PARALLELIZATION AND INCREMENTAL ALGORITHMS IN THE VERSE HYBRID SYSTEM VERIFICATION LIBRARY

BY

HAOQING ZHU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2023

Urbana, Illinois

Adviser:

Professor Sayan Mitra

ABSTRACT

Hybrid systems is a popular model for modeling and verifying cyber physical systems, combining the discrete transition logic and physical dynamics of agents. However, it is difficult for most users to adopt this technology without formal methods training. Verse is a verification library which tries to address this issue and make the hybrid system technology more usable. Verse has shown some promise and in a short amount of time is currently used by several research groups. But Verse has scalability issues yet to be solved. In this thesis, we present parallelization and incremental verification algorithms in Verse. Verse computes reachsets of a system as a reachability tree, and the parallelization algorithm can compute different parts of the tree concurrently in different processors. Using the popular Ray parallelization framework, we are able to efficiently parallelize the computations without the use of locks. The incremental verification algorithm can reuse computation from previous experiments and reduce computation time for similar scenarios. We evaluate the implementation of our algorithms on a variety of scenarios, and observed that we can achieve 2 to 4 times speedup on moderately large scenarios. In one experiment with 12 agents and 133 transitions, we are able to compute the reachsets in 8 minutes 30 seconds, a 3.5x speedup over the previous 30 minutes.

ACKNOWLEDGMENTS

This project would not have been possible without the support of many people. Many thanks to my adviser, Professor Sayan Mitra, who provided much guidance throughout the years and read numerous revisions of my paper and helped polish its presentation. Also thanks to those who created and contributed to the Verse project, Yangge Li, Katherine Braught, Keyi Shen, and Daniel Zhuang, without whom this work would not have completed.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Autonomous systems are becoming more important in our daily lives. Self driving cars and autonomous delivery drones and robots bring us great convenience, but can also cause great danger if not properly tested. Traditional testing methods have a hard time of revealing potential fatal flaws of the system, while verification methods like hybrid automaton verification can provide safety or reliability guarantees for a system.

1.1 HYBRID AUTOMATON VERIFICATION

Systems like an autonomous driving vehicle is what is called a *cyber-physical system*, where computer software needs to interact with an environment with physical properties [1]. Software verification typically make use of discrete-event models like automata, which have been studied extensively. However it is difficult to apply those technologies to cyber-physical systems. In order to properly model the states of systems, one popular formalism is the hybrid automaton model [1, 2].

A hybrid automaton model includes both the states and the behavior of a system it is trying to model. The state of the system can be represented by a set of state variables. These variables are divided into two categories: *continuous variables* and *discrete modes* (or just *modes*). The continuous variables model the behavior of physical quantities in the system, such as velocity and heading of a self-driving vehicle. The discrete modes model the states in the software, such as states in a Finite State Machine.

The behavior of the system is modeled by discrete transitions and continuous trajectories. The discrete transitions may be taken from the software in the system, and can be modeled by *transition graphs*. A *transition* in the system is a discrete jump between two states of the system with different modes. Formally, a transition is a relation between two system states. The transition behavior of a hybrid automaton model is described by *guards* and *resets*. The guards provide information on when transitions should occur, and resets describe how the values of the state variables should change after the transition. The continuous trajectories typically correspond to the physical environment surrounding the system, and the most common way to model these physical interactions is through *Ordinary Differential Equations* (ODEs). These equations describe changes in one continuous variable according to the values in other parts of the system state. These concepts will be defined more formally in Chapter 4.

As the system states evolve through time, it will produce a *trajectory* according to the its behavior, which describes the values in each state variable with respect to time. For some initial state S, a state in the system is said to be *reachable* if there exists a trajectory starting from S that contains it. The set of all reachable states is then called the *reachable set* or *reachset*. Note that reachsets can start from a single initial state, or more generally an initial set of states. Hybrid automata verification tools usually don't compute the reachset continuously, but rather after some predefined small *timesteps*. The evolution of the reachset after one timestep is encapsulated in the *post* operator. It can be separated into *continuous post* for evolving the reachset using the dynamics, and *discrete post* for processing the guards and resets. These two types of post operations are usually done in lockstep as the simulation progresses.

The verification of hybrid automata involves checking that the reachset of a system does not overlap the *unsafe set* of states. For example, in the context of autonomous driving vehicles, the unsafe set can be the road curbs or a certain distance within pedestrians in the environment. Ideally, the reachsets of a system is explored over an infinite amount of time. However, this is often not possible in practice, and hybrid automaton verification tools[3, 4, 5, 6] will instead compute a *bounded reachset* of a system, where the reachsets are only computed over a bounded duration of time. This length of time will be referred to as the *time horizon*.

1.2 HYBRID VERIFICATION USABILITY CHALLENGES AND VERSE

hybrid automaton verification tools have been used to analyze linear models with thousands of continuous dimensions [7, 8, 9] and nonlinear models inspired by industrial applications [5, 8, 10, 11]. Chen and Sankaranarayanan provide a survey of the state of the art [12]. 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 [6, 7, 9, 13, 14] require the input model to be written in a tool-specific language. Tools like C2E2 [15] 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 [2, 16, 17] which require the discrete 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 our recent paper, we presented Verse [18][1], a Python library that aims to make hybrid automaton technologies more usable for multi-agent scenarios. The key features implemented are as follows:

1. Verse introduces an additional structure, called *map*, for defining the modes and the transitions of a hybrid automaton. Map contains *tracks* that can capture geometric objects (e.g. lanes or waypoints), making 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.

2. In Verse, user-defined functions can be used to create complex *agents*, and invariant requirements can be written as `assert` statements. Multiple of these *agents* can interact with each other in the same *map*. The combination of a map and multiple agents is called a *scenario*, all of which can be created using Python. Verse parses this scenario and constructs an internal representation of the hybrid automaton for simulation and analysis.

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.

1.3 PARALLEL ALGORITHMS FOR HYBRID AUTOMATON VERIFICATION

Complex scenarios can have many agents, each with many possible transitions. These agents may also interact with each other repeatedly over a long analysis time horizon. These factors can cause the number of transitions to blow up exponentially with increasing time horizon. Intuitively, each of these transition branches should be independent of each other, and there should be opportunities for parallelization. At the same time, the reachability analysis algorithm currently implemented in Verse doesn't take advantage of the multi-core nature of contemporary processors. Being able to explore these opportunities will allow Verse to be more scalable, particularly on larger scenarios.

In addition, we observed that many experiments we perform are based on the same map and agent dynamics, and just change the initial condition and decision logic of the agents.

[1]Verse is available at https://github.com/AutoVerse-ai/Verse-library.

It's useful to test out how different initial conditions can affect the behavior of the system, or to tweak the decision logic to mitigate unsafe conditions. As some parts of the scenario may have been computed in a previous experiment, it is possible to cache and reuse those computations instead of computing everything from scratch. This is a particular type of incremental verification—a more general idea that has been explored in software model checking [19, 20, 21]. Our incremental verification algorithm tries to build on previous experiment results and incrementally simulate or compute reachability of new, slightly different scenarios.

In this thesis, we will present algorithms for parallelizing simulation and reachability analysis and performing incremental verification in Verse. In our experiments, we see that the parallelization algorithm can be effective in improving the performance, especially in more complex hybrid automata models where more nondeterministic branching can occur. The incremental verification algorithm is able to reuse some computations from experiment runs and speed up analysis in some situations, while in others it provides no benefits.

For the rest of the thesis, we will first cover related work (Chapter 2). We will then introduce the various constructs in Verse for creating multi-agent scenarios (Chapter 3) and define how they translate to hybrid automaton models (Chapter 4). We will present the design and proof correctness of our parallelization algorithms we have implemented in Verse (Section 5), and lastly we present experimental data and evaluate the results (Section 6).

CHAPTER 2: RELATED WORK

This work is closely related to reachability analysis of hybrid automata, and there have been many tools developed for creating, simulating and analysing hybrid automaton models.

Table 2.1: Parallelization methods used in hybrid automaton verification tools (None, CPU, or GPU) and types of dynamics supported (linear or nonlinear; nonlinear includes both linear and more general nonlinear ODEs)

Tool name	Parallelization method	Supported dynamics	Implementation language
HyTech [22]	None	linear	C
d/dt [23]	None	linear	C
C2E2 [3]	None	nonlinear	Python
checkmate [24]	None	nonlinear	Matlab/Simulink
SpaceEx [6]	None	linear	Java
Flow* [14]	None	nonlinear	C++
DryVR [10]	None	nonlinear	Python
XSpeed [25]	CPU & GPU	linear	C++
JuliaReach [26]	CPU	nonlinear	Julia
CORA [9]	None	nonlinear	Matlab
dreach [27]	CPU	nonlinear	C++
HyLAA [7]	CPU & GPU	linear	Python
PIRK [28]	CPU & GPU	nonlinear	C++
Verse [18]	None	nonlinear	Python

Many hybrid automaton verification tools have incorporated parallelization as an optimization technique. Table 2.1 briefly lists some hybrid automaton verification tools and the parallelization optimizations they support. Tools like JuliaReach [26] and dreach [27] utilizes CPU for parallelizing exploration of discrete mode transitions. Other tools like HyLAA [7] and XSpeed [25] also support using GPU for accelerating the computation of continuous posts. PIRK uses the pFaces [29] runtime to parallelize using CPUs and GPUs.

In particular, we have adapted the *AsyncBFS* algorithm implemented in XSpeed [25] to Verse. However, there are some major differences between our implementations. First, thanks to Ray, it is not necessary to explicitly handle locks in our implementation. Second, due to Python's limitations and Ray's design, the parallelization happens across persistent processes. This means that the cost of launching a parallelized computation becomes serializing, communicating and deserializing the arguments and return values, instead of spawning new threads. Lastly, we have chosen a different granularity of parallelization in Verse that is coarser than the algorithm presented in XSpeed.

CHAPTER 3: SCENARIOS IN VERSE

In Verse, the main concept that specifies a hybrid system model is a *scenario*. It contains 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 will describe these components below and in Section 4 we will discuss how they formally define a hybrid system. We will use a running example through this chapter to help illustrate these concepts.

Consider vehicles driving on a 4-way intersection, where vehicles can enter and exit the intersection through each of the 4 cardinal directions (the lane boundaries of the map is shown in 3.1). Each vehicle can have a simple collision avoidance logic, allowing it to switch lanes when it sees another vehicle closely ahead on the same lane. It can choose to switch to either the left or right lanes, provided that there is no other vehicle closely ahead in that lane.

3.1 TRACKS, MODES, AND MAPS

A *workspace* W is a Euclidean space in which the agents reside.This corresponds to a subset of $\mathbb{R}^2$ in the intersection example. 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 the intersection example, some of the tracks are along the center of the lanes, while others corresponds to switches between the lanes.

We assume that an agent's decision logic does not depend on exactly which of the infinitely many tracks it can follow, but instead, it depends only on which type of track it follows, or the *track mode*. 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. In the intersection example, a vehicle can come from the south side of the intersection, stay on the leftmost lane at all times, and exit through the west side. We can refer to the lane it is tracking as `SW-0`. A track mode corresponding to lane switches could be `SW-0-1` for denoting a switch from the leftmost lane to the right one, for the lanes going from south to west.

Finally, a Verse agent's decision logic can change its internal mode or *tactical mode*. For a simple car agent, its tactical mode may consist of `Normal`, `SwitchLeft` and `SwitchRight`. These specify the current state the agent is in, and instructs the underlying controller to steer

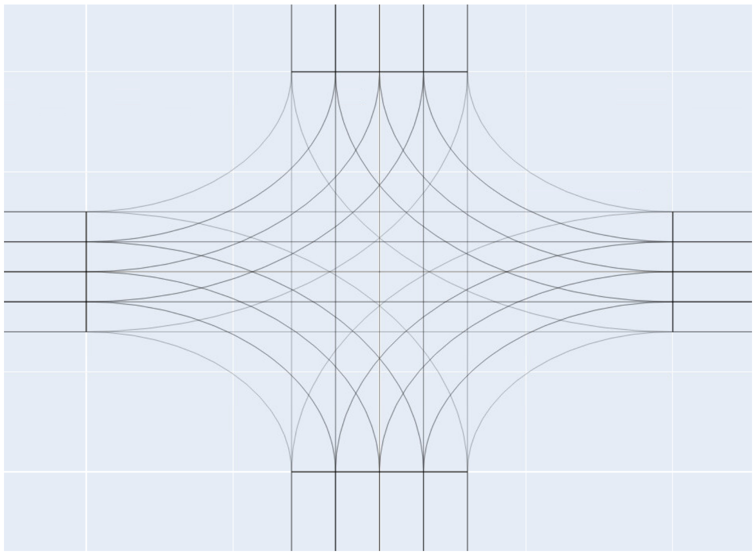

Figure 3.1: A 4-way intersection map with 2 lanes in each direction, showing the lane boundaries. Each lane extends very far outside of the picture.

the vehicle in a certain way. When an agent changes its tactical mode, it may also update the track it's following and this is encoded in another function: $h_{\mathcal{M}} : L_{\mathcal{M}} \times P \times P \to L_{\mathcal{M}}$ 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 vehicle changes from `Normal` to `SwitchRight` while it is on `SW-0`, this map function informs that $h_{\mathcal{M}}(\texttt{SW-0}, \texttt{Normal}, \texttt{SwitchRight}) = \texttt{SW-0-1}$ the vehicle should follow a track with mode `SW-1`.

3.2 AGENTS

A Verse *agent* is defined by discrete 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 intersection example, the continuous state variables can be x, y, θ, v for position, heading, and speed for the vehicle. The discrete modes are $\langle \texttt{Normal}, \texttt{SW-1} \rangle$, $\langle \texttt{SwitchRight}, \texttt{SW-0} \rangle$ etc.

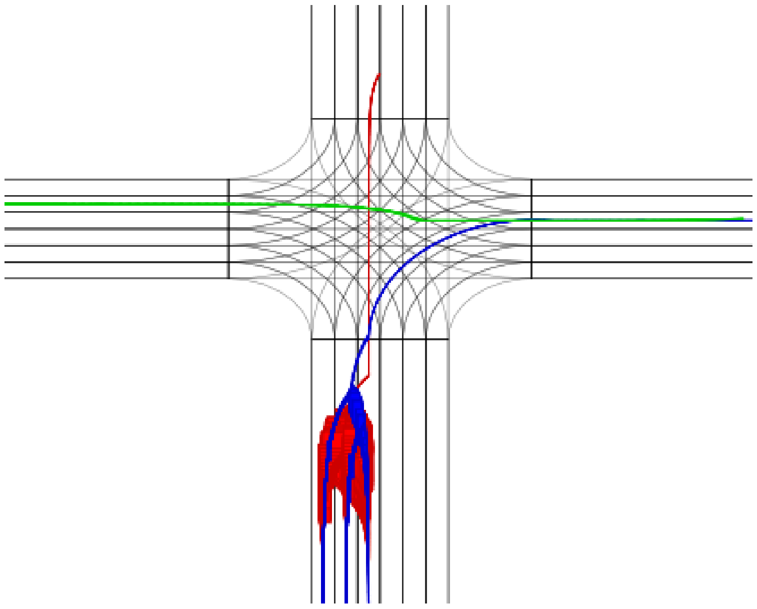

Figure 3.2: Computed reachsets with 3 agents (represented with 3 different colors) on the intersection map with 3 lanes in each direction. The red car goes from north to south, the green car goes from east to west, and the blue car goes from east to south.

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. 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. 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.

Figure 3.3 shows an example decision logic for a vehicle. For example, lines 60 to 63 can define a transition transitions $\langle \texttt{SwitchRight}, \texttt{SW-0-1} \rangle$ to $\langle \texttt{Normal}, \texttt{SW-1} \rangle$. The change of track mode is given by the h function. The guard for this transition comes from the `if` condition at Line 60. For example, $G(\langle \texttt{SwitchRight}, \texttt{SW-0-1} \rangle, \langle \texttt{Normal}, \texttt{SW-1} \rangle) = \{x \mid |\texttt{SW-1}.y - x.y| < 1 \forall x \in X\}$. Here continuous states remain unchanged after transition.

```python
def decisionLogic(ego: State, others: List[State], track_map):
    output = copy.deepcopy(ego)
    if ego.sw_time >= 1:
        car_front = car_ahead(ego, others, ego.track_mode, track_map, 7, 0)
        if ego.agent_mode == AgentMode.Accel and car_front:
            left_lane = track_map.h(ego.track_mode, ego.agent_mode,
            ↪  AgentMode.SwitchLeft)
            right_lane = track_map.h(ego.track_mode, ego.agent_mode,
            ↪  AgentMode.SwitchRight)
            if left_lane != None and not car_ahead(ego, others, left_lane,
            ↪  track_map, 8, -3):
                output.agent_mode = AgentMode.SwitchLeft
                output.track_mode = left_lane
                output.sw_time = 0
            if right_lane != None and not car_ahead(ego, others, right_lane,
            ↪  track_map, 8, -3):
                output.agent_mode = AgentMode.SwitchRight
                output.track_mode = right_lane
                output.sw_time = 0
        if ego.agent_mode == AgentMode.Brake and not car_front:
            output.agent_mode = AgentMode.Accel
            output.sw_time = 0
        lat_dist = track_map.get_lateral_distance(ego.track_mode, [ego.x, ego.y])
        lat = 2
        if ego.agent_mode == AgentMode.SwitchLeft and lat_dist >= lat:
            output.agent_mode = AgentMode.Accel
            output.track_mode = track_map.h(ego.track_mode, ego.agent_mode,
            ↪  AgentMode.Accel)
            output.sw_time = 0
        if ego.agent_mode == AgentMode.SwitchRight and lat_dist <= -lat:
            output.agent_mode = AgentMode.Accel
            output.track_mode = track_map.h(ego.track_mode, ego.agent_mode,
            ↪  AgentMode.Accel)
            output.sw_time = 0
    return output
```

Figure 3.3: Decision Logic Code Snippet from `intersection_car.py`.

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 thesis, we use F as a black-box function (see footnote 2).

3.3 SENSORS AND SCENARIOS

For simplifying exposition, in this thesis 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.

```python
    ctlr_src = "demo/vehicle/controller/intersection_car.py"
    scenario = Scenario()
    dirs = "WSEN"
    map = Intersection(lanes=LANES)
    scenario.set_map(map)
    for i in range(CAR_NUM):
        car = CarAgentDebounced(f"car{i}", file_name=ctlr_src,
        ↪  speed=rand(*CAR_SPEED_RANGE), accel=rand(*CAR_ACCEL_RANGE))
        scenario.add_agent(car)
        dir = random.randint(0, 3)
        src = dirs[dir]
        dst_dirs = list(dirs)
        dst_dirs.remove(src)
        dst = dst_dirs[random.randint(0, 2)]
        lane = random.randint(0, map.lanes - 1)
        start, off = map.size + rand(0, map.length * 0.3), rand(0, map.width)
        ↪  + map.width * lane
        pos = { "N": (-off, start), "S": (off, -start), "W": (-start, -off),
        ↪  "E": (start, off) }[src]
        init = [*pos, *(wrap_to_pi(dir * math.pi / 2 +
        ↪  rand(*CAR_THETA_RANGE)), rand(*CAR_SPEED_RANGE)), 0]
        modes = (AgentMode.Accel, f"{src}{dst}_{lane}")
        scenario.set_init_single(car.id, (init,), modes)
    traces = scenario.verify(60, 0.1)
```

Figure 3.4: Code snippet for creating the intersection scenario.

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 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., $Y_i = Y_j \forall i, j \in [0, k)$, but they can have different decision logics and different continuous dynamics.

An example is shown in 3.4 for how the intersection scenario is created in Verse. In this scenario, no sensor is explicitly set, thus every agent can observe all of the states for every other agent.

CHAPTER 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 [10]. As usual, the automaton has discrete and continuous states and discrete transitions defined by guards and resets. The only uncommon aspect in [10] is that the continuous flows may be defined by a black-box simulator functions, instead of white-box analytical models[2].

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. $\mathbf{TL}$ 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 $\forall t \in \mathbb{R}^{\geq 0}, \xi_i(t) = F_i(\mathbf{x}_i^0, \mathbf{d}_i)(t)$.

Proposition 4.1. $H(SC)$ is a valid hybrid system for a scenario with k agents $SC = \langle \mathcal{M}, \{\mathcal{A}_1, ..., \mathcal{A}_k\}, \mathcal{S}, P\rangle$ if it satisfies the following:

1. map and the agents are compatible

2. all agents have identical sets of states and modes, $Y_i = Y_j \forall i, j \in [0, k)$

[2]This design decision for Verse is relatively independent. For reachability analysis, Verse currently uses black-box statistical approaches implemented in DryVR [10] and NeuReach [30]. If the simulator is available as a white-box model, such as differential equations, then Verse could use model-based reachability analysis.

3. agent states match the input type of $\mathcal{S}$

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. $\forall i \in \{1, ..., m-1\}, \xi^i.lstate \in \mathbf{G}(\mathbf{d}^i, \mathbf{d}^{i+1})$ and $\xi^{i+1}.fstate = \mathbf{R}(\mathbf{d}^i, \mathbf{d}^{i+1})(\xi^i.lstate)$;

3. $\forall i \in \{1, ..., m-1\}, \xi^i.ltime = \delta$ for $i \neq m-1$ and $\xi^i.ltime \leq \delta$ for $i = m-1$.

We define $\alpha(\mathbf{X}^0, \mathbf{d}^0)$ to be an execution β where $\beta.fstate = \mathbf{X}^0$ and $\beta.fmode = \mathbf{d}^0$.

We define the 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 $\alpha.lmode = \mathbf{d}^{m-1}$.

4.1 BOUNDED REACH SETS

Consider a scenario SC with k agents and the corresponding hybrid automaton $H(SC)$. We define $Stride(\mathbf{x}^0, \mathbf{d}^0, \delta)$ to be $\bigcup_{t \in [0,\delta)} \xi^i(t)$ for any initial state $\mathbf{x}^0 \in \mathbf{X}$ and mode $\mathbf{d}^0 \in \mathbf{D}$, for some valid labeled trajectory $\langle \xi^0, \mathbf{d}^0 \rangle$. For any set of initial states $\mathbf{X}^0 \subseteq \mathbf{X}$, we define $Stride(\mathbf{X}^0, \mathbf{d}^0, \delta) = \bigcup_{\mathbf{x} \in \mathbf{X}^0} Stride(\mathbf{x}, \mathbf{d}^0, \delta)$.

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 $\mathbf{x}, \mathbf{x}' \in \mathbf{X}$,

$$post_{\mathbf{d},\mathbf{d}'}(\mathbf{x}) = \mathbf{x}' \iff \mathbf{x} \in \mathbf{G}(\mathbf{d}, \mathbf{d}') \tag{4.1}$$

$$post_{\mathbf{d},\delta}(\mathbf{x}) = \mathbf{x}' \iff \forall i \in 1, ..., k, \mathbf{x}'_i = F_i(\mathbf{x}_i, \mathbf{d}_i, \delta) \tag{4.2}$$

$$\mathbf{x}' = \mathbf{R}(\mathbf{d}, \mathbf{d}')(\mathbf{x}) \tag{4.3}$$

These operators are also lifted to sets of states in the usual way.

Note that for some mode $\mathbf{d}^0 \in \mathbf{D}$, initial set of states $\mathbf{X}^0 \subseteq \mathbf{X}$ and timestep δ, $post_{\mathbf{d},\delta}(\mathbf{X}^0)$ gives the frontier after time δ, while $Stride(\mathbf{X}^0, \mathbf{d}^0, \delta)$ gives the set of states reachable throughout the δ time.

We then denote the bounded reachable states of $H(SC)$ by $Reach(\mathbf{X}^0, \langle \mathbf{d}^0, \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle, \delta, T_{\max})$, where:

1. $\mathbf{X}^0 \subseteq \mathbf{X}$ and $\mathbf{d}^0 \in \mathbf{D}$ are the initial states of the hybrid automaton;

2. $T_{\max}$ is the time horizon such that $m \leq \frac{T_{\max}}{\delta}$;

3. $\langle \mathbf{d}^0, \mathbf{d}^1, ...\mathbf{d}^{m-1} \rangle$ is a path of discrete modes that the hybrid automaton follows.

$\mathit{Reach}(\mathbf{X}^0, \langle \mathbf{d}^0, \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle, \delta, T_{\max})$ can be defined recursively as:

Definition 4.1.

$$\mathit{Reach}(\mathbf{X}^0, \langle\rangle, \delta, T_{\max}) = \varnothing \tag{4.4}$$

$$\mathit{Reach}(\mathbf{X}^0, \langle \mathbf{d}^0, \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle, \delta, T_{\max}) = \varnothing \text{ if } T_{\max} < \delta \tag{4.5}$$

$$\mathit{Reach}(\mathbf{X}^0, \langle \mathbf{d}^0, \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle, \delta, T_{\max}) = \mathit{Reach}(\mathbf{X}^1, \langle \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle, \delta, T_{\max} - \delta) \tag{4.6}$$

$$\cup\, \mathit{Stride}(\mathbf{X}^0, \mathbf{d}^0, \delta) \text{ if } T_{\max} \geq \delta \tag{4.7}$$

where $\mathbf{X}^1 = \mathit{post}_{\mathbf{d}^0, \mathbf{d}^1}(\mathit{post}_{\mathbf{d}^0, \delta}(\mathbf{X}^0))$

Note that due to the nature of this definition there may be some subset $\mathbf{X}^{0\prime} \subseteq \mathbf{X}^0$ which doesn't visit the whole path but only a prefix of it. Those states are also contained within the same reachable set.

Finally, we define $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}) = \bigcup_{\mathit{path}} \mathit{Reach}(\mathbf{X}^0, \mathit{path}, \delta, T_{\max})$ for all valid paths of modes $\mathit{path} \in \mathbf{D}^n$ with initial states starting from $\mathbf{X}^0$, and $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}).\mathit{next} = \{\langle \mathbf{X}', \mathbf{d}' \rangle \mid \text{the path } \langle \mathit{path}, \mathbf{d}' \rangle \text{ is a valid path for all valid paths } \mathit{path}\}$.

Next, we discuss Verse functions for verification via reachability.

CHAPTER 5: BUILDING VERIFICATION ALGORITHMS IN VERSE

In this section we will describe the parallel algorithms with have implemented in Verse. But instead of presenting the algorithm itself directly, we will first introduce the basic serialized algorithm, and then gradually adding in different optimizations before getting to the final version of the algorithm.

5.1 REACHABILITY ANALYSIS

Recall that for a scenario SC and its hybrid system model $H(SC)$, $\mathbf{X}$ and $\mathbf{D}$ are respectively the continuous state space and discrete mode space of $H(SC)$. Verse provides `postCont`$(\mathbf{d}^0, \delta, \mathbf{X}^0)$ to compute both the reachable set $\mathit{Stride}(\mathbf{X}^0, \mathbf{d}^0, \delta)$ and the frontier $\mathit{post}_{\mathbf{d}^0,\delta}(\mathbf{X}^0)$, and `postDisc`$(\mathbf{d}^0, \mathbf{d}^1, \mathbf{X}^0)$ to compute $\mathit{post}_{\mathbf{d}^0,\mathbf{d}^1}(\mathbf{X}^0)$ for some set of states $\mathbf{X}^0 \subseteq \mathbf{X}$, modes $\mathbf{d}^0, \mathbf{d}^1 \in \mathbf{D}$, and timestep δ. Instead of computing the exact post, `postCont` and `postDisc` compute over-approximations using improved implementations of the algorithms in [10].

Verse's `verify` function implements a reachability analysis algorithm using these post operators (Algorithm 5.1). This algorithm constructs an execution tree $Tree = \langle V, E\rangle$ up to depth $T_{\max}$ in breadth first order. Each vertex $N = \langle \mathbf{X}, \mathbf{d}, t, \mathit{stride}, \mathit{children}\rangle \in V$ is a tuple of a set of states, a mode, the start time, the stride and children of the current node. In the pseudocode, we will use the dot field access notation to refer to fields of a node. For example for a node N, $N.\mathit{stride}$ and $N.\mathbf{X}$ refers to the stride and the set of initial states in N. The root is $\langle \mathbf{X}^0, \mathbf{d}^0, 0, \mathit{stride}, \mathit{children}\rangle$ given initial set of states $\mathbf{X}^0$ and mode $\mathbf{d}^0$. There is an edge from $\langle \mathbf{X}, \mathbf{d}, t, \mathit{stride}, \mathit{children}\rangle$ to $\langle \mathbf{X}', \mathbf{d}', t', \mathit{stride}', \mathit{children}'\rangle$ iff $\mathbf{X}' = \mathit{post}_{\mathbf{d},\delta}(\mathit{post}_{\mathbf{d},\mathbf{d}'}(\mathbf{X}))$ and $t' = t + \delta$.

5.1.1 Correctness proof of `verify` algorithm

In order to show the correctness of the `verify` algorithm, we will first proof the properties of the `verify_step` function in Proposition 5.1, and show the correctness of `verify` in Proposition 5.2 based on that.

Following the previous definitions, given a scenario SC, we have the hybrid automaton $H(SC)$. Let $\mathbf{X}$ and $\mathbf{D}$ be the continuous and discrete state space of $H(SC)$.

Proposition 5.1. For any set of states $\mathbf{X}^0 \subseteq \mathbf{X}$, mode $\mathbf{d}^0 \in \mathbf{D}$ and time t, the node

Algorithm 5.1

```
function VERIFY_STEP($N, \delta$)   where $N.stride = \varnothing,\ N.children = \varnothing$
    $\langle N.stride, \mathbf{X}' \rangle \leftarrow$ postCont($N.\mathbf{X}, N.\mathbf{d}, \delta$)
    for $\mathbf{d}' \in \mathbf{D}$ s.t. $\mathbf{G}(N.\mathbf{d}, \mathbf{d}') \cap N.stride \neq \varnothing$ do
        $N.children \leftarrow N.children \cup \langle$postDisc($\mathbf{X}', N.\mathbf{d}, \mathbf{d}'$)$, \mathbf{d}', N.t + \delta, \varnothing, \varnothing \rangle$
function VERIFY($H, \mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}$)
    $queue \leftarrow [\langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle]$
    $reachset \leftarrow \varnothing$
    while $queue \neq \varnothing \wedge \exists N \in queue$ s.t. $N.t < T_{\max}$ do
        $N \leftarrow queue$.dequeue()
        verify_step($N, \delta$)
        for $N' \in N.children$ do
            $queue$.add($N'$)
            $reachset \leftarrow reachset \cup N.stride$
    return $reachset$
```

$N = \langle \mathbf{X}^0, \mathbf{d}^0, t, stride, children \rangle$, after `verify_step`$(N, \delta)$ executes,

$$Stride(\mathbf{X}^0, \mathbf{d}^0, \delta) \subseteq N.stride \tag{5.1}$$

and $\forall C \in N.children$

$$\mathbf{G}(N.\mathbf{d}, C.\mathbf{d}) \cap N.stride \neq \varnothing \tag{5.2}$$

$$post_{N.\mathbf{d},C.\mathbf{d}}(post_{N.\mathbf{d},\delta}(N.\mathbf{X})) \subseteq C.\mathbf{X} \tag{5.3}$$

Proof. For (1), from line 2, $\langle N.stride, \mathbf{X}' \rangle =$ `postCont`$(N.\mathbf{X}, N.\mathbf{d}, \delta)$. As mentioned in the start of section 5.1, $N.stride \supseteq Stride(N.\mathbf{X}, N.\mathbf{d}, \delta)$ and $\mathbf{X}' \supseteq post_{N.\mathbf{d},\delta}(N.\mathbf{X})$.

For (2), for every children $C \in N.children$:

From the loop condition at line 3:

$$\mathbf{G}(\mathbf{d}^0, C.\mathbf{d}) \cap N.stride \neq \varnothing \tag{5.4}$$

$$C.\mathbf{X} = \texttt{postDisc}(\mathbf{X}', N.\mathbf{d}, C.\mathbf{d}) \tag{5.5}$$

$$\supseteq post_{N.\mathbf{d},C.\mathbf{d}}(\mathbf{X}') \tag{5.6}$$

$$\supseteq post_{N.\mathbf{d},C.\mathbf{d}}(post_{N.\mathbf{d},\delta}(N.\mathbf{X})) \tag{5.7}$$

QED.

Proposition 5.2. Given initial states $\mathbf{X}^0 \subseteq \mathbf{X}$ and $\mathbf{d}^0 \in \mathbf{D}$, time horizon $T_{\max}$,

$$\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}) \subseteq \mathtt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}) \tag{5.8}$$

Proof. We will first proof the correctness based on paths $\langle \mathbf{d}^0, \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle$

$$\mathit{Reach}(\mathbf{X}^k, \langle \mathbf{d}^k, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) \subseteq \mathtt{verify}(\mathbf{X}^k, \langle \mathbf{d}^k, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) \tag{5.9}$$

We will proof by induction on the length of the tail of the path $\langle \mathbf{d}^0, \mathbf{d}^1, ..., \mathbf{d}^{m-1} \rangle$.

The tree $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max})$ is with height m is just the same reachable set with time horizon equal to $m \times \delta$, i.e. $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, m \times \delta)$, resp. `verify`.

Base case:

$$\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta) \subseteq \mathtt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta) \tag{5.10}$$

and that the children of both trees are equal.

If let $\langle \mathit{stride}, \mathit{frontier} \rangle = \mathtt{postCont}(\mathbf{X}^0, \mathbf{d}^0, \delta)$, and $N = \langle \mathbf{X}^0, \mathbf{d}^0, \varnothing, \varnothing \rangle$, then

$$\begin{aligned}
\mathtt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta) &= \mathtt{verify\_step}(N, \delta) && (5.11)\\
&= \mathit{stride} && (5.12)\\
&\supseteq \mathit{Stride}(\mathbf{X}^0, \mathbf{d}^0, \delta) && (5.13)\\
&= \mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta) && (5.14)
\end{aligned}$$

In addition, according to proposition 5.1 point 2, the children of $\mathtt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta)$ which is $N.\mathit{children}$ after $\mathtt{verify\_step}(N, \delta)$ is the same as that of $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, \delta)$.

Induction hypothesis: Given $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta) \subseteq \mathtt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta)$ where $k \in [1, m)$, show $\mathit{Reach}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta) \subseteq \mathtt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta)$

Induction step:

$$\begin{aligned}
\mathtt{verify}(\mathbf{X}^k, \langle \mathbf{d}^k, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) &= \mathtt{reachStep}(\mathbf{X}^k, \mathbf{d}^k) && (5.15)\\
&\quad \cup \mathtt{verify}(\mathbf{X}^{k+1}, \langle \mathbf{d}^{k+1}, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) && (5.16)
\end{aligned}$$

$$\mathit{Reach}(\mathbf{X}^k, \langle \mathbf{d}^k, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) = \mathit{Stride}(\mathbf{X}^k, \mathbf{d}^k, \delta) \cup \mathit{Reach}(\mathbf{X}^{k+1}, \langle \mathbf{d}^{k+1}, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) \tag{5.17}$$

From the induction hypothesis:

$$\mathit{Reach}(\mathbf{X}^{k+1}, \langle \mathbf{d}^{k+1}, ..., \mathbf{d}^{m-1}, T_{\max}) \subseteq \mathtt{verify}(\mathbf{X}^{k+1}, \langle \mathbf{d}^{k+1}, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) \tag{5.18}$$

and from 5.1 with substitutions:

$$Stride(\mathbf{X}^k, \mathbf{d}^k, \delta) \subseteq \texttt{reachStep}(\mathbf{X}^k, \mathbf{d}^k) \tag{5.19}$$

Combining both we get:

$$Reach(\mathbf{X}^k, \langle \mathbf{d}^k, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) \subseteq \texttt{verify}(\mathbf{X}^k, \langle \mathbf{d}^k, ..., \mathbf{d}^{m-1} \rangle, T_{\max}) \tag{5.20}$$

QED.

5.2 PARALLELIZATION

In this section we show how we parallelize the verification algorithm shown above using Ray. Ray is a framework for parallelizing workloads in Python. It uses remote functions as an abstraction for performing parallelization. These are functions that can be called on one process, but will be executed in another process. These processes can be configured to be executed either on other cores of the same machine, or cores on other network-connected machines. Throughout this thesis we will assume that the remote functions execute on other cores within the same machine.

For some normal Python function with arguments `f(args)`, the function `f` can be turned into a remote function by decorating the definition of the function with the `ray.remote` decorator. Such remote functions can be called via `f.remote(args)`. In order to simplify the pseudocode, we will simply use a **remote** keyword before a function call to denote that it is a remote function call.

In Ray, two processes communicate through a distributed database. For a remote function, both the arguments and the return value will be stored in the database. From the caller's side, When a remote function is called, the arguments to the function will be sent automatically to the database, and a reference to the return value of the function is returned immediately. For the remote process, the arguments are first fetched from the database, then the function will run, and lastly the return value is sent back to the database. The main process can poll and fetch the value back by using the `ray.wait()` function. `ray.wait(refs)` blocks until one of the references in `refs` is available, fetches and returns that value along with the rest of the references.

The basic parallelized algorithm is shown in Algorithm 5.2. The `verify_parallel` algorithm uses a queue to explore the tree just like `verify`, however there are 2 branches in the loop. One of them pops nodes from the queue and calls the `verify_step` on the node as remote functions, while the other waits for the results to come back, processes the result, and adds new nodes to the queue. Because of the branching, the algorithm prioritizes sending out computations, which means there can be multiple remote computations inflight at the same time and the node computation can happen in a different order. The `verify_parallel` uses a slightly modified version of `verify_step`, as it's needed to recover the time and node information.

Algorithm 5.2

```
1: function VERIFY_PARALLEL_STEP($N, \delta$) where $N.stride = \varnothing,\ N.children = \varnothing$
2:     $\langle N.stride, \mathbf{X}' \rangle \leftarrow$ postCont($N.\mathbf{X}, N.\mathbf{d}, \delta$)
3:     for $\mathbf{d}' \in \mathbf{D}$ s.t. $\mathbf{G}(N.\mathbf{d}, \mathbf{d}') \cap N.stride \neq \varnothing$ do
4:         $N.children \leftarrow N.children \cup \langle$postDisc$(\mathbf{X}', N.\mathbf{d}, \mathbf{d}'), \mathbf{d}', N.t + \delta, \varnothing, \varnothing\rangle$
5:     return $N$
6: function VERIFY_PARALLEL($H, \mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}$)
7:     $root \leftarrow \langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle$
8:     $queue \leftarrow [root]$
9:     $refs \leftarrow \varnothing$
10:    $reachset \leftarrow \varnothing$
11:    while $(queue \neq \varnothing \vee refs \neq \varnothing) \wedge \exists N \in queue$ s.t. $N.t < T_{\max}$ do
12:        if $queue \neq \varnothing$ then
13:            $N \leftarrow queue$.dequeue()
14:            $refs$.add(remote verify_parallel_step ($N, \delta$))
15:        else                                        ▷ wait only when $queue$ is empty
16:            $\langle N, refs \rangle \leftarrow$ ray.wait($refs$)
17:            $reachset \leftarrow reachset \cup N.stride$
18:            for $N' \in N.children$ do
19:                $queue$.add($N'$)
20:    return $reachset$
```

5.2.1 Correctness proof of `verify_parallel` algorithm

For the `verify_parallel` algorithm to be sound, the reachset it computes needs to be an overapproximation over the result of `verify`. However, due to how the `verify_parallel` algorithm is implemented, this condition can be made stronger:

Proposition 5.3. For any set of states $\mathbf{X}^0 \subseteq \mathbf{X}$ and mode $\mathbf{d}^0 \in \mathbf{D}$,

$$\texttt{verify\_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}) = \texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max})$$

Proof. To prove the equality, we can show that the set of calls to `verify_step` in `verify` and `verify_parallel` are the same. In `verify`, `verify_step` is called at line 10; in `verify_parallel`, `verify_step` is called at line 14 as a remote function call. We assume that remote calls in Ray will always return, and that given the same arguments, remote function calls to `verify_step` will return the same values as non-remote calls. We can then compare the tree generated by both `verify` and `verify_parallel` and proof by induction on the height of the tree currently computed. Note that due to the nondeterministic ordering of node traversal, the `verify_parallel` can begin computing nodes that have $k+1$ depth before finishing nodes at depth k.

Base case:

$$\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta) = \texttt{verify\_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta) \tag{5.21}$$

and the children of both trees are the same.

Proof:

Let $N = \langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle$. After `verify_step`$(N, \delta)$ is called,

$$\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta) = N.\mathit{stride} \tag{5.22}$$

let $N' =$ `verify_parallel_step`(N, δ), then:

$$\texttt{verify\_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, 1 \times \delta) = N'.\mathit{stride} \tag{5.23}$$

Under our previous assumptions, $N = N'$. The children for each tree are simply $N.\mathit{children}$ and $N'.\mathit{children}$, and they are equal.

Induction step:

Given `verify`$(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta) =$ `verify_parallel`$(\mathbf{X}^0, \mathbf{d}^0, \delta, k \times \delta)$ where $k \in [1, m)$, show

$$\texttt{verify}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta) = \texttt{verify\_parallel}(\mathbf{X}^0, \mathbf{d}^0, \delta, (k+1) \times \delta) \tag{5.24}$$

Proof:

since the children of all nodes at depth k for `verify` and `verify_parallel` are the same, they must generate the same set of nodes at depth $k+1$.

QED.

Note that in practice, computing `verify_parallel_step` is cheap. Calling small remote functions like this will incur a lot of overhead due to the cost of communication and serialization/deserialization of data. When implementing the `verify_parallel` algorithm, we have chosen to batch together these computations, so that each remote function call computes as many timesteps as possible until a discrete mode transition is hit.

5.3 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 different[3]. SC_1 and SC_2 have the same sensors, maps, and agent flow functions. Let $Tree_1 = \langle V_1, E_1 \rangle$ and $Tree_2 = \langle V_2, E_2 \rangle$ 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 $\langle \mathbf{X}_1, \mathbf{d}_1 \rangle$ of $Tree_1$ with a possible mode involves a guard check, a computation of $post_{\mathbf{d},\mathbf{d}'}$, and $post_{\mathbf{d},\delta}$. The `verify_incremental` algorithm avoids performing these computations while constructing $Tree_2$ by reusing those computations from $Tree_1$, if possible. To this end, `verify_incremental` uses a cache that stores the result of a batch of `verify_step`. This is the same as that in Section 5.2, which simply batches together all the adjacent `verify_step` that have the same discrete modes. We'll call this batch operation `verify_batch`, and the corresponding cache C. `verify_batch` takes a set of states, a discrete mode, the timestep and time horizon, and returns a pair consisting of a set of reachable states and a set of all children nodes generated for that set of reachable states. Formally, here are the properties of `verify_batch`:

Proposition 5.4. For any set of states $\mathbf{X}^0 \subseteq \mathbf{X}$, mode $\mathbf{d}^0 \in \mathbf{D}$, time step δ and time horizon

[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).

$T_{\max}$, if let $\langle \textit{reachset}, \textit{branches} \rangle = \texttt{verify\_batch}(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max})$, then:

$$\textit{branches} = \bigcup_{i \in [0,m)} \{N' \mid N' \in N^i.\textit{children} \text{ s.t. } N'.\mathbf{d} \neq N^i.\mathbf{d}\} \tag{5.25}$$

$$\textit{reachset} = \bigcup_{i \in [0,m)} N^i.\textit{reachset} \tag{5.26}$$

$$\forall N \in N^{m-1}.\textit{children}, N.\mathbf{d} \neq N^{m-1}.\mathbf{d} \tag{5.27}$$

$$m \times \delta \leq T_{\max} \tag{5.28}$$

where m is the number of nodes in the batch, which we will refer to as the *batch size*, and

$$N^i \in N^{i-1}.\textit{children} \text{ after } \texttt{verify\_step}(N^{i-1}, \delta) \tag{5.29}$$

$$N^i.\mathbf{d} = N^{i-1}.\mathbf{d} \tag{5.30}$$

$$N^0 = \langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle \tag{5.31}$$

For `verify_batch`$(\mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max})$, the cache C will be indexed by $\langle \mathbf{X}^0, \mathbf{d}^0 \rangle$, and the value will be the same as that of `verify_batch`. Unlike normal caches, a cache hit can happen for C when the incoming key $\langle \mathbf{X}', \mathbf{d}' \rangle$ satisfies $\mathbf{X}' \subseteq \mathbf{X}^0 \wedge \mathbf{d}' = \mathbf{d}^0$.

The incremental verification algorithm is presented in Algorithm 5.3. `verify_incremental` checks C before every *post* computation to retrieve and reuse computations when possible. The caches can save information from any number of previous executions, so `verify_incremental` can be even more efficient than `verify_parallel` when running many consecutive verification runs. Note that in the pseudocode, in addition to what is described above, `verify_batch` will also take the current time as an argument, and return the initial conditions along with the end time. This is needed so that the contents of the cache can be correctly updated.

5.3.1 Correctness proof of `verify_incremental` algorithm

The correctness property of `verify_incremental` is the same as that of `verify_parallel` in Section 5.2, i.e. the reachset computed by `verify_incremental` for SC_2, when given a cache populated with data from SC_1, is an overapproximation of the reachset computed by `verify`. More formally:

Proposition 5.5. Given scenarios SC_1 and SC_2 with the same sensors, map and agent flow functions, for some empty cache $C = \varnothing$, any initial conditions $\mathbf{X}_1^0, \mathbf{X}_2^0 \subseteq \mathbf{X}$, $\mathbf{d}_1^0, \mathbf{d}_2^0 \in \mathbf{D}$, timestep δ and time horizon $T_{\max}$, after `verify_incremental`$(H_1, \mathbf{X}_1^0, \mathbf{d}_1^0, \delta, T_{\max}, C)$ is

Algorithm 5.3

```
function VERIFY_INCREMENTAL($H, \mathbf{X}^0, \mathbf{d}^0, \delta, T_{\max}, C$)
    $queue \leftarrow [\langle \mathbf{X}^0, \mathbf{d}^0, 0, \varnothing, \varnothing \rangle]$
    $refs \leftarrow []$
    $reachset \leftarrow \varnothing$
    while $(queue \neq \varnothing \vee refs \neq \varnothing) \wedge t < T_{\max}$ do
        if $queue \neq \varnothing$ then
            $N \leftarrow queue$.dequeue()
            if $C(N.\mathbf{X}, N.\mathbf{d}) \neq \varnothing$ then                ▷ queries the cache
                $\langle subreachset, branches \rangle \leftarrow C(N.\mathbf{X}, N.\mathbf{d})$
                $reachset \leftarrow reachset \cup subreachset$
                for $N' \in branches$ do
                    $queue$.add($N'$)
            else
                $refs$.add(remote verify_batch $(\mathbf{X}, \mathbf{d}, \delta, T_{\max}, t)$)
        else                                         ▷ wait only when queue is empty
            $\langle \langle subreachset, branches, N \rangle, refs \rangle \leftarrow$ ray.wait($refs$)
            $C(\mathbf{X}, \mathbf{d}) \leftarrow \langle subreachset, branches \rangle$      ▷ update the cache with results
            $reachset \leftarrow reachset \cup subreachset$
            for $N' \in branches$ do
                $queue$.add($N'$)
    return $reachset$
```

executed,

$$\texttt{verify}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}) \subseteq \texttt{verify\_incremental}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}, C)$$

Proof. For any initial conditions $\mathbf{X}_1^0, \mathbf{X}_2^0 \subseteq \mathbf{X}$, $\mathbf{d}_1^0, \mathbf{d}_2^0 \in \mathbf{D}$, let:

$$\langle reachset_1, branches_1 \rangle = \texttt{verify\_batch}(\mathbf{X}_1^0, \mathbf{d}_1^0, \delta, T_{\max}, t) \tag{5.32}$$

$$\langle reachset_2, branches_2 \rangle = \texttt{verify\_batch}(\mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}, t) \tag{5.33}$$

Given that `verify_batch` simply batches together `postCont` and `postDisc` operations,

$$\mathbf{X}_1^0 \subseteq \mathbf{X}_2^0 \implies reachset_1 \subseteq reachset_2 \tag{5.34}$$

As the cache C just stores the result of `verify_batch`

$$\mathbf{X}_1^0 \subseteq \mathbf{X}_2^0 \implies reachset_1 \subseteq C(\mathbf{X}_2^0, \mathbf{d}_2^0) \tag{5.35}$$

That is,

$$\texttt{verify\_incremental}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}, C) \supseteq \texttt{verify\_incremental}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}, \varnothing) \tag{5.36}$$

In other words, the reachset returned from a version of `verify_incremental` with caches would be an overapproximation of a version that doesn't have caches.

From Proposition 5.4, a `verify_batch` call can simply be decomposed into `verify_step` calls.

With the 2 conditions stated above, the algorithm for `verify_incremental` can be simplified to be the same as that of `verify_parallel`, which we have proven to be equivalent to `verify`. Thus,

$$\texttt{verify}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}) = \texttt{verify\_parallel}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}) \tag{5.37}$$

$$\subseteq \texttt{verify\_incremental}(H_2, \mathbf{X}_2^0, \mathbf{d}_2^0, \delta, T_{\max}) \tag{5.38}$$

QED.

CHAPTER 6: EXPERIMENTAL EVALUATION OF PARALLEL VERSE

We evaluate the performance of our parallel and incremental verification algorithms compared with the basic serial algorithm through various examples. The experiments will be divided into parts. We will first evaluate our parallel algorithm on a variety of examples. We will then evaluate the parallel algorithm on the 4-way intersection example introduced in Chapter 3. Finally, we compare the serial, parallel, and cached parallel algorithms using repeated runs of similar scenarios.

The algorithms will be evaluated in scenarios with different computation requirements. We will mainly use the number of agents and timesteps as the criteria for computation complexity. More agents in a scenario means that more reachsets need to be computed and more guards need to be checked, assuming the number of timesteps remain the same. More timesteps means that the agents potentially get to interact more often, and cause more transitions and branches to be explored.

In Chapter 5, we described verification algorithms where the initial conditions of continuous state variables are sets of states, we will also evaluate the effectiveness of these algorithms using simulation, which computes reachsets with the initial state as a point.

In these experiments, we consider two types of agents: a 4-d ground vehicle with bicycle dynamics and the Stanley controller [31] (labeled in the table below as C) and a 6-d drone with a NN-controller [32] (labeled as D). 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).

6.1 VARIETY EXPERIMENTS

In this first set of experiments, we use four 2-d maps (`map1` to `map4`) and two 3-d maps (`map5` and `map6`). `map1` and `map2` have 3 and 5 parallel straight tracks, respectively. `map3` has 3 parallel tracks with circular curve. `map4` is a race track imported from OpenDRIVE. `map5` is 3 vertically stacked straight tracks, and `map6` is 3 vertically stacked figure-8 tracks.

The scenario name denotes the number and type of agents and the map used in the scenario. For example, the scenario `3_C_map1` has 3 ground vehicle agents using `map1`. The last 4 rows respectively denote a Vanderpol, spacecraft, gearbox and thermostat scenarios. Note that in the scenario `7_C_map2`, only 3 of the 7 agents have collision avoidance logic, while all agents in all other experiments are collision avoiding.

Table 6.1: Runtime for simulating examples in Section 6. Columns are: scenario name, sensor type (Noisy $\mathcal{S}$), number of mode transitions (#TR), number of leaves in the execution tree (#leaves), the run time of the serial algorithm (Rt Ser), the run time of the parallel algorithm (Rt Par), and the ratio between the number of the serial and parallel algorithms (Speedup). N/A for not available. The unit of run time is second.

Scenario	Noisy $\mathcal{S}$	#Tr	#leaves	Rt Ser	Rt Par	Speedup
`2_D_map6`	No	1	1	0.59	3.58	0.16
`2_D_map5`	No	5	2	0.84	5.6	0.15
`3_D_map5`	No	7	2	1.49	6.65	0.22
`7_C_map2`	No	33	5	14.92	15.22	0.98
`3_C_map1`	No	5	2	0.50	4.23	0.12
`3_C_map3`	No	5	2	0.84	4.8	0.18
`3_C_map4`	No	7	2	4.12	8.52	0.48
`3_C_map1`	Yes	5	2	0.50	4.18	0.12
`2_C_map1`	No	5	2	0.73	4.56	0.16
`1_V`	N/A	1	1	0.04	1.78	0.02
`1_S`	N/A	3	1	0.05	1.85	0.03
`1_G`	N/A	3	1	0.12	2.05	0.06
`2_T`	No	85	64	0.42	6.85	0.06

From Table 6.1 we can see that the parallelization algorithm doesn't improve the runtime but rather increase them. The low computation requirements of these scenarios mean that the time it takes to simulate is too low and the overhead the algorithm brings is too high. This is especially evident for the last row, where the large number of branches would have meant more parallelism thus shorter runtime.

From Table 6.2 we can see roughly the same thing as before. However, because reachability analysis requires more computation, even though the number of transitions or leaves stay about the same compared to the simulation results, some experiments are able to benefit from the algorithm.

6.2 PARAMETRIC INTERSECTION SCENARIO

In the second set of experiments, we will use ground vehicles on the 4-way intersection map introduced in Chapter 3. All of the agents involved are collision avoiding agents. We will vary the number of tracks between 2 and 5, and the number of agents between 3 and 20 to show the effects of the parallelization algorithm under different levels of required computation.

From Table 6.3 we can see configurations with more than 1 leaf have a shorter runtime using `verify_parallel`. This means that a scenario can benefit from the parallel algorithm if it contains branches in the execution tree. Moreover, as the scenario gets more complex

Table 6.2: Runtime for computing reachable sets using examples in Section 6. Columns are: scenario name, sensor type (Noisy $\mathcal{S}$), number of mode transitions (#TR), number of leaves in the execution tree (#leaves), the run time of the serial algorithm (Rt Ser), the run time of the parallel algorithm (Rt Par), and the ratio between the number of the serial and parallel algorithms (Speedup). N/A for not available. The unit of run time is second.

Scenario	Noisy $\mathcal{S}$	#Tr	#leaves	Rt Ser	Rt Par	Speedup
`2_D_map6`	No	8	4	96.91	71.91	1.35
`2_D_map5`	No	5	2	28.38	32.46	0.87
`3_D_map5`	No	7	2	63.07	72.64	0.87
`7_C_map2`	No	37	7	493.00	228.82	2.15
`3_C_map1`	No	5	2	39.32	36.40	1.08
`3_C_map3`	No	4	2	58.22	58.66	0.99
`3_C_map4`	No	7	2	175.82	134.11	1.31
`3_C_map1`	Yes	5	2	50.92	43.59	1.17
`2_C_map1`	No	5	2	24.19	27.84	0.87
`1_V`	N/A	1	1	0.44	5.26	0.08
`1_S`	N/A	3	1	3.75	8.93	0.42
`1_G`	N/A	3	1	122.25	137.22	0.89
`2_T`	No	877	512	274.21	48.51	5.65

with increasing number of agents and lanes, the more benefit the parallel algorithm can provide. This can be clearly seen from Figure 6.4.

From Table 6.5 we can see that these results roughly match that of the simulation in terms of smaller scenarios. The agent and track numbers here are reduced compared to the simulation experiments as the run time quickly explodes for reachability analysis into a matter of hours. We decided experiments running for such a long time are not usable for benchmarking.

6.3 INCREMENTAL VERIFICATION EXPERIMENTS

In the third set of experiments, we will run repeated experiments on similar scenarios to test the effectiveness of the incremental verification algorithm. For each test, we will run experiments on 2 similar scenarios, one immediately after the other, and only record the result from the second run. Each will use `map2` and contain 7 agents, only 3 of which are collision avoiding agents. The agent dynamics and maps for all of the scenarios in this experiment will remain the same, as per the assumptions in Chapter 5.3. For simulation and reachability analysis, we will change the scenarios in 3 different ways:

1. under `repeat`, the 2 scenarios are identical;

2. under `change init`, the initial conditions of one of the agents will be changed;

Table 6.3: Runtime for simulating the intersection examples in Section 6. Columns are: number of tracks (#track), number of agents (#agent), number of mode transitions (#Tr), the number of leaves of the execution tree (#leaves), the total run time of the `verify` (Rt Ser), the total run time of `verify_parallel` (Rt Par), and the ratio between the number of the serial and parallel algorithms (Speedup). The unit of run time is in seconds.

#track	#agent	#timesteps	#nodes	#leaves	Rt Ser	Rt Par	Speedup
2	3	1200	1	1	5.32	9.81	0.54
2	4	1200	1	1	10.12	14.84	0.68
2	5	1200	3	1	17.58	23	0.76
2	6	1200	5	1	26.91	32.21	0.84
2	9	1200	5	1	55.81	63.51	0.88
2	12	1200	20	1	101.75	117.37	0.87
2	15	1200	20	1	155.52	178.03	0.87
3	3	1200	1	1	7	11.57	0.61
3	4	1200	1	1	14.1	18.9	0.75
3	5	1200	5	2	40.46	30.6	1.32
3	6	1200	5	2	59.14	40.22	1.47
3	9	1200	5	2	126.27	78.98	1.6
3	12	1200	37	8	544.07	185.73	2.93
3	15	1200	37	8	810.73	277.14	2.93
4	3	1200	1	1	6.48	10.52	0.62
4	4	1200	1	1	13.95	18.69	0.75
4	5	1200	5	2	43.52	31.35	1.39
4	6	1200	7	2	67.7	48.69	1.39
4	9	1200	7	2	143.27	90.19	1.59
4	12	1200	129	29	1527.05	465.22	3.28
4	15	1200	129	29	2688.21	779.13	3.45
5	3	1200	1	1	6.51	11.16	0.58
5	4	1200	1	1	14.21	18.48	0.77
5	5	1200	5	2	45.21	32.91	1.37
5	6	1200	5	2	64.18	44.7	1.44
5	9	1200	5	2	142.7	86.34	1.65
5	12	1200	133	32	1787.07	510.8	3.5
5	15	1200	133	32	2749.73	772.4	3.56

3. under `change dl`, the decision logic for one of the agents will be changed slightly.

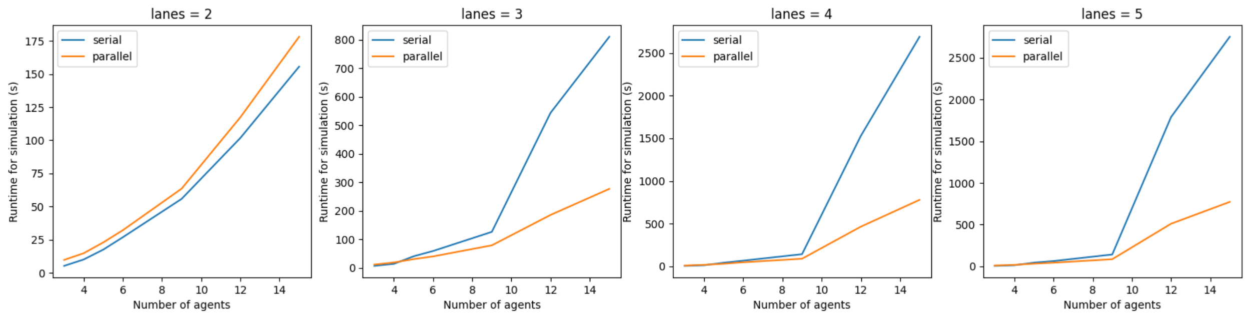

Figure 6.4: Runtime in simulation vs. the number of agents in the scenario

Table 6.5: Runtime for verifying the intersection examples in Section 6. Columns are: number of tracks (#track), number of agents (#agent), number of mode transitions (#Tr), the number of leaves of the execution tree (#leaves), the total run time of the `verify` (Rt Ser), the total run time of `verify_parallel` (Rt Par), and the ratio between the number of the serial and parallel algorithms (Speedup). The unit of run time is second.

#track	#agent	#timesteps	#nodes	#leaves	Rt Ser	Rt Par	Speedup
2	3	600	5	2	19.15	22.83	0.84
2	4	600	5	2	22.85	26.47	0.86
2	5	600	5	2	27.43	30	0.91
2	6	600	20	6	91.95	47.8	1.92
2	7	600	91	20	524.36	121.25	4.32
2	8	600	91	20	584.04	137.41	4.25
2	9	600	91	20	670.1	155.28	4.32
3	3	600	5	2	20.76	22.77	0.91
3	4	600	5	2	23.67	26.07	0.91
3	5	600	5	2	27.78	30.54	0.91
3	6	600	20	6	91.85	49.05	1.87
3	7	600	91	20	519.61	123.77	4.2
3	8	600	91	20	575.27	137.44	4.19
3	9	600	91	20	664.01	154.7	4.29
4	3	600	5	2	19.26	22.49	0.86
4	4	600	5	2	24.37	25.67	0.95
4	5	600	5	2	27.5	29.84	0.92
4	6	600	20	6	95.93	49.48	1.94
4	7	600	91	20	499.17	136.82	3.65
4	8	600	91	20	577.69	138.37	4.17
4	9	600	91	20	658.98	154.29	4.27
5	3	600	5	2	19.96	22.44	0.89
5	4	600	5	2	23.98	26.3	0.91
5	5	600	5	2	26.96	30.17	0.89
5	6	600	20	6	93.53	47.59	1.97
5	7	600	91	20	523.32	121.04	4.32
5	8	600	91	20	578.22	136.57	4.23
5	9	600	91	20	652.64	154.77	4.22

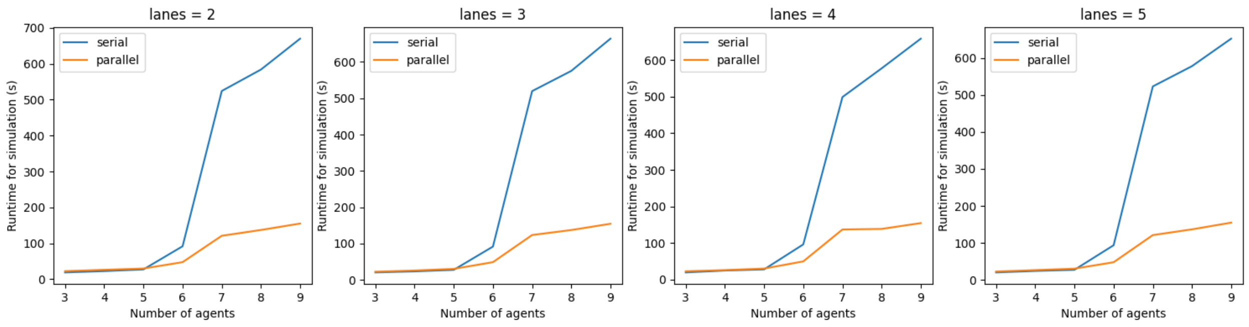

Figure 6.6: Runtime in verification vs. the number of agents in the scenario

Table 6.7: Experimental results for the `verify_incremental` algorithm in simulation, compared to the `verify` and `verify_parallel` algorithms. The columns are runtime in seconds (run time), the total memory usage of Verse in megabytes (memory), the size of C in megabytes (cache size), and the hit rate of the cache (hit rate).

	#Tr	algorithm	run time	memory	cache size	hit rate
repeat	45	`verify`	14.08	300	N/A	N/A
		`verify_parallel`	14.25	358	N/A	N/A
		`verify_incremental`	0.98	375	2.63	100%
change init	24	`verify`	16.5	300	N/A	N/A
		`verify_parallel`	9.35	388	N/A	N/A
		`verify_incremental`	9.19	398	3.72	84.9%
change dl	45	`verify`	13.75	301	N/A	N/A
		`verify_parallel`	13.29	358	N/A	N/A
		`verify_incremental`	8.73	388	3.51	71.9%

Table 6.8: Experimental results for the `verify_incremental` algorithm in verification, compared to the `verify` and `verify_parallel` algorithms. The columns are runtime in seconds (run time), the total memory usage of Verse in megabytes (memory), the size of C in megabytes (cache size), and the hit rate of the cache (hit rate).

	#Tr	algorithm	run time	memory	cache size	hit rate
repeat	105	`verify`	383	357	N/A	N/A
		`verify_parallel`	106	441	N/A	N/A
		`verify_incremental`	24.38	633	4.61	100%
change init	49	`verify`	202	365	N/A	N/A
		`verify_parallel`	99.8	1126	N/A	N/A
		`verify_incremental`	83.08	1142	6.24	89.06%
change dl	45	`verify`	334	367	N/A	N/A
		`verify_parallel`	153	1231	N/A	N/A
		`verify_incremental`	148	1247	6.51	72.6%

6.4 EVALUATION

From the experimental results we can see that the parallelization algorithm can provide significant run time benefits. For scenarios with more branching in the execution tree, the speedup can be up to 3 to 4x. For smaller scenarios or those without any branching, the algorithm only adds delay to running the experiment due to cold starts. However, the added delays don't seem to be serious.

For the incremental verification algorithm, we can see that for the `repeat` cases, the algorithm is able to provide more than 10 times speed up. However, it doesn't handle agent state changes very well. For the `change init` cases, incremental verification provides almost no improvement over just the parallelization algorithm, as the change happened at the start of the experiment. For the `change dl` cases, the change in system behavior does start changing around the half point through the experiment, and from the data we can see that the algorithm is able to reuse the cached result, lowering the runtime to around half of the original.

CHAPTER 7: CONCLUSIONS AND FUTURE DIRECTIONS

In this thesis, we presented algorithms for parallelization and incremental verification in the Verse library. The parallelization algorithm `verify_parallel` allows experiments using the Verse library to be finished in a shorter time, without incurring too much overhead. When given larger scenarios, the parallel algorithms can give significant improvements, up to 4 or 5 times. However when given smaller scenarios the algorithm either gives no improvement or simply introduces delays. The incremental verification algorithm `verify_incremental` makes it easier to iterate on previous scenarios and decision logics. When the states and decision logic of the agents are not changed much, the algorithm can give 100% run time improvements, or even up to 10 times the speed for identical scenarios. In the worst cases, the algorithm doesn't seem to introduce any penalties.

There are several directions along which improvements can be made.

As can be seen from the experiments, the parallelization algorithm doesn't do well with small scenarios. There are two improvements that can be made here. First, we could implement some kind of heuristics in the future so that the library can sense whether parallelization needs to enabled so that there is less overhead. In addition, currently the algorithms parallelize the computation on a per-branch basis. This is fine for scenarios with large amounts of branching, as from the experiment data. One method to improve on this is to parallelize with a granularity of agents. Being able to divide up tasks at a finer scale would mean more opportunities for parallelization, but we would also need to be careful of too small task sizes and develop batching algorithms that both take advantage of the parallelization while minimizing the overhead induced.

In incremental verification, the algorithm currently redoes the computation as soon as any of the agents reaches states or exhibit behaviors never seen before. This is fairly evident from the experiments, where the caching is not able to provide run time improvements for the `change init` cases. Finer grained analysis of agent interactions can be done so that changes in agents' states or behaviors will only trigger recomputation of agents that will be affected.

REFERENCES

[1] S. Mitra, *Verifying Cyber-Physical Systems: A Path to Safe Autonomy.* MIT Press.

[2] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata*, ser. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005, also available as Technical Report MIT-LCS-TR-917.

[3] P. S. Duggirala, S. Mitra, M. Viswanathan, and M. Potok, "C2e2: A verification tool for stateflow models," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035.* Berlin, Heidelberg: Springer-Verlag, 2015, p. 68–82.

[4] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, and R. Grosu, "Xspeed: Accelerating reachability analysis on multi-core processors," in *Hardware and Software: Verification and Testing*, N. Piterman, Ed. Cham: Springer International Publishing, 2015, pp. 3–18.

[5] B. Qi, C. Fan, M. Jiang, and S. Mitra, "Dryvr 2.0: A tool for verification and controller synthesis of black-box cyber-physical systems," in *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, ser. HSCC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3178126.3187008 p. 269–270.

[6] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "SpaceEx: Scalable verification of hybrid systems," in *Computer Aided Verification (CAV)*, 2011, pp. 379–395.

[7] S. Bak and P. S. Duggirala, "Hylaa: A tool for computing simulation-equivalent reachability for linear systems," in *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control.* ACM, 2017, pp. 173–178.

[8] S. Bak, H.-D. Tran, and T. T. Johnson, "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*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 23–32.

[9] M. Althoff, "An introduction to CORA 2015," in *Proc. of the Workshop on Applied Verification for Continuous and Hybrid Systems*, 2015.

[10] C. Fan, B. Qi, S. Mitra, and M. Viswanathan, "Dryvr: Data-driven verification and compositional reasoning for automotive systems," in *Computer Aided Verification (CAV)*, R. Majumdar and V. Kunčak, Eds. Cham: Springer, 2017, pp. 441–461.

[11] P. S. Duggirala, S. Mitra, and M. Viswanathan, "Verification of annotated models from executions," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, ser. EMSOFT '13. IEEE Press, 2013.

[12] X. Chen and S. Sankaranarayanan, "Reachability analysis for cyber-physical systems: Are we there yet?" in *NASA Formal Methods*, J. V. Deshmukh, K. Havelund, and I. Perez, Eds. Cham: Springer, 2022, pp. 109–130.

[13] R. Ray, A. Gurung, B. Das, E. Bartocci, S. Bogomolov, and R. Grosu, "Xspeed: Accelerating reachability analysis on multi-core processors," in *Hardware and Software: Verification and Testing*, N. Piterman, Ed. Cham: Springer, 2015, pp. 3–18.

[14] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification (CAV)*. Springer, 2013, pp. 258–263.

[15] C. Fan, B. Qi, S. Mitra, M. Viswanathan, and P. S. Duggirala, "Automatic reachability analysis for nonlinear hybrid models with C2E2," in *Computer Aided Verification (CAV)*, 2016, pp. 531–538.

[16] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The algorithmic analysis of hybrid systems," *Theoretical Computer Science*, vol. 138, no. 1, pp. 3–34, 1995.

[17] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, "What's decidable about hybrid automata?" *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94–124, 1998.

[18] Y. Li, H. Zhu, K. Braught, K. Shen, and S. Mitra, "Verse: A python library for reasoning about multi-agent hybrid system scenarios," 2023, accepted for Computer Aided Verification (CAV '23).

[19] K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, ser. CBSE '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2465449.2465456 p. 33–42.

[20] H. Günther and G. Weissenbacher, "Incremental bounded software model checking," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, ser. SPIN 2014. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2632362.2632374 p. 40–47.

[21] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Incremental verification of compiler optimizations," in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds. Cham: Springer International Publishing, 2014, pp. 300–306.

[22] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, "Hytech: A model checker for hybrid systems," in *Computer Aided Verification (CAV '97)*, ser. LNCS, vol. 1254, 1997, pp. 460–483.

[23] E. Asarin, O. Bournez, T. Dang, and O. Maler, "Approximate reachability analysis of piecewise-linear dynamical systems," in *Hybrid Systems: Computation and Control*, ser. LNCS, B. Krogh and N. Lynch, Eds., vol. 1790. Hybrid Systems: Computation and Control, 2000, pp. 20–31.

[24] C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi, "Computational techniques for the verification of hybrid systems," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 986–1001, 2003.

[25] A. Gurung, R. Ray, E. Bartocci, S. Bogomolov, and R. Grosu, "Parallel reachability analysis of hybrid systems in xspeed," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 4, pp. 401–423, Aug 2019. [Online]. Available: https://doi.org/10.1007/s10009-018-0485-6

[26] S. Bogomolov, M. Forets, G. Frehse, K. Potomkin, and C. Schilling, "Juliareach: a toolbox for set-based reachability," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019, pp. 39–44.

[27] S. Kong, S. Gao, W. Chen, and E. Clarke, "dreach: δ-reachability analysis for hybrid systems," in *International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 200–205.

[28] A. Devonport, M. Khaled, M. Arcak, and M. Zamani, "Pirk: Scalable interval reachability analysis for high-dimensional nonlinear systems," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_27 p. 556–568.

[29] M. Khaled and M. Zamani, "Pfaces: An acceleration ecosystem for symbolic control," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3302504.3311798 p. 252–257.

[30] D. Sun and S. Mitra, "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*, 2022, pp. 322–337.

[31] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun, "Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing," in *2007 American Control Conference*, 2007, pp. 2296–2301.

[32] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "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*, 2019, pp. 169–178.