The Verse Library

Presented By : ZHANG-Boyuan From CPS M1

Related paper :

Verse: *A Python Library for Reasoning About Multi-agent Hybrid System -Yangge Li , Haoging Zhu, Katherine Braught , Keyi Shen*

And documentation :

Welcome to Verse documentation! — verse 0.2 documentation (autoverse-ai.github.io)

CONTENT :

1. Introduction

The Verse library is developed to enhance the usability of hybrid system verification in multi-agent scenarios. In Verse, decision-making agents navigate within a map and interact through sensors. Each agent's decision logic is scripted in a subset of Python, while the continuous dynamics are managed by a black-box simulator. Verse allows the instantiation of multiple agents and their deployment across different maps to create diverse scenarios. Additionally, it offers functions for simulating and verifying these scenarios using existing reachability analysis algorithms.

2. Analyze the Structure of a Verse Scenario

2.0 Definition of Verse Scenario

A Verse scenario is defined by three main components:

- **Map**: Specifies the tracks that agents can follow.
- **Agents**: A set of agents participating in the scenario.
- **Sensor**: If multiple agents exist, a sensor function defines the observable variables among agents.

2.1 Instantiate Map

The map of a scenario outlines the permissible tracks for agent movement. Tracks are continuous functions within a defined workspace, and each track is associated with a specific track mode. The map abstraction ensures scenarios are succinct and agents can be ported across different maps seamlessly. Verse also provides functions to generate map objects from OpenDRIVE files, enhancing versatility.

2.2 Agents

Agents in Verse are characterized by:

- **Modes**: Define the agent's current state or behavior.
- **Continuous State Variables**: Represent the agent's dynamic properties.
- **Decision Logic**: Determines the agent's transitions between modes based on interactions.
- **Flow Function**: Defines the continuous evolution of the agent's state over time.

Agents are compatible with a map if their tactical modes are a subset of the map's allowed modes. This compatibility allows the same agent to operate on different maps without modification.

2.3 Sensors and Scenarios

Sensors define which variables of an agent are visible to others. By default, all variables are observable, but this can be customized to include bounded noise or limit visibility. A scenario is a combination of a map, a collection of compatible agents, and a sensor function. This setup allows for the creation of heterogeneous agents with varying decision logics and dynamics.

3. Verse Scenario to Hybrid Verification

Verse translates a scenario into an underlying hybrid system, facilitating verification tasks. The hybrid automaton representation includes:

- **State Space (X)**: Continuous state variables.
- **Mode Space (D)**: Discrete modes of agents.
- **Guard Conditions (G)**: Conditions for mode transitions.
- **Reset Functions (R)**: Define state updates during transitions.
- **Trajectory Set (TL)**: Defines the evolution of continuous states over time.

Verification involves checking whether the reachable states of the system violate any safety assertions. Verse supports various verification algorithms and allows users to implement custom reachability tools.

3.1 Verification Algorithms in Verse

Verse includes built-in verification algorithms and provides interfaces for integrating new ones. The primary algorithm constructs an execution tree up to a specified depth, exploring all possible mode transitions and checking safety conditions at each step. Users can plug in different reachability tools like DryVR and NeuReach to optimize verification processes.

3.2 Incremental Verification

Verse offers an incremental verification algorithm, `verifyInc`, which caches and reuses reachable sets to reduce computation time for similar scenarios. This approach significantly speeds up verification, especially when analyzing sequences of slightly altered scenarios.

4. Analyze with the Result of Example

I conducted experiments based on the examples provided in the `tutorial.ipynb` file, involving one or two drones flying forward while ascending to avoid obstacles. By adjusting certain parameters and adding some import statements, I achieved results that aligned with expectations. Among the various maps, **Map M3** proved to be the most complex, and I will use it as an example for detailed analysis.

4.0 Features of Map M3

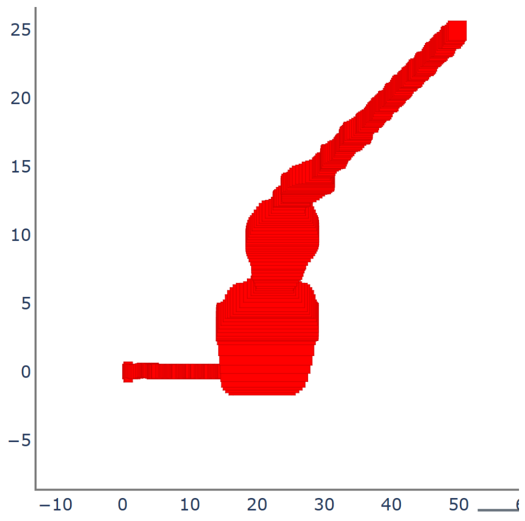
To understand why the example in the tutorial produces the observed images, it is essential to first consider the key features of **Map M3**:

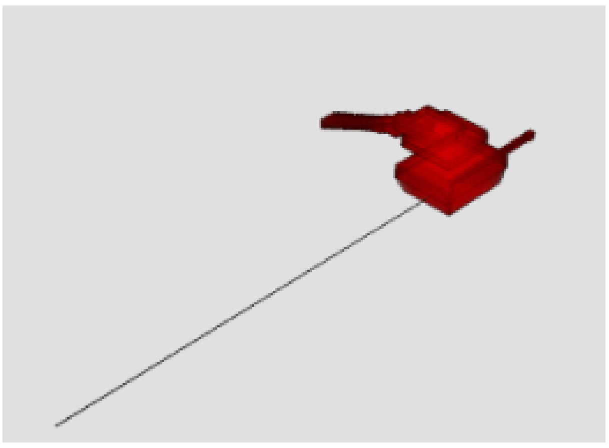
- **Map M3** contains two types of lanes:
    - **T0**: Horizontal straight segments.
    - **TAvoidUp**: Paths requiring drones to ascend to avoid obstacles. This mode connects to the end of **T0**.

It is important to note that the red color represents the **Reach Tube**, which is the set of possible positions a drone can occupy at different times.

4.1 Detailed Analysis: The Case with Only 1 Drone

By observing the 2D graph, it is evident that initially, the drone operates on the horizontal lane (**T0**) in a straight path. At this stage, the **Reach Tube** is almost a straight line, indicating low uncertainty. As the drone approaches the obstacle location, approximately at **x > 20**, it switches to the **AvoidUp** mode. Consequently, the trajectory follows an upward diagonal, and uncertainty increases sharply. This phenomenon is more pronounced in the 3D graph.





This behavior can be attributed to three main factors:

1. **Uncertainty in Control Inputs**
2. **Sensitivity of System Dynamics in the Vertical Direction**
3. **Accumulation of Time Errors**

These factors become even more evident in subsequent examples.

The initial error source causing the **Reach Tube** to expand continuously is defined within the drone's `runmodel`. Errors are assigned to movements in all three directions, representing an idealized simulation of real-world conditions. In reality, errors in each direction are not uniform, and environmental noise and disturbances contribute significantly. I believe that a crucial direction for future development is improving the representation of environmental bias.

```
bias1 = torch.FloatTensor(bias1)
bias2 = torch.FloatTensor(bias2)
bias3 = torch.FloatTensor(bias3)
weight1 = torch.FloatTensor(weight1)
weight2 = torch.FloatTensor(weight2)
weight3 = torch.FloatTensor(weight3)
```

4.1.1 Uncertainty in Control Inputs

When switching to the **AvoidUp** mode, new control inputs (such as upward velocity or acceleration) introduce uncertainty, especially in real systems where control commands are subject to noise or errors.

4.1.2 Sensitivity of System Dynamics in the Vertical Direction

- Vertical Sensitivity: The change in z_dot (vertical velocity) is entirely dependent on vz_dot (vertical input).
- Switching to the AvoidUp mode introduces rapid state changes due to vz_dot (upward acceleration).

Input Amplification Effect: Due to the dynamic changes in vertical acceleration, the state `z` undergoes significant changes in a short period, increasing sensitivity to input errors and accelerating the expansion of the **Reach Tube**. For example, the drone's dynamics equation `dynamic(t, state, u)` includes definitions such as

```
tmp1 = dvx * np.cos(sc) - dvy * np.sin(sc)
tmp2 = dvx * np.sin(sc) + dvy * np.cos(sc)
 dvx = tmp1
 dvy = tmp2
 dvz = u3 - 9.81
 dx = vx
 dy = vy
 dz = vz
```

4.1.3 Accumulation of Time Errors

Time accumulation errors arise from the discrete solving of dynamic equations. During simulation, the system state is numerically integrated, and errors accumulate over time. This is evident in the `TC_simulate(mode, init, time_bound, time_step, lane_map=None)` function, where time steps and numerical integration are implemented.

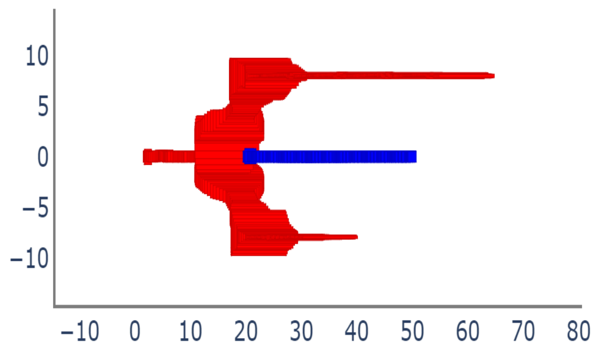
4.2 Computed Reach Tubes for a 2-Drone Scenario

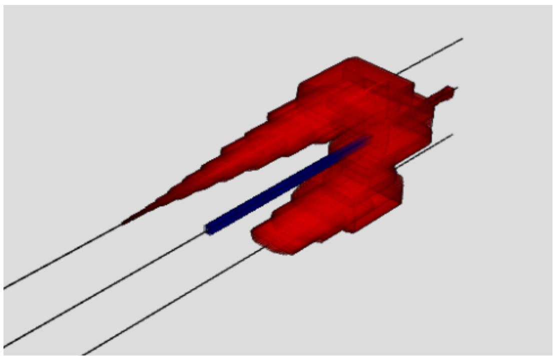
Building upon the single-drone scenario, safety requirements dictate that drones must not collide with obstacles or with each other. The visualization below illustrates:

- **Red (D-CA Agent)**: Represents the **Collision Avoidance (D-CA) agent**'s **Reach Tube**.
- **Blue (D-NPV Agent)**: Represents the **No Planned Variation (D-NPV) agent**'s **Reach Tube**.
- **Safety Constraint**: The distance between the two drones must always be at least **1 meter**.

Visualization of Verification

From the figure, it is clear that the **Reach Tubes** of the **D-CA agent** and **D-NPV agent** do not overlap, indicating that the drones do not occupy the same position at the same time and thus maintain the required safety distance.



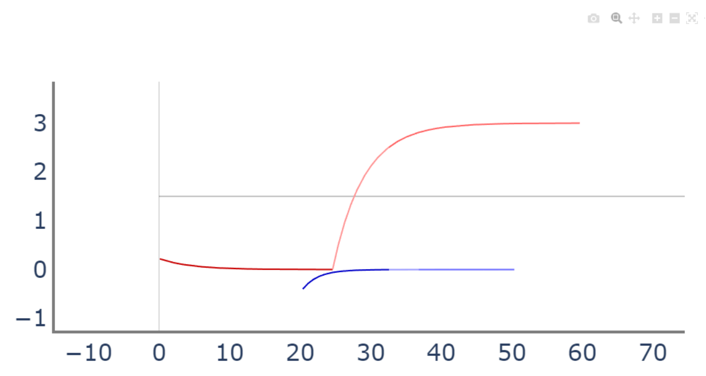


4.3 Detailed Analysis of Safety Constraints

The two charts below demonstrate the safety analysis scenario with two drones (**D-CA agent** and **D-NPV agent**), reflecting the system's state changes over time, particularly in the vertical (**z-axis**) and horizontal (**x-axis**) directions.

First Chart

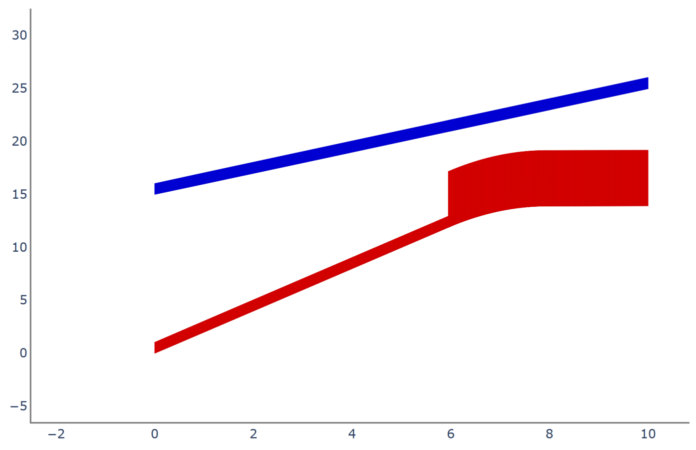
- **Trajectory Evolution**: Displays the evolution of a single time step's trajectory without uncertainty boundaries, making it easier to identify state transitions.
- **Blue Trajectory**: Shows a stable upward trend, indicating that the **D-NPV agent** remains in normal flight mode.
- **Red Trajectory**: Exhibits a noticeable curvature change around time **6-8**, indicating a state change (e.g., entering **AvoidUp** mode).

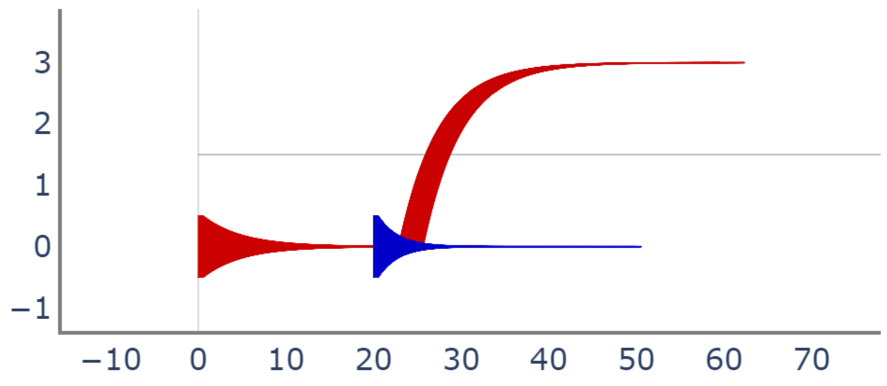


Second and Third Charts

- **Reach Tube Information**: Incorporates uncertainty and state evolution, demonstrating system safety over time and height.
- **Blue Area**: Relatively narrow, indicating low uncertainty for the **D-NPV agent**.
- **Red Area**: Rapidly expands around time **6-8**, showing significant state changes and increased uncertainty in the vertical (**z-axis**) direction after entering **AvoidUp** mode.

This rapid expansion of the **Reach Tube** reflects the sensitivity and error accumulation introduced by the **AvoidUp** control inputs and dynamic changes, consistent with the earlier analysis of a single agent.





5. Another Example: Analysis of Two Bouncing Balls

In this example, we consider a scenario with two **ball agents**—a **red-ball** and a **green-ball**—simulated within a bounded area. The key idea is that each ball moves under basic kinematic rules and bounces elastically off the boundaries. The conditions for bouncing are encoded in the **decisionLogic** function, which checks for collisions with boundaries at **x=0**, **y=0**, **x=20**, and **y=20**, and adjusts the velocities accordingly. When a ball hits a boundary, its velocity is reversed and scaled down by a factor (e.g., *cr=0.85*) to mimic energy loss during the impact.

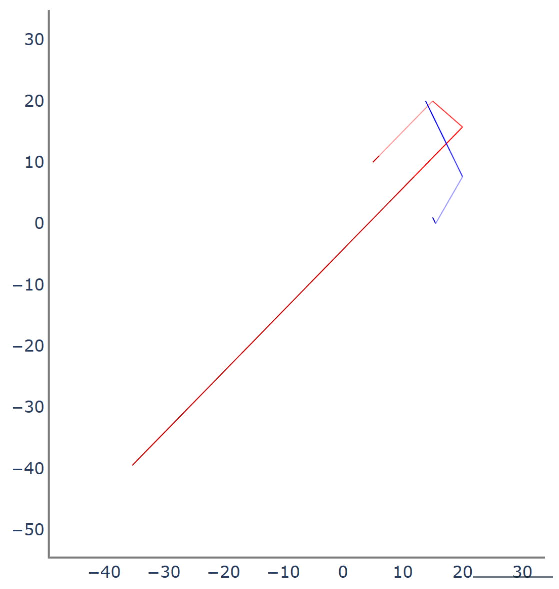


Figure Interpretation:
The provided figure shows trajectories for the two balls. The **red trajectory** corresponds to the **red-ball** and the **blue trajectory** corresponds to the **green-ball**. Each line segment represents the evolution of the ball's position over time. Initially, both balls move according to their initial velocities. As they approach the domain boundaries, their paths change direction due to bouncing events triggered by the **decisionLogic** function.

In the figure, the red-ball starts near **(5,10)** and moves diagonally towards the top-right region, while the green-ball begins near **(15,1)** and moves in a different direction. Over time, both balls may hit either the horizontal or vertical walls, causing them to bounce and reverse their trajectories. These directional changes are easily identifiable as points where the trajectory line changes slope abruptly.

Simulation Details and Traces:
During execution, the system prints out snapshots of the simulation state at various times:

- At **t=0**, the initial conditions show the red-ball at **(5, 10, vx=2.0, vy=2.0)** and the green-ball at **(15, 1, vx=1.0, vy=-2.0)**, both in **NORMAL** mode.
- At intermediate steps (e.g., **t=0.51**, **t=5.0**, **t=7.5**), the system logs updated positions and velocities. These logs illustrate how the green-ball moves upward and to the right

before bouncing and how the red-ball, after ascending, hits a boundary and rebounds with adjusted velocity.

Max Depth Reached:
The log entry *"max depth reached"* indicates that the simulation reached its predefined iteration or branching limit. In other words, to prevent infinite loops or excessive computation time, the simulation restricts the depth of its exploration. Once this limit is hit, the simulation halts. Such limits are common in scenario analysis to ensure tractability and to prevent runaway computations.

Conclusions for comparing 2 examples

The two-ball scenario demonstrates how Verse can simulate agents interacting with constrained environments. By encoding simple physical-like laws (elastic collisions with boundaries and reduced velocities after impacts), one can observe complex and realistic trajectories. The resulting visualization clearly shows how each ball's path evolves with time, the effect of boundary conditions, and how uncertainty or changes in the initial setup would influence the system's behavior.

Compared to the previous drone-based scenarios, this bouncing-ball setup is simpler in terms of dynamics and decision logic, yet it still provides insights into system behavior under bounded conditions. It highlights how even straightforward rules can produce a variety of behaviors when executed over extended time frames, ultimately reinforcing the importance of verification and controlled simulation depths in multi-agent and multi-modal scenarios.

6. Problem encountered and further developing

6.1 Code Implementation Issues

Compatibility with Python 3.13: Some functions conflict with the latest Python 3.13 environment, preventing certain visualization tools such as plotter and pyvista from functioning properly. As a workaround, setting
python `pio.renderers.default = "browser"`

1. **allows the output to be displayed in the web browser.**
2. **Dependency Management: Running the provided examples requires installing a large number of packages. Since these dependencies are not well-integrated**

into a unified requirements file, installation had to be done separately, leading to repeated installations and prolonged setup times.

3. **Windows Environment Permissions: On Windows, limited permissions in the Python environment may prevent modifications. To address this, creating a venv (virtual environment) is recommended for proper isolation and control.**

6.2 Application-Level Considerations

1. **Visualization Overlaps: The generated 2D images often suffer from overlapping elements. Additionally, to view detailed map information, manual adjustments in scripts (e.g., map.py) are needed. Offering a dedicated subcommand or configuration option would greatly simplify this process.**
2. **Distinct Boundaries in Reach Tubes: Currently, the boundaries between different reach tubes are not clearly defined. Introducing color gradients that vary with time or height would help visualize the temporal and spatial evolution of agent states more intuitively. This is especially useful for scenarios spanning extended durations or involving multiple sensors and agents.**
3. **Realistic Environmental Noise: Environmental noise and uncertainties significantly influence system behavior. A key direction for future development is improving how these factors are modeled, thereby producing more realistic simulations and enhancing the overall fidelity of the results.**

7. Overall Conclusion

Verse is a versatile Python library designed to simplify hybrid system verification in multi-agent environments. By leveraging Python for scenario scripting and supporting various reachability analysis tools, Verse bridges the gap between complex verification algorithms and practical usability. Its modular design allows for easy scenario creation, map switching, and integration of custom verification methods, making it a valuable tool for researchers and practitioners in hybrid systems verification.

8. limitations

Restricted Agent Interactions (Single-sensor model):
Currently, Verse assumes all agents interact through a single, shared sensor. This limits the fidelity of modeling complex, heterogeneous multi-agent interactions.
Chinese example: 想象有一台无人机和多辆无人驾驶汽车；无人机应当拥有全局视野的高空摄像头，而汽车只能依赖车载摄像头及雷达。Verse 的单一传感器模型无法准确体现这种感知差异。

Lack of Support for Heterogeneous Perceptions:
Verse does not easily accommodate agents with different types or qualities of sensors,

hindering realistic modeling of heterogeneous systems.
Chinese example: 在一个仓储机器人系统中，有的机器人具备高精度激光雷达，有的只有简单的红外测距仪。Verse 无法很好模拟这种异构传感器配置。

Limited Scenario Construction and Sampling Tools:
There are no robust built-in functions for automatically generating and systematically sampling diverse scenarios, making broad exploratory testing difficult.
Chinese example: 若要研究无人机在不同高度、障碍物分布和环境条件下的避障性能，当前必须人工修改参数并单独运行验证，增加了工作量和出错概率。

Insufficient Post-processing and Tool Integration:
Verse lacks direct integration with existing white-box analysis tools and post-processing capabilities, forcing manual conversion and limiting efficient workflows.
Chinese example: 当在 Verse 中验证一个多无人机协同避障场景后，想进一步将结果导入 SpaceEx、Flow* 等工具进行高级分析，必须手动转换输出格式，增加了研究者的负担。