Towards a Two-layer Framework for Verifying Autonomous Vehicles

Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist

Mälardalen University, Västerås, Sweden
`(first.last)@mdh.se`

Abstract. Autonomous vehicles rely heavily on intelligent algorithms for path planning and collision avoidance, and their functionality and dependability can be ensured through formal verification. To facilitate the verification, it is beneficial to decouple the static high-level planning from the dynamic functions like collision avoidance. In this paper, we propose a conceptual two-layer framework for verifying autonomous vehicles, which consists of a static layer and a dynamic layer. We focus concretely on modeling and verifying the dynamic layer using hybrid automata and UPPAAL SMC, where a continuous movement of the vehicle as well as collision avoidance via a dipole flow field algorithm are considered. In our framework, decoupling is achieved by separating the verification of the vehicle's autonomous path planning from that of the vehicle autonomous operation in its continuous dynamic environment. To simplify the modeling process, we propose a pattern-based design method, where patterns are expressed as hybrid automata. We demonstrate the applicability of the dynamic layer of our framework on an industrial prototype of an autonomous wheel loader.

1 Introduction

Autonomous vehicles such as driverless construction equipment bear the promise of increased safety and industrial productivity by automating repetitive tasks and reducing labor costs. These systems are being used in safety- or mission-critical scenarios, which require thorough analysis and verification. Traditional approaches such as simulation and prototype testing are limited in their scope of verifying a system that interacts autonomously with an unpredictable environment that assumes the presence of humans and varying site conditions. These techniques are either applied later in the system's development cycle (testing), or they simply cannot prove, exhaustively or statistically, the satisfaction of properties related to autonomous behaviors such as path planning, path following, and collision avoidance (simulation). Formal verification is usually adopted to compensate such shortage, yet verifying such a complex system in a continuous and dynamic environment is still considered a big challenge [1][4].

In this paper, we approach this challenge by proposing a two-layer framework consisting of a *static* and a *dynamic* layer, which facilitates verifying autonomous vehicles. The structure of the framework separates the static high-level path planning that assumes an environment with a predefined sequence of milestones that need to be reached, as well as static obstacles, from the dynamic functions like collision avoidance, thus providing a separation of concerns for the system's design, modeling, and verification. To improve on existing formal models of vehicle movement [17][26], in the dynamic layer, we propose a continuous model of the vehicle's motion, together with a model of the environment, where moving obstacles are either predefined or dynamically generated. The resulting models are hybrid automata, as accepted by the input language of UPPAAL Statistical Model Checker (SMC). The vehicle's dynamics is modeled as ordinary differential equations assigned to locations in the hybrid automata. In this paper, the hybrid automata only have non-deterministic time-bounded delays that are encoded based on the default uniform distributions assigned by UPPAAL SMC. We also consider the embedded control system of the autonomous vehicle including the involved processes, as well as the scheduling and communication among them. The path planning is following the Theta* algorithm [6], and the collision avoidance relies on the dipole flow field one [29]. Both algorithms are encoded as C-code functions in UPPAAL SMC, within the dynamic layer of our framework. Once this is accomplished, we can statistically model check the resulting network of hybrid automata, against probabilistic invariance properties expressed in weighted metric temporal logic [5]. To simplify the modeling process, we propose a pattern-based design method to provide reusable templates for various components of the framework. We demonstrate the applicability of our approach for modeling and analyzing the dynamic layer on an industrial autonomous wheel loader prototype that should meet certain safety-critical requirements.

This paper is organized as follows. In Section 2, we overview hybrid automata and UPPAAL SMC, as well as the Theta* algorithm for path planning, and the dipole flow field algorithm for collision avoidance. Section 3 describes the function of the autonomous wheel loader and its architecture. In Section 4, we present the conceptual two-layer framework, and in Section 5 we propose the pattern-based modeling of the components (of the dynamic layer) and their formal encoding. Next, we demonstrate the applicability of the framework on the autonomous wheel loader, and we present the verification results in Section 6. We compare to related work in Section 7, before concluding and outlining future lines of research in Section 8.

2 Preliminaries

In this section, we overview the background information needed for the rest of the paper, that is, hybrid automata and UPPAAL SMC, as well as the Theta* and dipole flow field algorithms.

2.1 Hybrid Automata and UPPAAL SMC

UPPAAL SMC [7] is an extension of the tool UPPAAL[21], which supports statistical model checking of hybrid automata (HA). A HA is defined as the following tuple:

$$HA = < L, l_0, X, \Sigma, E, F, I >, \tag{1}$$

where: L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, X is a finite set of continuous variables, $\Sigma = \Sigma_i \uplus \Sigma_o$ is a finite set of actions that are partitioned into inputs (Σ_i) and outputs (Σ_o), E is a finite set of edges of the form (l, g, a, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$, $a \in \Sigma$ is an action label, and φ is a binary relation on $\mathbb{R}^X$, $F(l)$ is a delay function for the location $l \in L$, and I assigns an invariant predicate $I(l)$ in/of L, which bounds the delay time in the respective location. In UPPAAL SMC, locations are marked as *urgent* (denoted by encircled u) or *committed* (denoted by encircled c), indicating that time cannot progress in such locations. Committed locations are more restrictive, requiring that the next edge to be traversed needs to start from a committed location. The delay function $F(l)$ for a simple clock variable x, which is used in (priced) timed automata, is encoded as the linear differential equation $x' = 1$ or $x' = e$ appearing in the invariant of l.

The semantics of the HA is defined over a timed transition system, whose states are pairs $(l, u) \in L \times \mathbb{R}^X$, with $u \vDash I(l)$, and transitions defined as: (i) delay transitions ($< l, u > \xrightarrow{d} < l, u + d >$ if $u \vDash I(l)$ and $(u + d') \vDash I(l)$, for $0 \leq d' \leq d$), and (ii) discrete transitions ($< l, u > \xrightarrow{a} < l', u' >$ if edge $l \xrightarrow{g,a,r} l'$ exists such that $a \in \Sigma, u \vDash g$, clock valuation u' in the target state (l', u') is derived from u by resetting all clocks in the reset set r of the edge, such that $u' \vDash I(l')$).

In UPPAAL SMC, the automata have a stochastic interpretation based on: (i) the probabilistic choices between multiple enabled transitions, and (ii) the non-deterministic time delays that can be refined based on probability distributions, either uniform distributions for time-bounded delays or user-defined exponential distributions for unbounded delays. In this paper, only the default uniform distributions for time-bounded delays are used. Moreover, the UPPAAL SMCmodel is a network of HA that communicate via broadcast channels and global variables. Only broadcast channels are allowed for a clean semantics of purely non-blocking automata, since the participating HA repeatedly race against each other, that is, they independently and stochastically decide on their own how much to delay before delivering the output, with the "winner" being the automaton that chooses the minimum delay.

UPPAAL SMC supports an extension of *weighted metric temporal logic* for probability estimation, whose queries are formulated as follows: `Pr[bound] (ap)`, where `bound` is the simulation time, `ap` is the statement that supports two temporal operators: "*Eventually*" ($\Diamond$) and "*Always*" ($\Box$). Such queries estimate the probability that `ap` is satisfied within the simulation time bound. Hypothesis testing (`Pr[bound]`(ψ) $\geq p_0$) and probability comparison (`Pr[bound]`(ψ_1) $\geq$ `Pr[bound]`(ψ_2)) are also supported.

2.2 Theta* Algorithm

In this paper, we employ the Theta* algorithm to generate an initial path for our autonomous wheel loader. The Theta* algorithm has been firstly proposed by Nash et al. [6] to generate smooth paths with few turns, from the starting position to the destination, for a group of autonomous agents. Similar to the A* algorithm that we have used in our previous study [17], the Theta* algorithm explores the map and calculates the cost of nodes by the function $f(n) = g(n) + h(n)$, where n is the current node being explored, $g(n)$ is the Euclidean distance from the starting node to n, and $h(n)$ is the estimated cheapest cost from n to the destination. In this paper, we use Manhattan distance [2] for $h(n)$. In each search iteration, the node with the lowest cost among the nodes that have been explored is selected, and its reachable neighbors are also explored by calculating their costs. The iteration is eventually ended if the destination is found or all reachable nodes have been explored. As an optimized version of A*, Theta* determines the preceding node of a node to be any node in the searching space instead of only neighbor nodes. In addition, Theta* adds a line-of-sight (LOS) detection to each search iteration to find an any-angle path that is less zigzagged than those generated by A* and its variants. For the detailed description of the algorithm, we refer the reader to the literature [6].

2.3 Dipole Flow Field for Collision Avoidance

Searching for a path from the starting point to the goal point, assuming a large map, is not an easy task and it is usually computationally intensive. Hence, some studies have adopted methods to generate a small deviation from the initial path, which is much easier to compute than an entirely new path, while being able to avoid obstacles. To avoid collisions, Trinh et al.[29] propose an approach to calculate the *static flow field* for all objects, and the *dynamic dipole field* for the moving objects in the map. In the theory of dynamic dipole field, every object is assumed to be a source of magnetic dipole field, in which the magnetic moment is aligned with the moving direction, and the magnitude of the magnetic moment is proportional to the velocity. In this approach, the static flow field is created within the neighborhood of the initial path generated by the Theta* algorithm. The flow field force is a combination of the attractive force drawing the autonomous wheel loader to the initial path, and the repulsive force pushing it away from obstacles. Unlike the dipole field force, the flow field force always exists, regardless of whether the vehicle is moving or not. As soon as the vehicle equipped with this algorithm gets close enough to a moving obstacle, the magnetic moment around the objects keeps them away from each other. The combination of the static flow field and the dynamic dipole field ensures that the vehicle moves safely by avoiding all kinds of obstacles and that it eventually reaches the destination, as long as a safe path exists. Compared with other methods [30][16], this algorithm provides a novel method for path planning of mobile agents, in the shared working environment of humans and agents, which suits our requirements well. For details, we refer the reader to the literature [29].

3 Use Case: Autonomous Wheel Loader

In this section, we introduce our use case, which is an industrial prototype of an autonomous wheel loader (AWL) that is used in construction sites to perform operations without human intervention [17]. On one hand, like other autonomous vehicles, autonomous wheel loaders need to be equipped with path-planning and collision-avoidance capabilities. On the other hand, they also ought to accomplish several special missions, e.g., autonomous digging, loading and unloading, often in a predefined sequence. Furthermore, autonomous wheel loaders usually work in unpredictable environments – dust and various sunlight conditions (from dim to extremely bright) that might cause inaccuracy or even errors in image recognition and obstacle detection. Moving entities, e.g., humans, animals, and other machines, might also behave unpredictably, for there are no traffic lights and lanes. Despite such disadvantages, the AWL's movements are less restricted if compared to, for instance, self-driving cars, as there are only a few traffic rules in sites. They can also stop and wait as long as they need without influencing the vehicles behind them. All these characteristics make our path-planning (Theta*) and collision-avoidance (Dipole Flow Field) algorithms applicable.

The architecture of the AWL's control system, presented in Figure 1, consists of three main units: a vision unit, a control unit, and an execution unit, which are connected by CAN buses. In this paper, we mainly focus on the con-

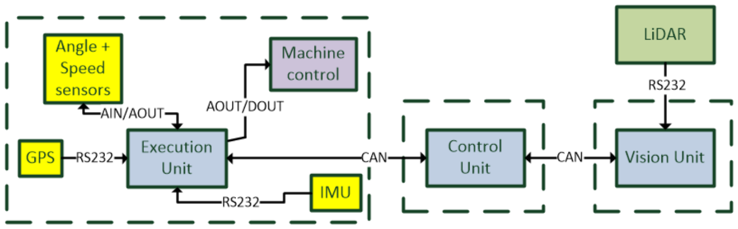


Fig. 1. The architecture of the AWL's embedded control system

trol unit that consists of three parallel processes, namely `ReadSensor`, `Main`, and `CalculateNewPath`, as depicted in Figure 2. These three processes are executed in parallel on independent cores. The process `ReadSensor` acquires data from sensors (e.g., LIDAR, GPS, angle and speed sensors, etc.) and sends them to the shared memory before they are accessed by process `Main` that runs the path-planning algorithm and invokes a function called `Execution Function`, in which three sub-functions are called. The function `AdjustAngle` adjusts the

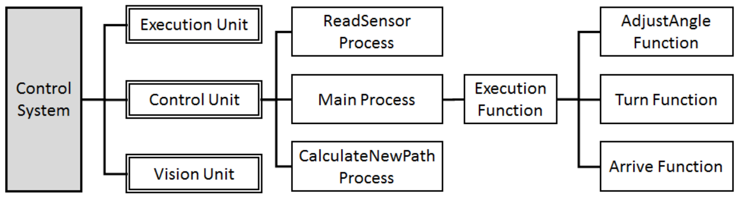


Fig. 2. Process allocation in the control system

moving angle of the AWL, based on its own and the obstacles' positions. Function `Turn` judges if the AWL arrives at one of the milestones on its initial path calculated by the path-planning algorithm, and changes its direction based on the result. Function `Arrive` judges if the AWL reaches the destination and sends the corresponding commands. Basically, the processes `Main` and `ReadSensor` are responsible for the AWL's regular routine. However, when an unforeseen obstacle suddenly appears in its vision, the process `Main` sends a request to process `CalculateNewPath`, in which the collision-avoidance algorithm is executed and a new and safe path segment is generated if it exists. Note that, although the AWL has more functionality, e.g., digging and loading, we focus only on the path planning and collision avoidance in this paper.

The loader's architecture (Figures 1, 2), including the parallel processes and functions, is hierarchical. Moreover, the distributed nature of the AWL's components, and the dynamic nature of its movement (including collision avoidance) call for a separation of concerns along the static and the dynamic dimensions of the system. Hence, in the following, we propose a two-layer framework to model and verify autonomous vehicles on different levels.

4 A Two-level Framework for Planning and Verifying Autonomous Vehicles

As it is shown in Figure 3, our two-level framework consists of a static layer and a dynamic layer, between which data is exchanged according to a defined/-chosen communication protocol. The *static layer* is responsible for path and mission planning for the AWL, according to possibly incomplete information of the environment. In this layer, known static obstacles are assumed, together with milestones representing points of operation of the loader. The *dynamic layer* is dedicated to simulating and verifying the system following the reference path given by the static layer, while considering continuous dynamics in an environment containing moving and unforeseen obstacles.

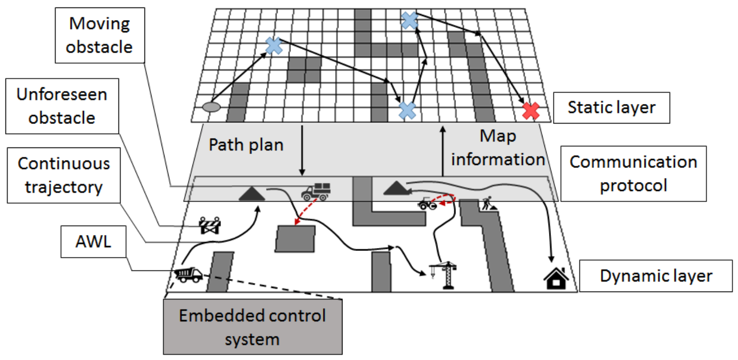


Fig. 3. Two-layer framework for planning and verifying autonomous vehicles

Static layer. The static layer is defined as a tuple $< E_s, S_s, M_s >$, where E_s denotes a discrete environment, S_s is a set of known static obstacles, and M_s is a set of milestones associated to missions (e.g., digging, loading, unloading, charging), including the order of execution, and timing requirements. As the path found by the path-planning algorithm is a connection of several straight-line segments on the map, realistic trajectories and continuous dynamics do not need to be considered in this layer. Hence, the environment is modeled as a discrete

Cartesian grid whose resolution is defined appropriately to present various sizes of static obstacles, e.g., holes, rocks, signs, etc. Even if not entirely faithful to reality, the Cartesian grid provides a proper abstraction of the map for path and mission planning. As the static layer is still at the conceptual stage currently, we propose several possible options for modeling and verification of this layer. DRONA [10] is a programming framework for building safe robotics systems. which has been applied in collision-free mission planning for drones. Rebeca is a generic tool for actor-based modeling and has been proven to be applicable for motion planning for robots [18]. Mission Management Tool (MMT) is a tool allowing a human operator an intuitive way of creating complex missions for robots with non-overlapping abilities [25].
Dynamic layer. The dynamic layer is defined as a tuple $< E_d, T_s, S_d, M_d, D_d >$, where E_d is a continuous environment, T_s is the trajectory plan input by the static layer, S_d is a set of static obstacles, M_d is a set of moving obstacles that are predefined, D_d is a set of unforeseen moving obstacles that are dynamically generated. The speed and direction of a moving obstacle $m_0 \in M_d$ are predefined as constant values in our model. The dynamically generated moving obstacle $d_0 \in D_d$ is instantiated during the verification when its initial location, moving speed and angle are randomly determined. Collision-avoidance algorithms are executed in this layer if the vehicle meets moving obstacles or unforeseen static obstacles. Ordinary differential equations (ODEs) are adopted to model the continuous dynamics of moving objects (e.g., vehicle, human, etc.), and the embedded control system of the autonomous vehicle is modeled in this layer.

This two-layer design has many benefits. Firstly, it provides a separation of concerns for the system's design, modeling, and verification. As a path plan does not concern the continuous dynamics of the vehicle, the discrete model in the static layer is a proper abstraction, which sacrifices some unnecessary realistic elements but preserves the possibility of exhaustive verification. The dynamic layer, which concerns the actual trajectories of moving objects, consists of hybrid models that contain relatively more realistic details of the system and environment, which enhance the truthfulness of the model. However, as a tradeoff, only probabilistic verification is supported in this layer. In addition, modification of algorithms or design is only restricted within the corresponding layer, so potential errors will not propagate in the entire system. Secondly, the two-layer framework is open for extension. It provides a possibility to add layers for new functions, such as artificial intelligence or centralized control.

5 Pattern-based Modeling of the Dynamic Layer

A classic control system consists of four components: a plant containing the physical process that is to be controlled, the environment where the plant operates, the sensors that measure some variables of the plant and the environment, and the controller that determines the system state and outputs timed-based signals to the plant [22]. In our case, as shown in Figure 1, the execution unit is the "plant" that describes the continuous dynamics of the AWL. The "sensors" are divided into two classes: vision sensors (LiDAR) connecting to the vision unit, and motion sensors (GPS, IMU, Angle and Speed sensors) connecting to the execution unit.

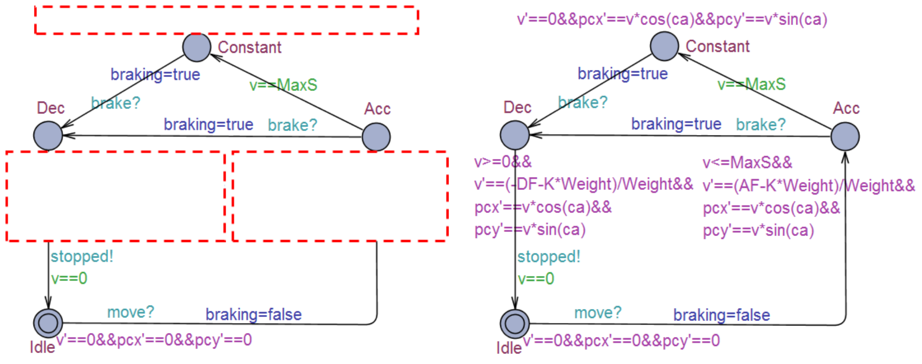
5.1 Patterns for the Execution Unit

Currently, the vision unit and vision sensors have no computation ability, so they are simply modeled as data structures. The execution unit is modeled in terms of hybrid automata, in which the motion of the AWL is given by a system of three ordinary differential equations:

$$\dot{x}(t) = v(t)cos\theta(t) \quad \dot{y}(t) = v(t)sin\theta(t) \tag{2}$$

$$\dot{\theta}(t) = \omega(t), \tag{3}$$

where, $\dot{x}(t)$ and $\dot{y}(t)$ are the projections of the linear velocity on x and y axes, $\omega(t)$ is the angular velocity, and $v(t)$ is the linear velocity, which follows the Newton's Law of Motion: $v(t) = \frac{F-k\times M}{M}$, where F is the force acting on the AWL, k is the friction coefficient, and M is the mass of the AWL.



(a) The skeleton of the pattern (b) The hybrid automaton of the pattern

Fig. 4. The pattern of the linear motion component in the execution unit

The pattern of the execution unit is a hybrid model consisting of two hybrid automata, namely linear motion and rotation. Here we use the linear motion component as an example to present the idea. As depicted in Figure 4(a), there are four locations indicating four moving states of the AWL, that is, stop at `Idle`, acceleration at `Acc`, moving at a constant speed at `Constant`, and deceleration at `Dec`. Therefore, the derivatives of the position (pcx', pcy') and the velocity (v') are assigned to zero at `Idle` for the stop state. According to different moving states, variations of equation 2 should be encoded in the refinement of each location in the blank boxes in 4(a). Figure 4(b) is an instance of the pattern, where v' is set to a positive value ($v' == (AF - k * m)/m$) at location `Acc` to present acceleration. Once the velocity reaches the maximum value ($maxS$) or the automaton receives a brake signal (denoted as a channel *brake*), it goes to location `Constant` or `Dec`, where the ODEs are changed to make the AWL move at a constant speed or decelerate.

5.2 Patterns for the Control Unit

As a part of an embedded system, the control unit model has three basic components: a scheduler, a piece of memory, and a set of processes. Currently, the memory is modeled as a set of global variables, hence the scheduler pattern and the processes patterns are the essence. Due to its safety-critical nature, the control unit is assumed to be a multi-core system and the processes are scheduled in a parallel, predictable, and non-preemptive fashion. This scheduling policy is inspired by *Timed Multitasking* [22], which tackles the real-time programming problem using an event-driven approach. However, instead of the preemptive scheduling, we apply a non-preemptive strategy. To illustrate this scheduling strategy, we use the three processes in the control unit (Figure 2) as an example. The process `ReadSensor` is firstly triggered at the moment $Trigger_1$ when the

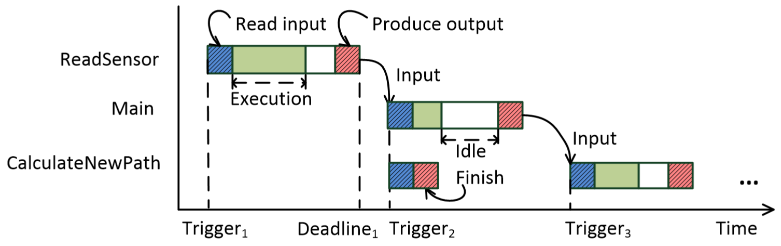


Fig. 5. Process scheduling

process reads data from sensors and runs its function as illustrated in Figure 5. Regardless of the exact execution time of a process, the inputs are consumed and the outputs are produced at well-defined time instances, namely trigger and deadline. As the input of `Main` is the output of `ReadSensor`, the former is triggered after the latter finishes. At same the moment, `CalculateNewPath` finishes its execution immediately as no input comes. This is actually reasonable, since process `CalculateNewPath` does not need to be executed every round, as it is responsible for generating a new path segment only when the AWL encounters an obstacle. For the benefits brought by the explicit execution time and deadline, we refer the interested readers to the literature [22] for detail.

The pattern of a process consists of two parts: a state module and an operation module. Similar to the state machine function-block and modal function-block in related work [19], the state module describes the mode transition structure of the processes, and the operation module describes the procedure or computation of the process. Because of their definition, the state modules are mod-

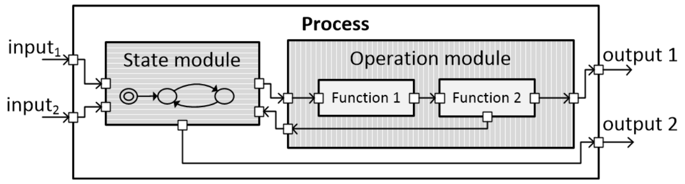


Fig. 6. A process model example

eled as discrete automata, and the operation modules are modeled as discrete automata or computation formulas according to their specific functionality. Figure 6 shows the inputs of the process coming to the state module in which the state of the process transfers according to the inputs. Some state transitions of the state module are detailed by the functions in the operation module in the sense that the former invokes the latter for concrete computation. Specifically, functions in the operation module could be modeled as discrete automata when they involve logic, or executable code when they are purely about computation. After executing the corresponding functions in the operation module, some results are sent out of the process as output, and some are sent back to the state module for state transitions, which might also produce output. The designs of the state module and operation module for different processes have both similarities and differences. They all need to be scheduled, to receive input, produce output, etc., but their specific functionality is different. To make our patterns reusable, we design fixed skeletons of the process patterns, which are presented as hybrid automata.

5.3 Encoding the Control Unit Patterns as Hybrid Automata

Scheduler. To model the scheduler as a hybrid automaton in UPPAAL SMC, we first discretize the continuous time as a set of basic time units to mimic the clock in an embedded system. As depicted in Figure 7, we use an invariant at location `Init` (clock `xd` $\leq$ `UNIT`), and a guard on its outgoing edge (`xd` == `UNIT`) to capture the coming basic time unit. We also declare a data structure representing processes, as follows:

```
typedef struct{
    int id; //process id
    bool running; // whether the process is being executed
    int period; //counter for the period of the process
    int executionTime; //counter for the execution time of the process
}PROCESS;
```

When a basic time unit comes, the scheduler transfers to location `Updating`. In the function `update()`, the period counters of all processes are decreased by one, and so are the execution time counters if the variable `running` in the process structure is true. When the `period` of a process equals zero, its `id` is inserted into a queue called `ready` and the variable `readyLen` indicating the length of the queue is increased by one. Similarly, when the `executionTime` equals zero, the process's id is inserted into a queue called `done`. The fact that the queue `done` is not empty (`doneLen` $>$ 0) implies that the execution times of some processes have elapsed, so the scheduler changes from `Updating` to `Finishing` to generate the outputs of those processes. The self loop at location `Finishing` indicates that the outputs of all the processes in queue `done` are generated orderly by the synchronization between the scheduler and the corresponding process automaton via the channel `output`. If the queue `ready` is not empty (`readyLen` $>$ 0), similarly, the scheduler moves to location `Execution` to trigger the top process in `ready` via the channel `execute`, and waits there until the process finishes, when the scheduler is then synchronized again with the process via channel `finish`. Note that the process finishes its function instantaneously and stores its output in the local variables, which will only be transferred to the other processes via global variables when the execution time passes.

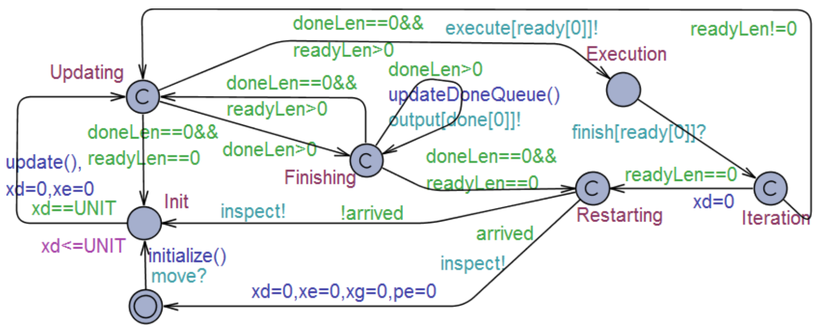
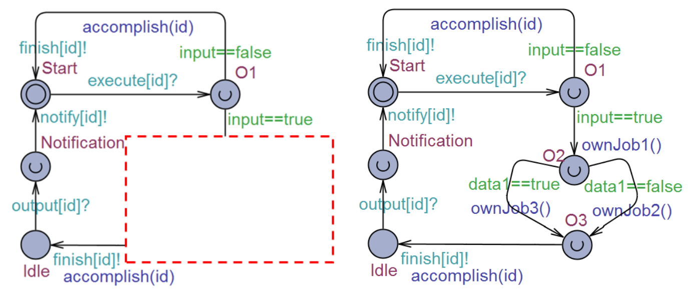


Fig. 7. The pattern of the scheduler

Process. A typical state module of a process consists of four states: being triggered, doing its own function, idle, and output. A typical pattern for it is shown in Figure 8(a). Except locations `Start` and `Idle`, all locations are urgent because the execution is instantaneous, and the output is generated when the execution time is finished. From location `Start` to `O1`, the process is being triggered by the scheduler by synchronizing on channel `execute[id]`, in which `id` is the process's ID. If the input is valid (input == true), the process starts to execute by leaving `O1` to the next location, otherwise, it finishes its execution immediately by going back to `Start` without any output generated, just as the description of the scheduling policy in Section 5.2. The blank box indicates the process's own function that is created in an ad-hoc fashion, so it is not part of the fixed skeleton of the pattern. After executing its own function, the process synchronizes again with the scheduler on channel `finish[id]`, when the process finishes and gives control back to the scheduler. The output is generated from location `Idle` to `Notification`. The broadcast channel `notify[id]` is for notifying other processes waiting for the output of the current process. Based on this idea, we give an example instantiated from this pattern in Figure 8(b). The automaton goes from `O2` to `O3` through two possible edges based on `data1`, which is the outcome of function `ownJob1()`. The concrete computation is encoded in functions `ownJob2()` and `ownJob3()`, which are the counterparts of the functions in the operation module of Figure 6. If the specific function of the process is more complex than in this example, or it includes function invocation, this blank box can be extended with synchronizations with other automata. We will elaborate this by revisiting our use case in the next section.

6 Use Case Revisited: Applying Our Method on AWL

As the patterns of linear motion and rotation components and the scheduler are totally applicable in the use case, they are simply transplanted in the model



(a) The skeleton of the pattern (b) An instance of the pattern

Fig. 8. The pattern of a generic process

of the AWL with parameter configuration. Hence, in this section, we mainly demonstrate how the processes in AWL's control unit are modeled using the proposed patterns, and present the verification results.

6.1 Formal Model of the Control Unit

The control unit contains three parallel processes (Figure 2). `ReadSensor` and `CalculateNewPath` are relatively simple because they do not invoke other functions, while `Main` calls function `Execution`, which calls other three functions: `AdjustAngle`, `Turn`, and `Arrive`. Therefore, The state modules of `ReadSensor` and `CalculateNewPath` are modeled as single automata and the operation modules are the functions at edges encoding the computation of their functionality. Differently, the state module of `Main` is a mutation of the process pattern extended with a preprocessing step calculating an initial path by running Theta* algorithm. Figure 9 depicts the automaton of the state module of `Main`, in which another automaton representing the function `Execution` is invoked via channel `invoke[0]`, where 0 is the ID of the function `Execution`. Note that the tran-

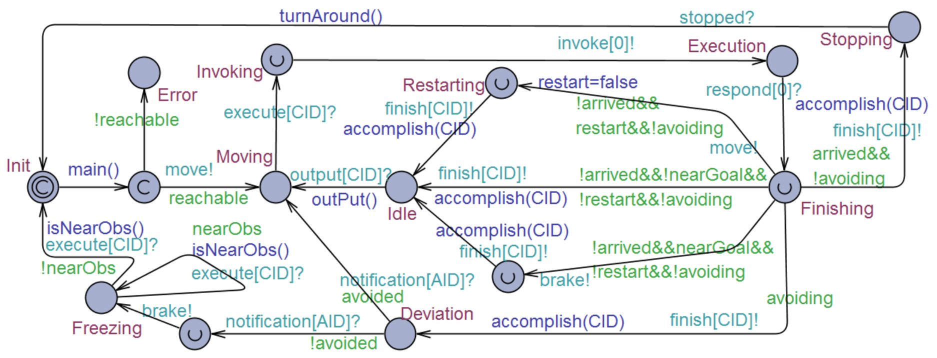


Fig. 9. The automaton of the state module of the process `Main`

sition from the location `Init` to `Moving` is the preprocessing step and Theta* algorithm is implemented in the function `main`, which will be moved to the static layer eventually after the entire framework is accomplished. As the process `Main` invokes other functions, its operation module is a network of automata containing the function `Execution`, `AdjustAngle`, `Turn`, and `Arrive`, which are called by using synchronizations between the state module automata and operation module automata (channels `invoke`, `respond`, `finish`). After calling other functions, `Main` goes to the location `Idle` via three edges based on the return values of the invoked functions and waits to generate output there.

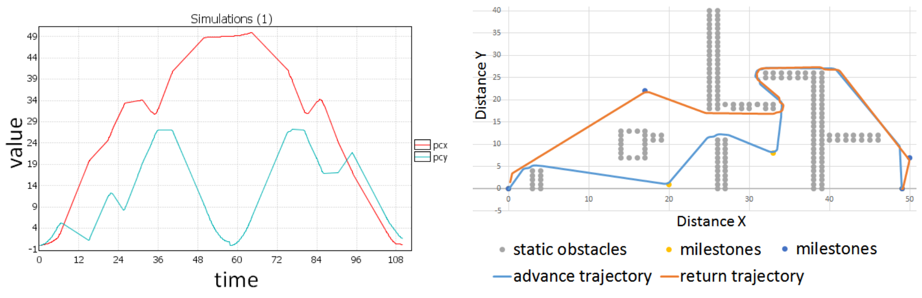
6.2 Statistical Model Checking of the AWL Formal Model

Environment configuration. In the following we consider a continuous map with the size 55×55, where five static obstacles and two moving obstacles are predefined, and another moving obstacle is dynamically generated during the verification. In order to achieve this, we leverage the spawning command of UPPAAL SMC to instantiate new time automata instance of the moving obstacle that "appears" in the map whenever it is generated by the automaton called `generator` and "disappears" from the map when its existence time terminates. The speed of the moving obstacles is a constant value indicating that they move one unit distance per second and their moving directions are either opposite or the same as it of the AWL. The parameters of the AWL are the weight of it, acceleration and deceleration force, friction coefficient and maximum speed, which are defined as constant values in UPPAAL SMC.

Path generation and following. Given a start and a goal and a set of milestones, the AWL must be able to calculate a safe path passing through them orderly avoiding static obstacles if the path exists and follow it. To verify this requirement, we first simulate the model in UPPAAL SMC using the command:

$$simulate\ 1[<= 110]\ \{pcx, pcy\} \tag{4}$$

where `pcx` and `pcy` are the real-valued coordinate of the AWL. Figure 10(a) shows the result of the simulation, and the result data is exported into Excel to depict the moving trajectory of the AWL shown in Figure 10(b). The AWL



(a) Coordinate changing of the AWL (b) Moving trajectory of the AWL in Excel

Fig. 10. Moving trajectory of the AWL generated by the command {`simulate 1[<=110] pcx,pcy`} in UPPAAL SMC and exported in Excel

perfectly follows the generated path that avoids all the static obstacles. But the simulation only runs one possible execution trace of the AWL model. Hence, we further verify the model with a query:

$$Pr[<= 70](<>\ arrived\ \&\&\ counter <= 60) \tag{5}$$

$$Pr[<= 110]([]\ followedPath) \tag{6}$$

where `arrived` and `counter` in query 5 are a Boolean variable and a clock that reflect if the AWL arrives at the destination and what the minimum time does it take, `followedPath` in query 6 is a Boolean variable indicating if the AWL has reached the destination and come back to the start by visiting all the milestones orderly. To update the value of `followedPath` timely and periodically during the verification, we create an independent automaton called `monitor` that checks the index of the model. The `monitor` is triggered by the `scheduler` every time unit that is small enough to ensure the position of the AWL does not change much during this time interval. The probability interval of satisfying these queries is [0.902606, 1] with 95% confidence obtained from 36 runs.

Collision avoidance. By the nature of the Theta* algorithm, AWL is able to avoid the static obstacles as long as it sticks to the initial path. When it meets an unforeseen static obstacle or a moving obstacle, the AWL must run the dipole flow field algorithm timely to avoid it. Two queries are designed to get the simulated moving trajectory and estimate the probability of satisfaction:

$$simulate\ 1[<= 110]\ \{pcx, pcy, ocx[0], ocy[0], ocx[1], ocy[1], ocx[3], ocy[3]\} \tag{7}$$

$$Pr[<= 110]([]\ !collided) \tag{8}$$

Arrays `ocx` and `ocy` in query 7 represent the positions of moving obstacles at x and y axes. The trajectories got from query 7 is shown in Figure 11, where "A" and "B" are two predefined moving obstacles and "C" is a dynamically generated obstacle that moves "recklessly" towards the AWL, so the latter turns around to avoid the obstacle. The overlap of two trajectories at "C" does not

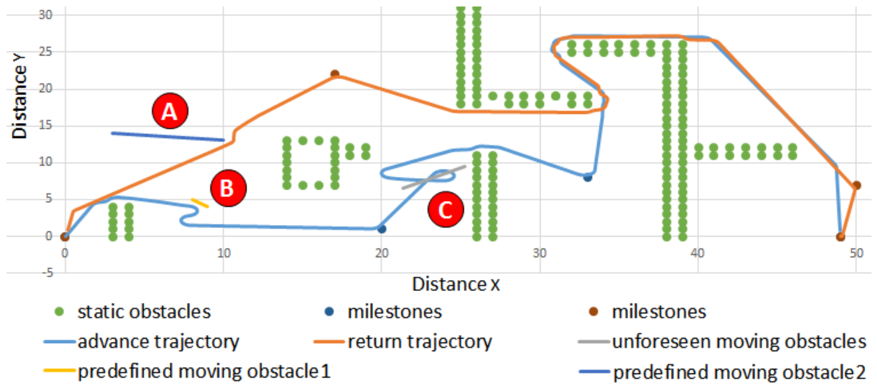


Fig. 11. The trajectory of the AWL in a map with three moving obstacles

imply a collision because the AWL and the moving obstacle are not at the same position at the same moment. To prove this, query 8 is designed, where `collided` is a Boolean variable indicating if the AWL has collided with any static or moving obstacles during the verification time. Similar to the verification of path generation and following, the automaton `monitor` is extended to update this variable periodically by checking if the current coordinate of the AWL is close to any obstacle in the map, and the threshold of the distance is 0.8 in this case. The probability interval of satisfying this query is [0.902606,1] with 95% confidence obtained from 36 runs.

7 Related Work

Automata-based methods [12][20][26][28] have been used for path or motion planning. Different from our work, these studies aim to solve the vehicle-routing problem by using temporal logic. These studies accomplish many typical autonomous tasks like searching for an object, avoiding an obstacle, and missions sequencing. However, as they focus on achieving collision avoidance in design, uncertainties in the real deployment like transmission time of sensors data in the embedded system and unforeseen obstacles have not been considered.

Runtime verification that monitors the behavior of autonomous systems complements this shortage to some extend [11][15][23][24]. This technique extracts information from a running system, based on which the behavior of the system is verified. Runtime overhead caused by the monitor is the most common problem introduced by this method.

Agent-based method is another widely studied approach for autonomous systems [3][8][11][13][14]. As the predominant form of rational agent architecture is that provided through the Beliefs, Desires, and Intentions (BDI) approach, these studies aim to translate the agent-based language to a formal language to verify the behavior of the agent. But this method usually does not concern the detail of the embedded control system and continuous dynamics of the vehicle.

There are also some studies providing a framework for verification of autonomous vehicles or robots. In [27], the authors captured the behavior of an unmanned aerial vehicle performing cooperative search mission into a Kripke model to verify it against the temporal properties expressed in Computation Tree Logic (CTL). Their model contains a decision making layer and a path planing layer. In [9], the authors propose an approach combining model checking with runtime verification to bridge the gap between software verification (discrete) and the actual execution of the software on a real robotic platform in the physical world. The software stack of a robotics system providing different verification capability focusing on different functionality has inspired our work. However, our framework provides an ability to encode the collision avoidance algorithm in the model and verifying it in a continuous environment.

8 Conclusions and future work

We have proposed a conceptual two-layer framework for formally verifying autonomous vehicles that decouples the high-level static planning from dynamic functions like collision avoidance, etc. The framework provides a separation of concerns for the complex modeling and verification of autonomous vehicles. The static layer focuses on making the optimal plan for the vehicle to accomplish a sequence of missions based on the incomplete information of the environment. While the dynamic layer concerns the execution of the plan with vehicle dynamics in a continuous environment model where unforeseen moving obstacles appear randomly. Hence, a collision avoidance algorithm relying on dipole flow field is implemented in the model of the embedded control system in this layer. We are currently engaged in modeling the dynamic layer using hybrid automata and UPPAAL SMC, and designing a pattern-based method to simplify the modeling process and increase reusability. The dynamic layer has been applied to model and verify a prototype of an autonomous wheel loader and the verification result shows the capability and applicability of statistical model checking adopted in autonomous vehicles. We expect to report our research of the static layer and the combination of these two layers in the years to come.

Acknowledgement: The research leading to the presented results has been performed within the research profile DPAC - Dependable Platform for Autonomous Systems and Control project, funded by grant 20150022 of the Swedish Knowledge Foundation that is gratefully acknowledged.

References

1. Bhatia, A., Maly, M.R., Kavraki, L.E., Vardi, M.Y.: Motion planning with complex goals. IEEE Robotics & Automation Magazine **18**(3), 55–64 (2011)
2. Black, P.E.: Manhattan distance. Dictionary of Algorithms and Data Structures **18**, 2012 (2006)
3. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying multi-agent programs by model checking. Autonomous agents and multi-agent systems **12**(2), 239–256 (2006)
4. Branicky, M.S., Borkar, V.S., Mitter, S.K.: A unified framework for hybrid control: Model and optimal control theory. IEEE transactions on automatic control **43**(1), 31–45 (1998)
5. Bulychev, P., David, A., Larsen, K.G., Legay, A., Li, G., Poulsen, D.B., Stainer, A.: Monitor-based statistical model checking for weighted metric temporal logic. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 168–182. Springer (2012)
6. Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research **39**, 533–579 (2010)
7. David, A., Du, D., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Statistical model checking for stochastic hybrid systems. arXiv preprint arXiv:1208.3856 (2012)
8. Dennis, L.A., Fisher, M., Webster, M.P., Bordini, R.H.: Model checking agent programming languages. Automated software engineering **19**(1), 5–63 (2012)
9. Desai, A., Dreossi, T., Seshia, S.A.: Combining model checking and runtime verification for safe robotics. In: International Conference on Runtime Verification. pp. 172–189. Springer (2017)
10. Desai, A., Saha, I., Yang, J., Qadeer, S., Seshia, S.A.: Drona: A framework for safe distributed mobile robotics. In: Proceedings of the 8th International Conference on Cyber-Physical Systems. pp. 239–248. ACM (2017)
11. Doherty, P., Kvarnström, J., Heintz, F.: A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. Autonomous Agents and Multi-Agent Systems **19**(3), 332–377 (2009)
12. Fainekos, G.E., Kress-Gazit, H., Pappas, G.J.: Temporal logic motion planning for mobile robots. In: Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on. pp. 2020–2025. IEEE (2005)

13. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational logics and agents: a road map of current technologies and future trends. Computational Intelligence **23**(1), 61–91 (2007)
14. Fisher, M., Dennis, L., Webster, M.: Verifying autonomous systems. Communications of the ACM **56**(9), 84–93 (2013)
15. Gat, E., Slack, M.G., Miller, D.P., Firby, R.J.: Path planning and execution monitoring for a planetary rover. In: Proceedings of the IEEE International Conference on Robotics and Automation. pp. 20–25 (1990)
16. Golan, Y., Edelman, S., Shapiro, A., Rimon, E.: Online robot navigation using continuously updated artificial temperature gradients. IEEE Robotics and Automation Letters **2**(3), 1280–1287 (2017)
17. 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)
18. Jafari, A., Nair, J.J.S., Baumgart, S., Sirjani, M.: Safe and efficient fleet operation for autonomous machines: an actor-based approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 423–426. ACM (2018)
19. Ke, X., Sierszecki, K., Angelov, C.: Comdes-ii: A component-based framework for generative development of distributed real-time control systems. In: null. pp. 199–208. IEEE (2007)
20. Kloetzer, M., Mahulea, C.: A petri net based approach for multi-robot path planning. Discrete Event Dynamic Systems **24**(4), 417–445 (2014)
21. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International journal on software tools for technology transfer **1**(1-2), 134–152 (1997)
22. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. Mit Press (2016)
23. Lotz, A., Steck, A., Schlegel, C.: Runtime monitoring of robotics software components: Increasing robustness of service robotic systems. In: Advanced Robotics (ICAR), 2011 15th International Conference on. pp. 285–290. IEEE (2011)
24. Luo, C., Wang, R., Jiang, Y., Yang, K., Guan, Y., Li, X., Shi, Z.: Runtime verification of robots collision avoidance case study. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). pp. 204–212. IEEE (2018)
25. Miloradović, B., Cürüklü, B., Ekström, M., Papadopoulos, A.: Extended colored traveling salesperson for modeling multi-agent mission planning problems. In: Proceedings of the 8th International Conference on Operations Research and Enterprise Systems - Volume 1: ICORES,. pp. 237–244. INSTICC, SciTePress (2019). https://doi.org/10.5220/0007309002370244
26. Quottrup, M.M., Bak, T., Zamanabadi, R.: Multi-robot planning: A timed automata approach. In: Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on. vol. 5, pp. 4417–4422. IEEE (2004)
27. Sirigineedi, G., Tsourdos, A., White, B.A., Zbikowski, R.: Modelling and verification of multiple uav mission using smv. arXiv preprint arXiv:1003.0381 (2010)
28. Smith, S.L., Tumova, J., Belta, C., Rus, D.: Optimal path planning for surveillance with temporal-logic constraints. The International Journal of Robotics Research **30**(14), 1695–1708 (2011)
29. Trinh, L.A., Ekström, M., Cürüklü, B.: Toward shared working space of human and robotic agents through dipole flow field for dependable path planning. Frontiers in neurorobotics **12** (2018)
30. Valbuena, L., Tanner, H.G.: Hybrid potential field based control of differential drive mobile robots. Journal of intelligent & robotic systems **68**(3-4), 307–322 (2012)