Verse: A Python Library for Reasoning About Multi-agent Hybrid System Scenarios

Yangge Li(✉), Haoqing Zhu(✉), Katherine Braught, Keyi Shen, and Sayan Mitra(✉)

Coordinated Science Laboratory, University of Illinois Urbana-Champaign, Champaign, USA
{li213,haoqing3,braught2,keyis2,mitras}@illinois.edu

Abstract. We present the Verse library with the aim of making hybrid system verification more usable for multi-agent scenarios. In Verse, decision making agents move in a map and interact with each other through sensors. The decision logic for each agent is written in a subset of Python and the continuous dynamics is given by a black-box simulator. Multiple agents can be instantiated, and they can be ported to different maps for creating scenarios. Verse provides functions for simulating and verifying such scenarios using existing reachability analysis algorithms. We illustrate capabilities and use cases of the library with heterogeneous agents, incremental verification, different sensor models, and plug-n-play subroutines for post computations.

Keywords: Scenario verification · Reachability · Hybrid Systems

1 Introduction

Automatic verification tools for hybrid systems have been used to analyze linear models with thousands of continuous dimensions [1,5,6] and nonlinear models inspired by industrial applications [6,14]. The state of the art and the challenges are discussed in a recent survey [11]. Despite the potentially large user base, currently this technology is inaccessible without formal methods training. Automatic hybrid verification tools [10,13,17,25,31] require the input model to be written in a tool-specific language. Tools like C2E2 [15] attempt to translate models from Simulink/Stateflow, but the language-barrier goes down to the underlying math models. The verification algorithms are based on variants of the hybrid automaton [3,21,24] which requires the discrete states (or *modes*) to be spelled out explicitly as a graph, with guards and resets labeling the transitions. We discuss related works in more detail in Sect. 6, including recently developed libraries that address usability barrier [5,7,8].

This research was funded in part by NASA University Leadership Initiative grant (80NSSC22M0070) Robust and Resilient Autonomy for Advanced Air Mobility.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13964, pp. 351–364, 2023.
https://doi.org/10.1007/978-3-031-37706-8_18

In this paper, we present Verse, a Python library that aims to make hybrid technologies more usable for multi-agent scenarios. The key features implemented are as follows: (1) In Verse, users write scenarios in Python. User-defined functions can be used to create complex *agents*, invariant requirements can be written as `assert` statements, and scenarios can be created by instantiating multiple agents, all using the standard Python syntax. Verse parses this scenario and constructs an internal representation of the hybrid automaton for simulation and analysis. (2) Verse introduces an additional structure, called *map,* for defining the modes and the transitions of a hybrid system. Map contains *tracks* that can capture geometric objects (e.g., lanes or waypoints) that make it possible to create new scenarios just by instantiating agents on new maps. With track modes, users do not have to explicitly write different modes for a vehicle following different waypoint segments. Finally, (3) Verse comes with functions for simulation and safety verification via reachability analysis. Developers can implement new functions, plug-in existing tools, or implement advanced algorithms, e.g., for incremental verification. In this tool paper, we illustrate use cases with heterogeneous agents and different scenario setups, the flexibility of plugging in different reachability algorithms and the ability to develop more advanced algorithms (Sect. 5). Verse is available at https://github.com/AutoVerse-ai/Verse-library.

2 Overview of Verse

We will highlight the key features of Verse with an example. Consider two drones flying along three parallel ∞-shaped tracks that are vertically separated in space (shown by black lines in Fig. 1). Each drone has a simple collision avoidance logic: if it gets too close to another drone on the same track, then it switches to either the track above or the one below. A drone on `T1` has both choices. Verse enables creation, simulation, and verification of such scenarios using Python, and provides a collection of powerful functions for building new analysis algorithms.

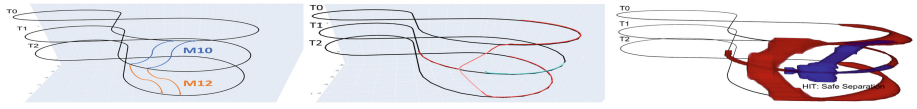

Fig. 1. *Left:* A 3-d ∞-shaped map with example track mode labels. *Center:* Simulation of a red drone nearing the blue drone on `T1` and nondeterministically moving to `T0` or `T2`. Both branches are computed by Verse's `simulate` function. *Right:* Computed reachable sets of the two drones cover more possibilities: either drones can switch tracks when they get close. All four branches are explored by Verse. The branch for blue drone moving downwards violates safety as it may collide with the red drone following T1.

Creating Scenarios. Agents like the drones in this example are described by a *simulator* and a *decision logic* in an expressive subset of Python (see code in Fig. 2 and [26] for more details). The decision logic for an ego agent takes as input its current state and the (observable) states of the other agents, and

updates the discrete state or the *mode* of the ego agent. For example, in lines 41–43 of Fig. 2 an agent updates its mode to begin a track change if there is any agent near it. It may also update the continuous state of the ego agent. The *mode* of an agent, as we shall see later in Sect. 3, has two parts—a *tactical mode* corresponding to agent's decision or discrete state, and a *track mode* that is determined by the map. Using the `any` and `all` functions, the agent's decision logic can quantify over other agents in the scene. User defined functions are also allowed (`is_close`, Fig. 2 line 41). Verse will parse this decision logic to create an internal representation of the transition graph of the hybrid model with guards and resets. The simulator can be written in any language and is treated as a black-box[1]. For the examples discussed in this paper, the simulators are also written in Python. Safety requirements can be specified using `assert` statements (see Fig. 5).

```
38    def decisionLogic(ego: State, others: List[State], track_map):
39      next = copy.deepcopy(ego)
40      if ego.tactical_mode == TacticalMode.Normal:
41        if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
          ↪  others):
42          next.tactical_mode = TacticalMode.MoveDown
43          next.track_mode = track_map.Tg(ego.track_mode, ego.tactical_mode,
            ↪  TacticalMode.MoveDown)
44        if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
          ↪  others):
45          next.tactical_mode = TacticalMode.MoveUp
46      # ...
47      if ego.tactical_mode == TacticalMode.MoveUp:
48        if in_interval(track_map.altitude(ego.track_mode)-ego.z, -1, 1):
49          next.tactical_mode = TacticalMode.Normal
50          next.track_mode = track_map.Tg(ego.track_mode, ego.tactical_mode,
            ↪  TacticalMode.Normal)
51      # ...
```

Fig. 2. Decision Logic Code Snippet from drone_controller.py.

Maps and Sensors. The map of a scenario specifies the tracks that the agents can follow. While a map may have infinitely many tracks, they fall in a finite number of *track modes*. For example, in this ∞-shaped map, each layer is assigned to a track mode (`T0-2`) and all the tracks between each pair of layers are also assigned to a track mode (`M10`, `M01` etc.). When an agent makes a decision and changes its tactical mode, the map object determines the new track mode for the agent. The map abstraction makes scenarios succinct and enables portability of agents across different maps. Besides creating from scratch, Verse provides functions for generating map objects from OpenDRIVE [4] files.

[1] This design decision for Verse is relatively independent. For reachability analysis, Verse currently uses black-box statistical approaches implemented in DryVR [14] and NeuReach [35]. If the simulator is available as a white-box model, such as differential equations, then Verse could use model-based reachability analysis.

The *sensor* function defines which variables from an agent are visible to other agents. The default sensor function allows all agents to see all variables; we discuss how the sensor function can be modified to include bounded noise in Sect. 5. A map, a sensor and a collection of (compatible) agents together define a scenario object (Fig. 3). In the first few lines, the drone agents are created, initialized, and added to the scenario object. A scenario can have heterogeneous agents with different decision logics.

```
scenario = Scenario()
drone_red = DroneAgent('drone_red', file_name='drone_controller.py')
drone_red.set_initial([init_l_1, init_u_1],(CraftMode.Normal, TrackMode.T1))
scenario.add_agent(drone_red)
drone_blue = DroneAgent('drone_blue', file_name='drone_controller.py')
scenario.add_agent(drone_blue)
# ...
scenario.set_map(M6())
scenario.set_sensor(BaseSensor())
#traces = scenario.simulate(40, time_step)
traces = scenario.verify(40, time_step)
```

Fig. 3. Scenario specification snippet.

Simulation and Reachability. Once a scenario is defined, Verse's `simulate` function can generate simulation(s) of the system, which can be stored and plotted. As shown in Fig. 1(Center), a simulation from a single initial state explores all possible branches that can be generated by the decision logics of the interacting agents, upto a specified time horizon. Verse verifies the safety assertions of a scenario by computing the over-approximations of the *reachable sets* for each agent, and checking these against the predicates defined by the assertions. Figure 1(Right) visualizes the result of such a computation performed using the `verify` function. In this example, the safety condition is violated when the blue drone moves downward to avoid the red drone. The other branches of the scenario are proved to be safe. The `simulate` and `verify` functions save a copy of the resulting execution tree, which can be loaded and traversed to analyze the sequences modes and states that leads to safety violations. Verse makes it convenient to plug in different reachability subroutines. It also provides powerful functions to implement advanced verification algorithms, such as incremental verification.

3 Scenarios in Verse

A *scenario* in Verse is specified by a map, a collection of agents in that map, and a sensor function that defines the part of each agent visible to other agents. We describe these components below, and in Sect. 4, we will discuss how they formally define a hybrid system.

Tracks, Track Modes, and Maps. A *workspace* W is an Euclidean space in which the agents reside (For example, a compact subset of $\mathbb{R}^2$ or $\mathbb{R}^3$). An agent's continuous dynamics makes it roughly follow certain continuous curves

in W, called *tracks*, and occasionally the agent's decision logic changes the track. Formally, a *track* is simply a continuous function $\omega : [0, 1] \to W$, but not all such functions are valid tracks. A map $\mathcal{M}$ defines the set of tracks $\Omega_{\mathcal{M}}$ it permits. In a highway map, some tracks will be aligned along the lanes while others will correspond to merges and exits.

We assume that an agent's decision logic does not depend on exactly which of the infinitely many tracks it is following, but instead, it depends only on which type of track it is following or the *track mode*. In the example in Sect. 2, the track modes are `T0`, `T1`, `M01`, etc. Every (blue) track for transitioning from point on T0 to the corresponding point on T1 has track mode `M01`. A map has a finite set of track modes $L_{\mathcal{M}}$ and a labeling function $V_{\mathcal{M}} : \Omega_{\mathcal{M}} \to L_{\mathcal{M}}$ that maps the track to a track mode. It also has a mapping $g_{\mathcal{M}} : W \times L_{\mathcal{M}} \to \Omega_{\mathcal{M}}$ that maps a track mode and a specific position in the workspace to a specific track.

Finally, a Verse agent's decision logic can change its internal mode or *tactical mode* P (E.g., `Normal` to `MoveUp`). When an agent changes its tactical mode, it may also update the track it is following and this is encoded in the track graph function: $T_{g_{\mathcal{M}}} : L_{\mathcal{M}} \times P \times P \to L_{\mathcal{M}}$ which takes the current track mode, the current and the next tactical mode, and generates the new track mode the agent should follow. For example, when the tactical mode of a drone changes from `Normal` to `MoveUP` while it is on `T1`, this map function $T_{g_{\mathcal{M}}}(\texttt{T1}, \texttt{Normal}, \texttt{MoveUp}) = \texttt{M10}$ informs that the agent should follow a track with mode `M10`. These sets and functions together define a Verse map object $\mathcal{M} = \langle L_{\mathcal{M}}, V_{\mathcal{M}}, g_{\mathcal{M}}, T_{g_{\mathcal{M}}} \rangle$. We will drop the subscript $\mathcal{M}$ when the map being used is clear from context.

Agents. A Verse *agent* is defined by modes and continuous state variables, a decision logic that defines (possibly nondeterministic) discrete transitions, and a flow function that defines continuous evolution. An agent $\mathcal{A}$ is *compatible* with a map $\mathcal{M}$ if the agent's tactical modes P are a subset of the allowed input tactical modes for T_g. This makes it possible to instantiate the same agent on different compatible maps. The *mode space* for an agent instantiated on map $\mathcal{M}$ is the set $D = L \times P$, where L is the set of track modes in $\mathcal{M}$ and P is the set of tactical modes of the agent. The *continuous state space* is $X = W \times Z$, where W is the workspace (of $\mathcal{M}$) and Z is the space of other continuous state variables. The (full) *state space* is the Cartesian product $Y = X \times D$. In the two-drone example in Sect. 2, the continuous states variables are the positions and velocities along the three axes of the workspace. The modes are $\langle \texttt{Normal}, \texttt{T1} \rangle$, $\langle \texttt{MoveUp}, \texttt{M10} \rangle$, etc.

An *agent* $\mathcal{A}$ in map $\mathcal{M}$ with $k-1$ other agents is defined by a tuple $\mathcal{A} = \langle Y, Y^0, G, R, F \rangle$, where Y is the state space, $Y^0 \subseteq Y$ is the set of initial states. The guard G and reset R functions jointly define the discrete transitions. For a pair of modes $d, d' \in D$, $G(d, d') \subseteq X^k$ defines the condition under which a transition from d to d' is enabled. The $R(d, d') : X^k \to X$ function specifies how the continuous states of the agent are updated when the mode switch happens. Both of these functions take as input the sensed continuous states of all the other $k-1$ agents in the scenario. The G and the R functions are not defined separately,

but are extracted by the Verse parser from a block of structured Python code as shown in Fig. 2. The discrete states in `if` conditions and assignments define the source and destination of discrete transitions. `if` conditions involving continuous states define guards for the transitions and assignments of continuous states define resets. Expressions with `any` and `all` functions are unrolled to disjunctions and conjunctions according to the number of agents k.

For example in Fig. 2, Lines 47–50 define transitions $\langle \texttt{MoveUp}, \texttt{M10} \rangle$ to $\langle \texttt{Normal}, \texttt{T0} \rangle$ and $\langle \texttt{MoveUp}, \texttt{M21} \rangle$ to $\langle \texttt{Normal}, \texttt{T1} \rangle$. The change of track mode is given by the T_g function. The guard for this transition comes from the `if` condition at Line 48, $G(\langle \texttt{MoveUp}, \texttt{M10} \rangle, \langle \texttt{Normal}, \texttt{T0} \rangle) = \{x \mid -1 < \texttt{T0}.pz - x.pz < 1\}$ for $x \in X$ given by user defined `in_interval` function. Here continuous states remain unchanged after transition.

The final component of the agent is the *flow* function $F : X \times D \times \mathbb{R}^{\geq 0} \to X$ which defines the continuous time evolution of the continuous state. For any initial condition $\langle x^0, d^0 \rangle \in Y$, $F(x^0, d^0)(\cdot)$ gives the continuous state of the agent as a function of time. In this paper, we use F as a black-box function (see Footnote 1).

Sensors and Scenarios. For a scenario with k agents, a *sensor* function $\mathcal{S} : Y^k \to Y^k$ defines the continuous observables as a function of the continuous state. For simplifying exposition, in this paper we assume that observables have the same type as the continuous state Y, and that each agent i is observed by all other agents identically. This simple, overtly transparent sensor model, still allows us to write realistic agents that only use information about nearby agents. In a highway scenario, the observable part of agent j to another agent i may be the relative distance $y_j = x_j - x_i$, and vice versa, which can be computed as a function of the continuous state variables x_j and x_i. A different sensor function which gives nondeterministic noisy observations, appears in Sect. 5.

A Verse *scenario* SC is defined by (a) a map $\mathcal{M}$, (b) a collection of k agent instances $\{\mathcal{A}_1 ... \mathcal{A}_k\}$ that are compatible with $\mathcal{M}$, and (c) a sensor $\mathcal{S}$ for the k agents. Since all the agents are instantiated on the same compatible map $\mathcal{M}$, they share the same workspace. Currently, we require agents to have identical state spaces, i.e., $Y_i = Y_j$, but they can have different decision logics and different continuous dynamics.

4 Verse Scenario to Hybrid Verification

In this section, we define the underlying hybrid system $H(SC)$, that a Verse scenario SC specifies. The verification questions that Verse is equipped to answer are stated in terms of the behaviors or *executions* of $H(SC)$. Verse's notion of a hybrid automaton is close to that in Definition 5 of [14]. The only uncommon aspect in [14] is that the continuous flows may be defined by a black-box simulator functions, instead of white-box analytical models (see Footnote 1).

Given a scenario with k agents $SC = \langle \mathcal{M}, \{\mathcal{A}_1, ... \mathcal{A}_k\}, \mathcal{S}, P \rangle$, the corresponding hybrid automaton $H(SC) = \langle \mathbf{X}, \mathbf{X}^0, \mathbf{D}, \mathbf{D}^0, \mathbf{G}, \mathbf{R}, \mathbf{TL} \rangle$, where

1. $\mathbf{X} := \prod_i X_i$ is the *continuous state space*. An element $\mathbf{x} \in \mathbf{X}$ is called a *state*. $\mathbf{X}^0 := \prod_i X_i^0 \subseteq \mathbf{X}$ is the set of *initial continuous states*.
2. $\mathbf{D} := \prod_i D_i$ is the *mode space*. An element $\mathbf{d} \in \mathbf{D}$ is called a *mode*. $\mathbf{D}^0 := \prod_i D_i^0 \subseteq \mathbf{D}$ is the finite set of *initial modes*.
3. For a mode pair $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{G}(\mathbf{d}, \mathbf{d}') \subseteq \mathbf{X}$ defines the continuous states from which a transition from $\mathbf{d}$ to $\mathbf{d}'$ is enabled. A state $\mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}')$ iff there exists an agent $i \in \{1, ..., k\}$, such that $\mathbf{x}_i \in G_i(\mathbf{d}_i, \mathbf{d}'_i)$ and $\mathbf{d}_j = \mathbf{d}'_j$ for $j \neq i$.
4. For a mode pair $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}') : \mathbf{X} \to \mathbf{X}$ defines the change of continuous states after a transition from $\mathbf{d}$ to $\mathbf{d}'$. For a continuous state $\mathbf{x} \in \mathbf{X}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x}) = R_i(\mathbf{d}_i, \mathbf{d}'_i)(\mathbf{x})$ if $\mathbf{x} \in G_i(\mathbf{d}_i, \mathbf{d}'_i)$, otherwise $= \mathbf{x}_i$.
5. **TL** is a set of pairs $\langle \xi, \mathbf{d} \rangle$, where the *trajectory* $\xi : [0, T] \to \mathbf{X}$ describes the evolution of continuous states in mode $\mathbf{d} \in \mathbf{D}$. Given $\mathbf{d} \in \mathbf{D}, \mathbf{x}^0 \in \mathbf{X}$, ξ should satisfy $\forall t \in \mathbb{R}^{\geq 0}, \xi_i(t) = F_i(\mathbf{x}_i^0, \mathbf{d}_i)(t)$.

We denote by $\xi.fstate$, $\xi.lstate$, and $\xi.ltime$ the initial state $\xi(0)$, the last state $\xi(T)$, and $\xi.ltime = T$. For a sampling parameter $\delta > 0$ and a length m, a δ-*execution* of a hybrid automaton $H = H(SC)$ is a sequence of m labeled trajectories $\alpha := \langle \xi^0, \mathbf{d}^0 \rangle, ..., \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$, such that (1) $\xi^0.fstate \in \mathbf{X}^0, \mathbf{d}^0 \in \mathbf{D}^0$, (2) For each $i \in \{1, ..., m-1\}$, $\xi_i.lstate \in \mathbf{G}(\mathbf{d}^i, \mathbf{d}^{i+1})$ and $\xi^{i+1}.fstate = \mathbf{R}(\mathbf{d}^i, \mathbf{d}^{i+1})(\xi^i.lstate)$, and (3) For each $i \in \{1, ..., m-1\}$, $\xi^i.ltime = \delta$ for $i \neq m-1$ and $\xi^i.ltime \leq \delta$ for $i = m-1$.

We define first and last state of an execution $\alpha = \langle \xi^0, \mathbf{d}^0 \rangle, ..., \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$ as $\alpha.fstate = \xi^0.fstate$, $\alpha.lstate = \xi^{m-1}.lstate$ and the first and last mode as $\alpha.fmode = \mathbf{d}^0$ and $\alpha.lmode = \mathbf{d}^{m-1}$. The set of reachable states is defined by $Reach_H := \{\alpha.lstate \mid \alpha \text{ is an execution of } H\}$. In addition, we denote the reachable states in a specific mode $\mathbf{d} \in \mathbf{V}$ as $Reach_H(\mathbf{d})$ and $Reach_H(T)$ to be the set of reachable states at time T. Similarly, denoting the unsafe states for mode $\mathbf{d}$ as $\mathbf{U}(\mathbf{d})$, the safety verification problem for H can be solved by checking whether $\forall \mathbf{d} \in \mathbf{D}$, $Reach_H(\mathbf{d}) \cap \mathbf{U}(\mathbf{d}) = \emptyset$. Next, we discuss Verse functions for verification via reachability.

Verification Algorithms in Verse. The Verse library comes with several built-in verification algorithms, and it provides functions that users can use to implement powerful new algorithms. We describe the basic algorithm and functions in this section.

Consider a scenario SC with k agents and the corresponding hybrid automaton $H(SC)$. For a pair of modes, $\mathbf{d}, \mathbf{d}'$ the standard discrete $post_{\mathbf{d},\mathbf{d}'} : \mathbf{X} \to \mathbf{X}$ and continuous $post_{\mathbf{d},\delta} : \mathbf{X} \to \mathbf{X}$ operators are defined as follows: For any state $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$, $post_{\mathbf{d},\mathbf{d}'}(\mathbf{x}) = \mathbf{x}'$ iff $\mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}')$ and $\mathbf{x}' = \mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x})$; and, $post_{\mathbf{d},\delta}(\mathbf{x}) = \mathbf{x}'$ iff $\forall i \in 1, ..., k$, $\mathbf{x}'_i = F_i(\mathbf{x}_i, \mathbf{d}_i, \delta)$. These operators are also lifted to sets of states in the usual way. Verse provides `postCont` to compute $post_{\mathbf{d},\delta}$ and `postDisc` to compute $post_{\mathbf{d},\mathbf{d}'}$. Instead of computing the exact post, `postCont` and `postDisc` compute over-approximations using improved implementations of the algorithms in [14]. Verse's `verify` function implements a reachability analysis algorithm using these post operators. The algorithm constructs an execution tree $Tree = \langle V, E \rangle$ up to depth m in breadth first order. Each vertex $\langle \mathbf{S}, \mathbf{d} \rangle \in V$

is a pair of a set of states and a mode. The root is $\langle \mathbf{X}^0, \mathbf{d}^0 \rangle$. There is an edge from $\langle \mathbf{S}, \mathbf{d} \rangle$ to $\langle \mathbf{S}', \mathbf{d}' \rangle$, iff $\mathbf{S}' = post_{\mathbf{d}',\delta}(post_{\mathbf{d},\mathbf{d}'}(\mathbf{S}))$. The safety conditions are checked when the tree is constructed. Currently, Verse implements only bounded time reachability, however, basic unbounded time analysis with fixed-point checks could be added following [14,32].

5 Experiments and Use Cases

We evaluate key features and algorithms in Verse through examples. We consider two types of agents: a 4-d ground vehicle with bicycle dynamics and the Stanley controller [22] and a 6-d drone with a NN-controller [23]. Each of these agents can be fitted with one of two types of decision logic: (1) a collision avoidance logic (CA) by which the agent switches to a different available track when it nears another agent on its own track, and (2) a simpler non-player vehicle logic (NPV) by which the agent does not react to other agents (and just follows its own track at constant speed). We denote the car agent with CA logic as agent C-CA, drone with NPV as D-NPV, and so on. We use four 2-d maps ($\mathcal{M}$1-4) and two 3-d maps $\mathcal{M}$5-6. $\mathcal{M}$1 and $\mathcal{M}$2 have 3 and 5 parallel straight tracks, respectively. $\mathcal{M}$3 has 3 parallel tracks with circular curve. $\mathcal{M}$4 is imported from OpenDRIVE. $\mathcal{M}$6 is the figure-8 map used in Sect. 2.

Safety Analysis with Multiple Drones in a 3-d Map. The first example is a scenario with two drones—D-CA agent (red) and D-NPV agent (blue)—in map $\mathcal{M}$5. The safety assertion requires agents to always separate by at least 1 m. Figure 4(*left*) shows the computed reachable set, its projection on x-position, and on z position. Since the agents are separated in space-time, the scenario is verified safe. These plots are generated using Verse's plotting functions.

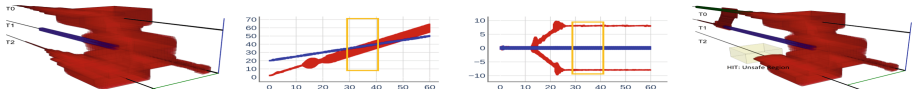

Fig. 4. Left to right: (1) Computed reachtubes for a 2-drone scenario; (2) same reachtube projected on x-dimension, and (3) on z-dimension. Since there is no overlap in space-time, no collision. (4) Reachtube for a 3-drone scenario, the red drone violates the safety condition by entering the unsafe region after moving downward. (Color figure online)

Checking Multiple Safety Assertions. Verse supports multiple safety assertions specified using `assert` statements. For example, the user can specify unsafe regions (Line 77–78) or safe separation between agents (Line 79–82) as shown in Fig. 5. We add a second D-NPV to the previous scenario and both safety assertions. The result is shown in the rightmost Fig. 4. In this scenario, D-CA violates the safety property by entering the unsafe region after moving downward to avoid collision. The behavior of D-CA after moving upward is not influenced. There is no violation of safe separation. Verse allow users to extract the set of reachable states and mode transitions that leads to a safety violation.

```
77        assert not (ego.x > 40 and ego.x < 50 and \
78            ego.y > -5 and ego.y < 5 and ego.z > -10 and ego.z < -6), "Unsafe Region"
79        assert not any(ego.x-other.x < 1 and ego.x-other.x > -1 and \
80            ego.y-other.y < 1 and ego.y-other.y > -1 and \
81            ego.z-other.z < 1 and ego.z-other.z > -1 \
82            for other in others), "Safe Separation"
```

Fig. 5. Safety assertions for three drone scenario.

Changing Maps. Verse allows users to easily create scenarios with different maps and port agents across compatible maps. We start with a scenario with one C-CA agent (red) and two C-NPV agents (blue, green) in $\mathcal{M}1$. The safety assertion is that the vehicles should be at least 1m apart in both x and y-dimensions. Figure 6(*left*) shows the verification result and safety is not violated. However, if we switch to map $\mathcal{M}3$ by changing one line in the scenario definition, a reachability analysis shows that a safety violation can happen after C-CA merges left Fig. 6(*center*). In addition, Verse allows importing map from OpenDRIVE [4] format. An example is included in the extended version of the paper [26].

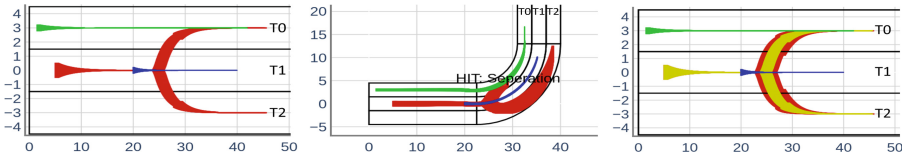

Fig. 6. *Left:* running the three car scenario on map with parallel straight lanes. *Center:* same scenario with a curved map. *Right:* same scenario with a noisy sensor. (Color figure online)

Adding Noisy Sensors. Verse supports scenarios with different sensor functions. For example, the user can create a noisy sensor function that mimics a realistic sensor with bounded noise. Such sensor functions are easily added to the scenario using the `set_sensor` function.

Figure 6(*right*) shows exactly the same three-car scenario with a noisy sensor, which adds ± 0.5 m noise to the perceived position of all other vehicles. Since the sensed values of other agents only impacts the checking of the guards (and hence the transitions) of the agents, Verse internally bloats the reachable set of positions for the other agents by ± 0.5 while checking guards. Compared with the behavior of the same agent with no sensor noise (shown in yellow in Fig. 6(*right*)), the sensor noise enlarges the region over which the transition can happen, causes enlarged reachtubes for the red agent.

Plugging in Different Reachability Engines. With a little effort, Verse allows users to plug in different reachability tools for the `postCont` computation. The user will need to modify the interface of the reachability tool so that given a set of initial states, a mode, and a non negative value δ, the reachability tool can output the set of reachable states over a δ-period represented by a set of

timed hyperrectangles. Currently, Verse implements computing `postCont` using DryVR [14], NeuReach [35] and Mixed Monotone Decomposition [12]. A scenario with two car agents in map $\mathcal{M}1$ verified using NeuReach and DryVR is included in the extended version of the paper [26].

Incremental Verification. We implemented an incremental verification algorithm in Verse called `verifyInc`. This algorithm improves `verify` by caching and reusing reachtubes, and can be effective when analyzing a sequence of slightly different scenarios. The function `verifyInc` avoids re-computing $post_{\mathbf{d},\mathbf{d}'}$ and $post_{\mathbf{d},\delta}$ when constructing the execution tree by reusing earlier execution runs. Experiments show that `verifyInc` reduces running time by 10x for two identical runs and 2x when the decision logic is slightly modified. (More details are provided in the extended version of paper [26]). This exercise illustrates a usage of Verse in creating alternative analysis algorithms.

Table 1 summarizes the running time of verifying all the examples in this section. We additionally include three standard benchmarks: van-der-pol (Agent V) [20], spacecraft rendezvous (Agent S) [20], and gearbox (Agent G) [2]. As expected, the running times increase with the number of discrete mode transition. However, for complicated scenario with 7 agents and 37 transitions, the verification can still finish in under 6 mins, which suggests some level of scalability. The choice of reachability engine can also impact running time. For the same scenario in rows 2, 3 and 10, 11, Verse with NeuReach[2] as the reachability engine takes more time than using DryVR as the reachability engine.

Table 1. Runtime for verifying examples in Sect. 5. Columns are: number of agents ($\#\mathcal{A}$), agent type ($\mathcal{A}$), map used (Map), reachability engine used (`postCont`), sensor type ($\mathcal{NS}$), number of mode transitions #TR, and the total run time (Rt). N/A for not available.

$\#\mathcal{A}$	$\mathcal{A}$	Map	`postCont`	$\mathcal{NS}$	#Tr	Rt (s)	$\#\mathcal{A}$	$\mathcal{A}$	Map	`postCont`	Noisy $\mathcal{S}$	#Tr	Rt (s)
2	D	$\mathcal{M}6$	DryVR	No	8	55.9	2	D	$\mathcal{M}5$	DryVR	No	5	18.7
2	D	$\mathcal{M}5$	NeuReach	No	5	1071.2	3	D	$\mathcal{M}5$	DryVR	No	7	39.6
7	C	$\mathcal{M}2$	DryVR	No	37	322.7	3	C	$\mathcal{M}1$	DryVR	No	5	23.4
3	C	$\mathcal{M}3$	DryVR	No	4	34.7	3	C	$\mathcal{M}4$	DryVR	No	7	118.3
3	C	$\mathcal{M}1$	DryVR	Yes	5	29.4	2	C	$\mathcal{M}1$	DryVR	No	5	21.6
2	C	$\mathcal{M}1$	NeuReach	No	5	914.9	1	V	N/A	DryVR	N/A	1	0.33
1	S	N/A	DryVR	N/A	3	2.3	1	G	N/A	DryVR	N/A	3	67.14

6 Related Work

Automatic hybrid verification tools typically require the input model to be written in a tool-specific language [10,13–15,17,25]. Libraries like JuliaReach [7]

[2] Runtime for NeuReach includes training time.

Hylaa [5] and HyPro [8] share our motivation to reduce the usability barrier by providing reachability analysis APIs for popular programming languages. Verse is distinct in this family in that it supports creation and analysis of multi-agent scenarios. The work in [33] also supports multiple agents, however, Verse significantly improves usability with maps, scenarios and decision logics written in Python.

Interactive theorem provers have been used for modeling and verification of multi-agent and hybrid systems [16,19,27,29]. KeYmeraX [19] uses quantified differential dynamic logic for specifying multi-agent scenarios and supports proof search and user defined tactics. Isabelle/HOL [16], PVS [27], and Maude [29] have also been used for limited classes of hybrid systems. These approaches are geared for a different user segment in that they provide higher expressive and analytical power to expert users. Verse is inspired by widely used tools for simulating multi-agent scenarios [9,18,28,30,36]. While the models created in these tools can be flexible and expressive, currently they are not amenable to formal verification.

7 Conclusions and Future Directions

In this paper, we presented the new open source Verse library for broadening applications of hybrid system verification technologies to scenarios involving multiple interacting decision-making agents. There are several future directions for Verse. Verse currently assumes all agents interact with each other only through the sensor in the scenario and all agents share the same sensor. This restriction could be relaxed to have different types of asymmetric sensors. Functions for constructing and systematically sampling scenarios could be developed. Functions for post-computation for white-box models by building connections with existing tools [1,10,15] would be a natural next step. Those approaches could obviously utilize the symmetry property of agent dynamics as in [32,34], but beyond that, new types of symmetry reductions should be possible by exploiting the map geometry.

References

1. Althoff, M.: An introduction to CORA 2015. In: Proceedings of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
2. Althoff, M., et al.: Arch-comp20 category report: continuous and hybrid systems with linear continuous dynamics. In: Frehse, G., Althoff, M. (eds.) 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20), ARCH20. EPiC Series in Computing, vol. 74, pp. 16–48. EasyChair (2020). https://doi.org/10.29007/7dt2
3. Alur, R., et al.: The algorithmic analysis of hybrid systems. Theoret. Comput. Sci. **138**(1), 3–34 (1995)
4. Association for Standardization of Automation and Measuring Systems (ASAM): Open dynamic road information for vehicle environment, August 2021. https://www.asam.net/standards/detail/opendrive/

5. Bak, S., Duggirala, P.S.: HyLAA: a tool for computing simulation-equivalent reachability for linear systems. In: Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control, pp. 173–178. ACM (2017)
6. Bak, S., Tran, H.D., Johnson, T.T.: Numerical verification of affine systems with up to a billion dimensions. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, pp. 23–32. Association for Computing Machinery, New York (2019)
7. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: JuliaReach: a toolbox for set-based reachability. In: Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, pp. 39–44 (2019)
8. Ábrahám, E., Schupp, S., Chen, X., Kowalewski, S., Makhlouf, I., Sankaranarayanan, S.: HyPro: a C++ library for the representation of state sets for the reachability analysis of hybrid systems (2023)
9. Brittain, M., Alvarez, L.E., Breeden, K., Jessen, I.: AAM-Gym: artificial intelligence testbed for advanced air mobility (2022)
10. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
11. Chen, X., Sankaranarayanan, S.: Reachability analysis for cyber-physical systems: are we there yet? In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.) NASA Formal Methods, NFM 2022. LNCS, vol. 13260. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06773-0_6
12. Coogan, S.: Mixed monotonicity for reachability and safety in dynamical systems. In: 2020 59th IEEE Conference on Decision and Control (CDC), pp. 5074–5085 (2020)
13. Devonport, A., Khaled, M., Arcak, M., Zamani, M.: PIRK: scalable interval reachability analysis for high-dimensional nonlinear systems. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 556–568. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_27
14. 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
15. 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
16. Foster, S., Huerta y Munive, J.J., Gleirscher, M., Struth, G.: Hybrid systems verification with Isabelle/HOL: simpler syntax, better models, faster proofs. In: Huisman, M., Păsăreanu, C., Zhan, N. (eds.) FM 2021. LNCS, vol. 13047, pp. 367–386. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-90870-6_20
17. Frehse, G., et al.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
18. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Seshia, S.A.: Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, pp. 63–78. Association for Computing Machinery, New York (2019)

19. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
20. Geretti, L., et al.: Arch-comp20 category report: continuous and hybrid systems with nonlinear dynamics. In: Frehse, G., Althoff, M. (eds.) 7th International Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH20). EPiC Series in Computing, vol. 74, pp. 49–75. EasyChair (2020). https://doi.org/10.29007/zkf6
21. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998)
22. Hoffmann, G.M., Tomlin, C.J., Montemerlo, M., Thrun, S.: Autonomous automobile trajectory tracking for off-road driving: controller design, experimental validation and racing. In: 2007 American Control Conference, pp. 2296–2301 (2007)
23. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: 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, pp. 169–178 (2019)
24. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science (2011). https://doi.org/10.1007/978-3-031-02003-2. Morgan Claypool (November 2005), also available as Technical Report MIT-LCS-TR-917, MIT
25. Kong, S., Gao, S., Chen, W., Clarke, E.: dReach: δ-reachability analysis for hybrid systems. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 200–205. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_15
26. Li, Y., Zhu, H., Braught, K., Shen, K., Mitra, S.: Verse: a Python library for reasoning about multi-agent hybrid system scenarios (2023)
27. Lim, H., Kaynar, D., Lynch, N., Mitra, S.: Translating timed I/O automata specifications for theorem proving in PVS. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 17–31. Springer, Heidelberg (2005). https://doi.org/10.1007/11603009_3
28. Lopez, P.A., et al.: Microscopic traffic simulation using SUMO. In: The 21st IEEE International Conference on Intelligent Transportation Systems. IEEE (2018)
29. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. Theoret. Comput. Sci. **285**(2), 359–405 (2002). rewriting Logic and its Applications
30. Jiang, M., et al.: GRAIC: a simulator framework for autonomous racing. https://popgri.github.io/Race/ (2021)
31. Ray, R., Gurung, A., Das, B., Bartocci, E., Bogomolov, S., Grosu, R.: XSpeed: accelerating reachability analysis on multi-core processors. In: Piterman, N. (ed.) HVC 2015. LNCS, vol. 9434, pp. 3–18. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26287-1_1
32. Sibai, H., Li, Y., Mitra, S.: SceneChecker: boosting scenario verification using symmetry abstractions. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 580–594. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_28
33. Sibai, H., Mokhlesi, N., Fan, C., Mitra, S.: Multi-agent safety verification using symmetry transformations. In: TACAS 2020. LNCS, vol. 12078, pp. 173–190. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_10

34. Sibai, H., Mokhlesi, N., Mitra, S.: Using symmetry transformations in equivariant dynamical systems for their safety verification. In: Chen, Y.-F., Cheng, C.-H., Esparza, J. (eds.) ATVA 2019. LNCS, vol. 11781, pp. 98–114. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31784-3_6
35. Sun, D., Mitra, S.: NeuReach: learning reachability functions from simulations. In: TACAS 2022. LNCS, vol. 13243, pp. 322–337. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_17
36. Wu, C., Kreidieh, A., Parvate, K., Vinitsky, E., Bayen, A.M.: Flow: Architecture and benchmarking for reinforcement learning in traffic control. arXiv:1710.05465 (2017)