SWERVE : Efficient Runtime Verification of Multi-Agent Systems Using Dynamical Symmetries and Cloud Computing

ABSTRACT

In this paper, we present SWERVE , an open source cloud computing toolkit for efficient run-time collision checking for multi-agent autonomous systems. SWERVE implements a remote server to check safety for different agents running in a shared workspace by using bounded-time reachability analysis. In addition, SWERVE implements a cache to store already computed reachable sets and reuses them to avoid repeated computations. We evaluated SWERVE on several scenarios involving more than 20 cars and drones following independent plans in realistic environments with hundreds of static obstacles. We are able to show that SWERVE is able to properly detect potential collision between agents and static obstacles. In addition, we show that with symmetry and caching, SWERVE is able to obtain 16x average speedup in service response time.

ACM Reference Format:

. 2018. SWERVE : Efficient Runtime Verification of Multi-Agent Systems Using Dynamical Symmetries and Cloud Computing. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The rapid deployment of large fleets of autonomous vehicles in the ground, water, air, and space has necessitated online planning and collision avoidance. Millions of small drones, many fully autonomous, are expected to be deployed in the US national airspace in the coming few years [10, 33]. Governmental agencies such as NASA and the Federal Aviation Agency (FAA) are developing planning and collision avoidance protocols for these drones to prevent their collision during their operations [2, 11, 20, 35]. Dozens of companies are developing autonomous cars and testing them on public roads [18, 47]. 15,000 satellites will be launched in the next decade. A major challenge for all of these technologies is safety, and in particular, collision avoidance.

Checking collision of independent autonomous agents, in the face of known uncertainties, can be viewed as a special type of formal safety verification for hybrid systems. Offline or design-time safety verification for hybrid systems is a well-recognized problem with significant progress in recent years [30, 31, 42]. Runtime verification complements offline verification by monitoring the behavior of the system and its surrounding environment during operation. It can check if the assumptions made in the offline design and verification are satisfied at runtime. If the assumptions are violated, a runtime verification method checks if that will in turn lead to the violation of safety under the current operating conditions. Runtime verification oversees unknown uncertainties that could have not been predicted at the design stage. Runtime verification has to be solved in real-time which imposes stricter constraints on computational time than offline verification, and therefore, is hard to solve on systems with limited power and computing footprints, such as small aerial and underwater drones, satellites, and delivery robots.

In similar edge-computing scenarios, where the device computation power is a bottleneck, cloud computing has proven to be an efficient alternative [15, 24, 39]. Cloud computing allows offloading computational tasks from small devices to servers with superior computational powers. An equally important feature of cloud computing is that the small devices can share data among each other through the server. This has been essential for training the machine learning models that power many of the features in modern mobile devices such as auto-correction and generation of driving trajectories [3]. In robotics, cloud computing has also been significantly utilized in the past decade to avoid expensive computations at the edge [26].

In this paper, we present SWERVE —an open source cloud computing toolkit for efficient runtime collision checking of multi-agent autonomous systems. Given a scenario with multiple autonomous systems operating in the same environment, SWERVE utilizes a remote server to perform bounded-time reachability analysis-based verification of safety by checking absence of collisions for the different agents at runtime. SWERVE uses ROS-based communication between the agents and the server. We use the Robot Operating System (ROS) as it has a community of users, interfaces to simulators like Gazebo [34], CARLA [16], and NVIDIA ISAAC [1], many industrial applications [37], and buy-in from the industry (see for example Baidu's Apollo [5], BlueRobotics' BlueROV2 [40]).

The SWERVE can use any existing reachability algorithm as a subroutine to compute reachable sets, which can then be used to check inter-agent collisions. More importantly, building on results demonstrating effectiveness of symmetries in improving offline verification [7, 12, 19, 21, 43–45], our Verification Server implements a cache to store reachable sets and reuse them to avoid repeated computations. The idea behind Verification Server is the same as that of the tool CacheReach [44, 45]. SWERVE implement symmetry transformation and caching to store already computed reachable sets and reuse those reachable sets for later computation.

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

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

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

1.1 Related Work

There has been significant efforts on developing efficient algorithms and frameworks for runtime verification of autonomous systems [8, 23, 25, 41, 46, 48, 49]. Rozier and Schumann developed the framework R2E2 for runtime verification of autonomous systems [41]. R2E2 is focused on the hardware and software components instead of the agent's continuous dynamics. In [29], Majumdar and Tedrake use a previously computed set of reachable sets for safe online motion planning. They use translation and rotation symmetries to compose the offline computed reachable sets in online manner to over-approximate future behavior and avoid collision. They do not compute new reachable sets at runtime nor do they consider multi-agent systems. The closest work to this paper is CacheReach, developed by Sibai et al. in [44]. They utilize symmetries to accelerate offline safety verification of scenarios with multiple agents. The symmetries are used, as in this paper, to avoid computing reachable sets by transforming previously computed ones. A followup to CacheReach is SceneChecker which implements a symmetry-based abstraction refinement for efficient offline verification for an agent following a predefined plan. However, SceneChecker does not consider multi-agent systems. None of the aforementioned works utilize cloud computing for faster reachability analysis computations. In a recent work, Khaled and Zamani presented the tool *pfaces* that utilizes parallel cloud computing to accelerate formal methods, but not runtime verification [27].

2 TOOLKIT ARCHITECTURE

Swerve is aimed to help online collision checking for multiple agents in a shared *workspace*. The workspace is a 3-dimensional zone, with a map defining static obstacles such as buildings and unsafe corridors as shown in Figure 1. An *agent* is an entity independently moving in this space. Each agent has a local planner that periodically generates a path for the agent to move from its current position to its next destination. However, the planner does not take into account the position or the plans of other agents. The agent's actual motion following the plan is governed by its dynamics and it is affected by various disturbances and uncertainties such as positioning error and wind. The role of the collision checker is to check whether any two such agents in the workspace are likely to collide with each other or with the static obstacles. This collision checking is the key functionality needed in smart intersections [9, 32, 38] and urban air-traffic management (UTM) [2, 11, 20].

Architecture. The architecture of Swerve is shown in Figure 2. It consists of the following components: the Verification Server, a set of agents, and the map of the environment with its static obstacles. The Verification Server has two ROS Services: *initialize* and

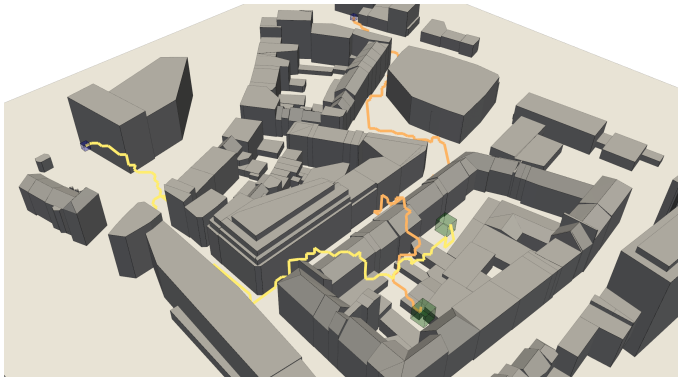

Figure 1: 3d map of Cologne with two quadrotors. Each agent is supposed to move from start position (small blue rectangle) to goal (green) following plan segments shown by the orange and yellow lines.

verifyQuery, which are presented to the user and the agents as remote procedure calls. The Verification Server has *verify*, which calls the Reachability Subroutine to compute the queried reachable set or retrieves it from the cache. In addition, *verify* checks if any of the intersections between the computed reachable set with the reachable sets of other agents or the static obstacles is non-empty. The map of the environment is provided to the Verification Server through the *initialize* service and will be stored, together with the computed reachable sets for all agents, in the *UnsafeSet*. The agents interact with the Verification Server through the *verifyQuery* service. A query to the *verifyQuery* service is fulfilled by calling the *verify*, which in turn uses the static obstacle map received from the *initialize* service, the cache of reachable sets, the Reachability Subroutine and the collision checker, to answer the query.

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

A query to the *verifyQuery* ROS Service includes its identifier *id*, the segment to be followed s, the set of possible current states, the dynamics of the agent, the symmetry map Φ, and the look-ahead time T. The dynamics of the agent can either be specified by ordinary differential equations (ODE) or by a black box simulator. The details about the symmetry map Φ will be discussed in Section 3.2

The Verification Server checks for collision by computing the reachable sets [31] of all querying agents. Given the set of possible current states of the agent Θ, the segment in the workspace it plans to follow s, and the look-ahead time T, the reachable set over-approximates the set of states that the agent might reach within time T starting from any state in Θ and following the segment s. Therefore, if the reachable set of an agent is not intersecting with any static obstacle or reachable set of another agent, then that agent is guaranteed to not reach a state in which it collides with static

obstacles or other agents within time T. The collision checker in the Verification Server stores the reachable set of the most recent query of each agent. The collision checker checks if a newly computed reachable set of an agent intersects with the stored reachable sets of the other agents or with the static obstacles. The Verification Server then replies with a Safe or Unsafe answer to the agent's query as well as with the computed reachable set. More details about the implementation of the server is discussed in Section 3.

Initialization. The server has to be initialized with the map of the environment listing the static obstacles. This is done in SWERVE by calling the *initialize* ROS Service. The static obstacle map can be specified in three possible ways: (1) a 3d city model in the `CityJSON` format [28], (2) a list of vertices defining polytopes in space-time, or (3) a list of linear inequalities also defining polytopic obstacles.

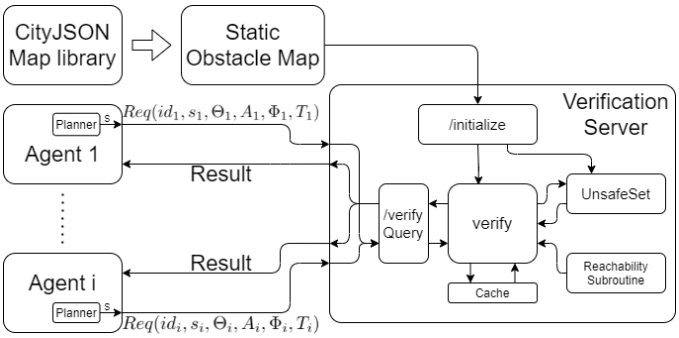

Figure 2: The figure shows the architecture of SWERVE .

3 VERIFICATION SERVER

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

3.1 *verify* overview

A sketch of the verification algorithm *verify* implemented on the server is shown in Algorithm 1. The execution time of *verify* is the major contributor to the response time described in Section 2. *verify* uses a *cache* and a *lock*. The *lock* is used to prevent multiple service calls from modifying the shared contents in the *cache* and the *UnsafeSet* at the same time. Each call to *verify* will have to obtain the *lock* first before executing the service (line 2) and will release the lock only after finishing all the computations (line 7 and 11). Therefore, only one service request can be fulfilled at a time.

In line 5, *verify* calls the *verifySegment* function described in Algorithm 2. The *verifySegment* function first obtains the reachable set for the agent starting from Θ following segment s for the given time bound T. Then, it checks for collision by checking the non-emptiness of the intersections between the obtained reachable set of the querying agent and the static obstacles and the stored reachable sets of the other agents in *UnsafeSet*. The *verifySegment* function then returns the verification result. If the verification result is *safe* or *unsafe*, *verify* returns it (line 8). Otherwise, *verify* calls the refine function, which is described in Section 3.4, to obtain a more accurate reachable set from the cache (line 9). If the *cache* did not store such a reachable set, *verify* sets its c flag to True. That means that in the next call to *verifySegment* in the following iteration of the for loop, the reachable set is computed using the reachability subroutine instead of retrieved from the cache (line 10). The for loop terminates after the refine threshold is reached.

Algorithm 1 $verify(id, s, \Theta, A, T, \Phi)$

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

3.2 Symmetry and Caching

The reachability subroutine in SWERVE can be any of the existing tools such as Flow* [13], CORA [4], DryVR [17], C2E2 [14], and HyLaa [6]. Our current implementation of SWERVE uses DryVR. In addition, SWERVE utilizes the underlying symmetries in the physics models of the vehicles and a *cache* to speed-up the collision checks.

Symmetry. For a dynamical system described by the differential equation (1) and a corresponding symmetry map Φ, if x is a solution to (1), then $\Phi(x)$ will also be a solution to (1) [45].

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

where $x \in \mathbb{R}^n$ and $s \in \mathbb{R}^m$. In [44], it has been shown how symmetries can be used to transform previously computed reachable sets to new ones starting from symmetric initial sets of states and following symmetric segments. In SWERVE , we require the user to provide a family of symmetries for the agents' dynamics. The family of symmetries would be a set of pairs of maps $\Phi = \{(\gamma_s, \rho_s)\}_{s \in S}$, where S is the set of segments that might be followed by any of the agents in the workspace. For any $s \in S$, ρ_s transforms the agent's planned segment s to a representative symmetric segment s_v and γ_s transforms a given agent's state or trajectory following s to its trajectory following s_v. This input requirement is the same as that of SceneChecker in [43] and CacheReach in [44].

CACHING. SWERVE uses a cache to store computed reachable sets. The implementation of the cache is similar to the one used in CacheReach [43]. For a given query by an agent to the Verification Server to compute the reachable set following segment s, the key to the cache is the representative segment s_v of s, i.e. $s_v := \rho_s(s)$. Since a scenario might have agents with different dynamics, an identifier to the agents' dynamics A is added as part of the key of the cache. After the cache entry is selected, the initial set of states Θ in the query is transformed using γ_s resulting in a new symmetric set of states Θ_v.

In a given entry in $cache$, there might be multiple stored reachable sets starting from different sets of states. The tool CacheReach [44], returns the $\gamma_{s_v}^{-1}$-transformation of the union of all reachable sets stored at that entry with initial sets intersecting Θ_v. This might result in over-approximation errors in the retrieved reachable set [44]. SWERVE tackles this problem by a refinement algorithm described in Section 3.4.

3.3 *verifySegment* overview

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

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

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

The algorithm checks the intersection between r and static obstacles O (line 11). If the intersection is non-empty and r is retrieved from the cache, the function will return *unknown*, otherwise the function will return *unsafe*. The next step is to update the reachable sets in R, which is the dynamic part of *UnsafeSet* (line 14). Since R stores the reachable sets corresponding to the current behaviors of the agents, it can naturally be used to predict near-future collision between agents. Accordingly, *verifySegment* computes the intersection of r with all the reachable sets stored in R (line 15). If any non-empty intersection exists and r was retrieved from *cache*, *verifySegment* returns *unknown*. If r was computed using the reachability subroutine, *verifySegment* returns *unsafe* instead.

3.4 Refinement

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

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

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

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

Algorithm 2 $verifySegment(id, s, \Theta, A, T, \Phi = \{(\gamma_s, \rho_s)\}, c)$

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

4 EXPERIMENTAL EVALUATION

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

Agents and dynamics. We use two types of agents: a ground vehicle with 3D bicycle dynamics and 2 inputs (C), and a 6D quadrotor (Q) with 6 inputs. The bicycle is controlled using a PD controller [36], which is translation and rotation symmetric by design. The quadrotor is controlled by a Neural Network controller from the Verisig paper [22]. This controller is translation symmetric since it takes the difference between agent state and segment been followed. Furthermore, we modified the Neural Network controller so that it's also rotation symmetric. Therefore, we are using both translation and rotation symmetries with caching.

Scenarios. We experimented with a number of different scenarios. We name scenarios as $Mapi{-}D{-}N$, where (1) $Mapi$ indicates the static obstacle map used; maps with higher number are more complex and have more complex plans. $Map4$ is a partial 3D map of Cologne in CityJSON format. (2) $D \in \{2D, 3D\}$ indicates the dimension of the workspace. (3) N is the number of agents in the scenario; different types of agents may be mixed. We run the above scenarios with different time horizons to vary the load on the verification server.

Recall that the plan from initial position to goal for each agent is generated independently (using a RRT planner). Therefore, the plans from different agents will indeed intersect in space and time. While running a scenario, the load of the Verification Server can

Table 1: Swerve response time (in seconds) < travel time on the average, ≪ using symmetry and caching; in both cases using DryVR. The symmetry map Φ is TR. Columns: Number of agents (N), agents types (Agent), number of obstacles ($|O|$), total number of plan segments ($|S|$), number of calls to the reachability subroutine (Rc), average response time for each service call (ARt), max response time (MRt), max response time of 90% of agents ($MRt-90$), and the the average traversal time for each path segment (ASt).

				Swerve					Swerve Without Caching		
Sc.	Agent	$\|O\|$	$\|S\|$	Rc	ARt	MRt-90	MRt	ASt	ARt	MRt	ASt
Map1-2D-50	C	100	300	6	0.25	0.25	8.07	15.61	75.21	90.33	97.80
Map2-2D-12	C	236	1457	133	0.95	2.22	4.95	18.30	3.63	10.76	21.00
Map2-2D-17	C	236	1457	197	1.14	2.93	8.62	19.35	9.93	13.41	27.84
Map3-2D-34	C	462	4810	616	2.30	5.48	14.49	21.40	35.72	51.22	54.89
Map1-3D-50	Q	100	300	1	0.18	0.25	2.19	26.84	71.16	81.27	96.63
Map2-3D-17	Q	236	1457	1	0.21	0.30	1.57	16.65	6.48	8.34	21.83
Map4-3D-20	Q	252	2724	25	0.25	0.34	2.67	17.06	2.51	5.15	22.91
Map4-3D-20	C&Q	252	2724	163	0.71	2.09	8.19	18.15	10.84	30.05	45.51

Table 2: Response time (ARt) with different agent densities in space-time. Besides the values shown in Table 1, this table also shows the time horizon for each segments (Ts).

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

be determined by the number of agents in the scenario together with the time horizon for each plan segment. As the number of agents increase or the time horizon for each plan segment decrease, the Verification Server will be called more frequently and therefore adding loads to the Verification Server.

The plans are fed to the agents segment by segment, which means each agent receives the new segment to follow when it finishes following its current segment. An agent will wait and retry following the segment after 15 seconds when the plan is detected by *verify* to be unsafe. The initial set of states of verification will be an L_∞ ball centered at the agent's current state representing sensor and state estimation uncertainty. The reachability tool DryVR [17] is used in the $VerificationServer$ to compute reachable sets. In addition, we add a constant bloating factor to the reachable sets computed by DryVR to simulate the effect of sensor noise while the agent is execute the plan.

We ran each of the scenarios with and without symmetry and caching. The results are shown in Table 1. From these results we make the following observations.

Swerve *is able to perform online multi-agent collision checks.* Table 1 shows that the system is able to check potential collisions and safe plans in both 2D and 3D scenarios with different types of agents (Col 2). In the best case (Row 4), even with 50 agents, the average response time for each service call is only 0.18s. Figure 3 is showing the visualization of computed reachable sets for 5 agents in a 30s time interval. The reachable sets shown in orange represent the reachable sets computed in the first 10s of the 30s time interval. The reachable sets shown in yellow and light yellow represent the reachable sets computed in the 10-20s interval and the 20-30s interval respectively. The reachable set shown in red is represents an unsafe reachable set detected by the Verification Server. We can see that, the tool is able to detect a potential collision between Agent3 and Agent4, and therefore, marks the plan for Agent4 to be unsafe. On the other hand, although the reachable sets for Agent1 and Agent2 occupy the same physical space, since they are disjoint in time, the Verification Server decided that their planned segments are safe to follow.

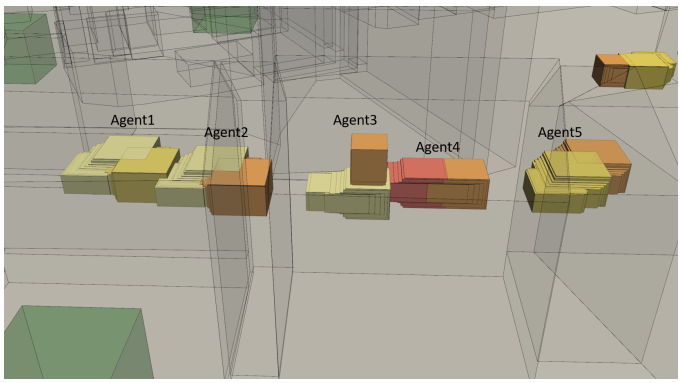

Figure 3: The picture shows the computed reachable set of 5 agents in scenario Comp3D-20 in a 30s time interval.

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

In addition, we can observe that as the number of agents in the scenario increases, savings from using symmetry-based caching increases. If we compare the average response time for scenario Map2-2D-12 and scenario Map2-2D-17, we can see that as the number of agents increase from 12 to 17, the saving from using symmetry-based caching increases from 3.8x to 8.7x. A similar trend can also be observed in most other scenarios.

From Table 1, we observe that with symmetry and caching, the number of calls to the reachability subroutine is much smllaer than the total number of segments in the scenario since many reachable sets were obtained by transforming stored reachable sets in $cache$. In the extreme case, as shown in scenario Map1-3D-50 and Map2-3D-17, only one reachable set was computed and the rest of the reachable sets were retrieved from $cache$.

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

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

The performance of Verification Server may decrease as the load increase. To further understand the performance of Verification Server, we varied its load applied by running scenario map2-2D with different number of agents and different time horizons for each of the planned segments. The results are shown in Table 2. From row 1-3 in Table 2, we can clearly see that as the number of agents in the workspace increase from 6 to 17, the average response time increases gradually from 0.71s to 1.14s. The max response time and 90% max response time increase as well. This decrease in performance happens not only because Verification Serverwill have to fulfill more $verifyQuery$ requests, but also since each $verifyQuery$ request may take longer to fulfill since the collision checker have to check more intersection with other agents.

From row 1, 3, and 4 in Table 2, we can see that as we decrease Ts from 20 to 10, the average response time increases from 0.90s to 1.42s, which also shows the decrease of performance in the Verification Server.

Row 5 in Table 2 shows an extreme case for the Verification Server. In this case, the number of agents is too large or Ts is too small so that the verification request is generated faster than the Verification Server can fulfill. In this case, we can see the response time increasing significantly and the verification tasks can on longer be performed in real time.

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

5 DISCUSSION AND FUTURE WORK

We present Swerve , an open source toolkit for online safety checking of multi-agent systems. It provides a ROS-based communication framework and an implementation of reachable set caching for rapid online collision checking. We apply the toolkit to scenarios with multiple agents with different dynamics. Our experiments suggest that Swerve is a promising approach for online checking of inter-agent and static obstacle collisions. In addition, we are able to identify the influence of load on the performance of Swerve and reasoning about our observation. In the future, it would be beneficial to explore how the sequence of handling verification requests can influence the performance of Swerve and to further boost the verification speed of the server. Finally, it will be interesting to experiment with Swerve in scenarios simulated in photo-realistic simulators and in real world deployments.

REFERENCES

[1] [n.d.]. ROS Bridge¶. https://docs.nvidia.com/isaac/isaac/packages/ros_bridge/doc/ros_bridge.html

[2] [n.d.]. unmanned aircraft system traffic management (utm) concept of operations version 2.0. ([n. d.]). https://www.faa.gov/uas/research_development/traffic_management/media/UTM_ConOps_v2.pdf

[3] 2009. The bright side of sitting in traffic: Crowdsourcing Road Congestion Data. https://googleblog.blogspot.com/2009/08/bright-side-of-sitting-in-traffic.html

[4] M. Althoff. 2015. An Introduction to CORA 2015. In *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems.*

[5] ApolloAuto. [n.d.]. ApolloAuto/apollo: An open autonomous driving platform. https://github.com/ApolloAuto/apollo

[6] Stanley Bak and Parasara Sridhar Duggirala. 2017. HyLAA: A Tool for Computing Simulation-Equivalent Reachability for Linear Systems. In *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control* (Pittsburgh, Pennsylvania, USA) *(HSCC '17)*. Association for Computing Machinery, New York, NY, USA, 173–178. https://doi.org/10.1145/3049797.3049808

[7] Stanley Bak, Zhenqi Huang, Fardin Abdi Taghi Abad, and Marco Caccamo. 2015. Safety and Progress for Distributed Cyber-Physical Systems with Unreliable Communication. *ACM Trans. Embed. Comput. Syst.* 14, 4, Article 76 (Sept. 2015), 22 pages. https://doi.org/10.1145/2739046

[8] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications.* Springer International Publishing, Cham, 135–175. https://doi.org/10.1007/978-3-319-75632-5_5

[9] Orly Barzilai, Nadav Voloch, Alon Hasgall, Orna Lavi Steiner, and Niv Ahituv. 2018. Traffic control in a smart intersection by an algorithm with social priorities. *Contemporary Engineering Sciences* 11, 31 (2018), 1499–1511.

[10] Wells C. Bennett. 2016. Unmanned at any speed: Bringing drones into our national airspace. https://www.brookings.edu/research/unmanned-at-any-speed-bringing-drones-into-our-national-airspace/

[11] Suda Bharadwaj, Tichakorn Wongpiromsarn, Natasha Neogi, Joseph Muffoletto, and Ufuk Topcu. 2021. Minimum-Violation Traffic Management for Urban Air Mobility. In *NASA Formal Methods*, Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez (Eds.). Springer International Publishing, Cham, 37–52.

[12] Manuela Bujorianu and Joost-Pieter Katoen. 2008. Symmetry reduction for stochastic hybrid systems. In *2008 47th IEEE Conference on Decision and Control : CDC ; Cancun, Mexico, 9 - 11 December 2008. - T. 1.* IEEE, Piscataway, NJ, 233–238. https://doi.org/10.1109/CDC.2008.4739086 Nebent.: Proceedings of the 47th IEEE Conference on Decision and Control.

[13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. 2013. Flow*: An analyzer for non-linear hybrid systems. In *CAV*. Springer, 258–263.

[14] Xin Chen, Sriram Sankaranarayanan, and Erika Abraham. 2015. Flow* 1.2: More Effective to Play with Hybrid Systems. In *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems (EPiC Series in Computing, Vol. 34)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 152–159. https://doi.org/10.29007/1w4t

[15] L. Minh Dang, Md. Jalil Piran, Dongil Han, Kyungbok Min, and Hyeonjoon Moon. 2019. A Survey on Internet of Things and Cloud Computing for Healthcare. *Electronics* 8, 7 (2019). https://doi.org/10.3390/electronics8070768

[16] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*. 1–16.

[17] Chuchu Fan, Bolun Qi, Sayan Mitra, and Mahesh Viswanathan. 2017. DryVR: Data-Driven Verification and Compositional Reasoning for Automotive Systems. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 441–461.

[18] Andrew J. Hawkins. 2021. Waymonbsp;and Cruise dominated autonomous testing in California in the first year of the pandemic. https://www.theverge.com/2021/2/11/22276851/california-self-driving-autonomous-cars-miles-waymo-cruise-2020

[19] Martijn Hendriks, Gerd Behrmann, Kim Larsen, Peter Niebert, and Frits Vaandrager. 2004. Adding Symmetry Reduction to UPPAAL.

[20] Chiao Hsieh, Hussein Sibai, Hebron Taylor, and S. Mitra. 2020. Unmanned air-traffic management (UTM): Formalization, a prototype implementation, and performance evaluation. *ArXiv* abs/2009.04655 (2020).

[21] C. Norris Ip and David L. Dill. 1993. 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 '93)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 97–111.

[22] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 2019. 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*. 169–178.

[23] Stefan Jaksic, Ezio Bartocci, Radu Grosu, and Dejan Nickovic. 2018. An Algebraic Framework for Runtime Verification. arXiv:1802.03775 [cs.LO]

[24] N. Joshi and S. Shah. 2019. A Comprehensive Survey of Services Provided by Prevalent Cloud Computing Environments. In *Smart Intelligent Computing and Applications*, Suresh Chandra Satapathy, Vikrant Bhateja, and Swagatam Das (Eds.). Springer Singapore, Singapore, 413–424.

[25] Sungjoo Kang, Ingeol Chun, and Hyeon-Soo Kim. 2019. Design and Implementation of Runtime Verification Framework for Cyber-Physical Production Systems. *Journal of Engineering* 2019 (2019), 2875236. https://doi.org/10.1155/2019/2875236

[26] Ben Kehoe, Sachin Patil, Pieter Abbeel, and Ken Goldberg. 2015. A Survey of Research on Cloud Robotics and Automation. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 398–409. https://doi.org/10.1109/TASE.2014.2376492

[27] Mahmoud Khaled and Majid Zamani. 2021. Cloud-Ready Acceleration of Formal Method Techniques for Cyber–Physical Systems. *IEEE Design Test* 38, 5 (2021), 25–34. https://doi.org/10.1109/MDAT.2020.3034048

[28] Hugo Ledoux, Ken Arroyo Ohori, Kavisha Kumar, Balázs Dukai, Anna Labetski, and Stelios Vitalis. 2019. CityJSON: a compact and easy-to-use encoding of the CityGML data model. *Open Geospatial Data, Software and Standards* 4, 1 (17 Jun 2019), 4. https://doi.org/10.1186/s40965-019-0064-0

[29] Anirudha Majumdar and Russ Tedrake. 2017. Funnel Libraries for Real-Time Robust Feedback Motion Planning. arXiv:1601.04037 [cs.RO]

[30] James Bret Michael, Doron Drusinsky, and Duminda Wijesekera. 2021. Formal Verification of Cyberphysical Systems. *Computer* 54, 9 (2021), 15–24. https://doi.org/10.1109/MC.2021.3055883

[31] Sayan Mitra. 2021. *Verifying cyber-physical systems a path to safe autonomy*. The MIT Press.

[32] Tanja Niels, Nikola Mitrovic, Klaus Bogenberger, Aleksandar Stevanovic, and Robert L Bertini. 2019. Smart intersection management for connected and automated vehicles and pedestrians. In *2019 6th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*. IEEE, 1–10.

[33] U.s. Department of Transportation. 2013. *Integration of Civil Unmanned Aircraft Systems (UAS) in the National Airspace System (NAS) roadmap*. U.S. Department of Transportation, Federal Aviation Administration.

[34] Osrf. [n.d.]. osrf/gazebo: Open source robotics simulator. https://github.com/osrf/gazebo

[35] Michael P. Owen, Adam Panken, Robert Moss, Luis Alvarez, and Charles Leeper. 2019. ACAS Xu: Integrated Collision Avoidance and Detect and Avoid Capability for UAS. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*. 1–10. https://doi.org/10.1109/DASC43569.2019.9081758

[36] Brian Paden, Michal Cap, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. 2016. A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. arXiv:1604.07446 [cs.RO]

[37] Neel V. Patel. 2021. NASA's next lunar rover will run open-source software. https://www.technologyreview.com/2021/04/12/1022420/nasa-lunar-rover-viper-open-source-software

[38] Mahmoud Pourmehrab, Lily Elefteriadou, and Sanjay Ranka. 2017. Smart intersection control algorithms for automated vehicles. In *2017 Tenth International Conference on Contemporary Computing (IC3)*. IEEE, 1–6.

[39] Medara Rambabu, Swati Gupta, and Ravi Shankar Singh. 2021. Data Mining in Cloud Computing: Survey. In *Innovations in Computational Intelligence and Computer Vision*, Manoj Kumar Sharma, Vijaypal Singh Dhaka, Thinagaran Perumal, Nilanjan Dey, and João Manuel R. S. Tavares (Eds.). Springer Singapore, Singapore, 48–56.

[40] Blue Robotics. 2016. BlueROV2. *Datasheet, June* (2016).

[41] Kristin Yvonne Rozier and Johann Schumann. 2017. R2U2: Tool Overview. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (Kalpa Publications in Computing, Vol. 3)*, Giles Reger and Klaus Havelund (Eds.). EasyChair, 138–156. https://doi.org/10.29007/5pch

[42] Gesina Schwalbe and Martin Schels. 2020. A survey on methods for the safety assurance of machine learning based systems. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*.

[43] Hussein Sibai, Yangge Li, and Sayan Mitra. 2021. SceneChecker: Boosting Scenario Verification using Symmetry Abstractions. arXiv:2011.10713 [eess.SY]

[44] Hussein Sibai, Navid Mokhlesi, Chuchu Fan, and Sayan Mitra. 2020. Multi-agent Safety Verification Using Symmetry Transformations. In *Tools and Algorithms for the Construction and Analysis of Systems*, Armin Biere and David Parker (Eds.). Springer International Publishing, Cham, 173–190.

[45] Hussein Sibai, Navid Mokhlesi, and Sayan Mitra. 2019. Using Symmetry Transformations in Equivariant Dynamical Systems for Their Safety Verification. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 98–114.

[46] Hoang-Dung Tran, Luan Viet Nguyen, Patrick Musau, Weiming Xiang, and Taylor T. Johnson. 2019. Decentralized Real-Time Safety Verification for Distributed Cyber-Physical Systems. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE'19)*, Jorge A. Pérez and Nobuko Yoshida (Eds.). Springer International Publishing, Cham, 261–277.

[47] WAYMO. 2020. Waymo Safety Report.

[48] Eleni Zapridou, Ezio Bartocci, and Panagiotis Katsaros. 2020. Runtime Verification of Autonomous Driving Systems in CARLA. In *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12399)*, Jyotirmoy Deshmukh and Dejan Nickovic (Eds.). Springer, 172–183. https://doi.org/10.1007/978-3-030-60508-7_9

[49] Xi Zheng, Christine Julien, Rodion Podorozhny, and Franck Cassez. 2015. BraceAssertion: Runtime Verification of Cyber-Physical Systems. In *2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems*. 298–306. https://doi.org/10.1109/MASS.2015.15