Formal Verification of Collision Avoidance for Nonlinear Autonomous Vehicle Models

Rong Gu, Cristina Seceleanu, Eduard Enoiu, Kristina Lundqvist

Mälardalen University, Sweden
firstname.lastname@mdh.se

Abstract. Autonomous vehicles are expected to be able to avoid static and dynamic obstacles automatically, along their way. However, most of the collision-avoidance functionality is not formally verified, which hinders ensuring such systems' safety. In this paper, we introduce formal definitions of the vehicle's movement and trajectory, based on hybrid transition systems. Since formally verifying hybrid systems algorithmically is undecidable, we reduce the verification of nonlinear vehicle behavior to verifying discrete-time vehicle behavior overapproximations. Using this result, we propose a generic approach to formally verify autonomous vehicles with nonlinear behavior against reach-avoid requirements. The approach provides a Uppaal timed-automata model of vehicle behavior, and uses Uppaal STRATEGO for verifying the model with user-programmed libraries of collision-avoidance algorithms. Our experiments show the approach's effectiveness in discovering bugs in a state-of-the-art version of a selected collision-avoidance algorithm, as well as in proving the absence of bugs in the algorithm's improved version.

1 Introduction

Autonomous vehicles (AV) such as driverless cars, robots, and construction equipment are becoming increasingly promising, hence prompting a wide interest in industry and academia. Safety of vehicle operation is the most important concern, requiring these systems to move and perform their tasks without colliding with other static or dynamic objects in the environment, such as big rocks, humans, and other mobile machines. Algorithms like A* [24], Rapidly-exploring Random Tree (RRT) [19], and Theta* [7] are able to navigate the AV by avoiding static obstacles towards reaching their destinations. However, when encountering dynamic obstacles that could appear and move arbitrarily in the environment, these algorithms are not enough for collision avoidance, and have to be complemented by algorithms such as those based on dipole flow fields [26] or dynamic window approach [12], which are capable of circumventing dynamic obstacles.

Although many collision-avoidance algorithms are being proposed by researchers and practitioners in recent years, few of them have been formally verified. However, formal verification is a very important tool for discovering problems in the early stage of algorithm design, or proving the absence of bugs

before the algorithms are deployed into real systems. In this paper, we consider two main challenges that can turn formal verification of AV models and their algorithms into a daunting task: (i) nonlinearity of the vehicle kinematics, and (ii) complexity and uncertainty of the environment where AV move. On the one hand, ordinary differential equations are used to describe the continuous dynamics and kinematics of the often nonlinear vehicles. The trajectories formed by these vehicle models are consequently nonlinear, which is the nonlinearity that we consider throughout the paper. On the other hand, discrete decisions made by vehicle control systems influence the movement of vehicles. In the model-checking world, verification of these so-called *nonlinear hybrid systems* is undecidable [14, 17]. In addition, AV that aim at tracking planned paths are inevitably diverted by their tracking errors caused by the inaccuracy of their sensors and actuators, and the disturbance from the complex environment. Dynamic obstacles can appear at any moment during the AV's movement, and go in any direction, at any speed. All these render exhaustive model checking of nonlinear vehicle models that move in an environment containing static obstacles and uncertain dynamic obstacles an unsolved problem.

In this paper, we solve this problem by addressing challenges (i) and (ii) mentioned above. First, we introduce safe zones of the trajectories formed by nonlinear vehicle models, which overcomes challenge (i). When AV drive along pre-computed and piece-wise-continuous (PWC) reference paths, their deviations from the reference paths are bounded as long as their tracking errors have Lyapunov functions [10]. The boundaries of tracking errors form the safe zone of the AV, assuming the reference path as the axis. As long as the dynamic obstacles do not intrude into these zones, the vehicles are guaranteed to be safe. Based on this observation, we reduce the verification of nonlinear trajectories to the verification of the PWC reference trajectories, and further to the verification of discrete-time models of trajectories. The various vehicle dynamics and kinematics, together with the uncertain tracking errors are all subsumed by the safe zones, so the undecidable verification problem is simplified to a decidable one without losing completeness. The introduction of safe zones also solve the problem of deciding the discretization granularity. Studies using discrete models often have troubles of deciding the sizes of cells in discretized maps [13, 16]. Too large sizes result in an over-approximation of obstacles, thus forbidding many feasible paths, whereas too small sizes increase computation time unnecessarily. In this paper, the boundary of tracking errors is used to compute the accurate size of units discretizing the map.

Next, we solve challenge (ii) by leveraging the non-determinism of timed automata models in Uppaal STRATEGO [8]. The initialization and movement of dynamic obstacles are modeled as timed automata, in which their positions, velocities, etc., are non-deterministically initialized and updated. In this way, the vehicle model passes the verification only when it is able to reach the destination without a collision under any circumstance, meaning that the vehicle meets its reach-avoid requirements when all possible obstacles are considered. Uppaal STRATEGO is an extension of the model checker Uppaal that en-

ables us to build and verify timed-automata models in a user-friendly manner. Most importantly, it supports calling external libraries so that we can treat the user-designed algorithms as black boxes, which makes our approach applicable for any collision-avoidance algorithms. When multiple dynamic obstacles are involved, or the ranges of their parameters are wide, the state space of the model becomes large and the verification becomes computationally expensive or even unsolvable. Consequently, we also propose a way of reducing the state space by splitting the verification into multiple tractable phases.

Note that, our approach is orthogonal to the methods of controller synthesis (e.g., [9, 11]). The latter targets the construction of motion plans that avoid static and dynamic obstacles, whereas our method can be used to verify the correctness of these methods, regardless of the path-planning and collision-avoidance algorithms considered. To summarize, our main contributions are:

1. A proven correct transformation of the verification of nonlinear vehicle models to the verification of PWC models and discrete-time models (Section 3).
2. A generic verification approach for model checking reach-avoid requirements of AV equipped with different collision-avoidance algorithms and kinematic features (Section 5).
3. An implementation of the approach in Uppaal STRATEGO, and a demonstration showing the ability of the approach to discover bugs in a state-of-the-art collision-avoidance algorithm based on dipole flow fields, and to prove the absence of bugs in an improved version of the same algorithm (Section 6).

The remainder of the paper is organized as follows. Section 2 overviews the preliminaries. In Section 3, we introduce the systems to be verified and the reach-avoid requirements. In Section 4, we concretely define the movement and trajectories of AV and prove two theorems of transforming the verification of nonlinear vehicle models to the verification of PWC models and discrete-time models. A detailed description of the verification approach and tool support is presented in Section 5, followed by experiments in Section 6. We compare our study to related work in Section 7, and conclude the paper in Section 8.

2 Preliminaries

In this paper, we denote a vector x by $\vec{x}$, the module of $\vec{x}$ by $\|\vec{x}\|$, and multiplications between two scalars and between a vector and a scalar by "×".

2.1 Timed Automata in UPPAAL

We start by giving the definitions of the syntax and semantics of basic timed automata, after which we briefly overview some of the extensions of timed automata supported by Uppaal [18], which are used in this paper.

A *timed automaton* (TA) [4] is a tuple $< L, l_0, C, A, E, I >$, where L is a set of *locations*, $l_0 \in L$ is the initial location, C is the set of clocks, A is the set

of actions, including synchronization actions and the internal τ-actions, $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard, a set of clocks to be reset, and $I : L \to B(C)$ that assigns clock *invariants* to locations. $B(C)$ stands for the set of formulas obtained as conjunctions of atomic constraints of the form $x \bowtie c$, or $x - y \bowtie c$, where $x, y \in C$, $c \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. The elements of $B(C)$ are called *clock constraints* over C. A clock constraint is *downward closed* if $\bowtie \in \{<, \leq, =\}$.

The semantics of a TA is defined as a *timed transition system* over states (l, u), (where l is a location and $u \in \mathbb{R}^C$), with the initial state $s_0 = (l_0, u_0)$, where u_0 assigns all clocks in C to zero. There are two kinds of transitions:

(i) *delay transitions*: $(l, u) \xrightarrow{d} (l, u \oplus d)$, where $u \oplus d$ is the result obtained by incrementing all clocks of the automaton with the delay amount d, and

(ii) *discrete transitions*: $(l, u) \xrightarrow{a} (l', u')$, corresponding to traversing an edge $l \xrightarrow{g,a,r} l'$ for which the guard g evaluates to *true* in the source state (l, u), a is an action, r is the reset set, and clock valuation u' of the target state (l', u') is obtained from u by resetting all clocks in r such that $u' \vDash I(l')$.

A trace σ of a TA is a sequence of delay and discrete transitions: $\sigma = (l_0, u_0) \xrightarrow{d_1,a_1} (l_1, u_1) \xrightarrow{d_2,a_2} ... \xrightarrow{d_n,a_n} (l_n, u_n)$.

The UPPAAL model checker [18] uses an extension of the timed-automata language with a number of features such as constants, data variables (Boolean variables, bounded global and local integer variables), arithmetic operations, arrays, broadcast channels, urgent and committed locations, and a C-like programming language [5]. In UPPAAL, a location can be marked *urgent* (u) or *committed* (c) to indicate that the time is not allowed to progress in the specified location(s), the latter being a stricter form indicating further that the next edge to be traversed needs to start from a *committed* location. Synchronization between timed automata is modeled via *channels* with rendezvous or broadcast semantics. The synchronization is done by annotating edges in the model with synchronization labels (e.g., **e**! and **e**?), where e is a side-effect-free expression evaluating to a channel. Two automata can synchronize on enabled edges annotated with complementary synchronization labels, that is, two edges in different automata can synchronize if the guards of both edges are satisfied, and they have synchronization labels **e**! and **e**? respectively. When two automata synchronize, both edges are traversed at the same time, that is, the current location of each automata is changed. The broadcast semantics allows 1-to-many synchronizations, and the broadcast sending is never blocking as compared to the rendezvous one. A *network* of UPPAAL TA, $A_1 \parallel ... \parallel A_n$, can be expressed as a parallel composition of n TA over C and A, synchronizing on actions and using shared variables [6].

UPPAAL uses symbolic semantics and symbolic reachability techniques to analyze dense-time state spaces against properties formalized in a simplified *timed computation tree logic* (TCTL) [3], which basically contains a subset of *computation tree logic* (CTL) plus clock constraints. The UPPAAL queries that we verify in this paper are CTL properties of the form: (i) **Invariance**: $A\Box p$ means

that for all paths, for all states in each path, p is satisfied, and (ii) **Liveness**: $A\Diamond p$ means that for all paths, p is satisfied by at least one state in each path. For further details, we refer the reader to the UPPAAL tutorial [5].

2.2 UPPAAL STRATEGO

UPPAAL STRATEGO [8] combines techniques to generate, optimize, compare and explore consequences and performance of strategies synthesized for stochastic priced timed games, in a user-friendly manner. In particular, the tool allows for efficient and flexible "strategy-space" exploration before adaptation in a final implementation. In UPPAAL STRATEGO, strategies become first class citizens, by introducing strategy assignment `strategy S =` and strategy usage `under S` where S is an identifier. These are applied to the queries already used in UPPAAL. The latest version of UPPAAL STRATEGO[1] provides a function of calling external libraries, which enables us to treat the user-designed collision-avoidance algorithm as a black box in our model. In the next section, we define the research problem in this paper.

3 Problem Description

Vehicles that are capable to calculate paths to their destinations, which avoid collision with any obstacles in the environment, and follow them without human intervention, are called *autonomous vehicles* (AV). As depicted in Fig. 1, when

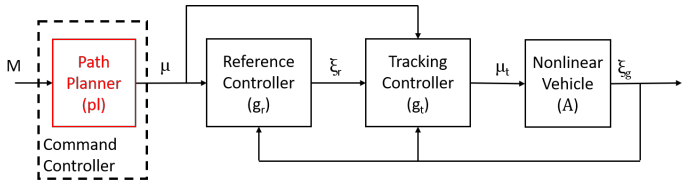


Fig. 1. A controller of autonomous vehicles without a collision-avoidance module

the environment contains only static obstacles whose positions are already known by the AV, paths are calculated by the path planner inside the controller of the AV. Path planners are usually equipped with path-planning algorithms, e.g., Theta* [7] or RRT [19], which explore the map (M) to find a path that avoids the static obstacles and reaches the destination. The reference controller (g_r) uses the output of the path planner and generates a trajectory of the state variables of the system, e.g., position, linear velocity, acceleration, rotational velocity, and heading of the vehicle, as a reference (ξ_r) for the tracking controller (g_t) to follow. The tracking controller aims to produce an input (μ_t) to the vehicle to drive it to track the reference trajectory. As the architecture of controllers is outside the scope of this paper, we refer the reader to the literature [10] for details including the definition of nonlinear control systems.

[1] UPPAAL 4.1.20-stratego-7 is at: https://people.cs.aau.dk/~marius/stratego/

Since the dynamics and kinematics of a real AV are nonlinear, and tracking errors between the actual trajectory and reference trajectory inevitably exist, path planners do not guarantee the safety of AV driving. Moreover, formally verifying if the actual trajectories ever hit the static obstacles is an undecidable problem, due to the reachability verification of nonlinear hybrid system models being undecidable [17]. Over approximation is a method of linearizing the vehicle model, to facilitate verification. Fan et al. [10] propose a method that proves that, as long as the dynamics of tracking errors has a Lyapunov function, the tracking errors are bounded by a piece-wise constant value, which depends on the initial tracking error and the number of segments of the reference trajectory. Fig. 2 shows an example of a reference trajectory and the boundary of tracking errors. As long as the safe regions of AV (green color) do not overlap with the grey areas, the actual trajectory is guaranteed to be safe.

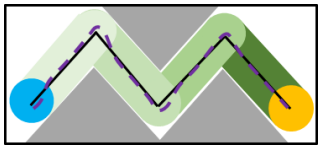


Fig. 2. The reference trajectory of the AV's position is depicted as solid black lines, and the actual trajectory is depicted as violet dotted lines. The boundaries of tracking errors are green. Static obstacles are grey [10].

Thanks to this result, one can reduce the problem of verifying whether the actual trajectory (ξ_g) ever overlaps with obstacles to a simplified problem of verifying whether the distance between the reference trajectory (ξ_r) and the obstacles is larger than the respective boundary of tracking error on each segment of ξ_r [10]. In other words, the reachability verification of nonlinear vehicle models is reduced to the verification of their piece-wise-continuous reference trajectories. Although the problem is much simplified, when dynamic obstacles appear, the verification becomes intractable. Since the static obstacles are known by the AV when it is planning the path, the controller of the vehicle can be designed as a closed system that does not need to perceive the environment while the vehicle is moving. However, dynamic obstacles cannot be known completely before the AV starts to move and encounter them. Therefore, the controller must be additionally equipped with a collision-avoidance module that perceives the environment periodically, via sensors. Fig. 3 shows such a controller, where the tracking module (g_r) and the nonlinear vehicle (A) are compacted as one for brevity. The path planner still calculates a path that avoids known static

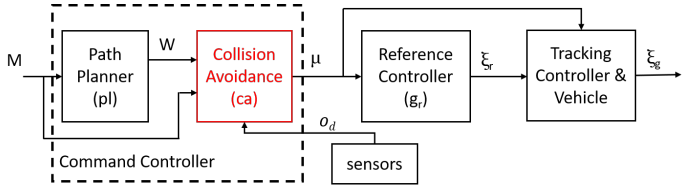


Fig. 3. The architecture of the controller of autonomous vehicles

obstacles and goes to the destination. The path serves as input to the collision-avoidance module as a sequence of waypoints (positions of turning directions, denoted as W), as well as the information of the map (M) and dynamic obsta-

cles (o_d). The command controller should meet the following two requirements, which are the focus of verification in this paper:

- Collision avoidance (invariance property): always circumventing the static and dynamic obstacles;
- Destination reaching (liveness property): always eventually reaching the goal area.

4 Definitions and Verification Reduction Theorems

In this section, we introduce the definitions of the important concepts used in this paper and the collision-avoidance verification theorems that eventually reduce the nonlinear trajectory verification to discrete-time trajectory verification. We denote AV and dynamic obstacles collectively by the term *agents*.

First, let us establish an overall view of the different types of models that are used in this section. So far, we have stated that model-checking *liveness* properties (e.g., destination reaching) and *invariance* properties (e.g., collision avoidance) of nonlinear hybrid systems is undecidable. Note that hybrid systems are described by syntactic models with an underlying semantics defined as hybrid transition systems (HTS), used in the following definitions. As depicted in Fig. 4, the continuous trajectories of agents are modeled as HTS. By incorporating the tracking errors of agents, the continuous trajectories are simplified into piece-wise-continuous (PWC) trajectories. However, the verification of PWC trajectories is still undecidable, so we transform the PWC trajectories into discrete-time trajectories, whose verification is decidable. Furthermore, the two-step transformation from continuous trajectories to discrete-time trajectories is proved to preserve the *liveness* and *invariance* properties that we want to verify (Theorem 1 and Theorem 2).

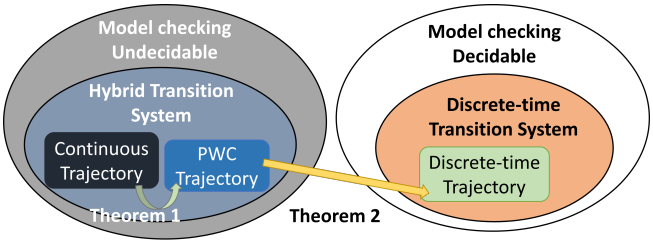


Fig. 4. Overall description of models and their decidability

4.1 Definitions of Maps, Agent States, and Trajectories

In this section, we first define the agent states and the map where they move. Next, we define the command controllers and agent-state trajectories.

Definition 1 (Map). *A map is a 4-tuple* $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$*, where (i)* $\mathcal{X} \in \mathbb{R}^d$ *is the moving space, with* $d \in \{2, 3\}$ *being the dimension of the map, (ii)* $\boldsymbol{O}_u \subseteq \mathcal{X}$ *is the unsafe area, (iii)* $\mathcal{I} \subseteq \mathcal{X}$ *is the initial area of AV, and (iv)* $\mathcal{G} \subseteq \mathcal{X}$ *is the goal area where the AV aims to go.*

An example of a map is illustrated in Fig. 2. Note that the unsafe area (a.k.a., static obstacle) is a polygon when $d = 2$, and a polyhedron when $d = 3$, where all agents must not enter.

Definition 2 (Agent State). *Given a map $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$, an agent state is a 5-tuple $\mathcal{S} =< \vec{p}, \vec{v}, \vec{a}, \theta, \omega >$, where (i) $\vec{p} \in \mathcal{X}$ is the position vector, (ii) $\vec{v}$ is the linear velocity vector, $\|\vec{v}\| \in [0, V_{max}] \subset \mathbb{R}_{\geq 0}$, (iii) $\vec{a}$ is the acceleration vector, $\|\vec{a}\| \in [A_{min}, A_{max}] \subset \mathbb{R}$, (iv) $\theta \in [-\pi, \pi] \subset \mathbb{R}$ is the heading, and (v) $\omega \in [\Omega_{min}, \Omega_{max}] \subset \mathbb{R}$ is the rotational velocity.*

The agent states are states of AV and dynamic obstacles. Some elements in the tuple of agent states $\mathcal{S}$ evolve continuously and some are assumed to change instantaneously. We define the trajectories of the evolution of the agent states in Definition 4. Before that, we first define the controller of AV, where dynamic obstacles ($\mathbf{O}_d$) are instances of agent states $\mathcal{S}$, as follows:

Definition 3 (Controller). *Given a map $\mathcal{M}$, and a set of dynamic obstacles $\boldsymbol{O}_d$, we define a command controller of AV as a 3-tuple $\mathcal{C} =< pl, ca, \Lambda >$, where (i) $pl : \mathcal{M} \longrightarrow \mathcal{W}$ is a path-planning function, $\mathcal{W} \subseteq \mathcal{X}$ is a set of waypoints, (ii) $ca : \mathcal{M} \times \mathcal{W} \times \boldsymbol{O}_d \longrightarrow \Lambda$ is a collision-avoidance function, and (iii) $\Lambda = \{ACC, BRK, TR^+, TR^-, STR\}$ is a set of commands.*

The commands are signals sent from the controllers to the actuators of the AV: ACC means acceleration, BRK means brake, TR^+ and TR^- mean turning counter-clockwise and clockwise, respectively, and STR means moving straightly at a constant speed. An example of the AV's controller architecture is shown in Fig. 3. When an AV starts to move, the transitions of its agent states form a trajectory, in which its position, linear velocity, and heading evolve continuously according to corresponding dynamic functions, whereas its acceleration and rotational velocity change discretely based on the commands.

Definition 4 (Continuous Trajectory). *Given an AV, whose command controller is $\mathcal{C} =< pl, ca, \Lambda >$, we define its movement by a hybrid transition system $< S, s_0, \Sigma, X, \rightarrow >$, where S is a set of states, s_0 is the initial state, $\Sigma \subseteq \Lambda$ is the alphabet, $X = X_d \cup X_c$ is a set of variables combining discrete variables in X_d and continuous variables in X_c, and $\rightarrow$ is a set of transitions defined by the following rules, with kinematic functions of the AV denoted by f:*

- *Delayed transitions: $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{\Delta t} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $t \in X_c$, $\vec{p}' = \vec{p} + \int_l^u \vec{v} dt$, $\vec{v}' = \vec{v} + \int_l^u \vec{a} dt$, $\vec{a}' = \vec{a}$, $\theta' = \theta + \int_l^u \omega dt$, $\omega' = \omega$, $l \in \mathbb{R}_{\geq 0}$ and $u \in \mathbb{R}_{>0}$ are the upper and lower time bounds, respectively, and $\Delta t = u - l$;*
- *Instantaneous transitions: $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{cmd} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $\vec{p}' = \vec{p}$, $\vec{v}' = \vec{v}$, $\vec{a}' = ca(\vec{a}, cmd)$, $\theta' = \theta$, $\omega' = ca(\omega, cmd)$, $cmd \in \Sigma$.*

A run of the transition system defined above over a duration U is a *trajectory* of agent states, also described by the function $\xi : [0, U] \rightarrow \mathcal{S}$. Henceforth, we name the agent-state trajectory as *trajectory* for brevity, and denote $\xi(t)$ as a point of ξ at time t, the projection of ξ on a dimension of an agent-state as $\xi{\downarrow}dimension$, e.g., positions on a trajectory are $\xi{\downarrow}\vec{p}$. The continuous variables of

actual trajectories of agents are generated by their nonlinear kinematic functions, yet these variables are piece-wise-continuous (PWC) in reference trajectories (see Figure 2). More specific, a reference trajectory ξ_r is a sequence of concatenated trajectory segments $\xi_{r,1} \frown ... \frown \xi_{r,k}$. The concatenating points $\{\vec{p}_i\}_{i=0}^{k}$ are the waypoints calculated by path-planners, where the discontinuity of the vehicle's heading θ happens. Therefore, the definition of agent movement on a reference trajectory changes as follows:

Definition 5 (Reference Trajectory). *Let us assume an AV, whose command controller is $\mathcal{C} =< pl, ca, \Lambda >$, and a PWC trajectory ξ_r of the AV, which is a sequence of trajectories $\xi_{r,1} \frown ... \frown \xi_{r,k}$ concatenated by a set of waypoints $\{\vec{P}_i\}_{i=0}^{k}$. Then, the AV's movement along the reference trajectory is a hybrid transition system similar to that of Definition 4, and its transitions are defined by the following rules:*

- *Delayed transitions on $\xi_r{\downarrow}\vec{p} \notin \{\vec{P}_i\}_{i=0}^{k}$: $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{\Delta t} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $\vec{p}' = \vec{p} + (\vec{v} + \frac{\vec{a} \times \Delta t}{2}) \times \Delta t$, $\vec{v}' = \vec{v} + \vec{a} \times \Delta t$, $\vec{a}' = \vec{a}$, $\theta' = \theta$, $\omega' = 0$;*
- *Instantaneous transitions: $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{cmd} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $\vec{p}' = \vec{p}$, $\vec{v}' = \vec{v}$, $\vec{a}' = ca(\vec{a}, cmd)$, $\theta' = \begin{cases} arctangent(\vec{P}_i, \vec{P}_{i+1}), & \text{if } \vec{p} \in \{\vec{P}_i\}_{i=0}^{k-1} \\ \theta, & \text{if } \vec{p} \notin \{\vec{P}_i\}_{i=0}^{k-1} \end{cases}$, $\omega' = 0$*

Intuitively, when an agent is moving along its reference trajectory (ξ_r), its heading ($\xi_r{\downarrow}\theta$) remains unchanged before it arrives at a waypoint, which means the rotational velocity ($\xi_r{\downarrow}\omega$) is irrelevant and remains 0. Therefore, the reference trajectory is infeasible to be tracked exactly by the agents. Although the integration of $\xi_r{\downarrow}\vec{p}$ and $\xi_r{\downarrow}\vec{v}$ on delayed transitions is simplified to polynomial functions, the nonlinearity of $\xi_r{\downarrow}\vec{p}$ still renders undecidability. The trigonometric function in the definition also causes a computational difficulty when running verification. In practice, we use linear speed vector ($\vec{v}$) to describe both the linear speed and the orientation of the agent. The acceleration ($\xi_r{\downarrow}\vec{a}$) changes instantaneously based on the commands from the command controller. Last but not least, the trajectories of dynamic obstacles are similar to Definition 4, but without a well-defined controller. On their instantaneous transitions, accelerations and rotational velocities are changed arbitrarily within the valid ranges.

4.2 Collision-Avoidance Verification Reduction

We use ξ_r and ξ_g to denote the reference and actual trajectory of AV, respectively, and ξ_o for the actual trajectories of dynamic obstacles.

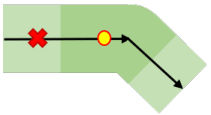


Fig. 5. A dynamic obstacle is at the red cross, while the current position of AV on the reference path is the yellow dot. The safety-critical area is dark green.

Let $d(var_1, var_2)$ denote the distance between var_1 and var_2, e.g., $d(\vec{p}_i, \xi_j{\downarrow}\vec{p})$ is the distance from position $\vec{p}_i$ to trajectory $\xi_j{\downarrow}\vec{p}$, and $d(\xi_i{\downarrow}\vec{p}, \mathbf{O}_u)$ is the distance from trajectory $\xi_i{\downarrow}\vec{p}$ to static obstacles. For brevity, we omit the projection when using this notation, i.e., $d(\vec{p}_i, \xi_j{\downarrow}\vec{p}) = d(\vec{p}_i, \xi_j)$. Let $\xi(t_1, t_2)$ denote

a segment of trajectory ξ between time points t_1 and t_2. The problem of verifying if AV hit static obstacles $\mathbf{O}_u$ is relatively simple, as $\mathbf{O}_u$ does not change. However, checking if AV hit moving obstacles is different and much harder, because both trajectories are formed dynamically while the agents are moving. Dynamic obstacles might meet an AV's reference trajectory, yet far enough from its current position(see Fig. 5). Therefore, we introduce the concept of *safety-critical segments*:

Definition 6 (Safety-Critical Segment). *Let C be the current time. Given a trajectory ξ, a time span of length $T \in \mathbb{R}_{>0}$, we define a safety-critical segment $sc(\xi)$ of ξ, as $\xi(C-T, C+T)$*[2].

The length of time-span T, so that the safety-critical area covers the actual current position of AV, can be delivered by design engineers with knowledge of vehicle dynamics, so this is not within the scope of this paper. Now, instead of checking if any part of the AV's entire trajectory (ξ_g) overlaps with a moving obstacle's trajectory (ξ_o), we check if the safety-critical segments of these two trajectories ($sc(\xi_g)$ and $sc(\xi_o)$) overlap.

Definition 7 (Collision-Avoidance Verification). *Given a map $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$, a nonlinear AV, whose actual continuous trajectory is ξ_g, and a set of dynamic obstacles whose trajectories are in set Ξ_o, we say that the collision-avoidance verification of the AV's actual trajectory equates with verifying that condition $\xi_g{\downarrow}\vec{p} \cap \mathcal{G} \neq \varnothing \wedge \xi_g{\downarrow}\vec{p} \cap \boldsymbol{O}_u = \varnothing \wedge sc(\xi_g{\downarrow}\vec{p}) \cap sc(\xi_o{\downarrow}\vec{p}) = \varnothing$ holds, where $\xi_o \in \Xi_o$.*

Since model-checking ξ_g is undecidable, we prove next that its verification can be reduced to one over the PWC trajectory ξ_r that ξ_g tracks.

Theorem 1 (Non-linearity to PWC). *Assume the collision-avoidance verification condition of Definition 7, a position $\vec{p}_g \in \mathcal{G}$ whose distance to the closest boundary of $\mathcal{G}$ is B, and that the tracking errors of the AV have a Lyapunov function. Then, it follows that if the condition $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \varnothing \wedge d(\xi_r, \boldsymbol{O}_u) > L \wedge d(sc(\xi_r), sc(\xi_o)) > L$, with $L \in \mathbb{R}_{>0}$ and $L \leq B$ holds, then the collision-avoidance condition of Definition 7 holds too.*

Proof. Based on Lemmas 2 and 3 proven by Fan et al. [10], if the tracking errors of the AV have a Lyapunov function, its ξ_g is bounded within a certain distance to its ξ_r. Let the distance be L, then $d(\xi_g, \xi_r) < L \leq B$. Hence, if $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \varnothing$, then $\xi_g{\downarrow}\vec{p} \cap \mathcal{G} \neq \varnothing$. Since $d(\xi_r, \mathbf{O}_u) > L > d(\xi_g, \xi_r)$ and $d(sc(\xi_r), sc(\xi_o)) > L > d(\xi_g, \xi_r)$, then $\xi_g{\downarrow}\vec{p} \cap \mathbf{O}_u = \varnothing \wedge sc(\xi_g{\downarrow}\vec{p}) \cap sc(\xi_o{\downarrow}\vec{p}) = \varnothing$. □

Note that these two problems are not equivalent. When the actual trajectory is not colliding with any obstacles, the distance from the reference trajectory to the obstacles could be less than L. The method of calculating L is not the concern of this paper. We refer the reader to literature [10] for details.

[2] When $C < T$, $sc(\xi) = \xi(0, C+T)$.

4.3 Discretization of Trajectories

Although the verification of nonlinear trajectories is simplified by Theorem 1, model-checking PWC trajectories is still difficult. PWC trajectories are described by hybrid systems, in which variables, e.g., $\vec{p}$ and $\vec{v}$, change continuously (specifically, $\vec{p}$ is nonlinear), whereas variables, e.g., θ, $\vec{a}$ and ω, change instantaneously (Definition 5). Unfortunately, the algorithmic verification of such model is undecidable [23]. To make the problem tractable, we discretize PWC trajectories into a discrete-time model, where the movement of agents (including AV and dynamic obstacles) is sampled synchronously:

Definition 8 (Discrete-Time Trajectory). *Given a PWC trajectory named ξ_r, whose concatenating points (waypoints) are $\{\vec{P}_i\}_{i=0}^{k}$, a discretized trajectory ξ_{rd} of ξ_r is a run of a corresponding discrete-time transition system $< D, d_0, \Pi, \rightarrow >$, where D is the set of states, d_0 is the initial state, $\Pi \subseteq \Lambda \cup \{sync\}$ is the set of labels consisting of controller commands and a label for synchronization with other discretized trajectories, and $\rightarrow$ is a transition relation, in which the instantaneous transitions of $\theta, \vec{a}$ and ω remain the same as defined in Definition 5, and the delayed transitions are sampled at the time points when $\Delta t = \varepsilon$, where $\varepsilon \in \mathbb{R}_{>0}$ is the granularity of sampling:*

- *if $\Delta t < \varepsilon$, $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega >$ does not change,*
- *if $\Delta t = \varepsilon$, $< \vec{p}, \vec{v}, \vec{a}, \theta, \omega > \xrightarrow{\Delta t, sync} < \vec{p}', \vec{v}', \vec{a}', \theta', \omega' >$, where $\theta' = \theta, \omega' = \omega, \vec{a}' = \vec{a}, \vec{v}' = \begin{cases} \vec{v} + \vec{a} \times \varepsilon, & \text{if } \|\vec{v} + \vec{a} \times \varepsilon\| < V_{max}, \\ \frac{\vec{v}}{\|\vec{v}\|} \times V_{max}, & \text{if } \|\vec{v} + \vec{a} \times \varepsilon\| \geq V_{max} \end{cases}$, $\vec{p}' = \begin{cases} \vec{P}_i, & \text{if } \vec{p} + (\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon \succ \vec{P}_i, \\ \vec{p} + (\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon, & \\ & \text{if } \vec{p} + (\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon \preccurlyeq \vec{P}_i \end{cases}$*

To denote if the position passes (resp., does not pass) the next waypoint, we use the syntactic sugar $\succ$ (resp., $\preccurlyeq$). The algorithm of judging this is given in Appendix A.1. Intuitively, when the time interval Δt is less than a small period ε, the environment is not observed, so the trajectories of the agents are not sampled; when Δt reaches ε, the agent states are observed and sampled. When an agent reaches or passes its target waypoint in the current period ε, it stops at the waypoint until the next period comes when the new waypoint and heading are updated by the instantaneous transitions.

Dynamic obstacles do not have pre-computed waypoints but appear and move arbitrarily in the map. However, a reasonable obstacle would not change its direction too frequently, e.g., every sampling period. We design dynamic obstacles such that, initially, they choose their starting agent-states arbitrarily. Then, they keep moving for N sampling periods before choosing a new agent-state as a target. The straight path between the current and target positions is a reference trajectory that the dynamic obstacle tracks in the next N periods, and the tracking errors are also bounded.

The agents' accelerations and rotational velocities are assumed to be changing discretely in these definitions. If the assumption is violated in some applications, one can discretize these two variables in the same way as in the discretization of position and linear velocity. Next, we prove a theorem that reduces the verification of PWC reference trajectories to the one of discrete-time trajectories.

Theorem 2. *(PWC to discrete-time trajectories). Assume a map* $\mathcal{M} =< \mathcal{X}, \boldsymbol{O}_u, \mathcal{I}, \mathcal{G} >$*, a set of trajectories* Ξ_o *formed by dynamic obstacles, with the maximum linear velocity* V*, a reference trajectory* ξ_r *of an AV with concatenating points* $\{\vec{P}_i\}_{i=0}^{k}$*, whose safety-critical segment is* $sc(\xi_r)$*, and synchronized and discretized trajectories* ξ_{rd} *of* ξ_r*, and* ξ_{od} *of* $\xi_o \in \Xi_o$ *with a granularity of sampling* $\varepsilon \leq \frac{L}{\|V\|}$*; here,* $L = L_a + L_o$*, where* L_a *is the tracking-error boundary of the AV, and* L_o *is the smallest tracking-error boundary among dynamic obstacles*[3]*. Then, if* $\vec{p}_g \in \mathcal{G}$*, and* $\xi_{rd}{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \varnothing \wedge d(\xi_{rd}, \boldsymbol{O}_u) > L \wedge d(sc(\xi_{od}), sc(\xi_r)) > L$*, it follows that* $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \varnothing \wedge d(\xi_r, \boldsymbol{O}_u) > L \wedge d(sc(\xi_o), sc(\xi_r)) > L$.

Proof. By substituting Δt in the delay transitions of Definition 5 with ε, we can see that $\xi_{rd}(\varepsilon)$ is a sampling of the reference trajectory $\xi_r(t)$ at the time points when $\Delta t = \varepsilon$. Hence, $\xi_{rd}{\downarrow}\vec{p} \subseteq \xi_r{\downarrow}\vec{p}$. Therefore, if $\xi_{rd}{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \varnothing$, which means ξ_{rd} can reach $\vec{p}_g$, then $\xi_r{\downarrow}\vec{p} \cap \{\vec{p}_g\} \neq \varnothing$ as well.

Based on Definition 8, waypoints $\{\vec{P}_i\}_{i=0}^{k} \subseteq \xi_{rd}{\downarrow}\vec{p}$, where turning occurs. Therefore, if t_i and t_{i+1} are two consecutive sampling points of ξ_{rd}, the line segment connecting t_i and t_{i+1} must be on ξ_r, denoted by $\xi_{rd}(t_i, t_{i+1})$. Therefore, if $d(\mathbf{O}_u, \xi_{rd}(t_i, t_{i+1})) > L$[4], then the concatenation of $\{\xi_{rd}(t_i, t_{i+1})\}_{i=0}^{n-1}$, which is ξ_r, satisfies $d(\mathbf{O}_u, \xi_r) > L$.

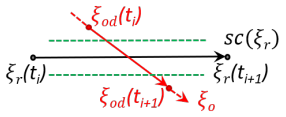


Fig. 6. The trajectory of a dynamic obstacle is red (ξ_o). The reference trajectory of AV is black (ξ_r). Dotted greens lines are the boundaries of tracking errors.

For $\xi_o \in \Xi_o$, similarly, t_i and t_{i+1} are two consecutive sampling points. As depicted in Fig. 6, $\xi_o(t_i, t_{i+1})$ and $\xi_r(t_i, t_{i+1})$ are the segments of $sc(\xi_o)$ and $sc(\xi_r)$, respectively. Assume $d(sc(\xi_{od}), sc(\xi_r)) > L$, but $d(sc(\xi_o), sc(\xi_r)) \leq L$, which means $d(\xi_{od}(t_i), \xi_r(t_i, t_{i+1})) > L$ and $d(\xi_{od}(t_{i+1}), \xi_r(t_i, t_{i+1})) > L$, but $d(\xi_o(t_i, t_{i+1}), \xi_r(t_i, t_{i+1})) \leq L$, then $\xi_o(t_i, t_{i+1})$ and $\xi_r(t_i, t_{i+1})$ must be intersecting, and thus $d(\xi_o(t_i), \xi_o(t_{i+1})) > L$ (see Fig. 6). Based on Definition 8, $d(\xi_o(t_i), \xi_o(t_{i+1})) = \|(\vec{v} + \frac{\vec{a} \times \varepsilon}{2}) \times \varepsilon\| \leq \|V\| \times \varepsilon$. Therefore, $\|V\| \times \varepsilon > L$, which contradicts the assumption $\varepsilon \leq \frac{L}{\|V\|}$. Hence, if $d(sc(\xi_{od}), sc(\xi_r)) > L$, then $d(sc(\xi_o), sc(\xi_r)) > L$. □

Based on Theorems 1 and 2, the reach-avoid verification of discretized trajectories is sufficient to entail that of nonlinear trajectories. The reach-avoid verification of discrete-time transition systems is decidable [14]. Therefore, the undecidable problem of model-checking nonlinear trajectories of agents is successfully simplified to a decidable one over discrete-time trajectories. In the next section, we introduce our approach of verifying the discrete-time models.

5 Verification Approach and Tool Support

In our verification approach, we employ UPPAAL Timed Automata (UTA) [18] to build the discrete-time model of the agents and dynamic obstacles, and UPPAAL

[3] When no dynamic obstacle is detected, L_o is zero.
[4] Computation of $d(\mathbf{O}_u, \xi_{rd}(t_i, t_{i+1}))$ is in Appendix A.2

STRATEGO as the model checker to execute the verification. The reason of these choices is two-fold. First, the non-determinism of transitions in UTA enables us to capture all the possible discrete values of parameters of dynamic obstacles. Second, the latest version of Uppaal STRATEGO provides a function of calling external libraries. This function enables us to design a model for verification without knowing the implementation details of algorithms, hence modeling them as black boxes. Specifically, during the process of verification, whenever the AV needs to avoid dynamic obstacles, our approach simply invokes the external library of the collision-avoidance algorithm to obtain the next moving direction and velocity for the AV. In fact, the approach is not concerned with the implementation details of the algorithms, because as long as the AV, equipped with the algorithm, is able to avoid all obstacles and get to the destination, we conclude that the algorithm satisfies the desired properties. The new function of Uppaal STRATEGO enables us to achieve this.

5.1 General Description of the Approach

Fig. 7 shows the workflow of the verification approach. In Step 1, users write their collision-avoidance algorithms according to the type signature requested by the approach and compile them into executable libraries, e.g., Dynamic-Link Libraries (DLL) in Windows, or Shared Object (SO) in Linux. The type signature

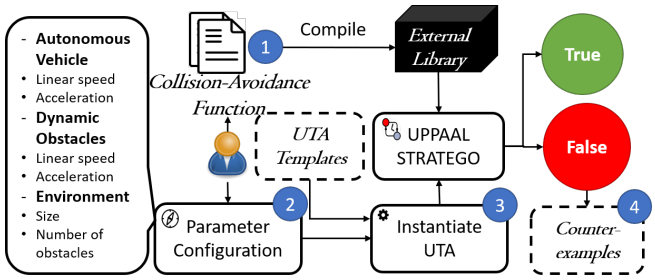
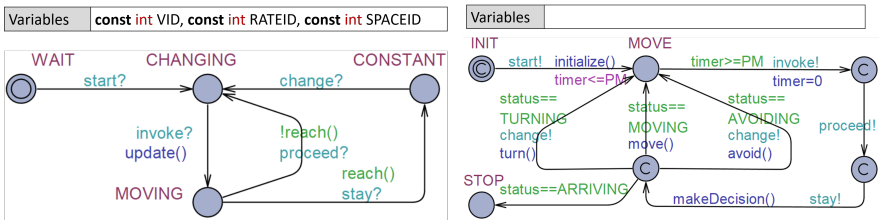


Fig. 7. The workflow of the verification approach

of this function is in Appendix A.3. In Step 2, users configure the parameters of the approach, which are used for instantiating the UTA models. Parameters like "Linear Speed" of the autonomous vehicles and dynamic obstacles regulate the minimum and maximum linear speeds of these agents. The detailed specification of the parameters is in Appendix A.4. In Step 3, UTA templates of the discrete-time models are instantiated into UTA models based on the configured parameters. After the instantiation, the model checker traverses the state space of the model, calls the external libraries of collision-avoidance algorithms when necessary, and verifies if the AV avoids all obstacles and reaches the destination under all circumstances. If the verification result is "true", the algorithms are guaranteed to be correct under the current parameter configuration; otherwise, counter-examples are returned by the model checker for the users to debug their algorithms or change the configuration of the parameters (Step 4).



(a) The UTA template of AV parameters (b) The UTA template of AV controllers

Fig. 8. UTA templates of AV

5.2 Design of the UTA Templates and CTL Queries

In this section, we introduce four UTA templates for updating AV parameters, implementing AV controllers, initializing the parameters of dynamic obstacles, and moving the dynamic obstacles. These templates are designed to be generic, so that users can easily extend the approach to verify their vehicle models. If the new vehicle models need more parameters, users do not need to design new UTA templates but only adding new instances of the UTA template of AV parameters.

UTA Templates of AV Parameters and Controllers. Based on Definition 8, after being initialized, the AV parameters either stay unchanged or update their values at the end of time periods, simultaneously. In another words, the "behaviors" of the UTA templates are the same, and the only difference is the amount of update for each parameter, e.g., positions update based on velocities. Therefore, the UTA template of AV parameters (UTAP) is designed as depicted in Fig. 8(a). Instances of this template are parameters of AV. For example, in a 2-D map, 2 parameters are needed for the position vector, and thus, 2 instances of this template are created.

The AV controllers mainly have three jobs: i) initializing the AV parameters, ii) invoking the UTA of AV parameters at the end of time periods, iii) making decisions, e.g., turning at waypoints, or avoiding obstacles, etc., based on the current status of AV. Fig. 8(b) depicts the UTA template of AV controllers (UTAC) that models these three behaviors. From *location INIT* to *location MOVE*, UTAC is synchronized with UTAP via *channel start*, meaning that the the AV starts to move. On this edge, a function named *initialize()* is invoked to initialize the AV parameters. Next, at *location MOVE*, UTAC stays PM time units, which is the sampling granularity defined in Definition 8 and must satisfy the condition in Theorem 2, and then it transits consecutively three times and comes back to this location. The locations in these three transitions are all committed, because sampling of the agent's status and decision making are done instantaneously at the end of time periods (see Definition 8). In these three transitions, the first one is synchronized with UTAP via *channel invoke*, meaning that AV parameters update their values at this moment (see Fig. 8(a)). The second and third transitions are synchronized with UTAP via *channel proceed* and *channel stay*, respectively, without guards, meaning that AV parameters should continue to be updated or remain the same in the next time period based on their own judgement. Correspondingly, in UTAP, these two synchronized edges are guarded by a function

named *reach()* that judges if the parameter has reached its goal value, e.g., if the velocity reaches its maximum value, it must not change until the controller changes the agent's acceleration and informs the velocity to start to change again. In UTAC, the edge labelled with *channel stay* has a function named *makeDecision()*, which makes the decision about whether the agent should simply continue moving straightly, or has reached a waypoint and must turn, or reached the destination and must stop, or detected an obstacle and must avoid. Based on the decision made by this function, from the next location, UTAC has four outgoing edges guarded by conditions in the format of "status==CMD", where *status* is an integer of UTAC changed by *makeDecision()*, which stores the decision made in this time period, and *CMD* is a constant integer that has four possible values representing the four kinds of decisions. The functions on these edges, namely *turn()*, *move()*, and *avoid()*, are responsible for performing the actions to execute the commands of the controller. Note that, the external function of the collision-avoidance algorithm is invoked in *avoid()*.

UTAC does not have any variables whereas UTAP has three. *VID* is for identifying AV parameters, *RATEID* refers to the ID of the AV parameter that determines the updating step of the current AV parameter, e.g., the *VID* of acceleration is the *RATEID* of linear velocity, and *SPACEID* is the dimension of this AV parameter if it is a vector, e.g., in a 2-D map, position vector has two instances of UTAP, namely positions on axes X and Y. Variables are declared to be constant to reduce the model's state space.

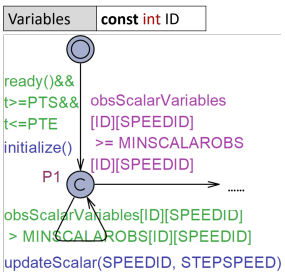


Fig. 9. A part of UTAI for initializing linear velocities of dynamic obstacles.

UTA Templates for Initializing and Moving Dynamic Obstacles. To save the state space of the model, the UTA templates for dynamic obstacles are less generic, so users need to change the template if they want to add new parameters of dynamic obstacles into the approach. However, the current version covers the most widely used parameters defined in Definition 2. Fig. 9 shows a part of UTA template for initializing dynamic obstacles (UTAI), in which an obstacle's linear velocity is decided. An outgoing edge is connected to the initial location, meaning that when time is between *PTS* and *PTE*, UTAI non-deterministically decides whether to initialize an obstacle in the map or wait. After all parameters are being initialized to their maximum values, UTAI transits to a sequence of locations to set the parameters to arbitrary initial values between their minimum and maximum. As depicted in Fig. 9, at *location P1*, the UTA either transits along the self-loop to decrease the value of linear velocity as long as it is greater than the minimum, or leaves this location when the value of linear velocity is greater than or equal to the minimum. The non-deterministic choices between these transitions assign an arbitrary value to the parameter. When running exhaustive verification, all possible values are thus enumerated and used to see if the AV can avoid dynamic obstacles driving at any speed. The

complete UTAI is in the Appendix A.5. Note that, for saving the state space of the model, the maximum and minimum values of parameters are designed wisely so that the state space is reduced while keeping the completeness of verification. Details are reported in Section 5.3.

The UTA template for moving dynamic obstacles (UTAM) is relatively simple[5]. It leaves the initial location with the dynamic obstacles' parameters initialized. When the AV's controller UTA (UTAC) arrives at the end of time periods of sampling, it also invokes UTAM via the channel named *invoke*, when the transitions for updating the parameters of dynamic obstacles are performed (transitions are defined in Definition 9). In this way, sampling of the AV and dynamic obstacles is synchronized at the same moments. The dynamic obstacles eventually end up to a location named *Disappear* when they are far away from AV and stay there until the end of verification. This eliminates the irrelevant movement of dynamic obstacles and saves the model's state space.

CTL Queries. There are two requirements that the collision-avoidance algorithms are supposed to fulfil, namely *obstacle avoiding* and *destination reaching*. The CTL queries are designed as following:

- Obstacle avoiding: `A[] !collision`, where `collision` is a Boolean variable that is updated in UTAC every time period. When the distances from the safety-critical segment of AV to any of the obstacles in the map is less than the boundaries of tracking errors, `collision` is turned to true, and remains *false* elsewhere. Therefore, this query asks whether for all execution paths, is it always the case that collision never occurs?
- Destination reaching: `A<> UTAC.STOP`, where `STOP` is a location in UTAC as depicted in Fig. 8(b). When UTAC goes to location *STOP*, it means that the AV has reached the destination. Therefore, this query asks whether for all the execution paths of the model, does AV eventually reach the destination?

We also leverage a special query in Uppaal STRATEGO to obtain extra information. The query is designed as following:

- Maximum distance from AV to obstacles: `inf{appear}:distance`, where `appear` is a Boolean variable indicating whether a dynamic obstacle appears or not, `distance` is an integer indicating the distances from the safety-critical segment of AV to any of the obstacles in the map, which is also used in the obstacle-avoiding requirement, `inf` returns the infimum of the variable `distance` when `appear` is *true*. This query acquires the closest distance from AV to obstacles, which reveals how conservative the algorithms are when they drive the AV to avoid obstacles.

As the verification against these queries requires exhaustive model checking, the state space and computation time increase rapidly when the size of the model becomes large, e.g., when the AV travels a long time. Therefore, we need a technique to tame this problem.

[5] UTAM is shown in the Appendix A.6.

5.3 Reduction of the State Space of the UTA Model

The purpose of this approach is to exhaustively verify if the two requirements are satisfied no matter when and where the unforeseen dynamic obstacles appear, and to which direction and at what speed they move. Therefore, in the worst-case scenario, we would have to explore the entire map, and enumerate all possible values of linear speeds, rotational speeds, and headings of dynamic obstacles. Most importantly, these obstacles can appear at any moment, i.e., at any sampling period in our model, which would exponentially increase the model's state space and make the problem unsolvable, especially when the number of dynamic obstacles is more than one. In this section, we introduce how to reduce the state space of the UTA model without damaging the completeness of the verification.

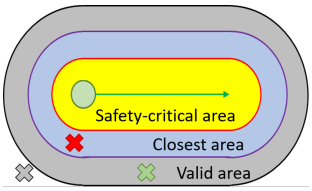


Fig. 10. The green arrow is the reference path. The green circle is the AV. The crosses are the dynamic obstacles, where red and grey ones are invalid positions, and the green cross is valid.

Reduction of the Number of Initial Values of Parameters. Even though the dynamic obstacles can appear at any positions in the map, some positions are too far away from the AV to be relevant at the current period, and some are too close to the AV to be possible to be avoided. Hence, we categorize positions into three classes, namely *safety-critical* area, *closest* area, and *valid* area. Fig. 10 depicts these three kinds of areas. The safety-critical area is defined in Definition 7. Positions from which the distance to the safety-critical segment of the reference path is shorter than or equal to $V \times n \times \varepsilon$ is called *closest* area, where V is the velocity of the dynamic obstacle, ε is the sampling period, and $n \in \mathbb{N}$ is a coefficient whose value depends on the physical limitations of the AV. Obstacles appearing within the closest area are impossible to be avoided, so they should be excluded from the valid initial positions. Similarly, positions from which the distance to the safety-critical segment is greater than $V \times n \times \varepsilon$ and less than or equal to $V \times m \times \varepsilon$ is called *valid* area, where $m \in \mathbb{N}$ is a coefficient for calculating the detection period of sensors. Obstacles outside this area cannot enter the safety-critical area within the current detection period, so they should be excluded from the verification in this period.

Collision-avoidance algorithms can turn the AV to any angle, so any heading of the dynamic obstacles can be dangerous. Hence, the initial value of heading is within π to $-\pi$ and cannot be reduced, and same for the linear velocity.

Phased Verification. Although the state space of the UTA model is much reduced by limiting the initial values of parameters of dynamic obstacles, they can still appear at any moment during the movement of AV, which increases the state space exponentially. However, our verification can be split into several phases, and in each phase, the state space is constrained under a solvable level. For example, the initial headings of dynamic obstacles can be split into four quadrants, and so can their initial positions. When the travelling time of AV is long, it can be split into multiple sections. As long as the concatenating states

between consecutive phases are unchanged, the logic conjunction of verification results of each phase implies the result throughout the entire verification. In the experiments of this study, we split the travelling time of AV at the points where it is travelling straightly at a constant speed, so verification can be statically separated into multiple phases before it starts.

6 Experimental Evaluation

The experiments are conducted on a server with Ubuntu 18.04, 48 CPU, and 256 GB memory. The verification is executed in UPPAAL 4.1.20-stratego-7 [8].

6.1 The Collision-Avoidance Algorithm to be Verified

In the following experiments, we employ a state-of-the-art algorithm to demonstrate the ability of our verification approach. The algorithm is based on dipole flow fields [26], and calculates static flow fields for all objects in the map, and dynamic dipole fields for moving objects. When the AV starts to move, the static flow fields generate attractive forces along the reference path to draw the AV to move towards the closest waypoint. When it encounters a dynamic obstacle, dipole fields are generated dynamically and centered by these two moving objects. Magnetic moments are thus calculated in these dipole fields, which push the moving objects away from each other. Therefore, the AV could possibly deviate from its planned path when meeting dynamic obstacles, and thus, it might encounter some static obstacles that are not taken into account by the reference path. Static flow fields now generate repulsive forces surrounding these static obstacles and push the AV away from them. Formulas for calculating these fields and forces can be found in the literature [26]. This algorithm has not been comprehensively verified considering all possible scenarios of dynamic obstacles.

6.2 Verification Results

In this study, we implement this algorithm as a C-code library and verify it using our approach. We demonstrate how to find the potential problems of this newly-designed algorithm by using counter-examples returned from the approach, followed by verifying iteratively the improved version.

Experiment Design. We report in Table 1 several statistics relevant to the obtained results. For each scenario S, we vary the following aspects relevant in real scenarios:(i) WP representing the number of waypoints, (ii) TT that stands for the travelling time of AV, (iii) AI representing the number of appearing intervals, (iv) DO, the number of dynamic obstacles, and (v) VA, the number of allowed velocities of dynamic obstacles. We note here that dynamic obstacles can appear within the detection range of AV's sensors at any moment during the verification, and this would make the size of the model's state space to grow exponentially. In order to avoid this problem, we use a concept named *appearing interval* for the sake of this experiment. It defines a time interval, between which

the dynamic obstacles appear. After appearing, the AV could be accompanied by the dynamic obstacles throughout the journey or get rid of them soon. One can expand an appearing interval to the same length as the entire travelling time of AV and set the number of appearing intervals (AI) to be 1; or split the travelling time to N segments (i.e., AI = N), and use our phased verification to obtain the final result.

Table 1. Overall Results for the Collision Avoidance Verification.

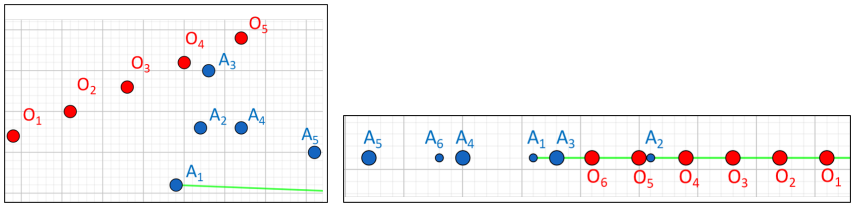
S	Environment		Obstacles		Avoiding Obstacles			Reaching Destination		
	WP	TT	DO	VA	NOS	CT	Result	NOS	CT	Result
S1	2	25	1	1	547,617	2.7 s	true	545,505	5.5 s	true
S2	6	25	1	1	411,747	1.8 s	true	411,168	3.6 s	true
S3	2	85	1	1	3,222,290	15.3 s	true	3,217,767	31.8	true
S3.1	1	30	1	1	1,532,082	7.4 s	true	1,527,811	15.7 s	true
S3.2	1	30	1	1	1,183,792	5.5 s	true	1,185,550	11.4 s	true
S3.3	1	25	1	1	506,416	2.4 s	true	504,406	4.7 s	true
S4	2	15	1	3	12,317,809	1.0 mins	true	12,498,924	2.1 mins	true
S5	2	15	2	1	1,398,011	7.6 s	false	226,896,902	43.2 mins	true

In scenarios S1 and S2, we use one appearing interval for the dynamic obstacle and one allowed velocity, which means that the dynamic obstacle can appear at any moment, always moving at the highest speed throughout the verification. S3 is similar to S1 but it prolongs the travelling time of the AV, and thus, the verification is split into three phases (S3.1 - S3.3). In S4, the dynamic obstacle has three possible velocities, which means its velocity has three initial values and changes arbitrarily during the verification. S5 increases the number of dynamic obstacles to 2, which means there could be at most 2 dynamic obstacles in the map at the same time. For each scenario S, we report the number of states (NOS) and the computation time (CT) needed to verify two necessary properties, namely obstacle avoiding and destination reaching (see Section 5.2 for details). These two values are useful indicators of our approach's performance dealing with various scenarios. All the dynamic obstacles are detected only when they get close to the AV, i.e., they are not foreknown by the AV.

Problems Discovered by Counter-examples. Initially, the proposed collision-avoidance algorithm could not pass the reach-avoid verification in any of these scenarios, and we have discovered several problematic scenarios by analyzing the counter-examples returned from our approach:

Problematic scenario 1. When there is only one dynamic obstacle whose maximum velocity is less than the maximum velocity of AV, the dipole flow fields generated by the algorithm sometimes draw the AV to the obstacle instead of pushing it away from it, until their distance is too short (see Fig. 11(a)). This happens because the magnetic moments could push or draw the moving objects. Here, we improve the algorithm by simply turning the direction of the magnetic moments before the AV and the dynamic obstacle get too close.

Problematic scenario 2. When the dynamic obstacle and AV move directly towards each other, the dipole fields can only generate magnetic moments on the line of their moving directions, which drive the AV to its opposite direction but on the same line. When the dynamic obstacle keeps moving towards the same



(a) Problematic scenario 1 (b) Problematic scenario 2

Fig. 11. Problematic scenarios discovered by counter-examples. AV's discretized trajectory is blue dots. The dynamic obstacle's discretized trajectory is red dots. AV's reference path is the green line. For differentiation, positions that are too close but belong to different time points are represented by small and large dots in scenario 2. A_n and O_m indicate the AV and obstacle, respectively, n and m are time points.

direction, the AV can only move backwards until their distance is longer than a certain value and turns 180° towards its next waypoint, which soon lets the AV get close to the dynamic obstacle again and turns backward (see Fig. 11(b)). According to the counter-examples, this scenarios keeps happening iteratively until the AV stops at the boundary of the map, and is hit by the dynamic obstacle eventually. This is the so-called "livelock" scenario that was also discovered by Gu et al. [13]. To overcome this, we force the AV to turn slightly when its heading is opposite to a dynamic obstacle's heading.

Experimental Results. Although the improved algorithm passes the verification in S1-S4, our results suggest that it still cannot satisfy the obstacle-avoiding requirement in the last scenarios (S5) that contain more than one dynamic obstacle (see Table 1). Note that the destination-reaching property is still satisfied in S5, because the vehicle models are not designed to stop when a collision happens. The rationale of this design is that collisions do not necessarily stop a car from continuing moving. We want to see if the dipole-flow field algorithm can draw the vehicle to its destination anyway when it deviates from the planned paths. Counter-examples are found relatively fast in S5, even though it is more complicated than other scenarios. We leave the further improvement of the algorithm to deal with multiple agents as a future work. The experiments have demonstrated the approach's ability of discovering problems in the early stage of designing collision-avoidance algorithms, and proving the absence of errors in some scenarios for the improved version of the algorithm.

7 Related Work

Luckcuck et al. [20] have carried out a systematic state-of-the-art survey of formal specification and verification approaches for autonomous systems. Their analysis shows that temporal logics and model checking are the most used formalism and verification approach in the literature, over the past decade.

Mitsch et al. [21] propose a method to verify safety properties of robots equipped with a collision-avoidance algorithm called dynamic window approach. Their method is based on hybrid system models and differential dynamic logic

for theorem proving in KeYMaera. Abhishek et al. [1, 2] also use KeYMaera for collision-avoidance verification. Their models consider the realistic geometrical shapes of vehicles, as well as the combination of maneuvre and braking. Heß et al. [15] propose a method to verify an autonomous robotic system during its operation, in order to cope with changing environments. Our work differs from the above studies in the following aspects: based on the bounded tracking errors of actual trajectories, we prove that the reach-avoid verification of nonlinear vehicle models can be simplified to a decidable problem of verifying discrete-time models. In addition, our approach provides counter-examples that are useful to improve the algorithms.

Shokri-Manninen et al. [25] have proposed maritime games as a special case of Stochastic Priced Timed Games and modelled the autonomous navigation using UPPAAL STRATEGO. Collision-free strategies are synthesized and verified taking into account several practically important side constraints such as wind, currents, etc. However, their models do not consider the nonlinear dynamics of the vessels, and the options of maneuvers for collision-avoidance are limited.

O'Kelly et al. [22] have developed a verification tool, called APEX, and investigated the combined action of a behavioral planner and state lattice-based motion planner to guarantee a safe vehicle trajectory. In contrast, our approach provides users a generic interface to verify their specific vehicle models equipped with their own collision-avoidance functions. This feature is beneficial to finding bugs in the early stage of designing new algorithms, or employing modified ones.

8 Conclusion and Future Work

In this paper, we propose a verification approach to formally verify reach-avoid requirements for autonomous vehicles, assuming nonlinear trajectories of movement. We overcome the difficulty of verifying nonlinear hybrid vehicle models by transforming the latter into discrete-time models whose verification we prove sufficient to guarantee meeting reach-avoid requirements of the original nonlinear model. Moreover, we engage tool support that provides users a generic interface to configure and verify their own vehicle models equipped with different collision-avoidance algorithms. We also propose a method for reducing the state space of the models and phasing the verification in order for it to handle complex contexts. Last but not least, we demonstrate the ability of the approach via relevant and comprehensive experiments. In these experiments, we show the verification by model checking of a state-of-the-art collision-avoidance algorithm based on dipole flow fields, which discovered bugs otherwise not detectable by simulation or testing. Based on the returned counter-examples, we have improved the algorithm by fixing the corresponding bugs, and have shown the absence of those bugs in this improved version.

Some interesting directions of future work include: (i) exploring ways of handling complex vehicle models that represent more detailed kinematic features; (ii) statistical verification on the cases where the distances between dynamic obstacles and AV are smaller than the tracking-error boundaries but collisions

do not necessarily occur.

Acknowledgement We acknowledge the support of the Swedish Knowledge Foundation via the profile DPAC - Dependable Platform for Autonomous Systems and Control, grant nr: 20150022, and via the synergy ACICS – Assured Cloud Platforms for Industrial Cyber-Physical Systems, grant nr. 20190038.

References

1. Abhishek, A., Sood, H., Jeannin, J.B.: Formal verification of braking while swerving in automobiles. In: Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control. pp. 1–11 (2020)
2. Abhishek, A., Sood, H., Jeannin, J.B.: Formal verification of swerving maneuvers for car collision avoidance. In: 2020 American Control Conference (ACC). pp. 4729–4736. IEEE (2020)
3. Alur, R., Courcoubetis, C., Dill, D.: Model-Checking in Dense Real-Time. Information and computation **104**(1), 2–34 (1993)
4. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science **126**, 183–235 (1994)
5. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on Uppaal 4.0 pp. 1–48 (2006)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal 4.0 (2006)
7. Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research **39**, 533–579 (2010)
8. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: TACAS. Springer (2015)
9. DeCastro, J.A., Alonso-Mora, J., Raman, V., Rus, D., Kress-Gazit, H.: Collision-free reactive mission and motion planning for multi-robot systems. In: Robotics research, pp. 459–476. Springer (2018)
10. Fan, C., Miller, K., Mitra, S.: Fast and guaranteed safe controller synthesis for nonlinear vehicle models. In: International Conference on Computer Aided Verification. pp. 629–652. Springer (2020)
11. Fan, C., Qin, Z., Mathur, U., Ning, Q., Mitra, S., Viswanathan, M.: Controller synthesis for linear system with reach-avoid specifications. IEEE Transactions on Automatic Control (2021)
12. Fox, D., Burgard, W., Thrun, S.: The dynamic window approach to collision avoidance. IEEE Robotics Automation Magazine **4**(1), 23–33 (1997)
13. Gu, R., Marinescu, R., Seceleanu, C., Lundqvist, K.: Formal verification of an autonomous wheel loader by model checking. In: Proceedings of the 6th Conference on Formal Methods in Software Engineering. pp. 74–83. ACM (2018)
14. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of computer and system sciences **57**(1), 94–124 (1998)
15. Heß, D., Althoff, M., Sattel, T.: Formal verification of maneuver automata for parameterized motion primitives. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 1474–1481. IEEE (2014)
16. Koo, T.J., Li, R., Quottrup, M.M., Clifton, C.A., Izadi-Zamanabadi, R., Bak, T.: A framework for multi-robot motion planning from temporal logic specifications. Science China Information Sciences **55**(7), 1675–1692 (2012)
17. Lafferriere, G., Pappas, G.J., Yovine, S.: A new class of decidable hybrid systems. In: International Workshop on Hybrid Systems: Computation and Control. pp. 137–151. Springer (1999)

18. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. In: International journal on software tools for technology transfer. pp. 134–152. Springer (1997)
19. LaValle, S.M.: Rapidly-exploring random trees: A new tool for path planning. Tech. rep., Computer Science Dept., Iowa State University (10 1998)
20. Luckcuck, M., Farrell, M., Dennis, L.A., Dixon, C., Fisher, M.: Formal specification and verification of autonomous robotic systems: A survey. ACM Computing Surveys (CSUR) **52**(5), 1–41 (2019)
21. Mitsch, S., Ghorbal, K., Vogelbacher, D., Platzer, A.: Formal verification of obstacle avoidance and navigation of ground robots. The International Journal of Robotics Research **36**(12), 1312–1340 (2017)
22. O'Kelly, M., Abbas, H., Gao, S., Shiraishi, S., Kato, S., Mangharam, R.: Apex: Autonomous vehicle plan verification and execution. SAE World Congress (2016)
23. Platzer, A.: Differential-algebraic dynamic logic for differential-algebraic programs. Journal of Logic and Computation **20**(1), 309–352 (2010)
24. Rabin, S.: Game programming gems, chapter a* aesthetic optimizations. Charles River Media (2000)
25. Shokri-Manninen, F., Vain, J., Waldén, M.: Formal verification of colreg-based navigation of maritime autonomous systems. In: International Conference on Software Engineering and Formal Methods. pp. 41–59. Springer (2020)
26. Trinh, L., Ekström, M., Çürüklü, B.: Dipole flow field for dependable path planning of multiple agents. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (September 2017)

A Appendix

A.1 Algorithm of Checking Waypoints Arrival

Algorithm 1: Given a sampling period ε, checking if an AV, whose current position is $\vec{p}_c$, acceleration is $\vec{a}_c$, velocity is $\vec{v}_c$, has reached/passed a waypoint ($\vec{w}$).

```
1 Main(p_c, w, v_c, a_c, ε)
2 p_new = (v_c + (a_c·ε)/2) · ε
3 dis_c = ‖p_c − w‖
4 dis_n = ‖p_c + p_new − w‖
5 if dis_c < dis_n & (dis_c < ‖p_new‖ ‖ dis_n < ‖p_new‖) then
6     return true
7 else
8     return false
```

A.2 Algorithm of Calculating the Distance from a Discrete-Time Trajectory to a Static Obstacle

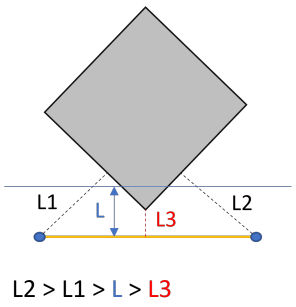


Fig. 12. A special case where the distances from the two ends of a line segment do not represent the distance from the static obstacle to the line segment.

In Algorithm 2, a static obstacle $\mathbf{O}_u$ is a polytope denoted by $Poly(\mathbf{H}, \vec{b}) = \{\vec{x} \in \mathbb{R}^n | \mathbf{H}\vec{x} \leq \vec{b}\}$, where matrix $\mathbf{H} \in \mathbb{R}^{r \times n}$ and vector $\vec{b} \in \mathbb{R}^r$. A surface of the polytope is denoted by S_j, and S_j is $\mathbf{H}^{(j)}\vec{x} = \vec{b}^{(j)}$, where $\mathbf{H}^{(j)}$ is the j^{th} row of $\mathbf{H}$, and $\vec{b}^{(j)}$ is the j^{th} entry of $\vec{b}$. A set $\{\vec{c}_{m,j}\}_{m=0}^{l}$ contains all the vertices of the surface S_j, where l is the total number of vertices. A discrete-time trajectory is denoted as ξ_{rd}. Given two consecutive sampling time points t_i and t_{i+1}, the vectors of positions sampled on ξ_{rd} are $\xi_{rd}(t_i)$ and $\xi_{rd}(t_{i+1})$. The line segment connecting $\xi_{rd}(t_i)$ and $\xi_{rd}(t_{i+1})$ is denoted as $\xi_{rd}(t_i, t_{i+1}) = \{\vec{x} \in \mathbb{R}^n | \vec{H}\vec{x} = b\}$, where $\vec{H} \in \mathbb{R}^n$. Apparently, $\xi_{rd}(t_i, t_{i+1})$ is presented in same form of S_j. Hence, we denote S_j and $\xi_{rd}(t_i, t_{i+1})$ collectively as $\{\vec{x} \in \mathbb{R}^n | \vec{H}\vec{x} = b\}$ in a function named *Distance* that is used in Algorithm 2. This function calculates the distance from a point $(\vec{p})$ to a surface $(\{\vec{x} \in \mathbb{R}^n | \vec{H}\vec{x} = b\})$ by using this formula: $\frac{|\vec{H}\vec{p} - b|}{\|\vec{H}\|}$.

Figure 12 illustrates the reason why Algorithm 2 is designed like this. When the two ends of a line segment stay at the same side of a static obstacle, in another word, the projections from the two ends to the static obstacle fall onto its same surface, the shortest distance between these two distances represent the distance from the line segment to the static obstacle. However, when the two ends of a line segment stay at the different sides of a static obstacle, as depicted in Fig. 12, then the distance is changed to be the one from a vertex of the static obstacle to the line segment. Hence, Algorithm 2 is designed to cover both cases.

Algorithm 2: Given a discrete-time trajectory ξ_{rd}, two sampling points t_i, t_{i+1}, and a static obstacle $\mathbf{O}_u$, calculating the shortest distance between $\xi_{rd}(t_i, t_{i+1})$ and $\mathbf{O}_u$. A surface of $\mathbf{O}_u$ is denoted by $S_j \in \mathbf{O}_u$.

Function `Main`(ξ_{rd}, t_i, t_{i+1}, $\boldsymbol{O}_u$)**:**
  $d_j = 0$, $d_{min} = 0$
  $\mathcal{D} = \varnothing$, $\mathcal{P} = \{\xi_{rd}(t_i), \xi_{rd}(t_{i+1})\}$, $S_{min} = \varnothing$
  for $S_j \in \boldsymbol{O}_u$ **do**
    d_j = `Distance`($\mathcal{P}, S_j$)
    $push(\mathcal{D}, d_j)$
  $d_{min} = min(\mathcal{D})$ `// the shortest distance in` $\mathcal{D}$
  `//` S_{min} `denotes the surface of` $\mathtt{O}_u$ `from where` d_{min} `is calculated`
  $S_{min} = minarg(\mathcal{D})$
  $\{\vec{c}_{m,j}\}_{m=0}^{l} = vertices(S_{min})$ `// get the vertices of` S_{min}
  d_{min} = `Distance`($\{\vec{c}_{m,j}\}_{m=0}^{l}, \xi_{rd}(t_i, t_{i+1})$)
  $push(\mathcal{D}, d_{min})$
    `//` $min(\mathcal{D})$ `now represents the distance from` $\xi_{rd}(t_i, t_{i+1})$ `to` $\mathtt{O}_u$
  $d_{min} = min(\mathcal{D})$
  return d_{min}

`//` `Assume` $target = \{\vec{x} \in \mathbb{R}^n : \vec{H}\vec{x} = b\}$

Function `Distance`($\mathcal{P}, target$)**:**
  $D = \infty$, $d_i = 0$
  for $p_i \in \mathcal{P}$ **do**
    $d_i = \frac{|\vec{H}\vec{p}_i - b|}{\|\vec{H}\|}$
    if $D > d_i$ **then**
      $D = d_i$
  return D

A.3 Type Signature of the Collision-Avoidance Function

```
typedef struct
{
    int x;
    int y;
    int z;//only used in 3-D maps
}Point;
typedef struct
{
    int speed;//rotational speed
    int heading;//direction
}RotationalState;
typedef struct
{
    int acc;//acceleration
    int speed;//linear speed
```

```
}LinearState;
void collisionAvoidance(int dynObsNum, bool *appeared, int staticObsNum,
    Point *staticObsVerdices, Point *dynObsPosition, RotationalState
    rotationalObsState, LinearState linearObsStates, Point avPosition,
    RotationalState rotationalAVState, LinearState linearAVState, Point
    goalPosition, Point *path, int pathLen, RotationalState *
    rotationalGoalState, LinearState *linearGoalState);
```

A.4 An Example of Configuration of the Framework Parameters

```
[Autonomous Vehicle]
LinearSpeedMin = 0
LinearSpeedMax = 45
AccelerationMin = -5
AccelerationMax = 9
SafetyCriticalLength = 5

[Dynamic Obstacle 1]
ID = 1
LinearSpeedMin = 0
LinearSpeedMax = 30
AccelerationMin = -5
AccelerationMax = 6

[Dynamic Obstacle 2]
ID = 1
LinearSpeedMin = 0
LinearSpeedMax = 50
AccelerationMin = -5
AccelerationMax = 10

[Map]
Dimension = 3
LengthX = 1000
LengthY = 500
LengthZ = 3000
NumberStatisObstacles = 5
NumberDynamicObstacles = 2
```

A.5 A UTA Template of Dynamic Obstacle Initialization

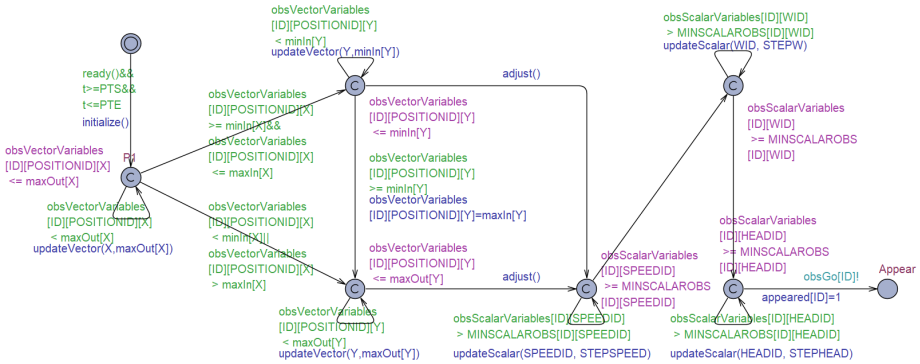


Fig. 13. The UTA template of dynamic obstacle initialization

A.6 A UTA Template of Dynamic Obstacle Movement

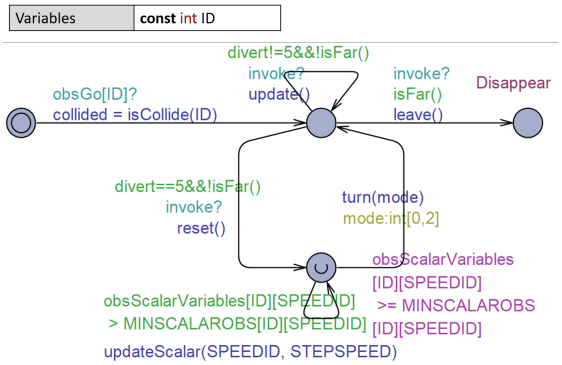


Fig. 14. The UTA template of dynamic obstacle movement