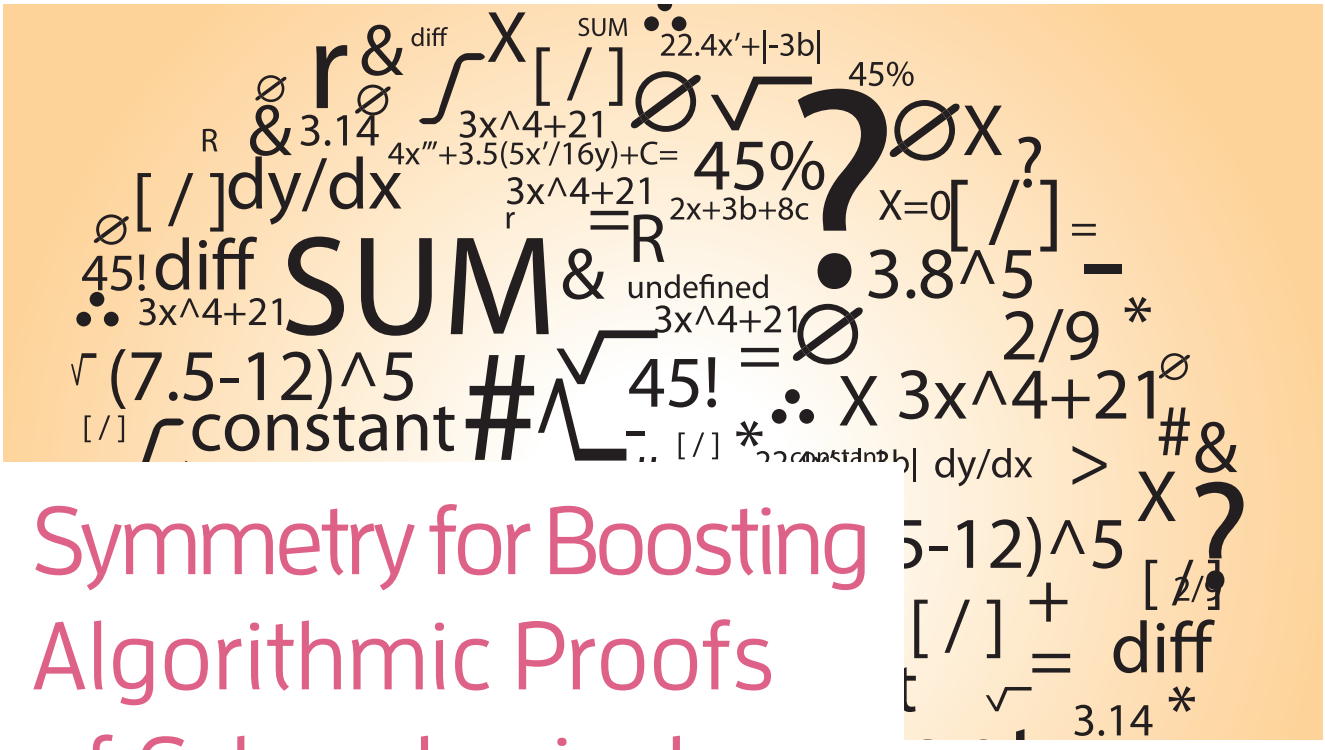

Symmetry for Boosting Algorithmic Proofs of Cyberphysical Systems

Sayan Mitra and Hussein Sibai, University of Illinois at Urbana-Champaign

Recent algorithms show how the availability of structural knowledge, such as symmetries, can significantly improve autonomous system verification in terms of both running time and sample complexity.

Verification is used both for constructing mathematical proofs as an assurance for quality and for bug hunting. Light- and heavyweight verification algorithms run every day on billions of lines of code at Google, Amazon, Facebook, and other software firms.[1–3] They ensure code-level the safety and security properties of mobile applications, Internet of Things operating systems, bootloaders, and device drivers. In cyberphysical systems (CPSs), like delivery robots and autonomous cars, the requirements to be verified are related to safety-critical system-level aspects like stability, robustness, and timeliness.

In the past decade, a number of software tools, like Flow*,[4] SpaceEx,[5] DryVR,[6] HyLAA,[7] and C2E2,[8] have been developed, and they have been successfully applied to verify realistic CPSs. On the one hand, algorithmic verification holds the promise of slashing development, testing, and certification costs. On the other hand, the scalability and usability of these approaches remain a challenge. In this article, we discuss how recent advances make it possible to exploit meta knowledge about models, such as symmetries, to improve the performance of CPS verification algorithms.

Digital Object Identifier 10.1109/MC.2022.3190954
Date of current version: 26 September 2022

EDITOR **DORON DRUSINSKY**
Naval Postgraduate School; ddrusins@nps.edu

Verification algorithms for CPSs work on mathematical models like hybrid automata—an expressive formalism that combines discrete transitions for describing programs and continuous flows for describing the evolution of physical quantities.[9] An example of a simple hybrid automaton is provided in the "Symmetry Abstraction and Refinement" section. The hybrid automaton model of an autonomous car, for instance, would consist of a program for computing the control decisions for steering, throttle, and brake as well as ordinary differential equations (ODEs) for describing how the vehicle moves in space according to the laws of physics, the vehicle characteristics, and the computed control decisions. The building block algorithm for the verification of hybrid automata is based on what is called *reachability analysis*. Before discussing how symmetry improves reachability algorithms for hybrid systems, we introduce the key metrics for these algorithms, namely, the running time and sample efficiency.

For the autonomous vehicle model, a simulation, or a test, produces a single behavior of the vehicle over time as its program drives through an environment—say, a road intersection or in a crowded urban space. A test is a record of the state of the system at different time points as it negotiates the merge; the state here includes both the values of the program or software variables as well as the physical variables, like the car's own position, velocity, and acceleration, and those of other actors in the environment, like pedestrians and vehicles. Running tests can help find design bugs but cannot show their absence: a model, like the autonomous car, has an uncountably infinite set of behaviors arising from different initial conditions; different environments; and, possibly, the innate nondeterminism in the controller program; therefore, no finite set of tests can cover all of these behaviors.

Reachability analysis, in contrast to testing, computes all behaviors of the model. Consider a quadrotor with a planned path to take off and follow a sequence of waypoints. When the quadrotor actually takes off and follows this path, many different trajectories may arise because of sensor errors and wind disturbances. As in Figure 1, reachability analysis computes a set, called the *reachset*, that contains all of these paths. From the reachset, we can then check for safety violations, like collision with obstacles.

One common use case for verification is offline or design-time analysis. Reachsets are used to generate safety or quality assurance certificates; found design bugs are analyzed by developers to improve the code or the design, amend the requirements, and discover new operating assumptions necessary for claiming product safety. The speed of the analysis is important. Reports from the software industry suggest that it has to finish in about 15 min for developers to react to the suggested bugs without being burdened by big context switches.[3] A second use case is in runtime verification. The onboard computer of the quadrotor could compute the reachset forward in time to check for potential collisions. For typical control loops, this analysis has to finish in 10–100 ms for the results to be relevant. In either use case, the running time for the analysis is a key metric for usability.

REACHABILITY AND SAFETY FROM SIMULATIONS AND SENSITIVITY

Reachability analysis for hybrid automata has been known to be computationally intractable since the early 1990s. The discovery of new data structures, like support functions[5] and generalized star sets,[7] have enabled the design of practical algorithms that compute approximate reachsets for restricted types of models.[9] These algorithms rely on the availability of analytical solutions of the linear differential equations, which are restrictive when targeting commonly used nonlinear models or systems where a complete model is unavailable.

The next generation of algorithms relied on numerical simulations and sensitivity analysis.[6,10] Consider an ODE $\dot{x} = f(x)$ describing the behavior

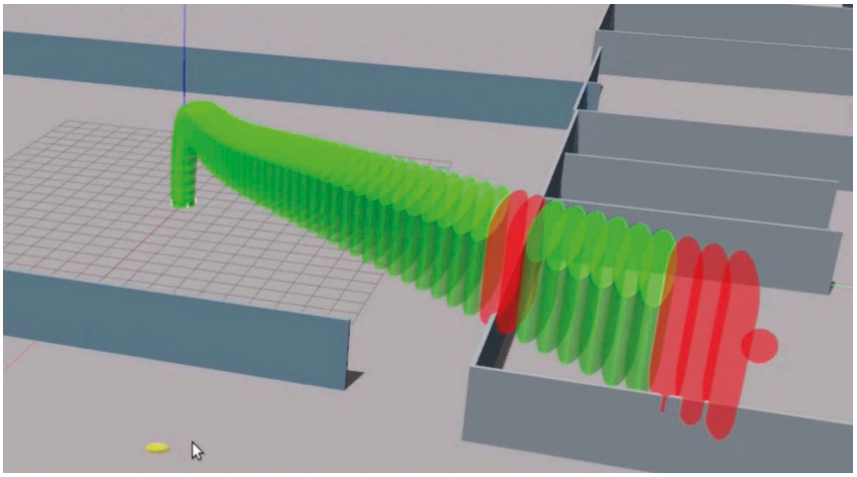

FIGURE 1. A reachability analysis for a quadrotor's planned path from a set of initial conditions shows possible unsafe intersections.

of a vehicle. A solution or a behavior of this system is a function of time $\xi(t)$ with the initial state $\xi(0)$. Given the initial state, the solution can be computed using numerical integration. This is done using the `Simulate()` function in Algorithm 1.

Suppose we can also somehow compute the sensitivity of $\xi(t)$. That is, for a given $\delta > 0$ perturbation to the initial state, for any solution ξ' that starts nearby, that is, $|\xi(0) - \xi'(0)| \leq \delta$, we have an upper bound $|\xi(t) - \xi'(t)| \leq \beta(t)$. Then, by bloating $\xi(t)$ by the factor $\beta(t)$, we can compute a set that contains all behaviors that start from the ball centered at $\xi(0)$ with radius δ. This set is going to contain the reachset of the system from the ball of radius δ.

By repeatedly performing this simulation and bloating operation over a number of such δ balls covering all of the initial states of interest, we can perform a reachability analysis using a finite number of simulations. It can also be shown that by shrinking δ—that is, the size of the balls covering the initial set—the computed reachset can be made arbitrarily precise at the expense of requiring more simulations.

Algorithm 1 uses this simulation plus bloating strategy for safety verification. Given a set of initial states K, a time horizon T, a set of `Unsafe` states, and the sensitivity β, it decides whether any solution from K hits the `Unsafe` set or not. It maintains a list of balls, `coverlist`, which is initialized as a cover of K with balls of radius δ. In the while loop, each ball (x, δ) in `coverlist` is analyzed with three possible outcomes: either

1. the reachset from the (x, δ) ball—computed by bloating the simulation from x by β—is disjoint from the `Unsafe` set and is removed from `coverlist`, or
2. a part of the simulation is contained in `Unsafe`, and the algorithm returns "Unsafe."

If neither holds, then

3. the (x, δ) ball is replaced with a finer cover of $\delta/2$ balls in `coverlist`.

This refinement of the cover ensures that a more precise reachset from the (x, δ)-ball will be computed in future iterations. The algorithm returns "Safe" only when the reachsets from all of the balls in `coverlist` are disjoint from the `Unsafe` set.

A key subroutine in Algorithm 1 is the computation of the sensitivity β. It is well known that, if the function f in the ODE is continuous with the Lipschitz constant L, then $\beta(t) := \delta e^{Lt}$ serves as a sensitivity function.[9,10] However, this function blows up with time, and, therefore, δ has to be made very small for a useful analysis, which, in turn, requires many simulations. In previous work,[11,12] it was shown that the notion of matrix measures of f can be used to compute much more precise sensitivity functions from general nonlinear models.

CYBERPHYSICAL SYMMETRIES FOR BOOSTING REACHABILITY

For models of vehicles and robots, often, we have additional knowledge about their symmetries. For example, we might know that the trajectory of a car making a left turn is just a reflection of its trajectory for a right turn. Could the knowledge of such symmetries be exploited to develop verification algorithms that use fewer samples and run faster? In the rest of this article, we discuss the ideas presented by Sibai et al.[13–15] that exploit model symmetries to make verification faster.

Formally, a symmetry for an ODE model $\dot{x} = f(x)$ is a function $g(\cdot)$ that transforms one state of the system to another, and this function commutes $f(\cdot)$. The commutation property implies that a behavior of the system $\xi'(t)$ starting from a transformed state $x'_0 = g(x_0)$ is identical to the transformed behavior $\xi(t)$ from x_0. This is, $g(\xi(t)) = \xi'(t)$, as shown in Figure 2.

Common symmetries include translations, rotations, and reflections. For physics models of vehicles, commutativity with respect to these symmetries holds because the vehicle models do not depend on the actual global position and orientation, but only on the relative positions. The car runs the same way at the 100th mile as it did at the 50th mile (ignoring fuel depletion and depreciation). For CPS testing, this means that a test run from the 50th to 52nd mile can be translated in position

ALGORITHM 1: BASIC SIMULATION-DRIVEN SAFETY VERIFICATION

```
Input: K,T,Unsafe,β
coverlist = Cover(K,δ)
While coverstack != empty
For each (x,δ) in coverlist
  ξ = Simulate(x,T)
  If Bloat(ξ,δ, β) ∩ Unsafe = empty
    coverlist = coverlist - (x,δ)
  ElseIf ∃ξ(t) ∈ Unsafe Return "Unsafe"
  Else coverlist := coverlist - (x,δ) + Cover(B(x,δ),δ/2)
Return "Safe"
```

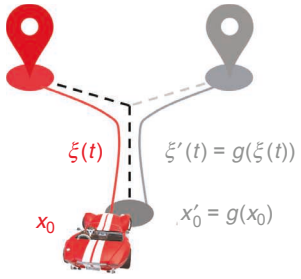

FIGURE 2. The symmetry allows trajectories starting from $g(x_0)$ to be obtained by transforming the trajectory from x_0.

to create another test run from the 100th to 102nd mile—for free. We do not really need to run the second test or do the simulation; we can simply apply the transformation $g()$ to the first test, which, in this case, is a translation of 50 mi, to create the new test.

All translations are not valid symmetries. The behavior of a vehicle speeding up from 60 to 80 mi/h will not simply be the shifted version of the run from 40 to 60 mi/h in the velocity coordinates. Similarly, behaviors of drones cannot be freely translated along the z-axis because the action of gravity breaks symmetry.

As the state in a CPS includes both computational and physical components, the types of symmetries can also go well beyond the common geometric symmetries. For example, consider an autonomous vehicle about to make a left turn. The controller software state includes a target waypoint for the vehicle to reach after the left turn (see Figure 2). Consider a state transformation that maps the left waypoint to the right waypoint and reflects the position of the vehicle from left across to the right. This state transformation is a symmetry of the waypoint-tracking control system. More generally, most waypoint-tracking controllers enjoy these types of translation, rotation, and reflection symmetries that combine the software and the physical state of the system.

Another symmetry that makes sense in multiagent CPSs is the notion of permutation symmetry. When analyzing an ego vehicle interacting with a collection of other vehicles, the identities of the other vehicles may not matter from the point of view of the ego vehicle. In such situations, considering one possible type or initial condition for each of the other vehicles may be adequate instead of considering all possible permutations of choices.

APPLICATION OF SYMMETRY IN VERIFICATION: CACHING

Just like a single new behavior can be obtained by transforming a previously computed behavior, a set of behaviors from a set of states $g(S)$ can be computed by transforming the previously computed behaviors from S. This idea can be incorporated in Algorithm 1 by caching the reachsets computed from different initial covers. For any subsequent reachability analysis from a new (x, δ) in `coverlist`, first, we whether the reachset from (x, δ) or any of its transforms has been cached. If so, it avoids the expensive reachability analysis by pulling out the cached value and transforming it appropriately. Otherwise, the algorithm proceeds to compute the reachset (x, δ) from scratch and cache the result.

Sibai et al.[13] show how a tree-like cache storing multiprecision reachsets can, indeed, speed up verification. The cache tree can be used as a "library" on top of existing reachability tools like Flow*,[4] SpaeEx,[5] DryVR,[6] HyLAA,[7] and C2E2.[8] We experimented with DryVR and tested the idea on several low-dimensional ODE models: the Lorenz attractor, a simple oscillator, and a system consisting of two 5D car models. These systems possess different symmetries, including scaling, reflection, translation, and permutation. With a single symmetry, the cache read and write overhead often dominates the savings in reachability. However, when multiple symmetries are used, such as translation invariance in the two-car system, symmetry caching sped up the reachability analysis by up to two orders of magnitude, and it computed up to four orders of magnitude fewer reachsets.

In hybrid models with multiple vehicles, for all of the software states, or modes, that can be mapped to each other using symmetries, their reachsets are cached together. This allows the verification algorithm to utilize computed reachsets across different modes. This increases the cache hit rate and the computational savings. Given the symmetries in the dynamics, the paths of the vehicles, and the obstacles, reachability with symmetry caching can check whether vehicles collide. Overall, this approach achieved a 64% speedup in the safety verification time of a fleet of fixed-wing aircraft using translation, rotation, and permutation symmetries and computed 26% fewer reachsets.[14]

SYMMETRY ABSTRACTION AND REFINEMENT

Symmetry can also be used to simplify or abstract hybrid models before reachability analysis is performed. Consider a set G of symmetry transformations of an ODE model and a set of unsafe states U. A verification algorithm can select, from each set of symmetric initial states, one representative, resulting in a smaller new set of representative initial states. Think of this as a coordinate transformation for states. The reachability analysis from this new smaller initial set in the new coordinates would be faster. The verification algorithm can then transform the computed reachset to get a reachset in the original coordinates and check safety with respect to U.

Another alternative is for the verification algorithm to transform the unsafe set U to the new coordinates and check the safety of the computed reachset in the new coordinates. The new set U' would be the union of the different transformations of U using all of the symmetries in G. Concretely, the set U' would represent the relative positions of U to the symmetric states in the original initial set that are represented by a single state in the new initial set. This approach saves the cache access and storage overhead of the approach described earlier.[15] The first approach implemented using the DryVR tool achieved an order of magnitude further speedup and computed two orders of magnitude fewer reachsets, compared to symmetry caching, in several verification scenarios involving cars cruising and braking on a single-lane road.

Abstractions of hybrid automata can be created similarly from a set of symmetries. The software states or modes that are symmetric to each other are represented by a single representative mode in the abstraction.

A side effect of this symmetry abstraction is that the guards and resets for a discrete transition of the abstract automaton would be the union of the symmetry-transformed guards and resets of the concrete automaton.

Consider the toy scenario in Figure 3(b), representing a car with simple dynamics traversing two possible paths, both from the bottom left (the beginning of segment s_0). The uncertainty in the initial position of the car is shown by the green box. As the car traverses the segments, the reachable states are shown in green. The corresponding hybrid automaton is shown in Figure 3(a). Each software state or the mode of the automaton corresponds to the segment of the path being followed. Discrete transitions represent the car switching from one path segment to another. The switch occurs when the car reaches a region around the endpoints of the segments.

The symmetry abstraction utilizing translation and rotation symmetries of the car is shown in Figure 3(c). Segments s_1–s_5 have the same length, while s_0 is shorter. With the right choice of translation vector and rotation angle for each segment, segments s_1–s_5 can be mapped to the same segment, while that of s_0 would not. Hence, the abstraction has two modes: s_v^0, representing s_0, and s_v^1, representing s_1–s_5. In the abstract automaton, the car starts from an initial set of states around the starting waypoint of segment s_v^0 and follows the x-axis toward the origin. Once the car reaches a region around the origin, its state would be reset nondeterministically to somewhere near the starting waypoint of s_v^1. Then the car would again follow the x-axis toward the origin, and so on. Since the abstraction is smaller, its reachability analysis is faster.

There are two alternatives for checking the safety of the abstract hybrid automaton. The first transforms the reachset of the abstract automaton to overapproximate the reachset of the concrete one and then checks the safety [Figure 3(e)]. The second maps the unsafe set to the abstract state space and checks the safety there [Figure 3(d)]. The five gray rectangles in Figure 3(d) represent the relative positions of the gray unsafe set in the original scenario with respect to each of the five segments represented by s_v^1. The green rectangle represents the relative position of the unsafe set with respect to s_0. The conservativeness of the abstraction may make the abstract automaton unsafe even when the actual system is safe. This can be seen by comparing the reachsets in Figure 3(b) and (e).

The conservativeness of symmetry abstractions can be controlled by refining the abstract automaton.[15] If the reachset of a mode intersects the unsafe set, the refinement algorithm splits the mode into two abstract modes and corresponding unsafe sets. In our toy example, the violet mode, s_v^1, gets split into the modes s_v^1 and s_v^2 in the refined abstraction in Figure 3(f). The refined automaton would be a more accurate abstraction of the concrete automaton. As seen in Figure 3(h), the overapproximation error in Figure 3(e) is eliminated by the refinement. In the worst case, the abstraction is refined back to the original automaton.

We implemented this abstraction-refinement algorithm in a new tool called *SceneChecker*[15] to assess the safety of motion plans in cluttered environments. The scenarios are specified in JavaScript Object Notation files, and the vehicle dynamics and their symmetries are written as Python functions. SceneChecker constructs the concrete

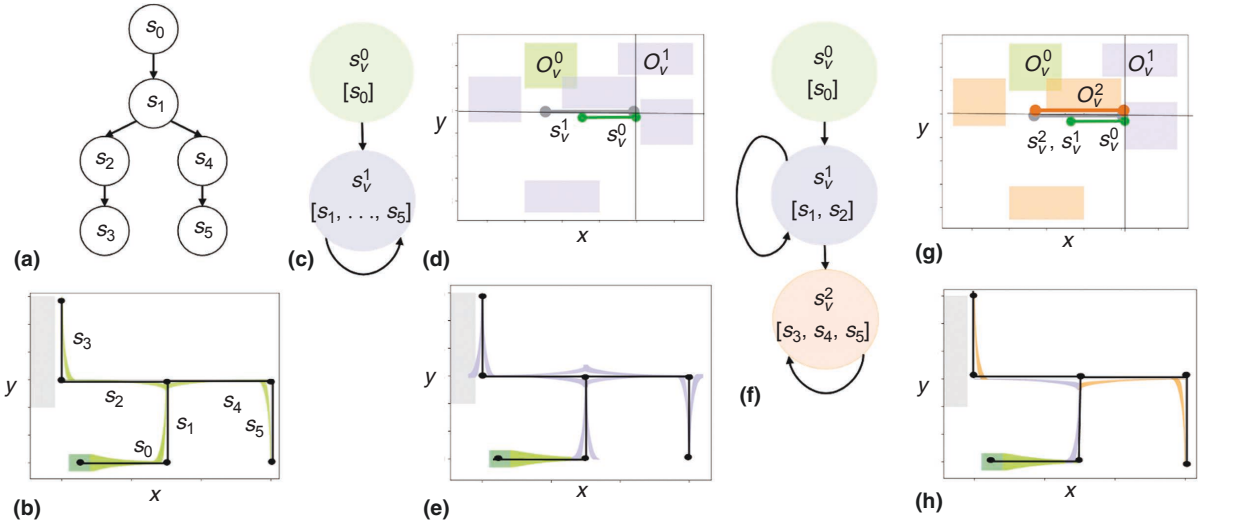

FIGURE 3. An abstraction refinement using symmetries. (a) Hybrid automaton model of a vehicle traversing two paths shown in (b). (b) The five mode transitions for five path segments. (c) Symmetry abstraction of the automaton. (d) Mapping of unsafe set to abstract state space. (e) Reachable states of (d) with spurious unsafety. (f) Refinement of symmetry abstraction. (g) Reachable states of (g) with no spurious unsafety.

hybrid automaton and then uses the provided symmetries to construct the abstraction. It uses DryVR[6] and Flow*[4] for reachability analysis. SceneChecker is able to successfully verify scenarios with quadrotors and cars having neural network controllers executing plans with hundreds of segments in environments with hundreds of polytopic obstacles. It achieves a 14× average speedup in the verification time over direct verification with DryVR and Flow*. On average, it computes two orders of magnitude fewer reachsets than the symmetry caching approach.

These recent advances suggest that symmetry caching and abstractions can boost the performance of CPS verification algorithms in terms of both their running times and their sample efficiency. That said, two research questions must be addressed before these ideas can be integrated in online or offline development processes. First, before applying these methods, we need to check, ideally automatically, that a given transformation is, indeed, a valid symmetry. This can be an easy static analysis problem in special cases but, in general, will require the development of new algorithmic checks. Second, complex autonomous systems interconnect different modules for perception, decision making, and control. Would new types of symmetries emerge by composing the symmetries of perception with those of dynamics? Discovering the right symmetries that maximally boost verification for artificial intelligence-powered CPSs will be an interesting direction for research.

ACKNOWLEDGMENT

This work was supported by a research grant from the National Science Foundation's Formal Methods in the Field program (award 1918531).

REFERENCES

1. C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspa, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018, doi: 10.1145/3188720.
2. N. Chong *et al.*, "Code-level model checking in the software development workflow," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng. Softw. Eng. Pract.*, New York, NY, USA, 2020, pp. 11–20, doi: 10.1145/3377813.3381347.
3. P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proc. 33rd Annu. ACM/IEEE Symp. Logic Comput. Sci. (LICS '18)*, New York, NY, USA, 2018, pp. 13–25, doi: 10.1145/3209108.3209109.
4. X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Proc. Comput. Aided Verification*, 2013, pp. 258–263, doi: 10.1007/978-3-642-39799-8_18.
5. G. Frehse *et al.*, "SpaceEx: Scalable verification of hybrid systems," in *Proc. Comput. Aided Verification*, 2011, pp. 379–395, doi: 10.1007/978-3-642-22110-1_30.
6. C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "DryVR: Data-driven verification and compositional reasoning for automotive systems," in *Computer Aided Verification*, R. Majumdar and V. Kunčak, Eds. Cham, Switzerland: Springer International Publishing, 2017, pp. 441–461.
7. S. Bak and P. S. Duggirala, "HyLAA: A tool for computing simulation-equivalent reachability for linear systems," in *Proc. 20th Int. Conf. Hybrid Syst., Comput. Contr.*, ACM, 2017, pp. 173–178, doi: 10.1145/3049797.3049808.
8. P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *Proc. 2013 Int. Conf. Embedded Softw. (EMSOFT)*, ACM, pp. 1–10, doi: 10.1109/EMSOFT.2013.6658604.
9. S. Mitra, *Verifying Cyber-Physical Systems: A Path to Safe Autonomy*. Cambridge, MA, USA: MIT Press, 2021.
10. A. Donzé and O. Maler, "Systematic simulation using sensitivity analysis," in *Proc. Int. Conf. Hybrid Syst., Comput. Contr.*, 2007, pp. 174–189, doi: 10.1007/978-3-540-71493-4_16.
11. C. Fan and S. Mitra, "Bounded verification with on-the-fly discrepancy computation," in *Proc. 2015 Int. Symp. Automat. Technol. Verification Anal. (ATVA)*, Shanghai, China, vol. 9364, pp. 446–463, doi: 10.1007/978-3-319-24953-7_32.
12. J. N. Maidens and M. Arcak, "Reachability analysis of nonlinear systems using matrix measures," *IEEE Trans. Autom. Control*, vol. 60, no. 1, pp. 265–270, 2015, doi: 10.1109/TAC.2014.2325635.
13. H. Sibai, N. Mokhlesi, and S. Mitra, "Using symmetry transformations in equivariant dynamical systems for their safety verification," in *Proc. 17th Int. Symp. Automat. Technol. Verification Anal.*, Springer-Verlag, 2019, pp. 98–114, doi: 10.1007/978-3-030-31784-3_6.
14. 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. Switzerland: Springer International Publishing, 2020, pp. 173–190.
15. H. Sibai, Y. Li, and S. Mitra, "SceneChecker: Boosting scenario verification using symmetry abstractions," in *Proc. 2021 33rd Int. Conf. Comput. Aided Verification*, Springer-Verlag, pp. 580–594, doi: 10.1007/978-3-030-81685-8_28.

SAYAN MITRA is a professor of electrical and computer engineering at the University of Illinois at Urbana-Champaign, Illinois, 61801, USA. Contact him at mitras@illinois.edu.

HUSSEIN SIBAI received his Ph.D. from the Department of Electrical and Computer Engineering of University of Illinois, Urbana-Champaign, Illinois, 61801, USA. He was advised by Prof. Sayan Mitra. Contact him at sibai2@illinois.edu.