlane [177] and repeated one of the metamorphic tests of the Hector Quadrotor. The result is shown in Figure 7.17c. Finally, we used absSymTest, the Python simulator of the NN-controlled quadrotor from Chapter 6[6], and the NN implementation of ACAS Xu [156], to execute the third test suite $tests_3$. The results are shown in Figure 7.14.

More symmetry results in more savings As can be observed from Tables 7.1 and 7.2, experiments where symTest used a more powerful virtual map have higher percentages of transformed trajectory segments and smaller percentages of cached ones. A virtual map is more powerful than another one if it represents more trajectory segments with the same representative one. The experimental results are expected, since with a more powerful virtual map, more trajectory segments would

[6]We used the modified NN-controller described in Appendix A.1, which guarantees rotation symmetry.

be mapped to the same cache entry. Hence, fewer segments have to be stored and more segments can be transformed from the same stored one.

An outlier to the analysis above is the 3D rot. experiment in Table 7.1. It has lower percentage of transformed segments and higher percentage of cached ones than that of the 1D rot. experiment. In 1D rot., γ_x does not alter the roll and pitch part of the state, and only rotates the mode in 1D, to the first intermediate frame shown in Figure 7.4. The roll and pitch are thus part of the key to the cache, which gets quantized by *getKey*. In 3D rot., the reference velocity vector or mode p will be rotated in 3D by ρ_x. In that case, $\gamma_x(x)$ will have zero pitch, roll, and yaw, and the pitch and roll would not be part of the key. Thus, the actual values of pitch and roll affect the value of the key, instead of the quantized ones. In summary, in 1D rot., the quantized pitch and roll are part of the key, while in 3D rot., the actual pitch and roll affect the key. Such difference might lead to having more unique cache keys, and consequently less cache hits, when using the 3D rot. virtual map than when using 1D rot..

Hence, in the implementation used for the presented experiments, *getKey* first computes $\gamma_x(x)$ and $\rho_x(p)$, then quantizes the result according to a predefined grid. In such an approach, the effects of quantization and symmetry are intertwined: while the symmetries are using the actual state x and mode p, the result of *getKey* is quantized. That is in contrast with quantizing the state and mode first according to a predefined grid, applying the transformations, and quantizing the result. More formally, in the latter approach, *getKey* returns $quantize(\gamma_{quantize(x)}(quantize(x)))$ and $quantize(\rho_{quantize(x)}(quantize(p)))$, where *quantize* is the quantization function according to the predefined grid. Such an approach results in a fairer comparison between the performances of symTest under different virtual maps, since only the quantized state and mode would affect the result of *getKey*. Such experiments are part of the plan for extending this chapter.

Calling the simulator after a cache miss results in less savings symTest simulating only the missed trajectory segment, and then continuing to check if the subsequent segments are in the cache, results in more savings than stopping and simulating the rest of the scenario after the first cache miss. This can be seen from comparing the rows with C vs. the rows with S in Tables 7.1 and 7.2. In Table 7.2, the rows with S have zero percent transformed trajectories because of the cache misses always happening in the first segment. In the same scenarios, symTest following C, achieved high percentage of transformed segments. However,

Table 7.1: Results of using symTest to execute test suite $tests_1$.

Virtual map (symmetries)	S / C	Transformed (%)	Cached (%)
1D rot.	C	21.7	18.8
1D rot.	S	5.5	19.7
3D rot.	C	17.9	20.7
3D rot.	S	1.6	21.5
3D rot. + vel. trans.	C	27.5	9.1
3D rot. + vel. trans.	S	2.9	6.9

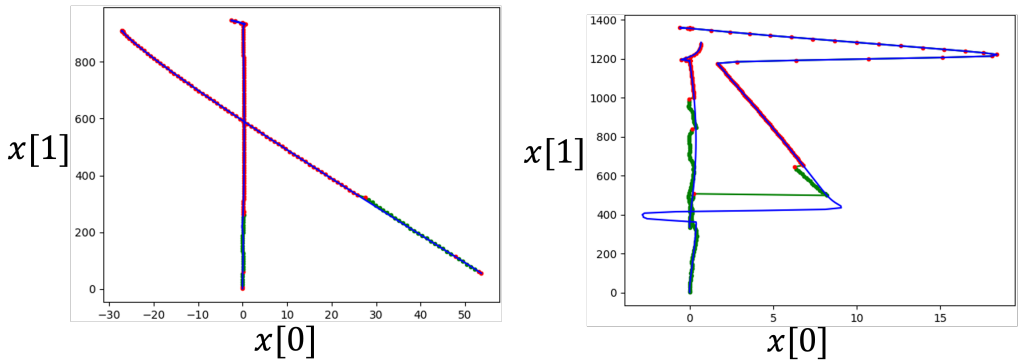
Table 7.2: Results of using symTest to execute test suite $tests_2$.

Virtual map (symmetries)	S / C	Transformed (%)	Cached (%)
1D rot.	C	26.3	64.9
1D rot.	S	0	64.9
3D rot.	C	28.8	64.9
3D rot.	S	0	64.9
1D rot. + vel. trans.	C	27.6	62.4
1D rot. + vel. trans.	S	0	54.1
3D rot. + vel. trans.	C	36.4	44.9
3D rot. + vel. trans.	S	0	45.3

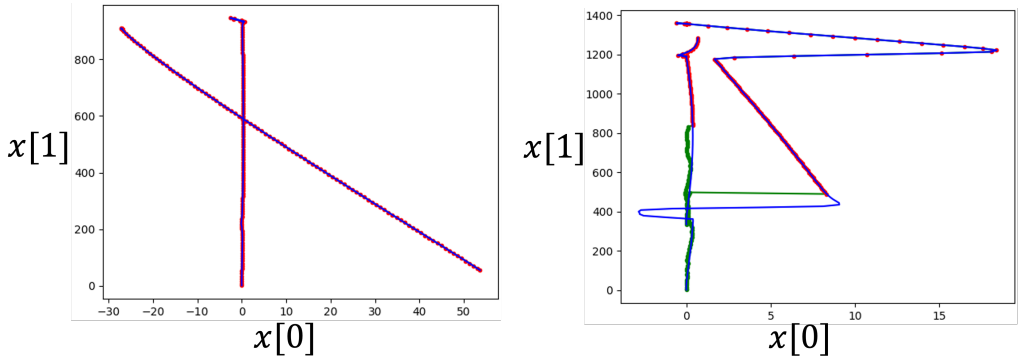
both tables do not show the testing time overhead in switching to simulation, and the potential need for relaunching Gazebo. In fact, recall that when following C, instead of simulating the missed segment, it is retrieved from the corresponding pre-generated trajectories using Gazebo. That is the reason for the horizontal jumps in the green trajectories in Figures 7.8a and 7.8b (right side). In the planned future experiments for this chapter, the segments would be simulated on-the-fly instead of pre-generated and statistics should show computational time and power consumed.

Simpler encounters result in more cached segments Comparing the last column of Tables 7.1 and 7.2 shows that in test cases with varying initial altitudes as those of $tests_2$, symTest cached a higher percent of the total segments than in test cases with equal initial altitudes as those of $tests_1$. This shows that the agents perform more uniques and less symmetric trajectories in more complicated scenarios.

absSymTest **is expected to achieve better savings than** symTest We did a preliminary evaluation of absSymTest using the 3D rot. + vel. trans. virtual map to execute test suite $tests_3$ on the NN-controlled six-dimensional quadrotor described in Appendix A.1. The trajectory of the simulated agent and the constructed trajec-

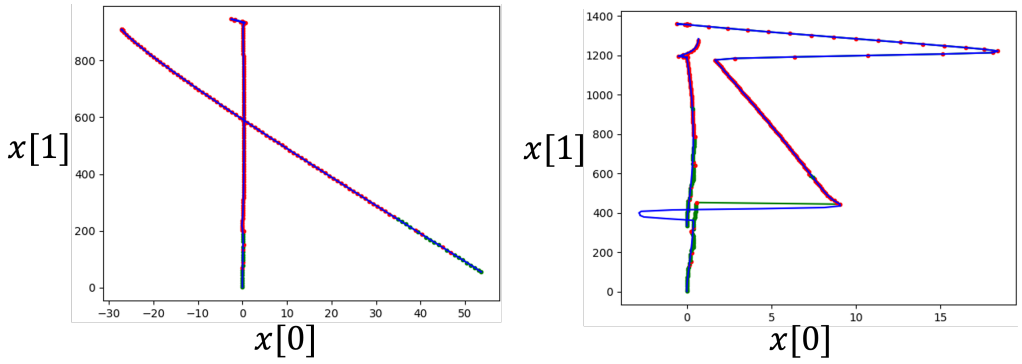


(a) Continuing after cache misses (C)

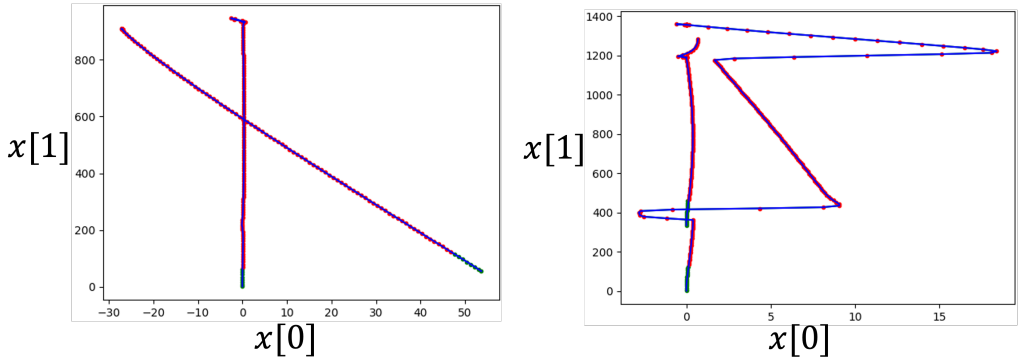


(b) Stopping after cache misses (S)

Figure 7.8: Trajectories constructed by symTest for two test cases from the test suite $tests_1$ using the virtual map 1D rot. The trajectories are projected to the first two dimensions of the state (position of the quadrotor in the plane). The test cases define two encounters: Right Beam (left) and Tail Chase (right). Right Beam: intruder coming from the east towards northwest and the ownship going straight north. Tail Chase: intruder coming from behind the ownship towards north and the ownship going north as well. Blue trajectories are the ones generated by standard testing using Gazebo without caching. Green trajectory segments represent the ones retrieved by symTest from *Cache*. Red dots represent the states at which symTest had cache misses and had to obtain the trajectory segments from the ones that were pre-generated using Gazebo, the blue ones.

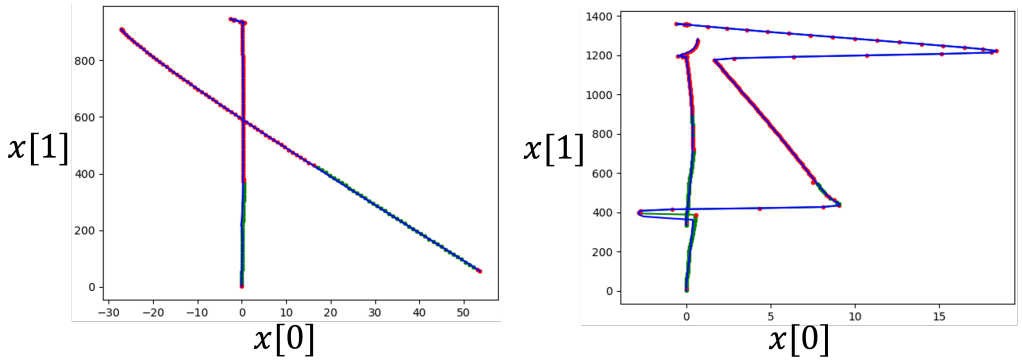


(a) Continuing after cache misses (C)

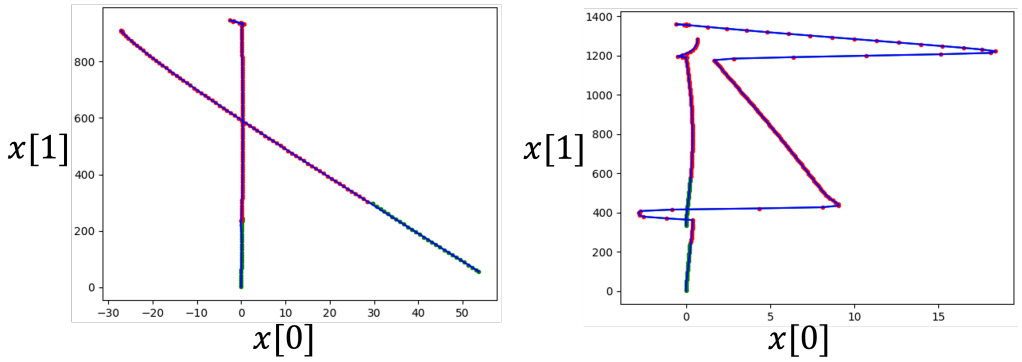


(b) Stopping after cache misses (S)

Figure 7.9: Using 3D rot. instead of 1D rot. in the experiments described in Figure 7.8.

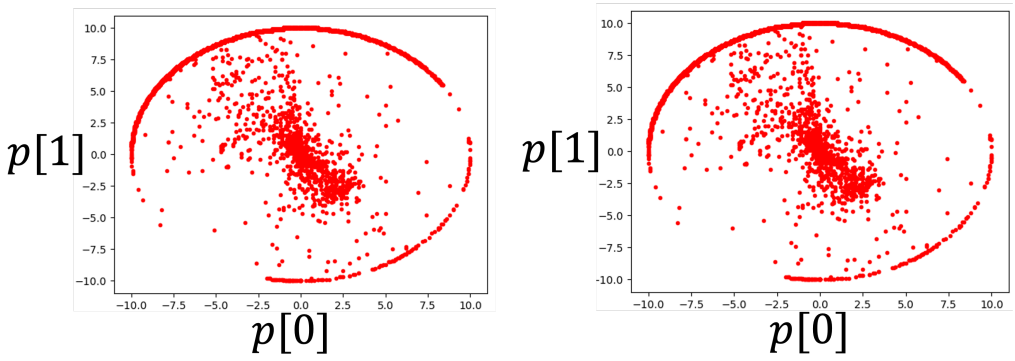


(a) Continuing after cache misses (C)

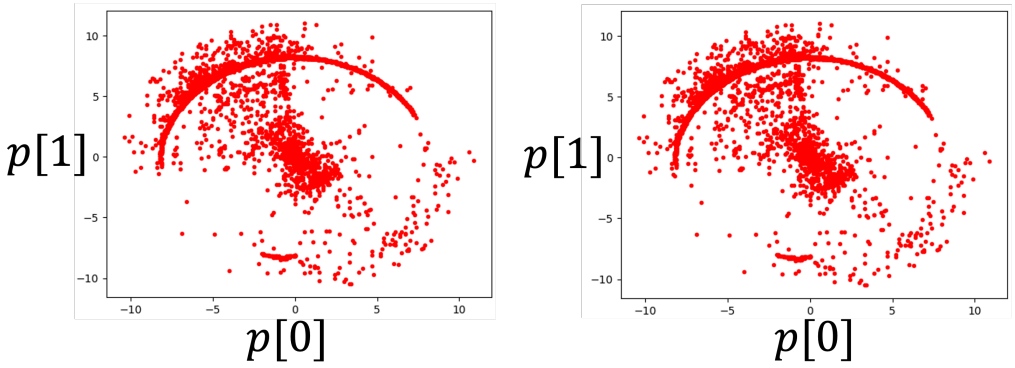


(b) Stopping after cache misses (S)

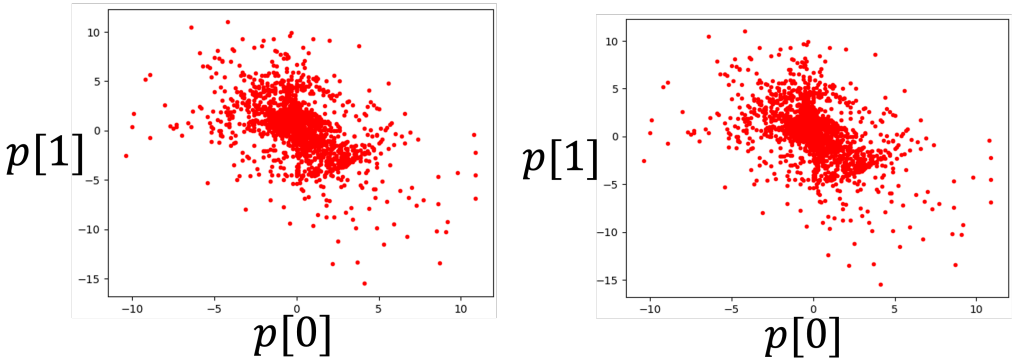
Figure 7.10: Using 3D rot. + vel. trans. instead of 1D rot. in the experiments described in Figure 7.8.



(a) 1D rot.

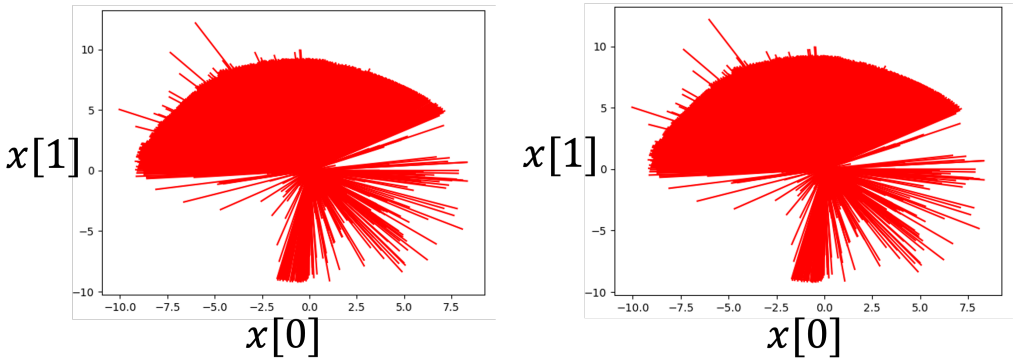


(b) 3D rot.

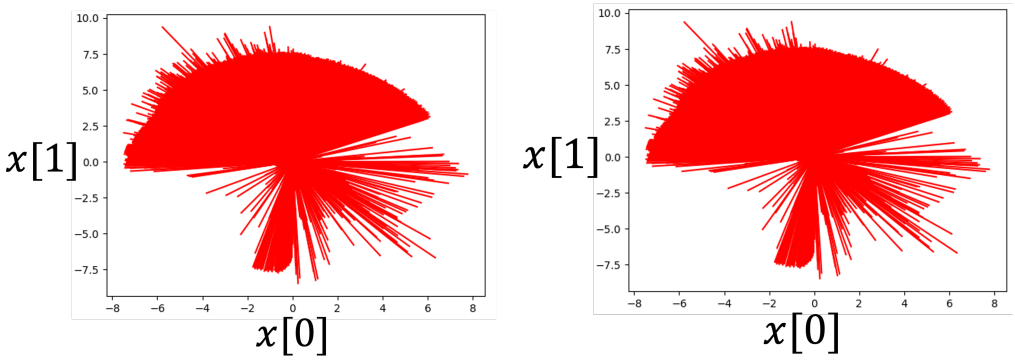


(c) 3D rot. + vel. trans.

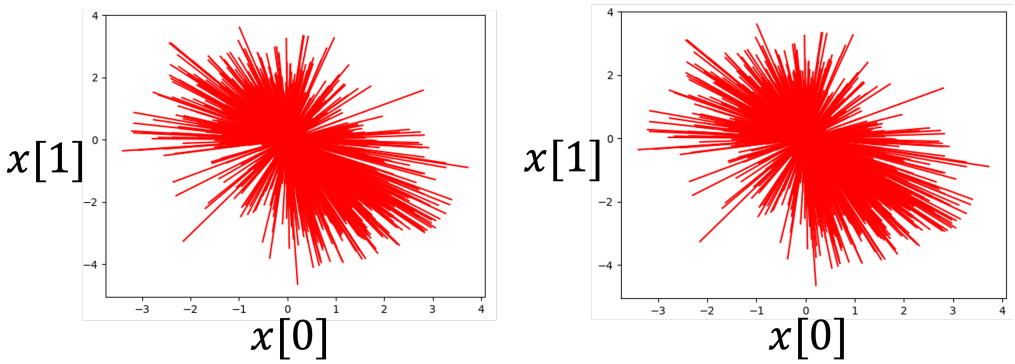
Figure 7.11: Cache keys, projected to the first two dimensions of the reference velocity vector, for which symTest did cache trajectory segments while generating the executions of the test suite $tests_1$. Continuing after cache miss (left). Stopping after cache miss (right).



(a) 1D rot.

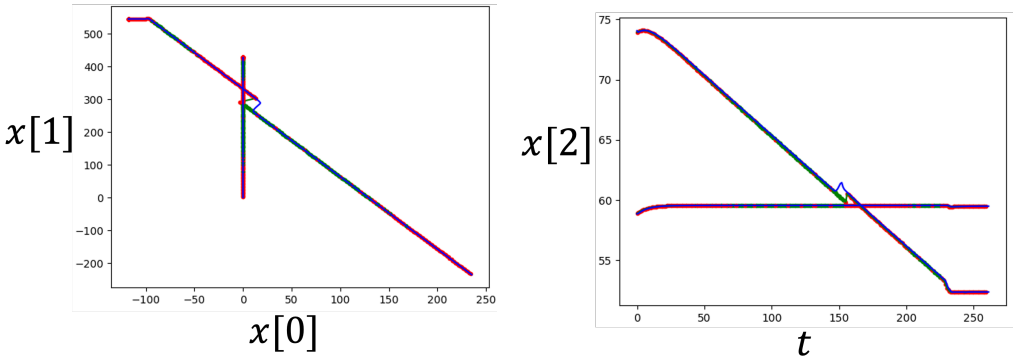


(b) 3D rot.

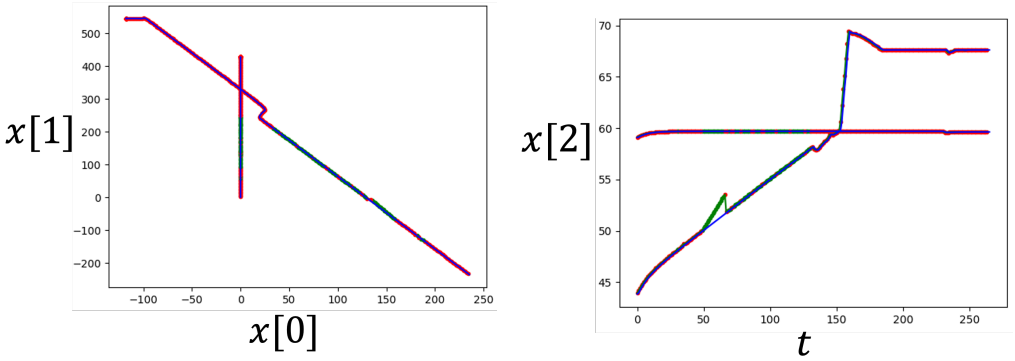


(c) 3D rot. + vel. trans.

Figure 7.12: Trajectory segments, projected to the first two dimensions, that were cached by symTest while generating the executions of the test suite $tests_1$. Continuing after cache miss (left). Stopping after cache miss (right).

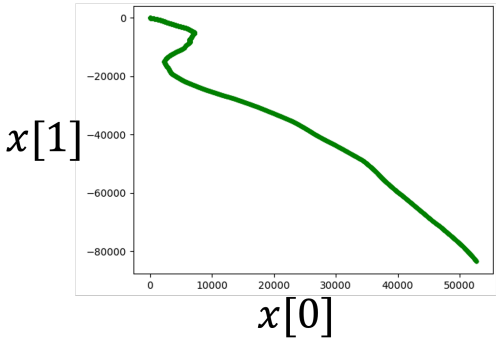


(a) Intruder starting from higher altitude than the ownship

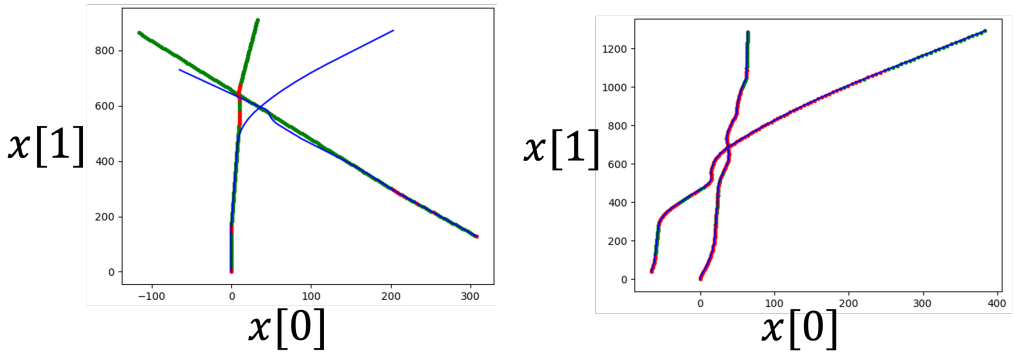


(b) Intruder starting from lower altitude than the ownship

Figure 7.13: Trajectories constructed by symTest for two test cases from the test suite $tests_2$ using the virtual map 3D rot. + vel. trans. The trajectories on the left are projected to the first two dimensions of the state (position of the quadrotor in the plane). The trajectories projected on the third dimension, the altitude of the quadrotor, with respect to time, is shown on the right. The test cases define the same encounter: Right Gaining, where the intruder is coming from the southeast towards northwest and the ownship is going straight north. However, the two test cases differ in the initial altitude of the intruder. Blue trajectories are the ones generated by standard testing using Gazebo without caching. Green trajectory segments represent the ones retrieved by symTest from *Cache*. Red dots represent the states at which symTest had cache misses and had to obtain the trajectory segments from the ones that were pre-generated using Gazebo, the blue ones.



(a) The trajectory of the reference agent, projected to the first two dimensions, constructed by absSymTest when generating the executions of the test suite $tests_3$.



(b) Trajectories constructed by absSymTest for two test cases from the test suite $tests_3$ using the virtual map 3D rot. + vel. trans. Blue trajectories are the ones generated by standard testing using Python ODEINT without caching. Green trajectory segments represent the ones retrieved by absSymTest from *Cache*. The red dots represent the states at which absSymTest had cache misses and had to simulate the trajectory segment.

Figure 7.14: Running absSymTest on test suite $tests_3$.

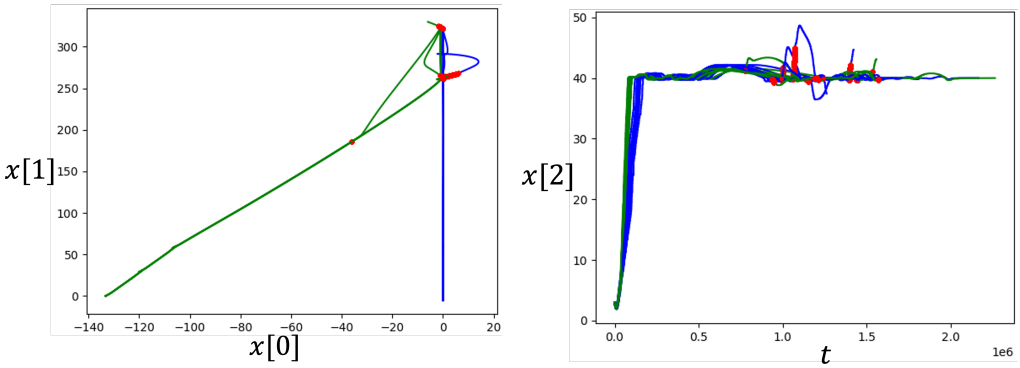
tories for the agents in two cases are shown in Figure 7.14. Simply simulating the quadrotor for the different test cases in $tests_3$ results in a total of 9300 trajectory segments by all agents. When we used absSymTest, the agent it simulated resulted in a trajectory with 3217 segments, shown in Figure 7.14a. The trajectories of the agents in all test cases of $tests_3$ can be constructed from the trajectory of the simulated agent. Thus, absSymTest achieved 65% of transformed trajectory segments and 35% of cached ones.

Although the experiment setup for symTest and absSymTest are different in this section, this preliminary experiment shows that absSymTest has the potential of achieving higher savings than symTest. This has to be further investigated in experiments with the same setup.

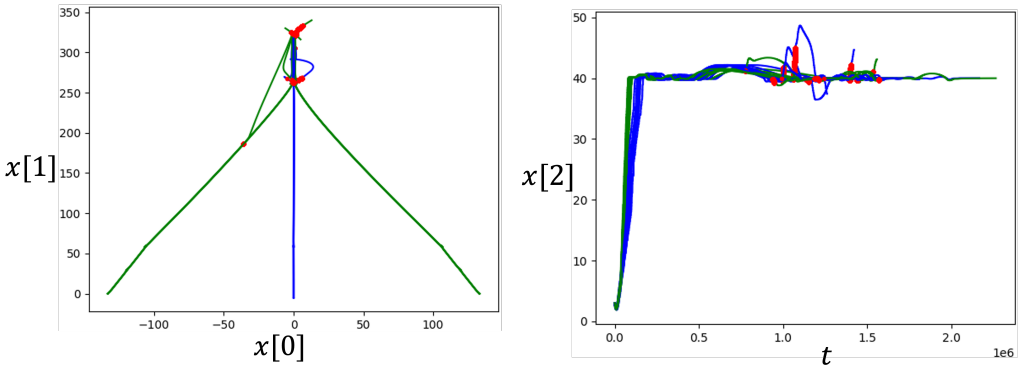
ACAS sXu satisfies several basic metamorphic relations Independent from evaluating symTest and absSymTest, we performed metamorphic testing of ACAS sXu using the constructed high-fidelity simulator. Our preliminary experiments show that ACAS sXu, in general, satisfies several metamorphic relations (or symmetric properties), such as reflection of initial states and waypoints results in reflected trajectories and advisories and increasing initial distance between agents results in fewer advisories. It is worth noting, that the non-determinism in Gazebo seems to be the main source of the discrepancy between repeated simulations of the same scenario or for symmetric scenarios.

7.6 Conclusions and future steps

We presented two algorithms for efficient testing of multi-agent autonomous systems using symmetry. We presented testing ACAS sXu in closed-loop as a case study. We built a high-fidelity simulation setup of ACAS sXu using Gazebo and ROS. We evaluated the performance of both algorithms on different test suites using different symmetries. The preliminary experiments show the potential of symmetry-based algorithms in significantly speeding up the high-fidelity testing time of autonomous systems.

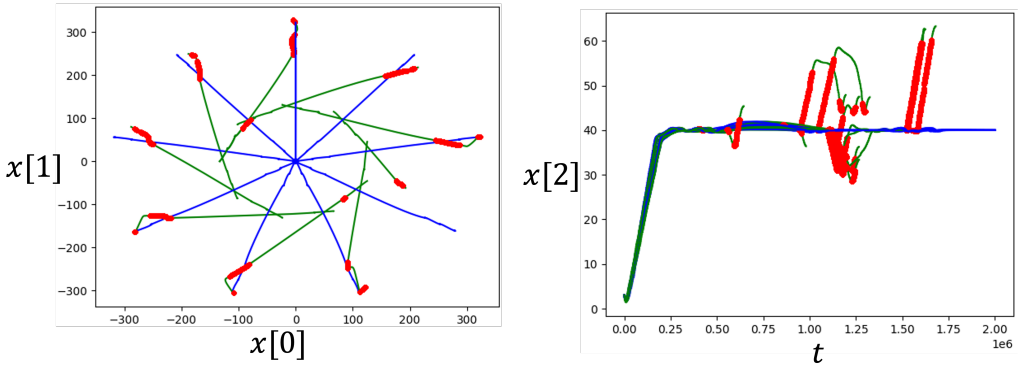


(a) Five Gazebo simulations of the same test case: Left Beam.

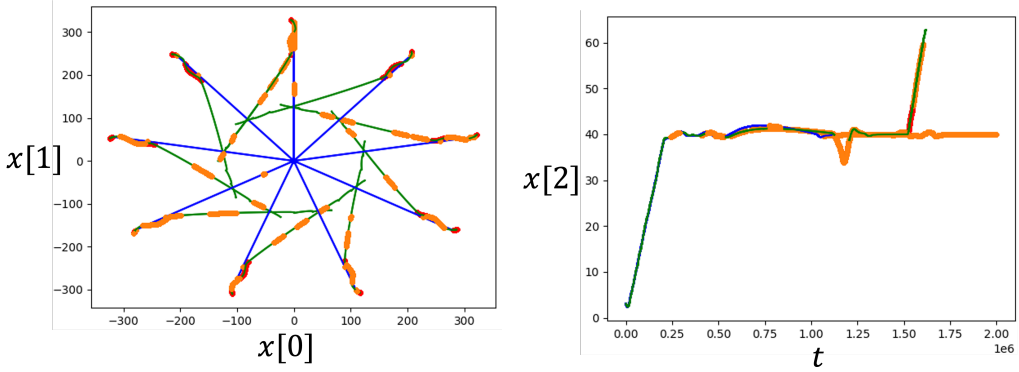


(b) Ten Gazebo simulations of reflected test cases: Left and Right Beams.

Figure 7.15: Metamorphic testing of ACAS sXu Version 3 in closed-loop with Hector Quadrotor in Gazebo. Trajectories are projected to the first two dimensions, showing the position of the quadrotor in the plane. The green and blue trajectories represent those of the intruder and the ownship, respectively. The red dots represent the states at which ACAS generated an advisory that is not clear-of-conflict.

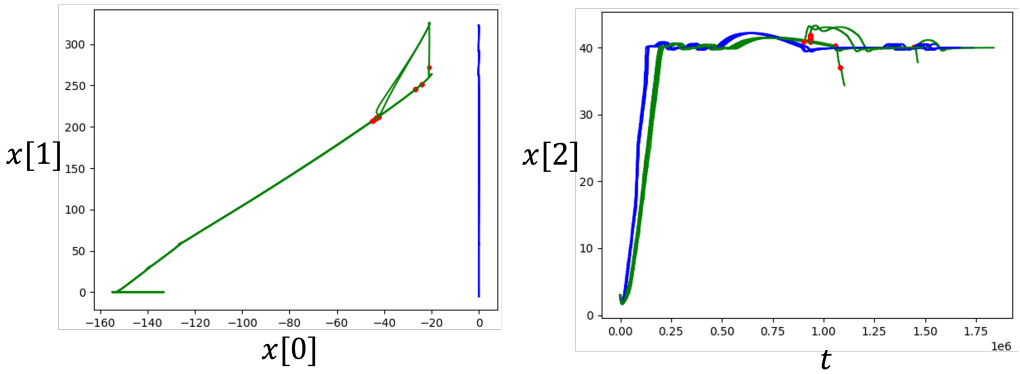


(a) Nine Gazebo simulations from initial states with rotated heading angles.

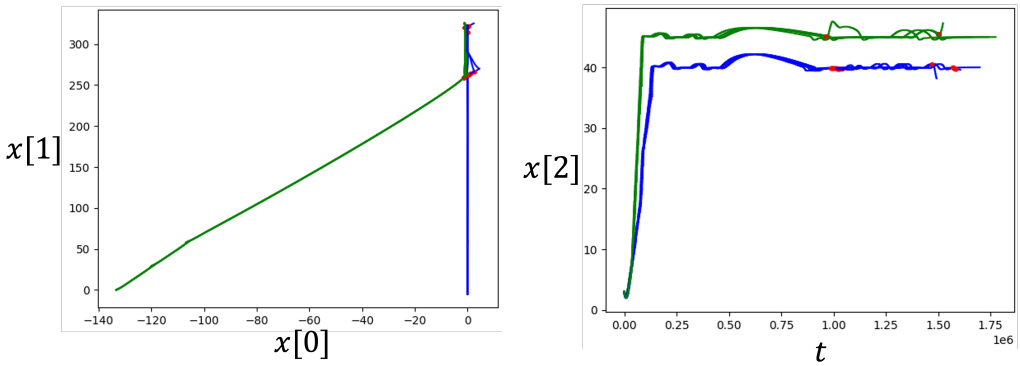


(b) The trajectories of the nine simulations in the figure above, Figure 7.16a, passed through ACAS sXu. The orange dots show the states at which ACAS sXu resulted in an advisory different from clear-of-conflict. But, of course, without the trajectories changing because of the advisories.

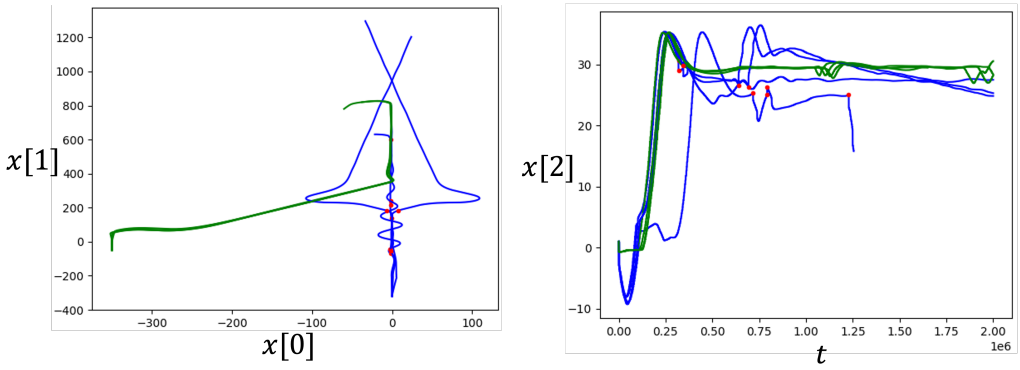
Figure 7.16: Metamorphic testing of ACAS sXu Version 3 in closed-loop with Hector Quadrotor in Gazebo. Trajectories are projected to the first two dimensions, showing the position of the quadrotor in the plane. The green and blue trajectories represent those of the intruder and the ownship, respectively. The red dots represent the states at which ACAS generated an advisory that is not clear-of-conflict. The orange dots represent the states at which ACAS generated an advisory that is not clear-of-conflict, but when the trajectories themselves are given to ACAS, instead of the agent responding to the advisories during simulation.



(a) Four Gazebo simulations of the Left Beam test case with increased initial distance between intruder and ownship.



(b) Four Gazebo simulations of the Left Beam test case with increased initial altitude difference between intruder and ownship.



(c) Four Gazebo simulations of the Left Beam test case with the ROSPlane [177].

Figure 7.17: Metamorphic testing of ACAS sXu Version 3 in closed-loop with Hector Quadrotor (first two rows) and ROSPlane (last row) in Gazebo. Trajectories are projected to the first two dimensions, showing the position of the quadrotor in the plane. The green and blue trajectories represent those of the intruder and the ownship, respectively. The red dots represent the states at which ACAS generated an advisory that is not clear-of-conflict.

Chapter 8

Conclusions and Future Research Directions

In this thesis, we present several algorithms to accelerate verification and testing of autonomous systems by exploiting the knowledge of the symmetries of the underlying dynamics. Symmetries transform trajectories of the system to other trajectories. The algorithms we propose utilize symmetry for caching computationally expensive trajectories and reachable sets and for abstracting or simplifying dynamical models. We present novel software tools that implement these algorithms. These tools utilize existing reachability analysis and simulation tools as subroutines. The experiments are mainly focused on autonomous vehicles following predefined paths. The symmetries utilized are mainly translation, rotation, and permutation, and are assumed to be provided by the users. The models of dynamical systems considered are ordinary differential equations and hybrid automata. This thesis is a first step towards utilizing symmetry for efficient formal verification and testing of cyber-physical systems.

8.1 Future research directions

There are diverse promising directions for extending the results of this thesis. We present some of these directions in this section.

Control synthesis using symmetry While the thesis focused on the analysis side, the tools presented can be extended to accelerate controller synthesis. Two novel tools have already used the results of this thesis for planning safe paths in complex environments [178, 179]. For example, CEGAR approaches for controller or program synthesis have shown promise in tackling challenging problems in the past [180]. Similarly, the symmetry-based CEGAR algorithm presented in Chapter 5, can be extended to efficiently synthesize safe plans for autonomous systems, instead of verifying existing ones. The general idea of such extension

would be to modify the refinement step to allow modifying the plan itself, rather just its abstraction.

Verification and synthesis of planners of autonomous vehicles Third, while the tools CacheReach and SceneChecker were designed for and applied to verify autonomous vehicles following a predefined plan, they have the potential to be extended to verify autonomous vehicles with on-board planners. The abstraction in Definition 5.1, would then generate a new abstract planner program that generates an abstract mode or segment instead of the concrete one, and that is easier to verify.

Reliability checking of interacting multi-agent systems the algorithm presented in Chapter 4 for symmetry utilization for multi-agent verification considers only non-interacting agents. The algorithm for efficient testing in Chapter 7 tackles the case of multi-agent scenarios that are interact periodically through a decision procedure, such as ACAS sXu. A unified framework that extends the CEGAR algorithm in Chapter 5 to interacting multi-agent scenarios would be useful. Intersection crossing of an autonomous vehicle while interacting with pedestrians and other vehicles, multiple quadrotors sharing the same airspace, or multiple aircraft landing in the same airport, are example scenarios where such a framework would be. Such an extension would benefit, in addition to symmetry, from existing partial order reduction algorithms to accelerate verification [181].

Learning symmetries of cyber-physical systems Designing algorithms for checking and learning symmetries for dynamical systems interacting with software, is an exciting future direction, that might reveal unknown symmetries. Revealing symmetries and natural laws is the main area of research in physics [25, 182]. Extending the ideas there to cyber-physical systems, would not only further enhance the performance and usability of the tools presented in this thesis, but open the doors for further collaboration between physicists and computer scientists to reveal (natural) laws of cyber-physical systems. Aside from discovering symmetries, a more approachable goal is to extend the presented algorithms to utilize approximate symmetries, those that approximately satisfy Definition 2.2. Such an extension would extend their reach to further applications. An approach would be to account for the approximate-symmetry errors as part of reachability analysis.

Verifying autonomous systems with perception modules Aside from Chapter 7, the systems considered in this thesis are closed and autonomous. Extending the results to systems with perception modules would tackle the fundamental scalability challenge facing the safety analysis of autonomous vehicles. These vehicles have sensors, such as Lidar, Radar, and camera, along with perception modules that analyze their data and feed information about the environment to the planner. There is a line of work on equivariant neural networks in computer vision [38, 35, 36, 37, 39, 41], reinforcement learning [42], and dynamics prediction [44]. A natural extension of the contributions of this thesis is a theory that join the symmetry in the dynamics with the symmetry in perception, for easier analysis, and better designs of autonomous systems.

In summary, the results presented in this thesis show the potential of symmetry in reducing computations for different safety analysis tasks. There are many directions in which this work, and extensions of it, can help tackle the scalability problem hindering the safe deployment of autonomous systems in safety-critical settings.

Appendix A

Chapter 6 discussions

A.1 NN-controlled quadrotor case study

In this section, we will describe a case study of a scenario having a planner, NN controller, and a quadrotor and model it as a hybrid automaton. We use the quadrotor model that was presented in [72] along, its trained NN controller (see Appendix A.2 on how we modify it to be rotation symmetric), and a RRT planner to construct its reference trajectories, independent of its dynamics.

The dynamics of the quadrotor are as follows:

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

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 $U \subset [-0.1, 0.1] \times [-0.1, 0.1] \times [7.81, 11.81]$ of eight possible control inputs. It was trained to mimic a model predictive control (MPC) controller to drive the quadrotor

to follow the planner trajectory. A NN is used for its faster runtime computation and reachability analysis and smaller memory requirements than traditional MPC controllers.

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

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

(a) $X = \mathbb{R}^6$, the space in which the state of the quadrotor q lives, and $\mathscr{P} = \mathbf{R}$, the space in which the graph segments live, where the first three and last three coordinates determine the start and end points $p.src$ and $p.dest$ of the segment, respectively,

(b) $\langle X_{init}, p_{init} \rangle := \langle [K, [-0.5, 0.5]^3], p_{init} \rangle$, where $[-0.5, 0.5]^3$ are the range of initial velocities of the quadrotor and p_{init} is the initial segment going out of K,

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

(d) $guard((r_i, r_{i+1}))$ is the 6D ball centered at $[r_i.dest, 0, 0, 0]$ with radius $[1, 1, 1, \infty, \infty, \infty]$, meaning that the quadrotor should arrive within distance 1 unit of the destination waypoint of the first segment, which is equivalent to the source waypoint of the second segment, at any velocity, to be able to transition to the second segment/mode,

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

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

A.2 Symmetry with non-symmetric controllers

In this section, we will specify the conditions that a controller should satisfy for a closed-loop system, with symmetric open-loop dynamics, to be symmetric as well. We then present a method to enforce the satisfaction of these symmetry conditions by the controller, by carefully transforming its inputs and outputs.

A.2.1 Symmetries of closed loop control systems

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

Fix an input space $U \subseteq \mathbb{R}^m$ and consider a right hand side of the ODE in Section 6.2 of the form:

$$f_c(x,p) := g(x,h(x,p)), \tag{A.2}$$

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

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

Definition A.1. *Given three maps $\beta : U \to U$, $\gamma : X \to X$, and $\rho : \mathscr{P} \to \mathscr{P}$. We call the control function h, (β,γ,ρ)-symmetric, if for all $x \in X$ and $p \in \mathscr{P}$, $\beta(h(x,p)) = h(\gamma(x),\rho(p))$.*

Definition A.1 means that if we transform the input of the controller, the state and the mode, using the maps γ and ρ, respectively, then its output gets transformed with the map β. Such a property formalizes intuitive assumptions about controllers in general. For example, translating the position of the quadrotor and the planned trajectory by the same vector should not change the controller output. The NN controller discussed in Appendix A.1 indeed satisfies this property since its input is the relative state $q - w$. We update the notion of equivariance for closed-loop control systems to account for the controller in the following definition.

Definition A.2. *We call the control system dynamic function f_c of equation (A.2) Γ-equivariant if for any $\gamma \in \Gamma$, there exist $\rho : \mathscr{P} \to \mathscr{P}$ and $\beta : U \to U$ such that h*

is (β, γ, ρ)*-symmetric and*

$$\forall\, x \in X, \forall\, u \in U,\ \frac{\partial \gamma}{\partial x} g(x,u) = g(\gamma(x), \beta(u)). \tag{A.3}$$

The following theorem repeats the results of Theorem 2.1 for the closed loop control system.

Theorem A.1. *If* f_c *of equation (A.2) is* Γ*-equivariant, then all maps in* Γ *are symmetries. Moreover, for any* $\gamma \in \Gamma$*, maps* $\rho : \mathscr{P} \to \mathscr{P}$ *and* $\beta : U \to U$ *that satisfy equation (A.3),* $x_0 \in X$*, and* $p \in \mathscr{P}$*,* $\gamma(\xi_c(x_0, p, \cdot)) = \xi_c(\gamma(x_0), \rho(p), \cdot)$*, where* ξ_c *is the trajectory of the dynamical system with RHS equation (A.2).*

Proof. Fix an initial state $x_0 \in X$, a mode $p \in \mathscr{P}$, and $\gamma \in \Gamma$ with its corresponding maps ρ and β that satisfy Definition 2.3 per the assumption of the theorem. For any $t \geq 0$, let $x = \xi_c(x_0, p, t)$ and $y = \gamma(x)$. Then,

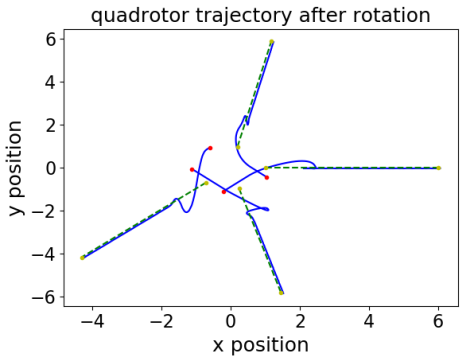
$$\begin{aligned}
\frac{dy}{dt} &= \frac{\partial \gamma}{\partial x}\frac{dx}{dt}, \text{ using the chain rule,} \\
&= \frac{\partial \gamma}{\partial x} g(x, h(x,p)), \text{ using equation (A.2),} \\
&= g(\gamma(x), \beta(h(x,p))), \text{ using equation (A.3),} \\
&= g(\gamma(x), h(\gamma(x), \rho(p))), \text{using Definition A.1,} \\
&= g(y, h(y, \rho(p))), \text{ by substituting } \gamma(x) \text{ with } y, \\
&= f_c(y, \rho(p)).
\end{aligned} \tag{A.4}$$

Hence, $\gamma(\xi_c(x_0, p, t))$ also satisfies equation (A.2) and thus a valid solution of the system. Therefore, γ is a symmetry per Definition 2.2. Moreover, y is a solution starting from $\gamma(x_0)$ in mode $\rho(p)$. This proof is similar to that of Theorem 10 in [26] with is the difference of having a controller h, which requires the additional assumption that h is symmetric. □

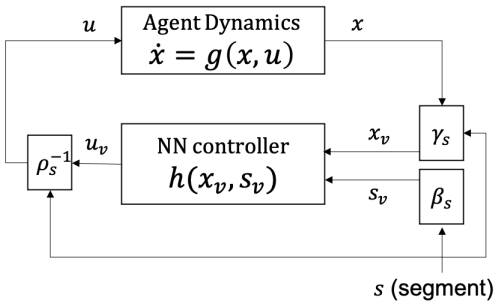
In Appendix A.2.2, we discuss how to make non-symmetric controllers symmetric, and apply that to the NN-controller of the quadrotor to make it rotation symmetric.

A.2.2 From non-symmetric controllers to symmetric ones

In some cases, the controller h is not symmetric. For example, the NN controller of the quadrotor in Section 6.7 and Appendix A.1 is not symmetric with respect to rotations in the xy-plane. We show a counter example in Figure A.1a.



(A.1a) NN controller of the quadrotor is not rotation symmetric.



(A.1b) Patching the controller to make it symmetric.

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

Such non-symmetric controllers will prevent the dynamics from being equivariant. Equivariance is a desirable, and expected, property of certain dynamical systems. For example, vehicles dynamics are expected to be translation and rotation invariant in the xy-plane. Thus, non-symmetric controllers violate intuition about systems dynamics. Such controllers may not be feasible to abstract using Definition 5.1. Next, we will suggest a way to make any controller, including NN ones, such as that of the quadrotor, symmetric, leading to better controllers and retrieving the ability to construct abstractions.

Consider again the closed loop control system of equation (A.2). Let $\Phi = \{(\gamma_p, \rho_p)\}_{p\in\mathscr{P}}$ be a set of maps that we want it to be a virtual map of system (A.2). As before let us define $rv : \mathscr{P} \to \mathscr{P}$ by $rv(p) = \rho_p(p)$, for all $p \in \mathscr{P}$. Assume that for every $p \in \mathscr{P}$, there exists $\beta_p : U \to U$, such that the open loop dynamic function g is symmetric in the sense that it satisfies equation (A.3). Moreover, assume that $\gamma_{rv(p)}, \beta_{rv(p)}$, and $\rho_{rv(p)}$ are identity maps for any $p \in \mathscr{P}$. This assumption means

that applying the same symmetry transformation twice would not change the state nor the mode. Now, let us define a new controller $h' : X \times \mathscr{P} \to U$, that is shown in Figure A.1b, as follows:

$$h'(x,p) = \beta_p^{-1}(h(\gamma_p(x), rv(p))). \tag{A.5}$$

Theorem A.2. *For any $p \in \mathscr{P}$, the controller h' is (β_p, γ_p, rv)-symmetric.*

Proof. Fix $x \in X$ and $p \in \mathscr{P}$. Then, $\beta_p(h'(x,p))$

$$\begin{aligned}
&= \beta_p(\beta_p^{-1}(h(\gamma_p(x), rv(p)))), \text{ using equation (A.5)},\\
&= h(\gamma_p(x), rv(p)), \text{ since } \beta_p\beta_p^{-1} \text{ is the identity map},\\
&= \beta_{rv(p)}^{-1} h(\gamma_{rv(p)}(\gamma_p(x)), \rho_{rv(p)}(rv(p))),\\
&\text{ [since } \gamma_{rv(p)} \text{ and } \beta_{rv(p)} \text{ are identity maps, } \rho_{rv(p)}(rv(p)) = rv(p)],\\
&= h'(\gamma_p(x), rv(p)), \qquad (A.6)
\end{aligned}$$

where the last equality follows from using equation (A.5) again. □

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

A.3 Trying other NN-controlled systems' verification tools as reachability subroutines

1. The state-of-the-art verification tool for NN-controlled systems Verisig needs up to 30 minutes to compute the reachsets for four segments in a quadrotor scenario [72]. We tried Verisig and it took similar or longer amount of time for scenarios with less than five segments. We decided to use DryVR for faster evaluation of SceneChecker in this paper.

2. We tried NNV [19], however the resulting reachsets had large over-approximation errors in our scenarios to the point of being not useful. We contacted its developers and they are working on the conservativeness problem.

3. We considered using the tool of "Reachability analysis for neural feedback systems using regressive polynomial rule inference", by Dutta et. al. [183]. We were unable to implement our scenarios in manageable time given that there is no manual to use the tool.

A.4 Car NN controller

The controller is trained to drive the car to follow a straight reference trajectory aligned with positive y-direction towards the origin. The controller follows method explained in Section A.2 and any given reference segment to be aligned with the y-axis and compute corresponding control inputs.

The inputs to the controller are the relative position of the car in the plane with respect to the reference position in the segment being followed and the cosine and sine of the relative orientation of the car with respect to the orientation of the segment being followed. The outputs from the controller are the speed v of the vehicle and the steering angle δ of the vehicle. The neural network have one hidden layer with 100 hidden neurons.

The training data for this controller is obtained by random sampling positions in range $x \in$[-3,3] and $y \in [-2.5, 1.5]$ and orientation in range $\theta \in [0, \pi]$. During the training process, the vehicle will start from the sampled position and orientation and drive towards the origin [x,y,θ] = [0,0,$\frac{\pi}{2}$] which is the reference state.

A.5 Long segments vs Short segments without using symmetry

To further utilize the symmetry property of the scenario, we will split mode that contains longer segments into several modes that contains shorter segments with equal length. In table A.1 shows the statistics for verifying the comp2D-1 scenario for quadrotor without using symmetry. For experiment run comp2D-1-S, modes that contain segments with length larger than three is split into mode that contain segments with length smaller than or equal to three. For experiment run comp2D-1, the modes are not split. The experiment is to compare the effect of splitting longer segments into short segments while not using symmetry. From the collected data, we can see that although number of modes and number calls to reachset

computer increase, the amount of time it takes for verification of the scenarios is not influenced. Therefore, for the rest of the experiments, while not using symmetry, we are verifying scenarios without splitting modes.

Table A.1: Results showing that without using symmetry, splitting modes with longer segments to modes with shorter ones won't influence performance.

Experiment run	complex2D-1	complex2D-1-S
# original modes	48	140
# virtual modes	47	139
# computed tubes	96	280
# segs computed	48	140
time	37.12	36.45

Bibliography

[1] Federal Aviation Administration, "FAA Aerospace Forecast Fiscal Year 2020-2040," 2020. [Online]. Available: https://www.faa.gov/data_research/aviation/aerospace_forecasts/media/FY2020-40_FAA_Aerospace_Forecast.pdf

[2] 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, 2004, pp. 2149–2154 vol.3.

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," 2017.

[4] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "Lgsvl simulator: A high fidelity simulator for autonomous driving," 2020.

[5] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *FSR*, 2017.

[6] J. K. Haas, "A history of the unity game engine," 2014.

[7] "Waymo safety report. https://waymo.com/safety/safety-report." [Online]. Available: https://waymo.com/safety/safety-report

[8] D. J. Fremont, E. Kim, Y. V. Pant, S. A. Seshia, A. Acharya, X. Bruso, P. Wells, S. Lemke, Q. Lu, and S. Mehta, "Formal scenario-based testing of autonomous vehicles: From simulation to the real world," 2020.

[9] Federal Aviation Administration, "UTM Pilot Program (UPP) Phase 2 Final Report," Oct. 2021. [Online]. Available: https://www.faa.gov/uas/research_development/traffic_management/utm_pilot_program/media/FY20_UPP2_Final_Report.pdf

[10] T. Akazaki, S. Liu, Y. Yamagata, Y. Duan, and J. Hao, "Falsification of cyber-physical systems using deep reinforcement learning," *Lecture Notes in Computer Science*, p. 456–465, 2018. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-95582-7_27

[11] 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

[12] H. Abbas, G. Fainekos, S. Sankaranarayanan, F. Ivančić, and A. Gupta, "Probabilistic temporal logic falsification of cyber-physical systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, May 2013. [Online]. Available: https://doi.org/10.1145/2465787.2465797

[13] A. Zutshi, J. V. Deshmukh, S. Sankaranarayanan, and J. Kapinski, "Multiple shooting, cegar-based falsification for hybrid systems," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2656045.2656061

[14] P. Du, Z. Huang, T. Liu, T. Ji, K. Xu, Q. Gao, H. Sibai, K. Driggs-Campbell, and S. Mitra, "Online monitoring for safe pedestrian-vehicle interactions," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, 2020, pp. 1–8.

[15] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "Data-driven verification and compositional reasoning for automotive systems," in *Computer Aided Verification*. Springer International Publishing, 2017, pp. 441–461.

[16] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2e2: A verification tool for stateflow models," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. Baier and C. Tinelli, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 68–82.

[17] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds. Springer Berlin Heidelberg, 2013, vol. 8044, pp. 258–263.

[18] A. Donzé, "Breach, a toolbox for verification and parameter synthesis of hybrid systems," in *Computer Aided Verification (CAV 2010)*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6174, pp. 167–170.

[19] H.-D. Tran, X. Yang, D. Manzanas Lopez, P. Musau, L. V. Nguyen, W. Xiang, S. Bak, and T. T. Johnson, "Nnv: The neural network verification tool for deep neural networks and learning-enabled cyber-physical systems," in *CAV*, S. K. Lahiri and C. Wang, Eds., 2020.

[20] 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*, ser. HSCC '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3302504.3311806 pp. 169–178.

[21] S. Bak, H.-D. Tran, and T. T. Johnson, "Numerical verification of affine systems with up to a billion dimensions," 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/10.1145/3302504.3311792 p. 23–32.

[22] S. Bogomolov, G. Frehse, M. Greitschus, R. Grosu, C. Pasareanu, A. Podelski, and T. Strump, "Assume-guarantee abstraction refinement meets hybrid systems," in *Hardware and Software: Verification and Testing*, E. Yahav, Ed. Cham: Springer International Publishing, 2014, pp. 116–131.

[23] G. Frehse, "Compositional verification of hybrid systems using simulation relations," Ph.D. dissertation, Radbout Universiteit Nijmegen, October 2005.

[24] D. Liberzon, "On topological entropy of interconnected nonlinear systems," *IEEE Control Systems Letters*, vol. 5, no. 6, pp. 2210–2214, 2021.

[25] P. W. Anderson, "More is different," *Science*, vol. 177, no. 4047, pp. 393–396, 1972.

[26] G. Russo and J.-J. E. Slotine, "Symmetries, stability, and control in nonlinear systems and networks," *Physical Review E*, vol. 84, no. 4, p. 041929, 2011.

[27] M. Golubitsky and I. Stewart, *The Symmetry Perspective: From Equilibrium to Chaos in Phase Space and Physical Space*, ser. Progress in Mathematics. Birkhäuser Basel, 2012.

[28] J. Hanc, S. Tuleja, and M. Hancova, "Symmetries and conservation laws: Consequences of noether's theorem," *American Journal of Physics - AMER J PHYS*, vol. 72, pp. 428–435, 04 2004.

[29] E. Noether, "Invarianten beliebiger differentialausdrücke," *Nachrichten von der Gesellschaft der Wissenschaften zu Göttingen, Mathematisch-Physikalische Klasse*, vol. 1918, pp. 37–44, 1918. [Online]. Available: http://eudml.org/doc/59011

[30] P. G. Mehta, G. Hagen, and A. Banaszuk, "Symmetry and symmetry-breaking for a wave equation with feedback," *SIAM Journal on Applied Dynamical Systems*, vol. 6, no. 3, pp. 549–575, 2007.

[31] M. W. Spong and F. Bullo, "Controlled symmetries and passive walking," *IEEE Transactions on Automatic Control*, vol. 50, no. 7, pp. 1025–1031, 2005.

[32] Q.-C. Pham and J.-J. Slotine, "Stable concurrent synchronization in dynamic system networks," *Neural Netw.*, vol. 20, no. 1, pp. 62–77, Jan. 2007.

[33] L. Gérard and J.-J. Slotine, "Neuronal networks and controlled symmetries, a generic framework," *arXiv preprint q-bio/0612049*, 2006.

[34] Y. S. Abu-Mostafa, "Learning from hints in neural networks," *Journal of Complexity*, vol. 6, no. 2, pp. 192 – 198, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0885064X9090006Y

[35] T. S. Cohen and M. Welling, "Group equivariant convolutional networks," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, p. 2990?2999.

[36] F. Fuchs, D. Worrall, V. Fischer, and M. Welling, "Se(3)-transformers: 3d roto-translation equivariant attention networks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/15231a7ce4ba789d13b722cc5c955834-Paper.pdf pp. 1970–1981.

[37] N. Thomas, T. Smidt, S. Kearnes, L. Yang, L. Li, K. Kohlhoff, and P. Riley, "Tensor field networks: Rotation- and translation-equivariant neural networks for 3d point clouds," 2018.

[38] T. S. Cohen, M. Geiger, and M. Weiler, "A general theory of equivariant cnns on homogeneous spaces," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. Alche-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/b9cfe8b6042cf759dc4c0cccb27a6737-Paper.pdf

[39] T. S. Cohen, M. Geiger, J. Kohler, and M. Welling, "Spherical CNNs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Hkbd5xZRb

[40] R. Gens and P. M. Domingos, "Deep symmetry networks," in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014. [Online]. Available: https://proceedings.neurips.cc/paper/2014/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf

[41] N. Keriven and G. Peyré, "Universal invariant and equivariant graph neural networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/ea9268cb43f55d1d12380fb6ea5bf572-Paper.pdf

[42] E. van der Pol, D. Worrall, H. van Hoof, F. Oliehoek, and M. Welling, "Mdp homomorphic networks: Group symmetries in reinforcement learning," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/2be5f9c2e3620eb73c2972d7552b6cb5-Paper.pdf pp. 4199–4210.

[43] F. Amadio, A. Colomé, and C. Torras, "Exploiting symmetries in reinforcement learning of bimanual robotic tasks," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1838–1845, 2019.

[44] N. Guttenberg, N. Virgo, O. Witkowski, H. Aoki, and R. Kanai, "Permutation-equivariant neural networks applied to dynamics prediction," *CoRR*, vol. abs/1612.04530, 2016. [Online]. Available: http://arxiv.org/abs/1612.04530

[45] K. Ganju, Q. Wang, W. Yang, C. A. Gunter, and N. Borisov, "Property inference attacks on fully connected neural networks using permutation invariant representations," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243834 p. 619–633.

[46] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, p. 61–67, Feb. 2019. [Online]. Available: https://doi.org/10.1145/3241979

[47] T.-C. Lin, Z. Hui, S. Huang, Z. Ren, and Y. Yao, "Metamorphic testing integer overflow faults of mission critical program: A case study," *Mathematical Problems in Engineering*, vol. 2013, p. 381389, 2013. [Online]. Available: https://doi.org/10.1155/2013/381389

[48] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3238147.3238187 p. 132–142.

[49] K. Pei, S. Wang, Y. Tian, J. Whitehouse, C. Vondrick, Y. Cao, B. Ray, S. Jana, and J. Yang, "Bringing engineering rigor to deep learning," *SIGOPS Oper. Syst. Rev.*, vol. 53, no. 1, p. 59–67, July 2019. [Online]. Available: https://doi.org/10.1145/3352020.3352030

[50] C. Robert, J. Guiochet, and H. Waeselynck, "Testing a non-deterministic robot in simulation - how many repeated runs ?" in *2020 Fourth IEEE International Conference on Robotic Computing (IRC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IRC.2020.00048 pp. 263–270.

[51] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, "Metamorphic model-based testing of autonomous systems," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, 2017, pp. 35–41.

[52] 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.

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

[54] 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.

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

[56] 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.

[57] 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.

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

[59] 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.

[60] 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

[61] T. A. Henzinger, "The theory of hybrid automata," in *11th Annual IEEE Symposium on Logic in Computer Science*, 1996, pp. 278–292.

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

[63] N. Roohi, Y. Wang, M. West, G. Dullerud, and M. Viswanathan, "Statistical verification of Toyota powertrain control verification benchmark," in *Proceedings of the 20th international conference on Hybrid systems: computation and control*. ACM, 2017.

[64] P. S. Duggirala, C. Fan, S. Mitra, and M. Viswanathan, "Meeting a powertrain verification challenge," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_37 pp. 536–543.

[65] 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. [Online]. Available: https://doi.org/10.1007/978-3-319-41528-4_29 pp. 531–538.

[66] P. S. Duggirala and M. Viswanathan, "Parsimonious, simulation based verification of linear systems," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, 2016. [Online]. Available: https://doi.org/10.1007/978-3-319-41528-4_26 pp. 477–494.

[67] 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*. ACM, 2017, pp. 173–178.

[68] C. Fan, J. Kapinski, X. Jin, and S. Mitra, "Locally optimal reach set over-approximation for nonlinear systems," in *EMSOFT*. ACM, 2016, pp. 6:1–6:10.

[69] C. Fan and S. Mitra, "Bounded verification with on-the-fly discrepancy computation," in *ATVA*, ser. Lecture Notes in Computer Science, vol. 9364. Springer, 2015, pp. 446–463.

[70] X. Chen, "Reachability analysis of non-linear hybrid systems using taylor models," 2015.

[71] 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 *Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 379–395.

[72] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: verifying safety properties of hybrid systems with neural network controllers," in *ACM HSCC*, 2019.

[73] S. Dutta, X. Chen, S. Jha, S. Sankaranarayanan, and A. Tiwari, "Sherlock - a tool for verification of neural network feedback systems: Demo abstract," ser. HSCC '19. New York, NY, USA: ACM, 2019. [Online]. Available: https://doi.org/10.1145/3302504.3313351 p. 262–263.

[74] M. Althoff and J. M. Dolan, "Online verification of automated road vehicles using reachability analysis," *IEEE Trans. Robotics*, vol. 30, no. 4, pp. 903–918, 2014. [Online]. Available: https://doi.org/10.1109/TRO.2014.2312453

[75] T. Kushner, B. W. Bequette, F. Cameron, G. P. Forlenza, D. M. Maahs, and S. Sankaranarayanan, "Models, devices, properties, and verification of artificial pancreas systems," in *Automated Reasoning for Systems Biology and Medicine*, 2019, pp. 93–131. [Online]. Available: https://doi.org/10.1007/978-3-030-17297-8_4

[76] C. Fan, B. Qi, and S. Mitra, "Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features," *IEEE Design & Test*, vol. 35, no. 3, pp. 31–38, 2018. [Online]. Available: https://doi.org/10.1109/MDAT.2018.2799804

[77] J. Maidens and M. Arcak, "Exploiting symmetry for discrete-time reachability computations," 2018.

[78] A. Irfan, K. D. Julian, H. Wu, C. Barrett, M. J. Kochenderfer, B. Meng, and J. Lopez, "Towards verification of neural networks for small unmanned aircraft collision avoidance," in *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, 2020, pp. 1–10.

[79] 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

[80] 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.

[81] 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 - 2008. - T. 1*. Piscataway, NJ: IEEE, 2008. [Online]. Available: https://publications.rwth-aachen.de/record/100535 pp. 233–238.

[82] A. Donzé and O. Maler, "Systematic simulation using sensitivity analysis," in *Hybrid Systems: Computation and Control*. Springer, 2007, pp. 174–189.

[83] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *EMSOFT*, 2013.

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

[85] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2e2: A verification tool for stateflow models," in *Proceedings of the 21st TACAS - Volume 9035*. Berlin, Heidelberg: Springer-Verlag, 2015. [Online]. Available: https://doi.org/10.1007/978-3-662-46681-0_5 p. 68–82.

[86] P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," *SAE International Journal of Transportation Safety*, vol. 4, no. 2016-01-0128, pp. 15–24, 2016.

[87] S. Mitra, *Verifying Cyber-Physical Systems: A Path to Safe Autonomy*, ser. Cyber Physical Systems Series. MIT Press, 2021. [Online]. Available: https://books.google.com/books?id=4y16zQEACAAJ

[88] R. Alur, *Principles of Cyber-Physical Systems*. The MIT Press, 2015.

[89] 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.

[90] L. Doyen, T. A. Henzinger, and J.-F. Raskin, "Automatic rectangular refinement of affine hybrid systems," in *Proceedings of the Third International Conference on Formal Modeling and Analysis of Timed Systems*, ser. FORMATS'05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: https://doi.org/10.1007/11603009_13 p. 144–161.

[91] S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke, "Reachability for linear hybrid automata using iterative relaxation abstraction," in *Proceedings of the 10th International Conference on Hybrid Systems: Computation and Control*, ser. HSCC'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 287–300.

[92] N. Roohi, P. Prabhakar, and M. Viswanathan, "HARE: A hybrid abstraction refinement engine for verifying non-linear hybrid automata," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, 2017. [Online]. Available: https://doi.org/10.1007/978-3-662-54577-5_33 pp. 573–588.

[93] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005, also available as Technical Report MIT-LCS-TR-917, MIT.

[94] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA 02139, September 2007. [Online]. Available: http://people.csail.mit.edu/mitras/research/thesis.pdf

[95] A. Girard, A. A. Julius, and G. J. Pappas, "Approximate simulation relations for hybrid systems," in *IFAC Analysis and Design of Hybrid Systems*, Alghero, Italy, June 2006.

[96] P. Koopman and M. Wagner, "Autonomous vehicle safety: An interdisciplinary challenge," *IEEE Intelligent Transportation Systems Magazine*, vol. 9, no. 1, pp. 90–96, 2017.

[97] L. E. Alvarez, I. Jessen, M. P. Owen, J. Silbermann, and P. Wood, "Acas sxu: Robust decentralized detect and avoid for small unmanned aircraft systems," in *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, 2019, pp. 1–9.

[98] J. Meyer, A. Sendobry, S. Kohlbrecher, U. Klingauf, and O. von Stryk, "Comprehensive simulation of quadrotor uavs using ros and gazebo," in *Simulation, Modeling, and Programming for Autonomous Robots*, I. Noda, N. Ando, D. Brugali, and J. J. Kuffner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 400–411.

[99] Stanford Artificial Intelligence Laboratory et al., "Robotic operating system." [Online]. Available: https://www.ros.org

[100] E. A. Coddington and N. Levinson, *Theory of ordinary differential equations [by] Earl A. Coddington [and] Norman Levinson*. McGraw-Hill New York, 1955.

[101] J. E. Marsden and T. S. Ratiu, *Introduction to Mechanics and Symmetry: A Basic Exposition of Classical Mechanical Systems*. Springer Publishing Company, Incorporated, 2010.

[102] P. J. Olver, *Applications of Lie Groups to Differential Equations*. Springer-Verlag New York, 1986.

[103] P. G. O. Freund, *Introduction to Supersymmetry*, ser. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 1986.

[104] E. N. Lorenz, "Deterministic nonperiodic flow," *Journal of Atmospheric Sciences*, vol. 20, no. 2, pp. 130 – 141, 1963. [Online]. Available: https://journals.ametsoc.org/view/journals/atsc/20/2/1520-0469_1963_020_0130_dnf_2_0_co_2.xml

[105] 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.

[106] R. Alur, C. C. T. A. Henzinger, and P. H. Ho., "Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems," in *Hybrid Systems*, ser. LNCS, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, Eds., vol. 736. Springer-Verlag, 1993, pp. 209–229.

[107] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[108] S. Blazic, "A novel trajectory-tracking control law for wheeled mobile robots," *Robotics and Autonomous Systems*, vol. 59, no. 11, pp. 1001 – 1007, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0921889011001023

[109] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" in *ACM Symposium on Theory of Computing*, 1995. [Online]. Available: citeseer.nj.nec.com/henzinger95whats.html pp. 373–382.

[110] N. Nedialkov, "VNODE-LP: Validated solutions for initial value problem for ODEs," McMaster University, Tech. Rep., 2006.

[111] "Computer assisted proofs in dynamic groups (capd)." [Online]. Available: http://capd.sourceforge.net/capdDynSys/docs/html/odes_rigorous.html

[112] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94 – 124, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000098915811

[113] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle, "Formal analysis of timing effects on closed-loop properties of control software," in *2014 IEEE Real-Time Systems Symposium*, Dec 2014, pp. 53–62.

[114] A. Devonport, M. Khaled, M. Arcak, and M. Zamani, "Pirk: Scalable interval reachability analysis for high-dimensional nonlinear systems," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 556–568.

[115] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "Dryvr: Data-driven verification and compositional reasoning for automotive systems," in *CAV*, R. Majumdar and V. Kunčak, Eds., 2017.

[116] M. Althoff, D. Grebenyuk, and N. Kochdumper, "Implementation of taylor models in cora 2018," in *Proc. of the 5th International Workshop on Applied Verification for Continuous and Hybrid Systems*, 2018.

[117] 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*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302504.3311804 p. 39–44.

[118] S. Bak, "nnenum: Verification of relu neural networks with optimized abstraction refinement," in *NASA Formal Methods Symposium*. Springer, 2021, pp. 19–36.

[119] L. Bu, A. Abate, D. Adzkiya, M. S. Mufid, R. Ray, Y. Wu, and E. Zaffanella, "Arch-comp20 category report: Hybrid systems with piecewise constant dynamics and bounded model checking," in *ARCH20. 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20)*, ser. EPiC Series in Computing, G. Frehse and M. Althoff, Eds., vol. 74. EasyChair, 2020. [Online]. Available: https://easychair.org/publications/paper/9xtZ pp. 1–15.

[120] Y. Deng, A. Rajhans, and A. A. Julius, "Strong: A trajectory-based verification toolbox for hybrid systems," in *QEST*, 2013.

[121] Z. Huang, C. Fan, A. Mereacre, S. Mitra, and M. Z. Kwiatkowska, "Invariant verification of nonlinear hybrid automata networks of cardiac cells," in *CAV*, ser. Lecture Notes in Computer Science, vol. 8559. Springer, 2014, pp. 373–390.

[122] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based cegar for rectangular hybrid systems," *Formal Methods in System Design*, vol. 46, no. 2, pp. 105–134, Apr 2015.

[123] Y. Annapureddy, C. Liu, G. Fainekos, and S. Sankaranarayanan, "S-TaLiRo: A tool for temporal logic falsification for hybrid systems," in *TACAS*, ser. Lecture Notes in Computer Science, vol. 6605. Springer, 2011, pp. 254–257.

[124] M. Golubitsky, I. Stewart, and A. Török, "Patterns of synchrony in coupled cell networks with multiple arrows," *SIAM Journal on Applied Dynamical Systems*, vol. 4, no. 1, pp. 78–100, 2005.

[125] T. Johnson and S. Mitra, "A small model theorem for rectangular hybrid automata networks," 2012.

[126] 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*, 2019, pp. 1–17.

[127] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds. Springer Berlin Heidelberg, 2013, vol. 8044, pp. 258–263.

[128] H. Sibai, N. Mokhlesi, C. Fan, and S. Mitra, "Cachereach: multi-agent safety verification using symmetry transformations software tool," 2020.

[129] A. Hartmanns and M. Seidl, "tacas20ae.ova. figshare," 2019.

[130] 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.

[131] R. Goebel, R. G. Sanfelice, and A. R. Teel, *Hybrid Dynamical Systems: Modeling, Stability, and Robustness*. Princeton University Press, 2012. [Online]. Available: http://www.jstor.org/stable/j.ctt7s02z

[132] A. van der Schaft and H. Schumacher, *An Introduction to Hybrid Dynamical Systems*. London: Springer, 2000.

[133] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, pp. 94–124, 1998.

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

[135] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid i/o automata," *Information and Computation*, vol. 185, no. 1, pp. 105–157, 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540103000671

[136] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," in *Proceedings of the IEEE*, 2000.

[137] V. Vladimerou, P. Prabhakar, M. Viswanathan, and G. E. Dullerud, "STORMED hybrid systems," in *Automata, Languages and Programming, ICALP 2008*, ser. LNCS, vol. 5126. Springer, 2008, pp. 136–147.

[138] G. Lafferriere, G. J. Pappas, and S. Sastry, "O-minimal hybrid systems," *Mathematics of control, signals and systems*, vol. 13, no. 1, pp. 1–21, 2000.

[139] A. Girard, "Controller synthesis for safety and reachability via approximate bisimulation," *Automatica*, vol. 48, no. 5, pp. 947 – 953, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S000510981200088X

[140] P. Tabuada, G. J. Pappas, and P. U. Lima, "Composing abstractions of hybrid systems." in *HSCC 2002*, ser. LNCS, C. Tomlin and M. R. Greenstreet, Eds., vol. 2289. Springer, 2002, pp. 436–450.

[141] P. Tabuada and G. J. Pappas, "Hybrid abstractions that preserve timed languages," in *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, ser. HSCC '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 501–514.

[142] A. Fehnker, E. M. Clarke, S. K. Jha, and B. H. Krogh, "Refining abstractions of hybrid systems using counterexample fragments," in *HSCC'2005*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds., vol. 3414. Springer, 2005, pp. 242–257.

[143] R. Alur, T. Dang, and F. Ivančić, "Counterexample-guided predicate abstraction of hybrid systems," *Theoretical Computer Science*, vol. 354, no. 2, pp. 250–271, 2006.

[144] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based CEGAR for rectangular hybrid systems," *Formal Methods in System Design*, vol. 46, no. 2, pp. 105–134, 2015. [Online]. Available: https://doi.org/10.1007/s10703-015-0225-4

[145] S. Sankaranarayanan, "Change-of-bases abstractions for non-linear hybrid systems," *Nonlinear Analysis: Hybrid Systems*, vol. 19, pp. 107 – 133, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1751570X15000540

[146] K. Hsu, R. Majumdar, K. Mallik, and A.-K. Schmuck, "Lazy abstraction-based control for safety specifications," in *57th IEEE Conference on Decision and Control, CDC 2018, Miami, FL, USA, December 17-19, 2018*, 2018. [Online]. Available: https://doi.org/10.1109/CDC.2018.8619659 pp. 4902–4907.

[147] T. Dang, O. Maler, and R. Testylier, "Accurate hybridization of nonlinear systems," 01 2010, pp. 11–20.

[148] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based cegar for rectangular hybrid systems," in *Verification, Model Checking, and Abstract Interpretation*, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 48–67.

[149] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan, "Hybrid automata-based cegar for rectangular hybrid systems," *Form. Methods Syst. Des.*, vol. 46, no. 2, p. 105–134, apr 2015. [Online]. Available: https://doi.org/10.1007/s10703-015-0225-4

[150] L. E. Kavraki, P. Svestka, J. . Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[151] S. M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.

[152] C. Fan, K. Miller, and S. Mitra, "Fast and guaranteed safe controller synthesis for nonlinear vehicle models," in *CAV*, S. K. Lahiri and C. Wang, Eds., 2020.

[153] 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.

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

[155] 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," ser. PLDI 2019. New York, NY, USA: ACM, 2019. [Online]. Available: https://doi.org/10.1145/3314221.3314633 p. 63–78.

[156] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, 2016, pp. 1–10.

[157] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, p. 61–67, feb 2019. [Online]. Available: https://doi.org/10.1145/3241979

[158] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3238147.3238187 p. 132–142.

[159] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[160] M. Lindvall, A. Porter, G. Magnusson, and C. Schulze, "Metamorphic model-based testing of autonomous systems," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, 2017, pp. 35–41.

[161] T.-C. Lin, Z. Hui, S. Huang, Z. Ren, and Y. Yao, "Metamorphic testing integer overflow faults of mission critical program: A case study," *Mathematical Problems in Engineering*, vol. 2013, p. 381389, 2013. [Online]. Available: https://doi.org/10.1155/2013/381389

[162] "Integration guidance for acas sxu version 3 (v3r0)." [Online]. Available: https://www.astm.org/COMMIT/Software.Description.zip

[163] C. Morris, "Midair collisions: Limitations of the see-and-avoid concept in civil aviation," *Aviation, space, and environmental medicine*, vol. 76, pp. 357–65, 04 2005.

[164] J. K. Kuchar and A. C. Drumm, "The traffic alert and collision avoidance system," 2007.

[165] "Investigation report for 2002 midair collision." [Online]. Available: https://reports.aviation-safety.net/2002/20020701-1_B752_A9C-DHL_T154_RA-85816.pdf

[166] Federal Aviation Administration, "UTM Pilot Program (UPP) Summary Report," Oct. 2019. [Online]. Available: https://www.faa.gov/uas/research_development/traffic_management/utm_pilot_program/media/UPP_Technical_Summary_Report_Final.pdf

[167] 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.

[168] S. Bak, H.-D. Tran, K. Hobbs, and T. T. Johnson, "Improved geometric path enumeration for verifying relu neural networks," in *Proceedings of the 32nd International Conference on Computer Aided Verification*. Springer, 2020.

[169] D. M. Lopez, T. T. Johnson, S. Bak, H.-D. Tran, and K. Hobbs, "Neural network verification methods for closed-loop acas xu properties," 2021.

[170] A. Irfan, K. D. Julian, H. Wu, C. Barrett, M. J. Kochenderfer, B. Meng, and J. Lopez, "Towards verification of neural networks for small unmanned aircraft collision avoidance," in *2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, 2020, pp. 1–10.

[171] Federal Aviation Administration, "Unmanned Aircraft System Traffic Management (UTM) Concept of Operations Version 2.0," Mar. 2020. [Online]. Available: https://www.faa.gov/uas/research_development/traffic_management/media/UTM_ConOps_v2.pdf

[172] R. W. Beard and T. W. McLain, *Small Unmanned Aircraft: Theory and Practice*. USA: Princeton University Press, 2012.

[173] T. Lee, M. Leok, and N. H. McClamroch, "Geometric tracking control of a quadrotor uav on se(3)," in *49th IEEE Conference on Decision and Control (CDC)*, 2010, pp. 5420–5425.

[174] R. Ghosh, J. P. Jansch-Porto, C. Hsieh, A. Gosse, M. Jiang, H. Taylor, P. Du, S. Mitra, and G. Dullerud, "Cyphyhouse: A programming, simulation, and deployment toolchain for heterogeneous distributed coordination," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 6654–6660.

[175] R. Ghosh, C. Hsieh, S. Misailovic, and S. Mitra, "Koord: A language for programming and verifying distributed robotics application," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428300

[176] C. Hsieh, H. Sibai, H. Taylor, Y. Ni, and S. Mitra, "Skytrakx: A toolkit for simulation and verification of unmanned air-traffic management systems," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021, pp. 372–379.

[177] G. Ellingson and T. McLain, "Rosplane: Fixed-wing autopilot for education and research," in *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2017, pp. 1503–1507.

[178] 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.

[179] K. Miller, C. Fan, and S. Mitra, "Planning in dynamic and partially unknown environments," in *In Proceedings of 7th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'21)*, 2021.

[180] T. A. Henzinger, R. Jhala, and R. Majumdar, "Counterexample-guided control," in *Automata, Languages and Programming*, J. C. M. Baeten, J. K. Lenstra, J. Parrow, and G. J. Woeginger, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 886–902.

[181] C. Fan, Z. Huang, and S. Mitra, "Approximate partial order reduction," in *Formal Methods*, K. Havelund, J. Peleska, B. Roscoe, and E. de Vink, Eds. Cham: Springer International Publishing, 2018, pp. 588–607.

[182] M. Schmidt and H. Lipson, "Distilling free-form natural laws from experimental data," *Science*, vol. 324, no. 5923, pp. 81–85, 2009.

[183] S. Dutta, X. Chen, and S. Sankaranarayanan, *Reachability Analysis for Neural Feedback Systems Using Regressive Polynomial Rule Inference*. New York, NY, USA: Association for Computing Machinery, 2019, p. 157–168. [Online]. Available: https://doi.org/10.1145/3302504.3311807