Using Symmetry Transformations in Equivariant Dynamical Systems for Their Safety Verification

Hussein Sibai(✉), Navid Mokhlesi, and Sayan Mitra

University of Illinois, Urbana, IL 61801, USA
{sibai2,navidm2,mitras}@illinois.edu

Abstract. In this paper, we investigate how symmetry transformations of equivariant dynamical systems can reduce the computation effort for safety verification. Symmetry transformations of equivariant systems map solutions to other solutions. We build upon this result, producing reachsets from other previously computed reachsets. We augment the standard simulation-based 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 reachsets. This new algorithm required the creation of a new cache-tree data structure for multi-resolution reachtubes. Our implementation has been tested on several benchmarks and has achieved significant improvements in verification time.

1 Introduction

Symmetry plays an important role in analysis of physical processes by summarizing the laws of nature independent of specific dynamics [14,25]. Symmetry related concepts have been used to explain and suppress unstable oscillations in feedback connected systems [21], show existence of passive gaits under changing ground slopes [26], and design control inputs for synchronization of neural networks [13,23].

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 [3,7]. 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 [17], but translating the idea 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 [2,19].

In this paper, we investigate how symmetry principles could benefit the analysis of cyberphysical systems (CPS). 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 [18]. The research program on data-driven verification and falsification has recently been met with some

Y.-F. Chen et al. (Eds.): ATVA 2019, LNCS 11781, pp. 98–114, 2019.
https://doi.org/10.1007/978-3-030-31784-3_6

successes [1,5,6,11]. The idea is to use simulation, together with model-based sensitivity analysis or property-specific robustness margins, to provide *coverage guarantees* or expedite the discovery of counterexamples. Software tools implementing these approaches have been used to verify embedded medical devices, automotive, and aerospace systems [1,5,9,11]. In this paper, we examine the question: how can we reduce the number of simulations needed to verify a CPS utilizing more information about the model in the form of its symmetries?

Contributions. The paper builds-up on the foundational results in symmetry transformations for dynamical systems [14,15,25] to provide results that allow us to compute the *reachable states* of a dynamical from a given initial set K', by transforming previously computed reachable states from a *different* initial set K. Since the computation of reachsets from scratch is usually more expensive than applying a transformation to a set, this reduces the number of reachset computations, and therefore, the number of simulations. Secondly, we identify symmetries that can be useful for analyzing CPS including translation, linear transforms, reflections, and permutations.

Third, 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 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 symGrpCacheTree.

Finally, we implemented the algorithms on top of the DryVR tool [10]. We augmented DryVR with symCacheTree and symGrpCacheTree. 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 paper starts with notations and definitions in Sect. 2. Examples of dynamical systems and symmetry transformations are given in Sect. 3. The main theorems of transforming reachtubes appear in Sect. 4. In Sect. 5, we present symCacheTree and symGrpCacheTree along with the key guarantees. The results of experiments are in Sect. 6 and conclusions and future directions are in Sect. 7.

2 Preliminaries

For any point $x \in \mathbb{R}^n$, we denote by x_i the i^{th} component of x. For any $\delta > 0$ and $x \in \mathbb{R}^n$, $B(x, \delta) \subseteq \mathbb{R}^n$ is a closed hypercube of radius δ centered at x. For a hyperrectangle

$S \subseteq \mathbb{R}^n$ and $\delta > 0$, $Grid(S, \delta)$, is a collection of 2δ-separated points along axis parallel planes such that the δ-balls around these points cover S. Given a positive integer N, we denote by $[N]$ the set of integers $\{1, \ldots, N\}$. Given an operator $\gamma : \mathbb{R}^n \to \mathbb{R}^n$ and a set $X \subseteq \mathbb{R}^n$, with some abuse of notation we denote by $\gamma(X)$ the subset of $\mathbb{R}^n$ that results from applying γ to every element on X. Let $D \in [N]$. We denote by the set $X \downarrow_D = \{x' : \exists x \in X, \forall\, i \in D, x_i = x'_i \text{ and } x'_i = 0, \text{otherwise}\}$. A continuous function $\beta : \mathbb{R}^+ \to \mathbb{R}^+$ is said to be a class-$\mathscr{K}$ function if it is strictly increasing and $\beta(0) = 0$.

Consider a dynamical system:

$$\dot{x} = f(x), \tag{1}$$

where $x \in \mathbb{R}^n$ is the state vector and $f : \mathbb{R}^n \to \mathbb{R}^n$ is a *Lipschitz continuous* function which guarantees existence and uniqueness of solutions [4]. The initial condition of the system is a compact set $K \subseteq \mathbb{R}^n$. A *solution* of the system is a function $\xi : \mathbb{R}^n \times \mathbb{R}^+ \to \mathbb{R}^n$ that satisfies (1) and for any initial state $x_0 \in K, \xi(x_0, 0) = x_0$. For a bounded time solution ξ, we denote the time domain by ξ.dom. Given an *unsafe set* $U \subset \mathbb{R}^n$ and a time bound $T > 0$, the *bounded safety verification problem* requires us to check whether there exists an initial state $x_0 \in K$ and time $t \leq T$ such that $\xi(x_0, t) \in U$.

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 (1) between times t_1 and t_2, starting from initial set $K \subset \mathbb{R}^n$ at time $t_0 = 0$ is defined as

$$Reach(K, [t_1, t_2]) = \{x \in \mathbb{R}^n \mid \exists\, x_0 \in K,\ t \in [t_1, t_2] \text{ s.t. } \xi(x_0, t) = x\}.$$

Thus, computing (or over-approximating) $Reach(K, [0, T])$ and checking $Reach(K, [0, T]) \cap U = \emptyset$ is adequate for verifying bounded safety. Instead of $Reach(K, [t, t])$ we write $Reach(K, t)$ in short for the set of state reachable from K after exactly t time units.

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* $Rtube(K, T) = \{(X_i, t_i)\}_{i=1}^{j}$ to be a sequence of time-stamped sets such that for each i, $X_i = Reach(K, [t_{i-1}, t_i])$, $t_0 = 0$ and $t_j = T$. The concatenation of two reachtubes $\{(X_i, t_i)\}_{i=1}^{j_1} \frown \{(X_i, t_i)\}_{i=1}^{j_2}$ is defined as the sequence $\{\{(X_i, t_i)\}_{i=1}^{j_1}, \{(X_i, t_i + t_{max})\}_{i=1}^{j_2}\}$, where t_{max} is the last time stamp in the first reachtube sequence.

A numerical simulation of system (1) is a reachtube with X_0 being a singleton state $x_0 \in K$. It is a discrete time representation of $\xi(x_0, \cdot)$. Several numerical solvers provide such representation of trajectories such as VNODE-LP[1] and CAPD Dyn-Sys library[2].

In this paper, we will find it useful to transform solutions and reachtubes using operators $\gamma : \mathbb{R}^n \to \mathbb{R}^n$ on the state space. Given a solution ξ and a reachtube $Rtube(K, T)$, we define the γ-transformed solution $\gamma \cdot \xi$ and reachtube $\gamma \cdot Rtube(K, T)$ as follows:

$$\forall\, t, (\gamma \cdot \xi)(x_0, t) = \gamma(\xi(x_0, t)) \text{ and } \gamma \cdot Rtube(K, T) = \{(\gamma(X_i), t_i)\}_{i=1}^{j}.$$

Notice that this transformation does not alter the time-stamps. Given a reachtube rt, $rt.last$ is the pair (X, t) with the maximum t in rt.

[1] http://www.cas.mcmaster.ca/~nedialk/vnodelp/.
[2] http://capd.sourceforge.net/capdDynSys/docs/html/odes_rigorous.html.

2.1 Data-Driven Verification

Data-driven verification algorithms answer the bounded safety verification question using numerical simulation data, that is, sample of simulations. The key idea is to generalize an individual simulation of a trajectory $\xi(x_0,\cdot)$ to over-approximate the reachtube $Rtube(B(x_0,\delta),T)$, for some $\delta > 0$. This generalization covers a δ-ball $B(x_0,\delta)$ of the initial set K, and several simulations can then cover all of K and over-approximate $Rtube(K,T)$, 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 $Rtube(K,T)$ have to be computed with smaller generalization radius δ and more simulation data.

Thus far, the generalization strategy has been 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 have soundness and relative completeness is formalized as *discrepancy function* [6].

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

The first condition in Definition 1 says that β upper-bounds the distance between two trajectories as a function of the distance between their initial states. The second condition makes the bound shrink as the initial states get closer.

Algorithms have been developed for computing this discrepancy function for linear, nonlinear, and hybrid dynamical models [5,8,11] as well as for estimating it for black-box systems [10]. The resulting software tools have been successfully applied to verify automotive, aerospace, and medical embedded systems [9,16,24].

Algorithm 1 without the boxed parts describes data-driven verification for a dynamical system (1). We refer to this algorithm as ddVer in this paper. Given the compact initial set of states $K \subseteq \mathbb{R}^n$, a time bound $T > 0$, and an unsafe set U, ddVer answers the safety verification question. It initializes a stack called *coverstack* with a cover of K. Then, it checks the safety from each element in the cover. For a given $B(x_0,\delta)$ in *coverstack*, ddVer simulates (1) from x_0 and bloats to compute an over-approximation of $Rtube(B(x_0,\delta),T)$. 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 $Rtube(B(x_0,\delta),T)$. If this over-approximation is disjoint from U then it is safe and is removed from *coverstack*. If instead, the over-approximation intersects with U 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 $Rtube(B(x_0,\delta),T)$. On the other hand, if the simulation hits U, that serves as a counterexample and ddVer returns Unsafe. Finally, if *coverstack* becomes empty, that implies that the algorithm reached a partition of K from which all the over-approximated reachtubes are disjoint from U, and then ddVer returns Safe.

Algorithm 1 ddVer safety verification algorithm

input: K, T, U, Γ, β
coverstack ← finite cover $\cup_i B(x_i, \delta) \supseteq K$
cachetree ← $\emptyset$
while *coverstack* $\neq \emptyset$ **do**
  $B(x_0, \delta) = coverstack.pop()$
  ans ← symGrpCacheTree $(U, \Gamma, cachetree, B(x_0, \delta))$
  if *ans* = Unsafe **then return:** *ans*
  else if *ans* = Safe **then continue**
  else
    sim ←simulate $\xi(x_0, \cdot)$ upto time T
    rt ← $sim \oplus \beta$
    $cachetree.insert(node(rt, sim))$
    if *sim* intersects with U **then**
      return: Unsafe
    else if *rt* intersects with U **then**
      Refine cover and add to the *coverstack*
return: Safe

2.2 Symmetry in Dynamical Systems

Symmetry takes a central place in analysis of dynamical systems [20]. The research line pertinent to our work develops the conditions under which one can get a solution by transforming another solution [12,20,22]. Symmetries of dynamical systems are modeled as groups of operators on the state space.

Definition 2 (Definition 2 in [25]). *Let Γ be a group of operators acting on $\mathbb{R}^n$. We say that $\gamma \in \Gamma$ is a symmetry of (1) if for any solution, $\xi(x_0, t)$, $\gamma \cdot \xi(x_0, t)$ is also a solution. Furthermore, if $\gamma \cdot \xi = \xi$, we say that the solution ξ is γ-symmetric.*

Thus, if γ is a symmetry of (1), then new solutions can be obtained by just applying γ to existing solutions. Herein lies the opportunity of exploiting symmetries in data-driven verification.

How can we know that γ is a symmetry for (1)? It turns out that, a sufficient condition exists that can be checked without finding the solutions (potentially hard problem), but only by checking commutativity of γ with the dynamic function f. Systems that meet this criterion are called *equivariant*.

Definition 3 (Definition 3 in [25]). *Let Γ be a group of operators acting on $\mathbb{R}^n$. The dynamic function (vector field) $f : \mathbb{R}^n \to \mathbb{R}^n$ is said to be Γ-*equivariant *if $f(\gamma(x)) = \gamma(f(x))$, for any $\gamma \in \Gamma$ and $x \in \mathbb{R}^n$.*

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

Theorem 1 ([14,25]). *If (1) is Γ-equivariant and ξ is a solution, then so is $\gamma \cdot \xi$, $\forall \gamma \in \Gamma$.*

3 Symmetries in Cyber-Physical 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. We start with a simple 2-dimensional linear system.

Example 1. Consider the circle system

$$\dot{x}_1 = -x_2, \dot{x}_2 = x_1. \tag{2}$$

where $x_1, x_2 \in \mathbb{R}$. Let Γ be the set of matrices of the form: $B = [[a, -b], [b, a]]$ where $a, b \in \mathbb{R}$ and B is not the zero matrix. Let $\circ$ be the matrix multiplication operator, then system (2) is Γ-equivariant.

Example 2. Lorenz attractor models the two-dimensional motion of a fluid in a container. Its dynamics are as follows:

$$\dot{x} = -px + py, \dot{y} = -xz + rx - y, \dot{z} = xy - bz. \tag{3}$$

where p, r, and b are parameters and x, y and $z \in \mathbb{R}$. Let Γ be the group that contains $\gamma : (x, y, z) \to (-x, -y, z)$ and the identity map. Then, system (3) is Γ-equivariant[3].

Example 3. Third, a car model is equivariant to the group of all translations of its position. The car model is described with the following ODEs:

$$\dot{x} = v\cos\theta, \dot{y} = v\sin\theta, \dot{\phi} = u, \dot{v} = a, \dot{\theta} = \frac{v}{L}\tan(\phi). \tag{4}$$

where u and a can be any control signals and x, y, v, θ,and $\phi \in \mathbb{R}$. We denote $r = (x, y, v, \phi, \theta) = (p, \bar{p})$, where $p = (x, y)$. Let Γ be the set of translations of the form $\gamma : r = (p, \bar{p}) \to r' = (p + c, \bar{p})$, for all $c \in \mathbb{R}^2$. Then, system (4) is Γ-equivariant.

Example 4. Consider the system of two cars with states r_1 and r_2. Let Γ be the set containing the operator $\gamma : (r_1, r_2) \to (r_2, r_1)$ and the identity operator. Moreover, assume that u and a are the same for both cars. Then, the system is Γ-equivariant.

Example 5. Let Γ be the group generated by the set of transformations of the form $\gamma : (r_1, r_2) = ((p_1, \bar{p}_1), (p_2, \bar{p}_2)) \to (r'_1, r'_2) = ((p_1 + c_1, \bar{p}_1), (p_2 + c_2, \bar{p}_2))$, where c_1 and $c_2 \in \mathbb{R}^2$, along with the group described in Example 4. Then, the system is Γ-equivariant. Hence, it is equivariant to translations in the positions and permutation of both cars.

4 Symmetry for Verification

In this section, we present new results that use symmetry ideas of Sect. 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

[3] http://www.scholarpedia.org/article/Equivariant_dynamical_systems.

computation of new reachtubes is in general more expensive than transforming ones. We derived similar theorems for switched systems in the extended version of the paper. For convenience, we will fix a set of initial states $K \subseteq \mathbb{R}^n$, a time bound $T > 0$, a group Γ of operators on $\mathbb{R}^n$, and an operator $\gamma \in \Gamma$ throughout this section. The following theorem formalizes transformation of reachtubes based on symmetry. It follows from Theorem 1.

Theorem 2. *If (1) is Γ-equivariant, then $\forall \gamma \in \Gamma$, $\gamma(Rtube(K,T)) = Rtube(\gamma(K),T)$.*

Proof. By Theorem 1, given any solution $\xi(x_0,\cdot)$ of system (1), where $x_0 \in K$, $\gamma(\xi(x_0,\cdot))$ is its solution starting from $\gamma(x_0)$, i.e. $\gamma(\xi(x_0,\cdot)) = \xi(\gamma(x_0),\cdot)$.

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

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

Corollary 1 shows how a new reachtube from a set of initial states $K' \subseteq \mathbb{R}^n$ can be computed by γ-transforming an existing $Rtube(K,T)$.

Corollary 1. *If system (1) is Γ-equivariant, and $K' \subseteq \mathbb{R}^n$, then if there exists $\gamma \in \Gamma$ such that $K' \subseteq \gamma(K)$, then $Rtube(K',T) \subseteq \gamma(Rtube(K,T))$.*

Remark 1. Corollary 1 remains true if instead of $Rtube(K,T)$, we have a tube that over-approximates it. Moreover, Theorem 2 and Corollary 1 are also true if we replace the reachtubes with reachsets.

5 Verification Algorithm

In this section, we add to the ddVer procedure the symCacheTree one for caching, searching, and transforming reachtubes. The result is the new ddSymVer algorithm. symCacheTree uses symmetry to save ddVer from computing fresh reachtubes in lines 10–11 in case they can be transformed from already computed and cached reachtubes. Later we will replace symCacheTree with the more general symGrpCacheTree procedure.

The idea of symCacheTree (and symGrpCacheTree) is as follows: given a tree *cachetree* storing reachtubes as they are computed by ddVer, an initial set of states $initset_n$ that ddVer needs to compute the reachtube for, a symmetry operator γ (or a group of them Γ) for system (1) and the unsafe set U, it checks if the safety of the system starting from $initset_n$ can be decided by transforming reachtubes stored in *cachetree*.

Before getting into symCacheTree and symGrpCacheTree, we note the simple additions to ddVer (shown by boxes) that lead to ddSymVer. First, *cachetree* is initialized to an empty tree (line 3). Then, symCacheTree (or symGrpCacheTree) 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, fresh reachtube rt gets computed in line 11 and inserted as a new node in *cachetree* (line 12).

Tree Data Structure. Each node *node* in symCacheTree stores an initial set *initset*, a simulation *sim* of duration T from the center of *initset*, and an over-approximation *rt* of $Rtube(initset, T)$. The key invariants of symCacheTree for non *Null* nodes are:

$$root.initset = K, \tag{5}$$
$$\forall\, node, node.left.initset \subseteq node.initset, \tag{6}$$
$$\forall\, node, node.right.initset \subseteq node.initset, \tag{7}$$
$$\forall\, node, node.left.initset \cap node.right.initset = \emptyset, \tag{8}$$
$$\forall\, node, node.left.initset \cup node.right.initset = node.initset. \tag{9}$$

That is, the *initset* of the root node is equal to K; each child's *initset* is contained in the *initset* of the parent; the disjoint union of the *initset*s of the children partition the *initset* of the parent. Hence, by property (2) of the discrepancy function β (Definition 1) it follows that the union of the reachtubes of children is a tighter over-approximation of the reachtube of the parent, for the same initial set. Since the refinement in ddSymVer is done depth-first, symCacheTree is also constructed in the same way.

In brief, symCacheTree (symGrpCacheTree) uses symmetry to save ddVer from computing the reachtube $Rtube(initset_n, T)$ afresh in line 11 from initial set $initset_n$ in the case that safety of $Rtube(initset_n, T)$ can be inferred by transforming an existing reachtube in *cachetree*. That is, given an unsafe set U, a tree *cachetree* storing reachtubes (previously computed), and a symmetry operator γ (a group of symmetries Γ) for system (1), symCacheTree (Algorithm 2) or symGrpCacheTree (Algorithm 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*.

5.1 The symCacheTree Procedure

The core of the symCacheTree algorithm 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 1 to get an over-approximation of $Rtube(initset_n, T)$ and symCacheTree returns SymmetryNotUseful (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 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 $\gamma(Rtube(initset_c, T))$ and $Rtube(initset_n, T)$ do not intersect. Hence, the node is not useful for this initial set. Also, if the transformed reachtube $\gamma(node.rt)$ does not intersects U, 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 U, then we have a counter example by Theorem 1. Hence, it returns Unsafe (line 12). If the transformed reachtube $\gamma(node.rt)$ intersects U, it cannot know 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, it returns Compute, asking ddVer to compute the over-approximation from scratch (line 15). Otherwise, it enqueue all the children nodes in *traversalQueue* (line 14).

If *traversalQueue* gets empty, then we have an over-approximation of the reachtube starting from $initset_n$ that does not intersect with U. Hence, it returns Safe (line 16).

The following two theorems show the correctness guarantees of symCacheTree. The proofs are in the extended version of the paper. Theorem 3 shows that if *cachetree* has reachtubes that can prove that the system is safe using γ, it will return Safe. If it has a simulation that can prove that the system is unsafe using γ, it will either ask ddVer to compute the reachtube from scratch or will return Unsafe. Theorem 4 shows that if symCacheTree returns Safe, then the reachtube of the system starting from $initset_n$ does not intersect U. Moreover, if it returns Unsafe, then there exists a trajectory that starts from $initset_n$ and intersects U.

Algorithm 2. symCacheTree

1: **input:** $\mathrm{U}, \gamma, cachetree, initset_{\mathbf{n}}$
2: $initset_c := cachetree.root.initset$
3: **if** $initset_n \not\subseteq \gamma(initset_c)$ **then**
4:     **return:** SymmetryNotUseful
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 $\gamma(node.rt) \cap U = \emptyset$ **then**
10:         continue
11:     **if** $\exists j \mid \gamma(R_j) \cap U \neq \emptyset$ 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

Theorem 3 (Completeness). *If there exists a set of nodes S in cachetree with*

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

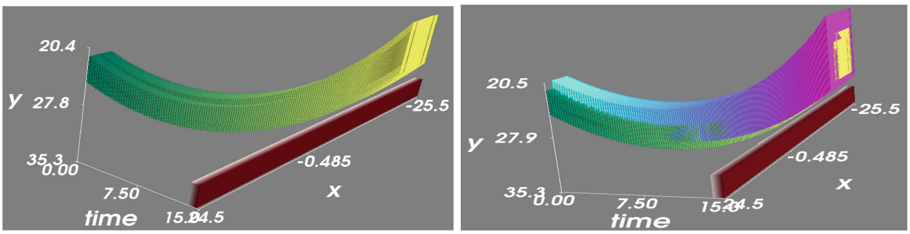
symCacheTree *will return* Safe*. Also, if there exists a node s in cachetree where* $\gamma(s.sim) \cap U \neq \emptyset$ *and starts from* $initset_n$*, then* symCacheTree *will return* SymmetryNotUseful*,* Unsafe*, or* Compute*.*

Theorem 4 (Soundness). symCacheTree *is sound: if it returns* Safe*, then the reachtube* $Rtube(initset_n, T)$ *does not intersect* U *and if it returns* Unsafe*, then there exists a trajectory starting from* $initset_n$ *that enters the unsafe set.*

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 to illustrate the need for multiple symmetry maps.

Circular Orbits and Scaling Symmetry. The linear system in Example 2 has circular orbits. Consider the initial set $K = [[21.5, 21.5], [24.5, 24.5]]$, the unsafe set $x_2 \geq 32$ after $t = 1.4$ s, and the time bound $T = 1.5$ s. 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 K 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 1).

That is because we are using a single transform which leaves symCacheTree useless in most of the input cases. Figure 1a shows the reachtube (colored green to yellow) computed using ddVer, unsafe set (brown). Figure 1b shows the reachtube computed using ddVer and symCacheTree. 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 instead if being computed. Next, we present symGrpCacheTree, a generalization of symCacheTree that uses a group of symmetries aiming for a bigger ratio of transformed to computed reachtubes.



(a) (b)

Fig. 1. (a) Reachtube using ddVer. (b) Reachtube using ddSymVer. (Color figure online)

5.2 The symGrpCacheTree Procedure

Procedure symGrpCacheTree (Algorithm 3) is a generalization of symCacheTree using a group of symmetries. The symGrpCacheTree 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). However, it is not sufficient in symGrpCacheTree since there may be another node that cover the same part of $initset_n$ which has a transformed reachtube that intersects U, 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, it cannot return Compute when the transformed reachtube of a visited node intersects U 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 because of the availability of multiple symmetries. Hence, it cannot return Compute unless it traversed the whole tree and still parts of $initset_n$ could not be proven to be safe. We show that symGrpCacheTree has the same

---

Algorithm 3. symGrpCacheTree

1: **input:** U, 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:** SymmetryNotUseful
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:     $X = \{x : \exists \gamma \in \Gamma, x \in \gamma(initset_c)$ and $\gamma(node.rt) \cap U = \emptyset\}$
11:     $leftstates \leftarrow leftstates \backslash X$
12:     **if** $\exists \gamma \in \Gamma, j \mid \gamma(R_j) \cap U \neq \emptyset$ 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

---

guarantees as symCacheTree in the following two theorems with the proof being in the extended version of the paper.

Theorem 5 (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*

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

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* symGrpCacheTree *will return* SymmetryNotUseful*,* Unsafe*, or* Compute*.*

Theorem 6 (Soundness). symGrpCacheTree *is sound: if it returns* Safe*, then the reachtube $Rtube(initset_n, T)$ does not intersect U and if it returns* Unsafe*, then there exists a trajectory starting from $initset_n$ that enters the unsafe set.*

The new challenge in symGrpCacheTree is in computing the union at line 3, computing X 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 from Sect. 3 to show that these operations are easy to compute in some cases.

5.3 Revisiting Arbitrary Translations

Recall that the car model in Example 3 in Sect. 3 is equivariant to all translations in its position. In this section, we show how to apply symGrpCacheTree not just for it, but to arbitrary differential equations. Let D be the set of components of the states that do not appear on the RHS of (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 $[n]\backslash D$ contains $initset_n$ projected to the same components. Since if it is true, $initset_c$ can be translated arbitrarily in its components in D so that the union contains $initset_n$.

Given two initial sets K and K' and the reachtube starting from K', we compute $\beta \subseteq \mathbb{R}^n$ such that $K' \downarrow_D \oplus \beta = K \downarrow_D$. Then, if $K \downarrow ([n]\backslash D) \subset K' \downarrow ([n]\backslash D)$, by Corollary 1, we can use that β to compute an overapproximation of $Rtube(K,T)$ by computing $Rtube(K',T) \oplus \beta$. Then, let β be such that $initset_c \downarrow_D \oplus \beta = leftstates \downarrow_D$, in line 10. We set X to be equal to $initset_c \oplus \beta$ if $node.rt \oplus \beta \cap U = \emptyset$ and to $\emptyset$, otherwise. To check the $\exists$ operator in line 12, we can treat the simulation as $node.rt$ and compute β accordingly. Then, compute $node.sim \oplus \beta$. The new condition would be then: if $R_j \oplus \beta \cap U$. Notice that we dropped $\gamma(R_0) \in leftstates$ from the condition since we know that $R_0 \in leftstates$ and β is bloating it to the extent it is equal to $leftstates$.

Optimized symGrpCacheTree *for Arbitrary Translations.* The size of $K' \downarrow_D$ above does not matter, i.e. even if it is just a point, one can compute β so that it covers $K \downarrow_D$. Hence, instead of computing $Rtube(K,T)$, we compute only $Rtube(K',T)$ and then compute β from K and K' and then bloat it. This decreases the number of dimensions that the system need to refine by $|D|$. This is in contrast with what is done in symGrpCacheTree where the reachtubes are computed without changing the initial set structure. This improvement resulted in verifying models in 1s when they take an hour on DryVR as shown in Sect. 6. We call this algorithm TransOptimized and refer to it as version 2 of symGrpCacheTree when applied to arbitrary translation invariance transformations.

6 Experimental Evaluation

We implemented symCacheTree and symGrpCacheTree in Python 2.7 on top of DryVR[4]. DryVr implements ddVer to verify hybrid dynamical systems. We augmented it and implemented ddSymVer. In our experiments, we only consider the (non-hybrid) dynamical systems. DryVR learns discrepancy from simulations as it is designed to work with unknown dynamical models. This learning functionality is unnecessary for our experiments, 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.

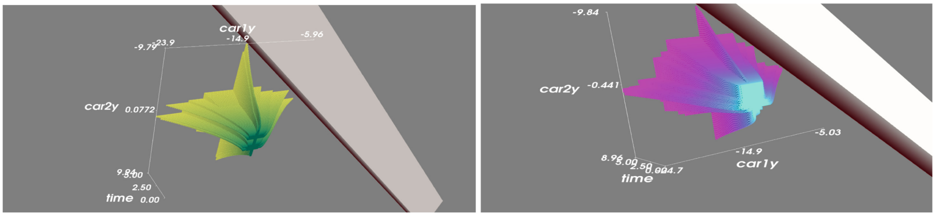
In this section, we present the experimental results on several examples using symCacheTree and symGrpCacheTree. The transformations used are linear. Two of the systems are linear and one is non-linear. The results of the experiments are shown in Table 1. The experiments were ran on a computer with specs shown in the extended version of the paper. 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 K of (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 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 r_2 with r_1. The system is unsafe if the cars are too close to each other. Assume that initially (y_1, y_2) belongs to $K = [[l_1, l_2], [u_1, u_2]]$. If the two intervals $[l_1, u_1]$ and $[l_2, u_2]$ do not intersect, γ would not be useful since for any $X \subseteq K$, $\gamma(X) \cap K = \emptyset$. However, if $K = [[l_1, l_2], [u_1, u_2]] \cup [[l_1', l_2'], [u_1', u_2']]$, 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 from the one 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. This can also be done for (x_1, x_2) and a combination of both.

We implemented ddVer for the disjoint initial sets case as follows: We first ran ddVer to compute the reachtube of the system starting from the first hyperrectangle and cached all the computed reachtubes in the process in a *cachetree*. Then, we used that *cachetree* in ddSymVer to check the safety of the system starting from the second hyperrectangle.

Cars and Permutation Invariance. For the car example (Example 4), we ran ddVer on an initial set where $(x_1, y_1, x_2, y_2) \in [[0, -2.42, 0, -22.28], [2, 3.93, 0.1, -12.82]]$ and running for 5s and the unsafe set being $|y_1 - y_2| < 5$ and cached all the tubes in a *cachetree* and saved it on the hard-drive. It returned Safe. Then, we used it in

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



(a) (b)

Fig. 2. (a) Cars reachtube using ddVer. (b) Cars reachtube using ddSymVer.

symCacheTree to verify the system starting from $[[0, -22.28, 0, -2.42], [0.1, -12.82, 2, 3.93]]$. 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 93 s; the output is shown in Figs. 2a and b. Figure 2a shows the tube when computed by ddVer and Fig. 2b when computed by ddSymVer. Figure 2b has only blue-to-violet colors since it was all computed using a symmetry transformation.

Lorenz Attractor and Circle Revisited. We used the disjoint initial sets verification implementation to use the symmetry transformation for the nonlinear lorenz attractor in its safety verification. Recall from Sect. 3 that its symmetry map is $(x, y, z) \rightarrow (-x, -y, z)$. So for any given initial set $K = [[l_x, l_y, l_z], [u_x, u_y, u_z]]$ and a corresponding overapproxiation of the reachtube, we automatically get an overapproximation of the reachtube with the initial set $[[-u_x, -u_y, l_z], [-l_x, -l_y, u_z]]$. We generated the *cachetree* from the initial set $[[14.9, 14.9, 35.9], [15.1, 15.1, 36.1]]$, unsafe set $x \geq 20$ and $T = 10$ s that returned Safe and used that *cachetree* in symCacheTree to verify the system starting from $[[-15.09, -15.09, 35.91], [-14.91, -14.91, 36.09]]$. The resulting statistics are in Table 1. Lorenz1 is the one corresponding to the first initial set and Lorenz2 to the for which we use permutation symmetry.

We revisit the circle example from Sect. 5 and test symCacheTree performance with the transformation being: $\gamma : (x, y) \rightarrow (-y, x)$ instead of the scaling one. Then, we compute the reachtube starting from the same initial set as before and created its *cachetree*. After that, we used ddSymVer with symCacheTree to get the one starting from $[[-24.49, 21.51]$, $[-21.51, 24.49]]$ and running for 1.5 s. The statistics are shown in Table 1. The figures of the reachtubes are in the extended version of the paper. Again, the whole tube is blue-to-violet since it is computed fully by transforming parts of *cachetree*.

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 transforma-

Table 1. Results. Columns 3–5: number of times symCacheTree (or symGrpCacheTree) 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	Transformations (Γ)	Compute	Safe	Unsafe	SRefs	DryVR+sym	NoSRefs	DryVR
Circle1	$(0.95x_1, 0.95x_2)$	5	1	0	6	1.78	7	0.54
Circle2	$(-x_2, x_1)$	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. subset	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

tions, i.e. using symGrpCacheTree, storing compressed reachtubes, and optimizing the code.

Cars and General Translation. Finally, we ran ddSymVer with the two versions of symGrpCacheTree for translation invariance described in Sect. 5.3 on three different scenarios of the 2-cars Example 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 $T = 5$ s 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 symGrpCacheTree is used without modifications.

7 Conclusions

Equivariant 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 symGrpCacheTree) that use symmetry transformations, to verify the safety of the equivariant system 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. This paper opens the doors for more investigation of the role that symmetry can help in testing, verifying, and synthesizing dynamical and hybrid systems.

Acknowledgments. The authors are supported by a research grant from The Boeing Company and a research grant from NSF (CPS 1739966). We would like to thank John L. Olson and Arthur S. Younger from The Boeing Company for valuable technical discussions.

References

1. Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-TaLiRo: a tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_21
2. Antuña, L., Araiza-Illan, D., Campos, S., Eder, K.: Symmetry reduction enables model checking of more complex emergent behaviours of swarm navigation algorithms. In: Dixon, C., Tuyls, K. (eds.) TAROS 2015. LNCS (LNAI), vol. 9287, pp. 26–37. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22416-9_4
3. Clarke, E.M., Jha, S.: Symmetry and induction in model checking. In: Computer Science Today: Recent Trends and Developments, pp. 455–470 (1995)
4. Coddington, E.A., Levinson, N.: Theory of Ordinary Differential Equations. McGraw-Hill, New York (1955)
5. Donzé, A.: Breach, a toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_17
6. Duggirala, P.S., Mitra, S., Viswanathan, M.: Verification of annotated models from executions. In: EMSOFT (2013)
7. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Computer Aided Verification, 5th International Conference, CAV 1993, Elounda, Greece, Proceedings, 28 June–1 July 1993, pp. 463–478 (1993)
8. Fan, C., Mitra, S.: Bounded verification with on-the-fly discrepancy computation. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 446–463. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_32
9. Fan, C., Qi, B., Mitra, S.: Data-driven formal reasoning and their applications in safety analysis of vehicle autonomy features. IEEE Design Test **35**(3), 31–38 (2018)
10. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: DryVR: data-driven verification and compositional reasoning for automotive systems. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 441–461. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_22
11. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 531–538. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_29
12. Freund, P.G.O.: Introduction to Supersymmetry. Cambridge Monographs on Mathematical Physics. Cambridge University Press, Cambridge (1986)
13. Gérard, L., Slotine, J.J.: Neuronal networks and controlled symmetries, a generic framework. arXiv preprint q-bio/0612049 (2006)
14. Golubitsky, M., Stewart, I.: The Symmetry Perspective: From Equilibrium to Chaos in Phase Space and Physical Space. Progress in Mathematics, Birkhäuser Basel (2012)

15. Golubitsky, M., Stewart, I., Török, A.: Patterns of synchrony in coupled cell networks with multiple arrows. SIAM J. Appl. Dyn. Syst. **4**(1), 78–100 (2005)
16. Huang, Z., Fan, C., Mereacre, A., Mitra, S., Kwiatkowska, M.: Invariant verification of non-linear hybrid automata networks of cardiac cells. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 373–390. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_25
17. Ip, C.N., Dill, D.L.: Better verification through symmetry. In: Proceedings of the 11th IFIP WG10.2 International Conference Sponsored by IFIP WG10.2 and in Cooperation with IEEE COMPSOC on Computer Hardware Description Languages and Their Applications, CHDL 1993, pp. 97–111. North-Holland Publishing Co., Amsterdam, The Netherlands (1993)
18. Koopman, P., Wagner, M.: Challenges in autonomous vehicle testing and validation. SAE Int. J. Transp. Saf. **4**(2016–01–0128), 15–24 (2016)
19. Kwiatkowska, M.Z., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: 18th International Conference on Computer Aided Verification, CAV 2006, Seattle, WA, USA, 17–20 August 2006, Proceedings, pp. 234–248 (2006)
20. Marsden, J.E., Ratiu, T.S.: Introduction to Mechanics and Symmetry: A Basic Exposition of Classical Mechanical Systems. Springer, New York (2010). https://doi.org/10.1007/978-0-387-21792-5
21. Mehta, P.G., Hagen, G., Banaszuk, A.: Symmetry and symmetry-breaking for a wave equation with feedback. SIAM J. Appl. Dyn. Syst. **6**(3), 549–575 (2007)
22. Olver, P.J.: Applications of Lie Groups to Differential Equations. Springer, New York (1986). https://doi.org/10.1007/978-1-4684-0274-2
23. Pham, Q.C., Slotine, J.J.: Stable concurrent synchronization in dynamic system networks. Neural Netw. **20**(1), 62–77 (2007)
24. Prabhakar, P., Duggirala, P.S., Mitra, S., Viswanathan, M.: Hybrid automata-based cegar for rectangular hybrid systems. Form. Methods Syst. Des. **46**(2), 105–134 (2015)
25. Russo, G., Slotine, J.J.E.: Symmetries, stability, and control in nonlinear systems and networks. Phys. Rev. E **84**(4), 041929 (2011)
26. Spong, M.W., Bullo, F.: Controlled symmetries and passive walking. IEEE Trans. Autom. Control **50**(7), 1025–1031 (2005)