

Verse: A Python library for reasoning about multi-agent hybrid system scenarios

Yangge Li¹, Haoqing Zhu¹, Katherine Braught¹, and Sayan Mitra¹

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
`{li213, haoqing3, braught2, 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 several capabilities and use cases of the library with heterogeneous agents, incremental verification, different sensor models, and the flexibility of plugging in different subroutines for post computations.

Keywords: Scenario verification · Reachability analysis · Hybrid Systems.

1 Introduction

Hybrid system verification tools have been used to analyze linear models with thousands of continuous dimensions [6,7,3] and nonlinear models inspired by industrial applications [7,12]. Chen and Sankaranarayanan provide a survey of the state of the art [10]. Despite the large potential user base, current usage of this technology remains concentrated within the formal methods community. We conjecture that usability is one of the key barriers. Most hybrid verification tools [16,29,9,6,3] require the input model to be written in a tool-specific language. Tools like C2E2 [13] attempt to translate a subclass of models from the popular Simulink/Stateflow framework, but the language-barrier goes deeper than syntax. The verification algorithms are based on variants of the hybrid automaton [4,20,23] which require the discrete states (or *modes*) to be spelled out explicitly as a graph, with guards and resets labeling the edges.

In contrast, the code for *simulating* a multi-agent scenario would be written in an expressive programming language. Each agent will have a decision logic and some continuous dynamics. A complex scenario would be composed by putting together a collection of agents; it may use a *map* which brings additional structure and constraints to the agent’s decisions and interactions. Describing or

translating such scenarios for hybrid verification is a far cry from the capabilities of current tools.

In this paper, we present Verse¹, a Python library that aims to make hybrid technologies more usable for multi-agent scenarios. An agent’s decision logic is written in an expressive subset of Python (See Fig. 2). This program can access the relevant features of the map and parts of the states of the other agents. The continuous dynamics of an agent has to be supplied as a black-box simulation function. Multiple agents with different dynamics and different decision logics are instantiated to create a Verse scenario.

Verse provides functions for performing systematic simulation and verification of such scenarios through reachability analysis. An agent’s decision logic can allow nondeterministic choices. If multiple Python `if` conditions are satisfied at a given state, then both branches are explored. For example, Fig. 1(*center*) shows simulations in which the red drone on track T1 nears the blue and nondeterministically switches to tracks T0 and T2. Similarly, the `verify` function propagates uncertainty in the initial states through branches. Safety requirements written with `assert` can be checked via reachability analysis. While the functions for traversal and *post* computation are based on known algorithms, new ones can be implemented and the library vastly simplifies specification of hybrid multi-agent scenarios.

The key concept that makes the above functionalities tractable and specifications expressive is the *map* abstraction. A Verse map defines a set of *tracks* that agents can follow. While a map may have infinitely many tracks, they fall in a finite number of *track modes*. For example, in Fig. 1 each layer in the map is assigned to a track mode (T0–2) and the tracks between each pair of layers are also assigned to a track mode (M10, M01 etc.). Further, when an agent makes a decision and changes its internal mode (called *tactical mode*, which is different from the track mode), for example from `Normal` to `MoveUp`, the map object determines the new track mode for the agent. For an agent on track T1, its new track mode will be M10. This map abstraction allows portability of an agent’s decision logic across different maps as long as they have the same interface or track modes. It makes Verse suited for multi-agent scenarios, arising in motion control [19], air-traffic management [14], and the study of tactical collision avoidance [26].

In summary, the main contributions of this paper are: (1) an expressive framework for specifying and simulating multi-agent hybrid scenarios on interesting maps (Section 3), (2) a library of powerful functions for building verification algorithms based on reachability analysis (Section 5), and (3) illustrations of several capabilities and use cases of the library with heterogeneous agents, incremental verification, different sensor models, and the flexibility of plugging in different subroutines for *post* computations (Section 6).

Related work. There are a number of powerful tools for creating, simulating, and testing complex multi-agent scenarios [17,34,8,27]. For instance, Scenic [17] uses

¹ We will make the tool available for artifact evaluations. We omitted the online link in this submission to avoid compromising the double blind review requirements.

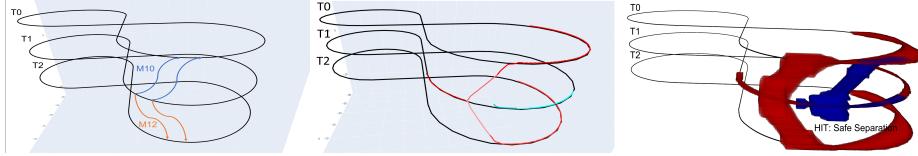


Fig. 1: *Left:* A 3-d figure-8 map with example track mode labels. *Center:* Simulation of a red drone nearing the blue drone on T1 and nondeterministically moving to T0 or T2. Both branches are computed by Verse’s `simulate` function. *Right:* Computed reachable sets of the two drones cover more possibilities: both drones are allowed to switch tracks when they get close. All four branches are explored by Verse and one is found to violate safety.

a probabilistic programming language for guided sampling, simulations, and falsification, Flow [34] integrates the SUMO [25] traffic simulator for reinforcement learning, AAM-GYM [8] can generate and simulate scenarios for testing AI algorithms in advanced air mobility, and GRAIC [27] has been used for testing racing controllers in dynamic environments. While the models created in these tools can be very flexible and expressive, they are not readily amenable to formal verification.

Interactive theorem provers have been used for modeling and verification of multi-agent, distributed and hybrid systems [15,18,24,35]. Most notably KeYmeraX [18] uses quantified differential dynamic logic for specifying multi-agent scenarios and supports speculative proof search and user defined tactics. Isabelle/HOL [15], PVS [24], and Maude [35] have also been used for limited classes of hybrid systems. These approaches require significant levels of user expertise to interact with.

This work is closest to the tool presented in [31], which also supports multiple agents and reachability analysis. However, Verse significantly improves usability by allowing (a) Python for the decision logics and (b) complex maps. The theoretical ideas of Sibai et al. [30,31], in exploiting symmetries and caching are complementary to our contributions and could indeed be incorporated to improve the verification algorithms in Verse.

2 Overview of Verse

We will highlight the key features of Verse with an example. Consider two drones flying along three parallel figure-eight 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 for such scenarios.

Creating scenarios. Agents like the drones in this example are described by a *simulator* and a *decision logic*. The decision logic has to be written in an expressive subset of Python (see code in Fig. 2 and Appendix B 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. It may also update the continuous state of the ego agent. The *mode* of an agent, as we shall see later in Section 3.2, 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. For example, in lines 41-43 of Fig. 2 an agent updates its tactical mode to begin a track change if there is any agent near it.

Verse will parse this decision logic to internally construct 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-box². 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
42                   others):
43                 next.tactical_mode = TacticalMode.MoveDown
44                 next.track_mode = track_map.h(ego.track_mode, ego.tactical_mode,
45                   TacticalMode.MoveDown)
45             if any((is_close(ego, other) and ego.track_mode==other.track_mode) for other in
46                   others):
47                 next.tactical_mode = TacticalMode.MoveUp
48             # ...
49             if ego.tactical_mode == TacticalMode.MoveUp:
50                 if in_interval(track_map.altitude(ego.track_mode)-ego.z, -1, 1):
51                     next.tactical_mode = TacticalMode.Normal
52                     next.track_mode = track_map.h(ego.track_mode, ego.tactical_mode,
53                       TacticalMode.Normal)
54             # ...
55         return next

```

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. Besides creating from scratch, Verse provides functions that automatically generates map objects from OpenDRIVE [5] files. The *sensor* function defines

² This design decision for Verse is relatively independent. For reachability analysis, Verse currently uses black-box statistical approaches implemented in DryVR [12] and NeuReach [33]. If the simulator is available as a white-box model, such as differential equations, then Verse could use model-based reachability analysis.

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 Section 6.1. 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.

```

32     scenario = Scenario()
33     drone_red = DroneAgent('drone_red', file_name='drone_controller.py')
34     drone_red.set_initial([init_l_1, init_u_1], (CraftMode.Normal, TrackMode.T1))
35     scenario.add_agent(drone_red)
36     drone_blue = DroneAgent('drone_blue', file_name='drone_controller.py')
37     scenario.add_agent(drone_blue)
38     # ...
39     scenario.set_map(M6())
40     scenario.set_sensor(BaseSensor())
41     # ...
42     #traces = scenario.simulate(40, time_step)
43     traces = scenario.verify(40, time_step)

```

Fig. 3: Scenario Specification File 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 earlier in Fig. 1, 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 can verify the safety assertions of a scenario with the `verify` function. It computes the over-approximations of the *reachable sets* for each agent, and checks these against the predicates defined by the assertions. Fig. 1 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.

Building advanced functions. Verse library provides powerful functions to implement advanced verification algorithms. Section 5.2 explains how users can modify the basic reachability algorithm to save computing time in repeated verification of similar scenarios using caching and incremental verification. Verse also makes it convenient to plug in different reachability subroutines. The experiments in Section 6.1 show how the sensor can be useful when modeling realistic inputs to the controllers and Section 6.3 describes verification when dynamics have uncertainty.

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 that is visible to other agents. We describe these components below and in Section 4 we will discuss how they formally define a hybrid system.

3.1 Tracks, 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 $\omega : [0, 1] \rightarrow W$, 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 Section 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 $L_{\mathcal{M}}$, a labeling function $V_{\mathcal{M}} : \Omega_{\mathcal{M}} \rightarrow L_{\mathcal{M}}$ that gives the track mode for a track. It also has a mapping $g_{\mathcal{M}} : W \times L_{\mathcal{M}} \rightarrow \Omega_{\mathcal{M}}$ that gives a specific track from a track mode and a specific position in the workspace.

Finally, a Verse agent’s decision logic can change its internal mode or *tactical mode* (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 another function: $h_{\mathcal{M}} : L_{\mathcal{M}} \times P \times P \rightarrow L_{\mathcal{M}}$ 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 $h_{\mathcal{M}}(T1, \text{Normal}, \text{MoveUp}) = M10$ informs that the agent should follow a track with mode M10. These sets and functions together define a Verse map object $\mathcal{M} = \langle L_{\mathcal{M}}, V_{\mathcal{M}}, g_{\mathcal{M}}, h_{\mathcal{M}} \rangle$. We will drop the subscript \mathcal{M} , when the map being used is clear from context.

3.2 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 h . 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 Section 2, the continuous states variables px, py, pz, vx, vy, vz are the positions and velocities along the three axes of the workspace. The modes are $\langle \text{Normal}, \text{T1} \rangle, \langle \text{MoveUp}, \text{M10} \rangle$, 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 $R(d, d') : X^k \rightarrow X$ 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. Details about the sensor which transmits state information across agents is discussed in Section 3.3. The G and the R functions are actually 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 the `if` condition and the assignments define the source and destination of discrete transition. The `if` conditions involving continuous states define the guard for the transitions and the assignments of continuous states define the reset. Expressions with `any` and `all` functions are unrolled to disjunctions and conjunctions according to the number of agents k .

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

The final component of the agent is the *flow* function $F : X \times D \times \mathbb{R}^{\geq 0} \rightarrow X$ 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 2).

3.3 Sensors and scenarios

For a scenario with k agents, a *sensor* function $\mathcal{S} : Y^k \rightarrow Y^k$ 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 Section 6.1.

A Verse *scenario* SC is defined by (a) a map \mathcal{M} , (b) a collection of k agent instances $\{\mathcal{A}_1 \dots \mathcal{A}_k\}$ 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 [12]. As usual, the automaton has discrete and continuous states and discrete transitions defined by guards and resets. The only uncommon aspect in [12] is that the continuous flows may be defined by a black-box simulator functions, instead of white-box analytical models (see footnote 2).

Given a scenario with k agents $SC = \langle \mathcal{M}, \{\mathcal{A}_1, \dots, \mathcal{A}_k\}, \mathcal{S}, P \rangle$, 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 $i \in \{1, \dots, k\}$, 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 $\mathbf{d}, \mathbf{d}' \in \mathbf{D}$, $\mathbf{R}(\mathbf{d}, \mathbf{d}') : \mathbf{X} \rightarrow \mathbf{X}$ 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. \mathbf{TL} is a set of pairs $\langle \xi, \mathbf{d} \rangle$, where the *trajectory* $\xi : [0, T] \rightarrow \mathbf{X}$ 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 $\forall t \in \mathbb{R}^{>0}, \xi_i(t) = F_i(\mathbf{x}_i^0, \mathbf{d}_i)(t)$.

Proposition 1. *If a scenario with k agents $SC = \langle \mathcal{M}, \{\mathcal{A}_1, \dots, \mathcal{A}_k\}, \mathcal{S}, P \rangle$, satisfies the following: (1) map and the agents are compatible, (2) all agents have identical sets of states and modes, $Y_i = Y_j$, and (3) agent states match the input type of \mathcal{S} , then $H(SC)$ is a valid hybrid system.*

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 $\alpha := \langle \xi^0, \mathbf{d}^0 \rangle, \dots, \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$, such that (1) $\xi^0.fstate \in \mathbf{X}^0, \mathbf{d}^0 \in \mathbf{D}^0$, (2) For each $i \in \{1, \dots, m-1\}$, $\xi_i.lstate \in \mathbf{G}(\mathbf{d}^i, \mathbf{d}^{i+1})$ and $\xi^{i+1}.fstate = \mathbf{R}(\mathbf{d}^i, \mathbf{d}^{i+1})(\xi_i.lstate)$, and (3) For each $i \in \{1, \dots, m-1\}$, $\xi^i.ltime = \delta$ for $i \neq m-1$ and $\xi^i.ltime \leq \delta$ for $i = m-1$.

We define the first and last state of an execution $\alpha = \langle \xi^0, \mathbf{d}^0 \rangle, \dots, \langle \xi^{m-1}, \mathbf{d}^{m-1} \rangle$ 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 $\text{Reach}_H := \{\alpha.lstate \mid \alpha \text{ is an execution of } H\}$. In addition, we denote the reachable states in a specific mode $\mathbf{d} \in \mathbf{V}$ as $\text{Reach}_H(\mathbf{d})$ and $\text{Reach}_H(T)$ 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 $\forall \mathbf{d} \in \mathbf{D}, \text{Reach}_H(\mathbf{d}) \cap \mathbf{U}(\mathbf{d}) = \emptyset$. Next, we discuss Verse functions for verification via reachability.

5 Building verification algorithms in Verse

The Verse library comes with several built-in verification algorithms, and it provides functions that users can be used to implement powerful new algorithms. We first describe the basic building blocks and in Sections 5.2 and 6.2 we discuss more advanced use cases and algorithms.

5.1 Reachability analysis

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 $\text{post}_{\mathbf{d}, \mathbf{d}'} : \mathbf{X} \rightarrow \mathbf{X}$ and continuous $\text{post}_{\mathbf{d}, \delta} : \mathbf{X} \rightarrow \mathbf{X}$ operators are defined as follows: For any state $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$, $\text{post}_{\mathbf{d}, \mathbf{d}'}(\mathbf{x}) = \mathbf{x}'$ iff $\mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}')$ and $\mathbf{x}' = \mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x})$; and, $\text{post}_{\mathbf{d}, \delta}(\mathbf{x}) = \mathbf{x}'$ iff $\forall i \in 1, \dots, k, \mathbf{x}'_i = F_i(\mathbf{x}_i, \mathbf{d}_i, \delta)$. These operators are also lifted to sets of states in the usual way. Verse provides `postCont` to compute $\text{post}_{\mathbf{d}, \delta}$ and `postDisc` to compute $\text{post}_{\mathbf{d}, \mathbf{d}'}$. Instead of computing the exact post, `postCont` and `postDisc` compute over-approximations using improved implementations of the algorithms in [12]. Verse's `verify` function implements a reachability analysis algorithm using these post operators (Algorithm 1). This 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 $\mathbf{S}' = \text{post}_{\mathbf{d}, \delta}(\text{post}_{\mathbf{d}, \mathbf{d}'}(\mathbf{S}))$.

Algorithm 1

```

1: function VERIFY( $\langle \mathbf{X}^0, \mathbf{d}^0 \rangle, H, m, \delta$ )
2:    $root \leftarrow \langle \mathbf{X}^0, \mathbf{d}^0 \rangle$ 
3:    $depth \leftarrow 0$ 
4:    $queue \leftarrow [\langle \mathbf{X}^0, \mathbf{d}^0 \rangle, depth]$ 
5:   while  $queue \neq \emptyset$  and  $depth \leq m$  do
6:      $\langle \mathbf{S}, \mathbf{d} \rangle, depth \leftarrow queue.dequeue()$ 
7:     for  $\mathbf{d}'$ , s.t.  $\mathbf{G}(\mathbf{d}, \mathbf{d}')$  do
8:       if  $\mathbf{S} \cap \mathbf{G}(\mathbf{d}, \mathbf{d}') \neq \emptyset$  then
9:          $\langle \mathbf{S}', \mathbf{d}' \rangle \leftarrow \langle \text{postCont}(\text{postDisc}(\mathbf{S}, \mathbf{d}, \mathbf{d}'), \mathbf{d}', \delta), \mathbf{d}' \rangle;$ 
10:         $\langle \mathbf{S}, \mathbf{d} \rangle.addChild(\langle \mathbf{S}', \mathbf{d}' \rangle)$ 
11:         $queue.add(\langle \mathbf{S}', \mathbf{d}' \rangle, depth + 1)$ 
12:   return  $root$ 

```

Proposition 2. *For hybrid automaton $H = H(SC)$, let $\text{Tree} = \langle V, E \rangle$ be the execution tree with depth T constructed by `verify`, then for each level $t \in \{0, \dots, T\}$, $\text{Reach}_H(\delta t) = V(t)$, where $V(t)$ is the union of the states in the vertices at depth t .*

Proposition 2 holds when the post computations are exact. However, typically, we rely on techniques that compute the over-approximations of the actual post, in which, $V(t)$ over-approximates $\text{Reach}_H(\delta t)$. Currently, Verse implements only bounded time reachability, however, basic unbounded time analysis with fixed-point checks could be added following [12,30].

5.2 Incremental Verification

During the design-analysis process, users perform many simulation and verification runs on slightly tweaked scenarios. Can we do better than starting each verification run from scratch? In Verse, we have implemented an incremental analysis algorithm that improves the performance of `simulate` and `verify` by reusing data from previous verification runs. This algorithm also illustrates how Verse can be used to implement different algorithms.

Consider two hybrid automata $H_i = H(SC_i)$, $i \in \{1, 2\}$ that only differ in the discrete transitions. That is, (1) $\mathbf{X}_2 = \mathbf{X}_1$, (2) $\mathbf{D}_2 = \mathbf{D}_1$, and (3) $\mathbf{TL}_2 = \mathbf{TL}_1$, while the initial conditions, the guards, and the resets are slightly different³. SC_1 and SC_2 have the same sensors, maps, and agent flow functions. Let $\text{Tree}_1 = \langle V_1, E_1 \rangle$ and $\text{Tree}_2 = \langle V_2, E_2 \rangle$ be the execution trees for H_1 and H_2 . Our idea of incremental verification is to reuse some of the computations in constructing the tree for H_1 in computing the same for H_2 .

Recall that in `verify`, expanding each vertex $\langle \mathbf{S}_1, \mathbf{d}_1 \rangle$ of Tree_1 with a possible mode involves a guard check, a computation of $\text{post}_{\mathbf{d}, \mathbf{d}'}$, and $\text{post}_{\mathbf{d}, \delta}$. The `verifyInc` algorithm avoids performing these computations while constructing Tree_2 by reusing those computations from Tree_1 , if possible. To this end, `verifyInc` is endowed with two caches: C_g stores information about discrete transitions and C_f stores information about continuous flows. The key to C_g is a vertex $\langle \mathbf{S}_2, \mathbf{d}_2 \rangle \in V_2$, a guard $\mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2)$ and a reset $\mathbf{R}_2(\mathbf{d}_2, \mathbf{d}'_2)$ for a mode pair. The retrieved data from C_g will be a pair $\langle s, v' \rangle$ where $s \in \{\text{sat}, \text{unsat}, \text{unknown}\}$ is the guard checking result for \mathbf{S}_2 and $\mathbf{G}_2(\mathbf{d}, \mathbf{d}')$ and $v' = \langle \text{post}_{\mathbf{d}, \mathbf{d}'}(\mathbf{S}_2), \mathbf{d}' \rangle$ is the vertex after applying post. A cache hit can happen if there exists an entry in the cache with key match exactly with $\langle \langle \mathbf{S}_2, \mathbf{d}_2 \rangle, \mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2), \mathbf{R}_2(\mathbf{d}_2, \mathbf{d}'_2) \rangle$. If $\langle \langle \mathbf{S}_2, \mathbf{d}_2 \rangle, \mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2), \mathbf{R}_2(\mathbf{d}_2, \mathbf{d}'_2) \rangle \in C_g$ and $s = \text{sat}$, then we know from C_g that \mathbf{S}_2 satisfies $\mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2)$ and the post of \mathbf{S} can be retrieved from the cache $v' = \langle \text{post}_{\mathbf{d}, \mathbf{d}'}(\mathbf{S}_2), \mathbf{d}' \rangle$. If $\langle \langle \mathbf{S}_2, \mathbf{d}_2 \rangle, \mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2), \mathbf{R}_2(\mathbf{d}_2, \mathbf{d}'_2) \rangle \in C_g$ but $s = \text{unsat}$, then \mathbf{S}_2 does not satisfy the guard $\mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2)$ and $v' = \text{none}$. If $\langle \langle \mathbf{S}_2, \mathbf{d}_2 \rangle, \mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2), \mathbf{R}_2(\mathbf{d}_2, \mathbf{d}'_2) \rangle \notin C_g$, then $s = \text{unknown}$, $v' = \text{none}$ and the guard checking and post computation from \mathbf{S}_2 will need to happen afresh for H_2 .

³ Note that in this section subscripts index different hybrid automata, instead of agents within the same automaton (as we did in Sections 3 and 4).

The second cache constructed is C_f . The key to the cache will be a vertex $\langle \mathbf{S}, \mathbf{d} \rangle$. A cache hit can happen if there exist an entry in the cache $\langle \mathbf{S}', \mathbf{d}' \rangle$ such that $\mathbf{S} \subseteq \mathbf{S}'$ and $\mathbf{d} = \mathbf{d}'$ and the retrieved value from the cache will be $\mathbf{S}'' = post_{\mathbf{d}, \delta}(\mathbf{S}') \supseteq post_{\mathbf{d}, \delta}(\mathbf{S})$.

Similar to `verify`, for each vertex in the tree, `verifyInc` expands all possible mode transitions (Line 10). The full algorithm is shown in Algorithm 2 for completeness and we skip the detailed description owing to space limitations. `verifyInc` checks C_g or C_f before every *post* computation to retrieve and reuse computations when possible. The caches can save information from any number of previous executions, so `verifyInc` can be even more efficient than `verify` when running many consecutive verification runs.

Proposition 3. *Given the caches C_g and C_f constructed while constructing $Tree_1$ for H_1 , the tree $Tree_2$ constructed by `verifyInc` is sound. That is*

$$Reach_{H_2}(\delta t) \subseteq V_2(t), \forall t \in \{0, \dots, T\}$$

Proof sketch. The proof relies on `verifyInc`'s design ensuring that the cache data is always an over approximation of the actual post computations. Consider C_g and C_f constructed while constructing $Tree_1$ and the tree growth algorithm `verifyInc` from a vertex $\langle \mathbf{S}_2, \mathbf{d}_2 \rangle \in Tree_2$ for tree for H_2 . When no cache hit happens, `verifyInc` works exactly the same as `verify`. If $\langle \langle \mathbf{S}_2, \mathbf{d}_2 \rangle, \mathbf{G}_2(\mathbf{d}_2, \mathbf{d}'_2), \mathbf{R}_2(\mathbf{d}_2, \mathbf{d}'_2) \rangle \in C_g$, there exists an entry in C_g with key $\langle \langle \mathbf{S}_1, \mathbf{d}_1 \rangle, \mathbf{G}_1(\mathbf{d}_1, \mathbf{d}'_1), \mathbf{R}_1(\mathbf{d}_1, \mathbf{d}'_1) \rangle$ from $Tree_1$ that matches exactly with it and the output from cache will be $v' = \langle post_{\mathbf{d}, \mathbf{d}'}(\mathbf{S}_1), \mathbf{d}'_1 \rangle = \langle post_{\mathbf{d}, \mathbf{d}'}(\mathbf{S}_2), \mathbf{d}'_2 \rangle$. Similarly for C_f , given $\langle \mathbf{S}_2, \mathbf{d}_2 \rangle$, a cache hit can happen only when there exists an entry $\langle \mathbf{S}_1, \mathbf{d}_1 \rangle$ in cache such that $\mathbf{d}_1 = \mathbf{d}_2$ and $\mathbf{S}_1 \supseteq \mathbf{S}_2$. Therefore, the output from cache will be $post_{\mathbf{d}, \delta}(\mathbf{S}_1) \supseteq post_{\mathbf{d}, \delta}(\mathbf{S}_2)$.

6 Experimental Evaluation

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 [21] and a 6-d drone with a NN-controller [22]. 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 Section 2.

Algorithm 2

```

1: Global  $C_g, C_f$ 
2: function FLOWCACHE( $\langle S, d \rangle$ )
3:   if  $\langle S, d \rangle \notin C_f$  then  $C_f(\langle S, d \rangle) \leftarrow \text{postCont}(S, d, \delta)$ 
4:   return  $\langle C_f(\langle S, d \rangle), d \rangle$ 
5: function VERIFYINC( $\langle X^0, d^0 \rangle, H, m$ )
6:    $root \leftarrow \langle X^0, d^0 \rangle$ ;  $depth \leftarrow 0$ 
7:    $queue \leftarrow [\langle X^0, d^0, depth \rangle]$ 
8:   while  $queue \neq \emptyset$  and  $depth \leq m$  do
9:      $\langle \langle S, d \rangle, depth \rangle \leftarrow queue.\text{dequeue}()$ 
10:    for  $d'$ , s.t.  $G(d, d')$  do
11:       $\langle s, \langle S', d' \rangle \rangle \leftarrow C_g(\langle S, d \rangle, G(d, d'), R(d, d'))$  ▷ Read  $C_g$ 
12:      if  $s \neq \text{unsat}$  then ▷ if  $s = \text{unsat}$  then continue to next  $d'$ 
13:        if  $s = \text{unknown}$  then
14:          if  $S \cap G(d, d') \neq \emptyset$  then
15:             $\langle S', d' \rangle \leftarrow \langle \text{postDisc}(S, d, d'), d' \rangle$ 
16:             $C_g(\langle \langle S, d \rangle, G(d, d'), R(d, d') \rangle) \leftarrow \langle \text{sat}, \langle S', d' \rangle \rangle$  ▷ Record in  $C_g$ 
17:          else
18:             $C_g(\langle \langle S, d \rangle, G(d, d'), R(d, d') \rangle) \leftarrow \langle \text{unsat}, \text{none} \rangle$  ▷ Record in  $C_g$ 
19:            continue
20:         $\langle S'', d'' \rangle \leftarrow \text{flowCache}(\langle S', d' \rangle)$ 
21:         $\langle S, d \rangle.\text{addChild}(\langle S'', d'' \rangle)$ 
22:         $queue.\text{add}(\langle \langle S'', d'' \rangle, depth + 1)$ 
23:    return  $root$ 

```

6.1 Evaluation of core features

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 1m. Fig. 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.

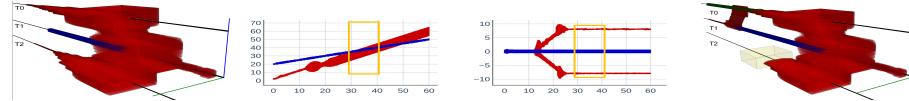


Fig. 4: Left to right: (1) Computed reachtubes for a 2-drone scenario; (2) same reachtube projected on x -dimension, and (3) on z -dimension. Since there is no overlap in space-time, no collision. (4) Reachtube for a 3-drone scenario, the red drone violates the safety condition by entering the unsafe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

Checking multiple safety assertions. Verse supports multiple safety assertions specified using `assert` statements. For example, the user can specify unsafe regions (Line 77-80) or check safe separation between agents (Line 81-83) as shown in Fig. 5. We extend the two-drone scenario described in the previous paragraph by adding a second D-NPV 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's no violation of safe separation between agents in this scenario. Verse allow users to extract 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"
83     return next

```

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. Fig. 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 [5] format, which enables users to easily create scenarios with interesting maps. An example is shown in Fig. 8 in Appendix.

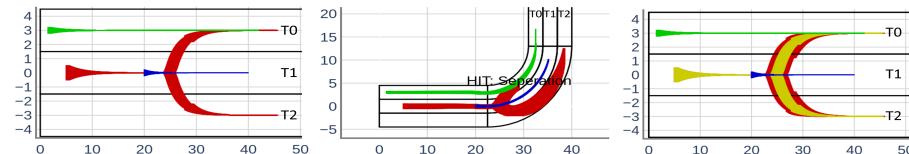


Fig. 6: *Left:* running the three car scenario on map with parallel straight lanes. *Center:* same scenario with a curved map. *Right:* same scenario with a noisy sensor.

Adding noisy sensors. Verse supports the ability to verify 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` Verse library function.

Fig. 6 shows exactly the same three-car scenario as in the previous paragraph but with a noisy sensor, which adds $\pm 0.5m$ 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 ego agent, 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, which results in 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 hyper-rectangles. Currently, Verse implements computing `postCont` using DryVR [12], NeuReach [33] and Mixed Monotone Decomposition [11]. A scenario with two car agents in map $M1$ verified using NeuReach and DryVR is shown in Fig. 9 in the Appendix.

Table 1 summarizes the running time of verifying all the examples in this section on a standard Intel Core i7-11700K @ 3.60GHz CPU desktop. 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 NeuReach⁴ as the reachability engine takes more time than using DryVR as the reachability engine.

6.2 Incremental Verification

The incremental verification algorithm `verifyInc` is evaluated by repeatedly verifying (and simulating) scenarios with slight modifications. In this example, the scenario contains 3 C-CA agents and 5 C-NPV agents in map $M2$. The simulation and verification results for the scenario are shown in Fig. 12 in Appendix C. The running time for simulation is 19.96s and that for verification is 470.01s. We show the results for 3 experiments for both simulation and verification, which corresponds to the 3 rows in each section of Table 2. In the *repeat* experiments, we perform analysis twice on the same scenarios ($H_2 = H_1$), which shows the most favorable benefits of incremental verification. In the *change init* and *change ctrl* experiments, we respectively change the initial conditions and the decision logic for one of the C-CA agents between the 2 experiment runs.

⁴ Run time for NeuReach includes training time.

Table 1: Runtime for verifying examples in Section 6.1. Columns are: number of agents ($\#\mathcal{A}$), agent type (\mathcal{A}), map used (Map), reachability engine used (`postCont`), sensor type (Noisy \mathcal{S}), number of mode transitions #Tr, and the total run time (Run time).

$\#\mathcal{A}$	\mathcal{A}	Map	<code>postCont</code>	Noisy \mathcal{S}	#Tr	Run time (s)
2	Q	$\mathcal{M}6$	DryVR	No	8	55.9
2	Q	$\mathcal{M}5$	DryVR	No	5	18.7
2	Q	$\mathcal{M}5$	NeuReach	No	5	1071.2
3	Q	$\mathcal{M}5$	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

We perform each of these experiments with incremental verification turned off and on, which corresponds to the `verify` and `verifyInc` columns.

From the experimental results we can see that `verifyInc` indeed reduces the time needed for simulation and verification. This time reduction is also correlated with the similarity between the scenarios, with the *repeat* experiment being able to achieve an almost 10x speedup, while changing the initial conditions leads to almost no benefit.

Table 2: Experimental results for the Incremental Verification algorithm. The table shows the number of mode transition (#Tr), run-times in seconds (run time), the memory usage of Verse in megabytes (memory), the size of C_g and C_f in megabytes (cache size), and the hit rate of the caches (hit rate).

		<code>verify</code>		<code>verifyInc</code>			
		#Tr	run time	memory	Run time	memory	cache size
Simulation	repeat	45	12.18	431	1.05	435	3.83
	change init	24	10.17	431	9.3	436	4.07
	change ctrl	45	11.45	429	5.98	438	4.38
Verification	repeat	105	450.89	498	55.34	482	3.23
	change init	49	365.91	485	349.68	498	3.7
	change ctrl	93	421.65	498	230.12	490	4.0

6.3 White-box uncertain continuous dynamics

Verse provides an implementation of `postCont` using algorithms from [1,2,11] which is applicable to agents with white-box uncertain continuous dynamics. Given a white-box dynamical system defined by a differential equation $\dot{x} =$

$f(x, w)$, where the state $x \in \mathcal{X} \subseteq \mathbb{R}^n$ and the bounded disturbance input $w \in [w_l, w_u]$. The idea of the approach in [11] is to find a decomposition function $d()$ which can then be used to define an augmented system:

$$\begin{bmatrix} \dot{x}_l \\ \dot{x}_u \end{bmatrix} = \begin{bmatrix} d(x_l, w_l, x_u, w_u) \\ d(x_u, w_u, x_l, w_l) \end{bmatrix},$$

such that the reachable set of the original noisy system at time T , from any initial set $\mathcal{X}_0 = [\underline{x}, \bar{x}] \subseteq \mathcal{X}$ is contained in $[x_l(T), x_u(T)]$. The latter can be computed by simulating the decomposed system from the singleton initial state $\langle x_l = \underline{x}, x_u = \bar{x} \rangle$. Verse currently requires users to provide the decomposition function $d()$ to handle systems with uncertainty in dynamics. Consider an example system and the corresponding manually derived decomposition function:

$$\begin{aligned} \dot{x}_1 &= x_1(1.1 + w_1 - x_1 - 0.1x_2) \\ \dot{x}_2 &= x_2(4 + w_2 - 3x_1 - x_2) \end{aligned} \quad d(x, \hat{x}, w, \hat{w}) = \begin{bmatrix} x_1(1.1 + w_1 - x_1 - 0.1\hat{x}_2) \\ x_2(4 + w_2 - 3\hat{x}_1 - x_2) \end{bmatrix}$$

With $x_1, x_2 \in [0.3, 2]$ and $w_1, w_2 \in [-0.1, 0.1]$. Verse can automatically construct the augmented system and compute the reachable states for that system (Sample results are shown in Fig. 10 of Appendix C).

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. Verse allows users to create agents with decision logics expressed in Python, scenarios with different types of agents, and it provides functions for performing systematic simulation and verification through reachability analysis. Verse maps can be imported from a standard open format, and they allow agents to be ported across different compatible maps, offering flexibility in creating scenarios. The agent decision logics allow non-deterministic decision making and the reachability functions can propagate the uncertainty in the initial condition through all decision branches. The safety requirements written using `assert` statements enable various safety conditions. The incremental verification algorithm in Verse enables the user to rapidly verify and improve their decision logic. We illustrate useful capabilities and use cases of Verse through various examples.

There are several exciting future directions for and around 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. In incremental verification (and simulation), our `verifyInc` merely opens the door for a invention that exploit small changes in maps and continuous dynamics. Functions for constructing and systematically sampling scenarios could be developed. Functions for post-computation for white-box models by building connections with existing tools [3,9,13] would be a natural next step. Those approaches could obviously utilize the symmetry property of agent dynamics as in [30,32], but beyond that,

new types of symmetry reductions should be possible by exploiting the map geometry.

References

1. Abate, M.: Mixed Monotonicity for Efficient Reachability with Applications to Robust Safe Autonomy. Ph.D. thesis, Georgia Institute of Technology (2020)
2. Abate, M., Coogan, S.: Computing robustly forward invariant sets for mixed-monotone systems. In: 2020 59th IEEE Conference on Decision and Control (CDC). pp. 4553–4559 (2020)
3. Althoff, M.: An introduction to CORA 2015. In: Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems (2015)
4. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. *Theoretical Computer Science* **138**(1), 3–34 (1995)
5. Association for Standardization of Automation and Measuring Systems (ASAM): Open dynamic road information for vehicle environment (Aug 2021), <https://www.asam.net/standards/detail/opendrive/>
6. 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)
7. 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. p. 23–32. HSCC ’19, Association for Computing Machinery, New York, NY, USA (2019)
8. Brittain, M., Alvarez, L.E., Breeden, K., Jessen, I.: AAM-Gym: Artificial intelligence testbed for advanced air mobility (2022)
9. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Computer Aided Verification (CAV). pp. 258–263. Springer (2013)
10. 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. pp. 109–130. Springer, Cham (2022)
11. 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)
12. Fan, C., Qi, B., Mitra, S., Viswanathan, M.: Dryvr: Data-driven verification and compositional reasoning for automotive systems. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification (CAV). pp. 441–461. Springer, Cham (2017)
13. Fan, C., Qi, B., Mitra, S., Viswanathan, M., Duggirala, P.S.: Automatic reachability analysis for nonlinear hybrid models with C2E2. In: Computer Aided Verification (CAV). pp. 531–538 (2016)
14. Federal Aviation Administration: Unmanned Aircraft System Traffic Management (UTM) Concept of Operations Version 2.0 (Mar 2020)
15. 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.) Formal Methods. pp. 367–386. Springer, Cham (2021)

16. Frehse, G., Guernic, C.L., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Computer Aided Verification (CAV). pp. 379–395 (2011)
17. Fremont, D.J., Dreossi, T., Ghosh, S., Yue, X., Sangiovanni-Vincentelli, A.L., Sesia, 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. p. 63–78. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019)
18. 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.) Automated Deduction - CADE-25. pp. 527–538. Springer, Cham (2015)
19. Goldenberg, D.K., Lin, J., Morse, A.S.: Towards mobility as a network control primitive. In: MobiHoc '04: Proceedings of the 5th ACM international symposium on Mobile ad hoc networking and computing. pp. 163–174. ACM Press (2004)
20. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? *Journal of Computer and System Sciences* **57**(1), 94–124 (1998)
21. 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)
22. 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)
23. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Synthesis Lectures on Computer Science, Morgan Claypool (November 2005), also available as Technical Report MIT-LCS-TR-917, MIT
24. Lim, H., Kaynar, D., Lynch, N., Mitra, S.: Translating timed I/O automata specifications for theorem proving in PVS. In: Proceedings of Formal Modelling and Analysis of Timed Systems (FORMATS'05). No. 3829 in LNCS, Springer, Uppsala, Sweden (September 2005)
25. Lopez, P.A., Behrisch, M., Bieker-Walz, L., Erdmann, J., Flötteröd, Y.P., Hilbrich, R., Lücke, L., Rummel, J., Wagner, P., Wießner, E.: Microscopic traffic simulation using SUMO. In: The 21st IEEE International Conference on Intelligent Transportation Systems. IEEE (2018)
26. Manfredi, G., Jestin, Y.: An introduction to ACAS Xu and the challenges ahead. In: 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC). pp. 1–9. IEEE (2016)
27. Minghao Jiang and Zexiang Liu and Kristina Miller and Dawei Sun and Arnab Datta and Yixuan Jia and Sayan Mitra and Necmiye Ozay: Graic: A simulator framework for autonomous racing. <https://popgri.github.io/Race/> (2021)
28. Python Software Foundation: Python full grammar specification (Oct 2022)
29. Ray, R., Gurung, A., Das, B., Bartocci, E., Bogomolov, S., Grosu, R.: Xspeed: Accelerating reachability analysis on multi-core processors. In: Piterman, N. (ed.) Hardware and Software: Verification and Testing. pp. 3–18. Springer, Cham (2015)
30. Sibai, H., Li, Y., Mitra, S.: SceneChecker: Boosting scenario verification using symmetry abstractions (2021)
31. Sibai, H., Mokhlesi, N., Fan, C., Mitra, S.: Multi-agent safety verification using symmetry transformations. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 173–190. Springer, Cham (2020)

32. 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.) *Automated Technology for Verification and Analysis*. pp. 98–114. Springer, Cham (2019)
33. Sun, D., Mitra, S.: Neureach: Learning reachability functions from simulations. In: Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I. pp. 322–337 (2022)
34. Wu, C., Kreidieh, A., Parvate, K., Vinitsky, E., Bayen, A.M.: Flow: Architecture and benchmarking for reinforcement learning in traffic control. ArXiv [abs/1710.05465](https://arxiv.org/abs/1710.05465) (2017)
35. Ölveczky, P.C., Meseguer, J.: Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science* **285**(2), 359–405 (2002), rewriting Logic and its Applications

A Example Maps

The figure for maps used in the examples in Section 6 are shown in Fig. 7.

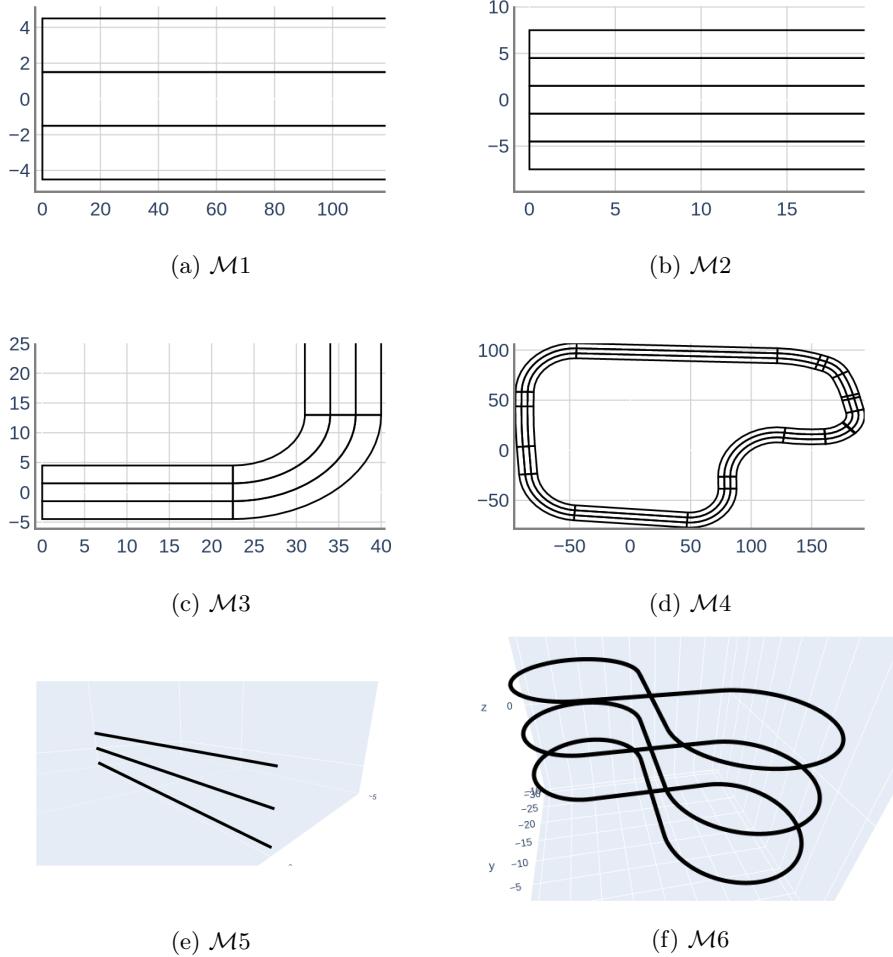


Fig. 7: Maps in examples

B Grammar for Decision Logic Code

This is the subset of the Python grammar [28] that Verse support for coding the decision logic of agents. Some details about operator precedence are not given

here. Currently, the supported OPERATOR include logic operators (`and`, `or`, `and not`), comparison operators (`<=`, `<`, `==`, `!=`, `>`, and `>=`), and arithmetic operators (`+`, `-`, `*` and `/`).

```

expression:
    | expression OPERATOR expression
    | 'lambda' parameters ':' expression
    | literal
    | dotted_name tuple
literal:
    | '[' '-' ] NUMBER
    | STRING+
    | 'None' | 'True' | 'False'
    | tuple | group | genexp
expressions: ',' expression*
tuple: '(' expressions ')'
group: '(' expression ')'
genexp: '(' expression for_if_clauses+ ')'
for_if_clause: 'for' IDENT 'in' expression ('if' expression )*
function_def: 'def' IDENT '(' [params] ')' ['->' expression ] ':' block
params: (maybe_typed_name ',')* maybe_typed_name?
maybe_typed_name: IDENT [':'] (expression | STRING)
block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts
statements: statement+
statement: compound_stmt | simple_stmts
statement_newline: compound_stmt NEWLINE | simple_stmts | NEWLINE | EOF
compound_stmt: function_def | if_stmt | class_def
simple_stmts: ';' simple_stmt+ ';' NEWLINE
simple_stmt: assignment | return_stmt | assert_stmt | if_stmt
assignment: IDENT '=' expression
return_stmt: 'return' [expression]
assert_stmt: 'assert' expression [',' expression ]
if_stmt: 'if' expression ';' block [else_block]
else_block: 'else' ':' block
dotted_as_names: ',' dotted_as_name+
dotted_as_name: dotted_name ['as' NAME ]
dotted_name: dotted_name '.' NAME | NAME

```

C Additional Examples

The verification result for a 3-car scenario with a map imported from Open-DRIVE format is shown in Fig. 8.

The verification result with `postCont` computed by NeuReach and DryVR is shown in Fig. 9.

The plotted reachtube for x_1 and x_2 for system with uncertain dynamics in Section 6.3 is shown in Fig. 10.

The reachtube plot for the 7-car scenario described in row 5 of Table 1 is shown in Fig. 11.

The simulation and verification result for the baseline 8-car system used for experimenting incremental verification in Section 6.2 is shown in Fig. 12

The baseline scenario for the incremental verification experiments is specified in Fig. 13 for simulation and in Fig. 14 for verification. In the *change init*

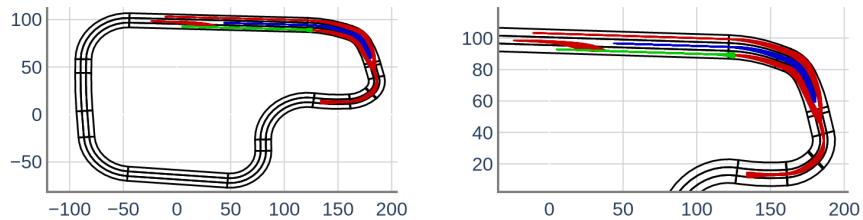


Fig. 8: Picture showing verification result for a scenario with map imported from OpenDRIVE format.

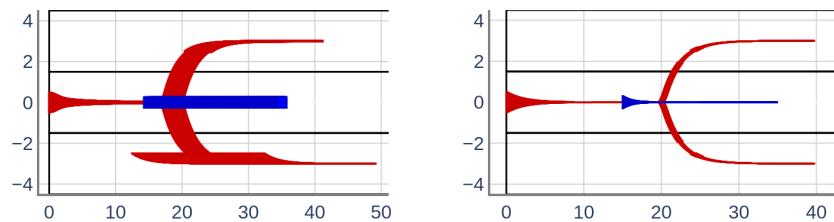


Fig. 9: Picture on the left showing scenario with `postCont` computed by NueReach. Picture on the right showing same scenario with `postCont` computed by DryVR.

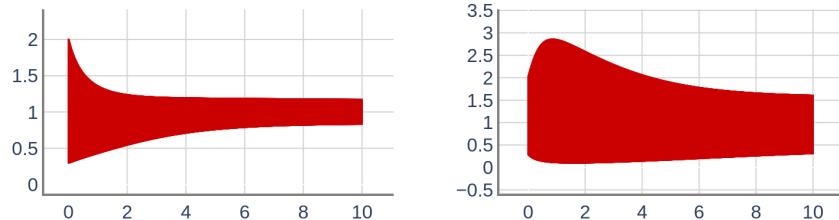


Fig. 10: Reachtube plot for x_1 (left) and x_2 (right) for the example system.

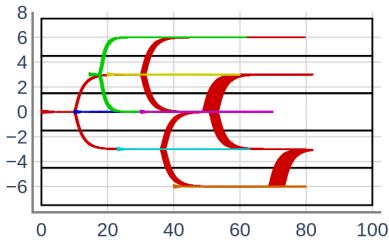


Fig. 11: Reachtube plot for 7-car scenario

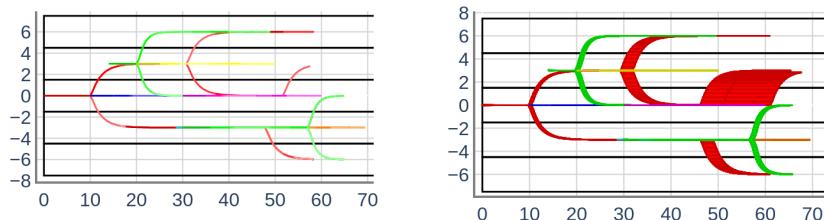


Fig. 12: Simulation (left) and reachtube plot (right) for scenario in incremental verification experiment.

experiment, the initial condition for `car7` is changed to $[[50, -3, 0, 0.5], [50, -3, 0, 0.5]]$. In the *change ctr* experiment, the controller for `car8` is changed so that it switches tracks if there is another car less than 4.5 meters in front of it instead of 5 meters.

Fig. 13: The scenario specification for the simulation baseline in the incremental verification experiments

```

scenario = Scenario()
controller_file = '...'
car1 = CarAgent('car1', file_name=controller_file)
car1.set_initial([[0, 0, 1.0], [0, 0, 1.0]], (TacticalMode.Normal, TrackMode.T1))
scenario.add_agent(car1)
car2 = NPCAgent('car2')
car2.set_initial([[10, 0, 0, 0.5], [10, 0, 0, 0.5]], (TacticalMode.Normal, TrackMode.T1))
scenario.add_agent(car2)
car3 = CarAgent('car3', file_name=controller_file)
car3.set_initial([[14, 3, 0, 0.6], [14, 3, 0, 0.6]], (TacticalMode.Normal, TrackMode.T0))
scenario.add_agent(car3)
car4 = NPCAgent('car4')
car4.set_initial([[20, 3, 0, 0.5], [20, 3, 0, 0.5]], (TacticalMode.Normal, TrackMode.T0))
scenario.add_agent(car4)
car5 = NPCAgent('car5')
car5.set_initial([[30, 0, 0, 0.5], [30, 0, 0, 0.5]], (TacticalMode.Normal, TrackMode.T1))
scenario.add_agent(car5)
car6 = NPCAgent('car6')
car6.set_initial([[28.5, -3, 0, 0.5], [28.5, -3, 0, 0.5]], (TacticalMode.Normal, TrackMode.T2))
scenario.add_agent(car6)
car7 = NPCAgent('car7')
car7.set_initial([[39.5, -3, 0, 0.5], [39.5, -3, 0, 0.5]], (TacticalMode.Normal, TrackMode.T2))
scenario.add_agent(car7)
car8 = CarAgent('car8', file_name=controller_file)
car8.set_initial([[30, -3, 0, 0.6], [30, -3, 0, 0.6]], (TacticalMode.Normal, TrackMode.T2))
scenario.add_agent(car8)

```

The simulation and verification result for the 8-car system after changing initial condition is shown in Fig. 15. The agent with changed controller is marked with the red box in the plot.

The simulation and verification result for the 8-car system after changing controller is shown in Fig. 16. The agent with changed controller is marked with the red box in the plot.

Fig. 14: The scenario specification for the verification baseline in the incremental verification experiments

```

scenario = Scenario()
controller_file = '...'
car1 = CarAgent('car1', file_name=controller_file)
car1.set_initial([[0, -0.05, 0, 1.0], [0, 0.05, 0, 1.0]], (TacticalMode.Normal, TrackMode.T1))
scenario.add_agent(car1)
car2 = NPCAgent('car2')
car2.set_initial([[10, 0, 0, 0.5], [10, 0, 0, 0.5]], (TacticalMode.Normal, TrackMode.T1))
scenario.add_agent(car2)
car3 = CarAgent('car3', file_name=controller_file)
car3.set_initial([[14, 2.95, 0, 0.6], [14, 3.05, 0, 0.6]], (TacticalMode.Normal, TrackMode.T0))
scenario.add_agent(car3)
car4 = NPCAgent('car4')
car4.set_initial([[20, 3, 0, 0.5], [20, 3, 0, 0.5]], (TacticalMode.Normal, TrackMode.T0))
scenario.add_agent(car4)
car5 = NPCAgent('car5')
car5.set_initial([[30, 0, 0, 0.5], [30, 0, 0, 0.5]], (TacticalMode.Normal, TrackMode.T1))
scenario.add_agent(car5)
car6 = NPCAgent('car6')
car6.set_initial([[28.5, -3, 0, 0.5], [28.5, -3, 0, 0.5]], (TacticalMode.Normal, TrackMode.T2))
scenario.add_agent(car6)
car7 = NPCAgent('car7')
car7.set_initial([[39.5, -3, 0, 0.5], [39.5, -3, 0, 0.5]], (TacticalMode.Normal, TrackMode.T2))
scenario.add_agent(car7)
car8 = CarAgent('car8', file_name=controller_file)
car8.set_initial([[30, -3.05, 0, 0.6], [30, -2.95, 0, 0.6]], (TacticalMode.Normal, TrackMode.T2))
scenario.add_agent(car8)

```

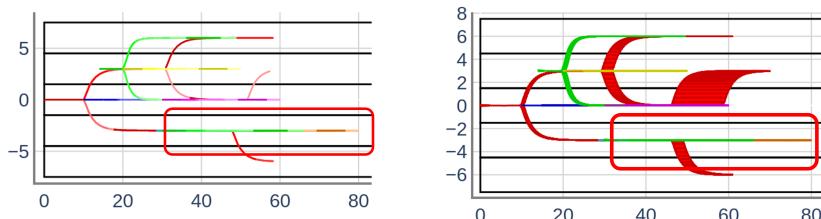


Fig. 15: Simulation (left) and reachtube plot (right) for scenario after changing initial condition in incremental verification experiment.

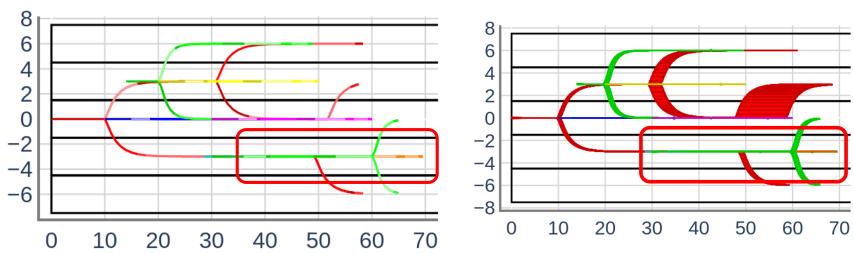


Fig. 16: Simulation (left) and reachtube plot (right) for scenario after changing controller in incremental verification experiment.