Formal Verification of an Autonomous Wheel Loader by Model Checking

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

Mälardalen University, Västerås, Sweden

(first.last)@mdh.se

ABSTRACT

In an attempt to increase productivity and the workers' safety, the construction industry is moving towards autonomous construction sites, where various construction machines operate without human intervention. In order to perform their tasks autonomously, the machines are equipped with different features, such as position localization, human and obstacle detection, collision avoidance, etc. Such systems are safety critical, and should operate autonomously with very high dependability (e.g., by meeting task deadlines, avoiding (fatal) accidents at all costs, etc.).

An Autonomous Wheel Loader is a machine that transports materials within the construction site without a human in the cab. To check the dependability of the loader, in this paper we provide a timed automata description of the vehicle's control system, including the abstracted path planning and collision avoidance algorithms used to navigate the loader, and we model check the encoding in UPPAAL, against various functional, timing and safety requirements. The complex nature of the navigation algorithms makes the loader's abstract modeling and the verification very challenging. Our work shows that exhaustive verification techniques can be applied early in the development of autonomous systems, to enable finding potential design errors that would incur increased costs if discovered later.

KEYWORDS

Autonomous Vehicle, Collision Avoidance, Formal Verification, Timed Automata, UPPAAL, Model Checking

1 INTRODUCTION

Industrial robots are used in modern manufacturing sites to automate repetitive tasks and reduce labor costs. Advances in self-driving vehicles have propelled similar developments in the construction industry, by the outset of autonomous construction equipment, which are heavy vehicles that operate without human intervention.

The environment where the autonomous construction equipment operates is hazardous, that is, dusty, with possibly harsh weather conditions, and populated with static and dynamic obstacles that need to be discovered and avoided by all means. These vehicles are designed to perform predefined tasks, and, unlike industrial robots, they operate in large construction sites, alongside other vehicles and humans. On the one hand, their environment is contained and controlled, thus their autonomy is bounded. On the other hand, being complex safety-critical systems, the autonomous construction equipment's dependability is crucial for ensuring safety and increased productivity, hence verifying formally an abstraction of the system's behavior could be highly beneficial. In this paper, we take upon such a task and formally model and verify an industrial prototype of an autonomous wheel loader against functional, timing, and safety requirements. The complexity of the system stems from the integrated intelligent algorithms, such as path planning, obstacle detection, and collision avoidance, etc. The crux of our work is the formalization of an abstraction of the vehicle's motions, control system, path-planning and collision-avoidance algorithms, such that resulting model is analyzable via exhaustive model checking. We use the timed automata (TA) [6] framework for modeling, and the UPPAAL [4] model checker for verification.

In comparison to related efforts of verifying autonomous vehicles [2, 3, 13, 20], our approach encodes the A* algorithm [19] for initial path planning, as well as the dipole flow field algorithm [23] used for avoiding static and dynamic obstacles, which are two algorithms that resolve many issues of implementing reliable collision avoidance effectively. Both algorithms are encoded as C functions in UPPAAL. To create the model of the machine's control system, we map the activity diagrams of components to TA representations. The system requirements, initially described in natural language, are formalized in Timed Computation Tree Logic (TCTL), as UPPAAL queries that the formal model needs to satisfy for any possible behavior. We show that under the mentioned abstractions, the exhaustive verification of the autonomous loader is possible, and we also discuss some identified issues of verifying a more faithful model.

This paper is organized as follows. In Section 2, we present the architecture of the autonomous wheel loader, as well as its natural language requirements. Section 3 overviews the preliminaries, that is, timed automata and UPPAAL, as well as the A* algorithm for the loader's initial path planning, and the dipole flow field algorithm for collision avoidance. In Section 4, we show the TA model of the loader's control tasks and algorithms, the verification queries and model checking results. A short discussion and lessons learned are provided in Section 5, after which we compare to related work in Section 6. Finally, Section 7 concludes the paper.

2 AUTONOMOUS WHEEL LOADER: ARCHITECTURE AND REQUIREMENTS

In this section, we introduce the Autonomous Wheel Loader (AWL), which is an industrial prototype and serves as our use case. The AWL is a heavy vehicle used in the construction site to transport materials (e.g., blasted rocks), which works independently, without any manual intervention. The AWL operates in a quarry (see Figure 1), where it transports rocks between a stone pile and a crusher. To be able to operate autonomously, the AWL is equipped with a path planning system that computes the initial path from the stone pile to crusher and back, which the AWL should follow. We assume

Figure 1: The AWL in its working environment

that there are various obstacles in the quarry, such as humans, other machines, holes, signs, etc. Other functions like autonomous digging, unloading etc. are not considered here. To ensure safety, the AWL is equipped with a collision avoidance system that identifies nearby objects, and deviates from the planned path (i.e., changes the direction and possibly the speed of the AWL), if needed, to avoid collision. This mechanism should cope with different light conditions (from bright sunlight to complete darkness), possibly bad weather (heavy rain or snow), dust, etc. To ensure it perceives its surroundings accurately, the AWL has a set of sensors, including GPS and IMU (Inertial Measurement Unit) for localization, and LIDAR, radar and camera for obstacles capture and identification.

The architecture of the AWL's control system, presented in Figure 2, consists of three main units: the vision unit, the control unit, and the execution unit, which are connected via Ethernet. The roles of these units are as follows:

- The vision unit is connected to the LIDAR and camera, and is responsible for detecting obstacles within the vision range.
- The control unit collects data (e.g., position of the AWL, obstacles, system status, etc.) from other units, plans the path, schedules the tasks, and sends commands to the execution unit.
- The execution unit controls the actuators, the steering and the brakes, based on the commands received from the control unit. It also collects data from the GPS and IMU, and sends them to the control unit.

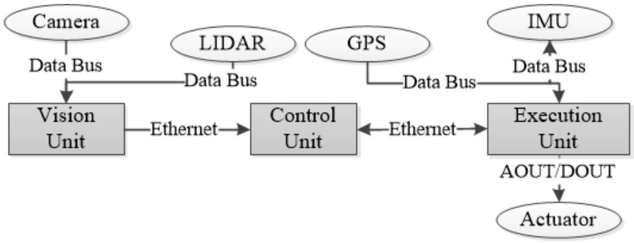


Figure 2: The architecture of the AWL's control system

AWL's Functionality. The functionality of the system is implemented through a set of tasks that are assigned and executed on the three units respectively, as depicted in Figure 3.

The obstacle detection relies on the Do Obstacle Task in the Vision Unit. This task is responsible for: (i) acquiring data from the sensors (e.g., LIDAR, camera), and (ii) executing the recognition algorithms to determine the presence and the type of the obstacles (e.g., human, other machines, holes).

The Control Unit executes three parallel tasks, described below:

- Read Position Task that reads the loader's position from the Execution Unit,
- Main Task that is responsible for generating the initial path, analyzing the environment, and devising control strategies to avoid different obstacles,
- Calculate Path Task that calculates a new path when the AWL encounters an obstacle and deviates from the initial path.

Three parallel tasks are assigned to the Execution Unit, namely Receive Command Task, Do Command Task, and Calculate Position Task. The tasks are responsible for getting commands from the Control Unit, executing the commands to move or brake the AWL, and calculating the position of the AWL and sending it to the Control Unit, respectively.

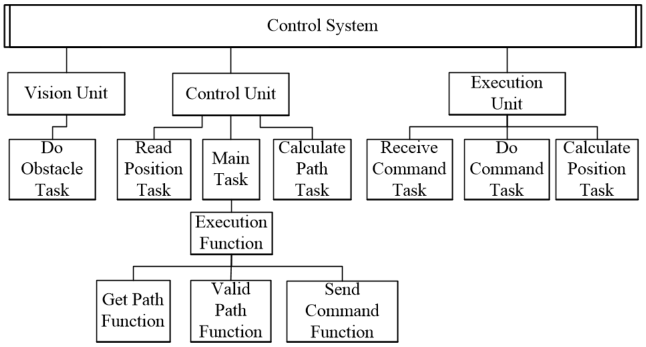


Figure 3: Task allocation in the control system

The communication among these tasks is asynchronous, that is, the tasks do not await response after they send out data. The tasks interact and cooperate with each other to accomplish specific missions of the control system, e.g., perceiving information from the environment, formulating an efficient (or close to optimal) path to avoid a dynamic obstacle, etc. Figure 4 depicts the partial interaction between tasks. Main Task takes one path segment of the initial path from Path Stack 1, which stores the initial path in the control unit. Next, it calls the Valid Path Function to check if the path segment leads to any collision. If the validation passes, the path segment is sent to Receive Command Task in the execution unit. Otherwise, the AWL might encounter an obstacle or malfunction, in which case Calculate Path Task will receive a new path request from the Main Task. Consequently, the corresponding algorithm employed for collision avoidance, called the dipole field algorithm [23], is executed in Calculate Path Task before a new path segment is sent to Receive Command Task, if it exists. If the calculation does not return any new path segment, Calculate Path Task will send a

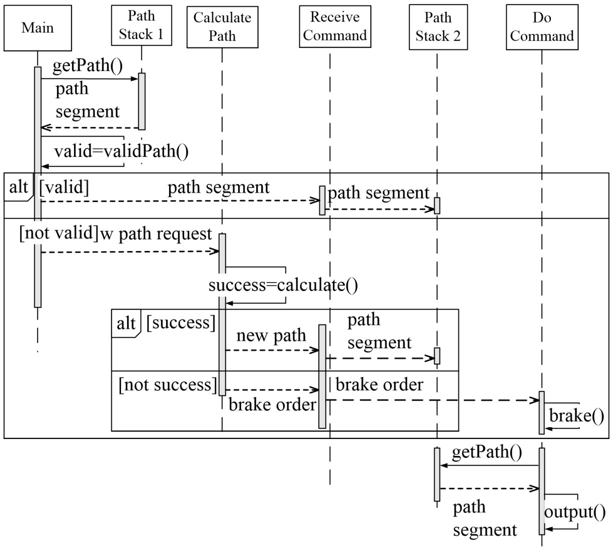


Figure 4: Sequence diagram of tasks in the control system

braking command to Receive Command Task, which then stores the path segment into Path Stack 2 where Do Command Task gets path segments. In the end, Do Command Task generates an output to the actuator, based on the commands.

System Requirements. The AWL has a large set of functional and extra-functional requirements. Below, we present some of these requirements, which are formally verified in this paper.

1) **Initial path computation**: during initialization, the AWL must compute an initial path to the destination, which must avoid all the static obstacles identified in the quarry;
2) **Obstacle avoidance and path recalculation**: the AWL must avoid static and dynamic objects around it in due time before returning to the initial path;
3) **Mode switch**: when a critical error occurs (e.g., an obstacle cannot be safely avoided or be reported to the control unit), the AWL must switch to the safety mode in order to freeze all motions within a certain time limit, to avoid further damage. In this case, the reaction time limits are error-specific;
4) **End-to-end deadline**: To guarantee a certain productivity, the AWL must reach the destination within 2200 milliseconds.

3 PRELIMINARIES

In this section, we overview the background information needed for the rest of the paper: timed automata and UPPAAL, as well as the A* and dipole flow field algorithms.

3.1 Timed Automata and UPPAAL

UPPAAL [6, 15] is a tool suite for modeling, simulation, and model checking of real-time systems. The modeling formalism of UPPAAL is an extension of *timed automata* (TA) [1], which is defined as the following tuple:

$$< L, l_0, A, V, C, E, I > \qquad (1)$$

where: L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $A = \Sigma \cup \tau$ is a set of *actions*, where Σ is a finite set of *synchronizing actions* and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization, V is a set of *data variables*, C is a set of *clocks*, $E \subseteq L \times B(C,V) \times A \times 2^C \times L$ is the set of *edges*, where $B(C,V)$ is the set of *guards* over C and V, that is, conjunctive formulas of clock constraints ($B(C)$), of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in C$, $n \in \mathbb{N}$, $\bowtie \in \{<, \leq, =, \geq, >\}$, and non-clock constraints over V ($B(V)$), and $I : L \longrightarrow B_{dc}(C)$ is a function that assigns *invariants* to locations, where $B_{dc}(C) \subseteq B(C)$ is the set of downward-closed clock constraints with $\bowtie \in \{<, \leq, =\}$. The invariants bound the time that can be spent in locations, hence ensuring progress of TA's execution. An edge from location l to location l' is denoted by $l \xrightarrow{g,a,r} l'$, where g is the guard of the edge, a is an update action, and r is the clock reset set, that is, the clocks that are set to 0 over the edge.

In UPPAAL, 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. Variables and clocks can be set to certain values by the updates along the edges. In UPPAAL, an update can be a comma-separated list of expressions, or a C-code style function that is implemented in the declaration of TA.

The semantics of TA is a *labeled transition system*. The states of the labeled transition system are pairs (l, u), where $l \in L$ is the current location, and $u \in R^C_{\geq 0}$ is the clock valuation in location l. The initial state is denoted by (l_0, u_0), where $\forall x \in C,\ u_0(x) = 0$. Let $u \vDash g$ denote that clock value u satisfies guard g. We use $u + d$ to denote the time elapse where all the clock values have increased by d, for $d \in \mathbb{R}_{\geq 0}$. There are two kinds of transitions $\longrightarrow$:

(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) Action transitions: $< l, u > \xrightarrow{a} < l', u' >$ if $l \xrightarrow{g,a,r} l', a \in \Sigma, u \vDash g$, 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 $u' \vDash I(l')$.

TA are composed into a *network of TA* over a common set of clocks and actions [4]. In this paper, we model the communication between TA via synchronization channels (e.g., a! and a?) with rendezvous semantics: a sender (a!) synchronizes with a receiver (a?), provided that the sending and receiving edges are enabled, that is, their guards are satisfied. The UPPAAL model checker supports the verification of queries written in a decidable subset of Timed Computation Tree Logic (TCTL) [4]. The syntax of a TCTL formula consists of quantifiers over paths and path-specific temporal operators. There are two types of path quantifiers: the universal one, "A" meaning "for all paths", and the existential one, "E" denoting "there exists a path". We are interested in two path-specific temporal operators, that is, "*Always*" ($\Box$) temporal operator meaning that a given formula is true in all states of a path, and the "*Eventually*" ($\Diamond$) operator meaning that a formula becomes true in finite time, in some state along a path. The UPPAAL queries that we verify in this paper are properties of the form: (i) **Invariance**: $A\Box p$ means that for all paths, for all states in each path, p is satisfied, (ii) **Liveness**: $A\Diamond p$ means that for all paths, p is satisfied by at least one state in each path, (iii) **Reachability**: $E\Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (iv) **Time-bounded Leads to**: $p \leadsto_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter; it is equivalent to the property: $A\,\Box\,(p \Rightarrow A\,\Diamond_{\leq t}\, q)$.

3.2 A* Algorithm

The A* algorithm is a widely used algorithm for path finding and graph traversal [19], and it was first introduced by Hart et al. [11]. In this paper, we use it to compute the initial path for AWL. It is an extension of Dijkstra's algorithm that uses a heuristic function to guide the graph traversal in order to achieve better performance. The basic idea of the A* algorithm is to find a lowest cost path from all possible paths to the destination, similar to Dijkstra's algorithm. While exploring the graph, the cost of the current node is calculated by the following function: $f(n) = g(n) + h(n)$, where n is the current node, $g(n)$ is the cost from the starting node to n, and $h(n)$ is the estimated cheapest cost from n to the destination. Intuitively, the A* algorithm aims to find the path that minimizes $f(n)$.

The pseudo code of the A* algorithm [16] is shown by Algorithm 1. It works in weighted graphs and constructs a tree of paths starting from a specific node of the graph, which is defined as the input.

Algorithm 1: A* Algorithm

```
   Input: Node $start$, Node $destination$
   Output: If the path is found or not
1  $closed := open := \emptyset$
2  $parent(start) := start$
3  $g(start) := 0$
4  $open.Insert(start, g(start) + h(start))$
5  while $open \neq \emptyset$ do
6      $current := open.top()$ /*return and remove the node with
         the minimum cost in open*/
7      if $current = destination$ then
8          return "arrived"
9      end
10     closed.Insert(current)
11     foreach $n \in neighbors(current)$ do
12         if $n \notin closed$ then
13             if $n \notin open$ then
14                 $g(n) := \infty$
15                 $parent(n) := NULL$
16             end
17             $g_{old} := g(n)$
18             if $g(current) + c(current, n) < g(n)$ then
19                 $parent(n) = current$
20                 $g(n) = g(current) + c(current, n)$
21             end
22             if $g(n) < g_{old}$ then
23                 if $n \in open$ then
24                     $open.Remove(n)$
25                 end
26                 $open.Insert(n, g(n) + h(n))$
27             end
28         end
29     end
30     return "no path found"
31 end
```

From line 1 to line 4, two arrays are initialized, that is, *open*: the set of currently discovered nodes that are not evaluated yet, and *closed*: the set of nodes that have been evaluated already. From lines 5 to 16, the main loop starts, in which the node with the minimum cost in *open* is selected. If this node is the destination, the calculation ends. Otherwise, the neighbors of this node and denoted by *n*, which are one-cell distance away around the node, are considered one by one as candidates to the *open* set, and evaluated in the rest of the code. From lines 17 to 21, the cost of node *n* is updated to the minimum and its parent node is changed accordingly. And between lines 22 to 26, the open set either updates the cost of node *n*, or inserts a new node *n* and its cost into the set.

3.3 Dipole Flow Field for Collision Avoidance

Modeling the paths of moving vehicles or other dynamic objects is not an easy task. Some studies have adopted the so-called *static flow field* and *dynamic dipole field* algorithms to represent the interactions of such moving objects [23]. In such scenarios, a vehicle moves within a certain area, called the map, and travels along a preset path that avoids the static obstacles, and approaches the destination. As soon as it discovers a moving obstacle within its vision range, the vehicle runs the collision avoidance algorithm to stay away from the obstacle as well as move towards the destination.

In this case, the static flow field force attracts the vehicle to its goal, ensuring that the vehicle avoids the static obstacles on the map. Meanwhile, as soon as a dynamic obstacle is encountered (be it another moving vehicle or a human), the dynamic dipole field algorithm generates forces that push the vehicle away from the dynamic obstacle, based on the latter's respective moving direction and velocity when within a close range from the original vehicle. The static flow field force is calculated by the following equation: $F_a = \frac{k_a q_0 Q}{D^2}$, $F_r = \frac{k_r q_0 q_1}{d^2}$, and $F_{\text{flow}} = F_a + F_r$, where F_a is the attractive force that draws the vehicle back to its initial path, F_r is the repulsive force from the nearby static obstacles, k_a, k_r, q_0, q_1, Q are coefficients whose values are problem specific, whereas D and d are the distances between the vehicle and its goal, and between the vehicle and the static obstacle, respectively. Unlike the dipole field forces, the attraction and repulsive forces always exist, regardless of whether the vehicle is moving or not.

In the theory of 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. Concretely, the repulsive force of a moving obstacle acting on the vehicle can be formulated as follows:

$$\vec{m} = k_m \vec{v} \tag{2}$$

$$\begin{aligned}\vec{F_d} = \frac{k_d}{d^5}[(\vec{m}_0 \cdot \vec{r}) \times \vec{m}_i + (\vec{m}_i \cdot \vec{r}) \times \vec{m}_0 + (\vec{m}_0 \cdot \vec{m}_i) \times \vec{r}] - \\ \frac{5 \cdot (\vec{m}_0 \cdot \vec{r}) \cdot (\vec{m}_i \cdot \vec{r})}{d^2} \times \vec{r}],\end{aligned} \tag{3}$$

where $\vec{r}$ is the distance vector between the two objects ($k_m, k_d \in R^+$). The combination of the static flow field and the dynamic dipole field ($F = F_{\text{flow}} + F_d$) guarantees that the vehicle moves safely by avoiding all detected obstacles, and reaches the destination eventually as long as the path is safe.

4 AWL'S MODELING AND VERIFICATION

In this section, we present the formal model of the AWL, as a network of TA, and the verification results after employing UPPAAL on the formal model. The model consists of three parts: the map, the AWL's movements, and the AWL's control system. Figure 5 depicts the verification methodology proposed in this paper. First, the map

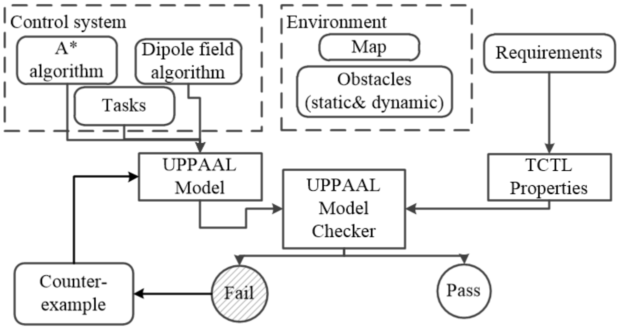


Figure 5: AWL's modeling and verification process

is modeled as a data structure. Next, the movements of dynamic obstacles and AWL, which include straight moving, turning and braking, are designed, assuming the actors are functionally correct in the given map. Then, we model the AWL's control system as a network of timed automata, in which tasks are TA that communicate via shared global variables and synchronize via channels. The UPPAAL model checker is then applied to verify whether the timed automata network satisfies the AWL requirements that are formalized as TCTL properties.

4.1 Map Abstraction

The loader's working environment consists of a map and several obstacles. The map is abstracted into a 2-dimensional Cartesian grid of disjoint cells with resolution $\epsilon \in \mathbb{R}_+$. As Figure 6 shows, the location of an object on the map is denoted by (x, y), with $x, y \in \mathbb{R}_{\geq 0}$.

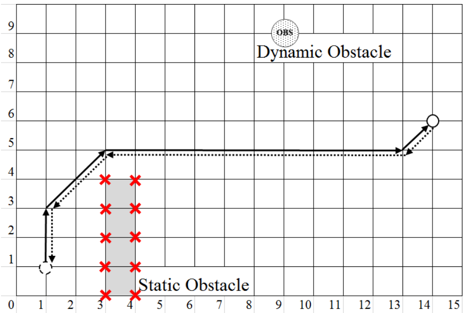


Figure 6: Abstraction of the map

The grid is encoded as (z_x, z_y), with $z_x, z_y \in \mathbb{Z}_{\geq 0}$. The mapping from reals to integers on two axes is given by the following:

$$f_1 : \mathbb{R}^2_+ \to \mathbb{Z}^2_+ \quad f(x, y) = (z_x, z_y)$$
$$\text{if } x - \frac{\epsilon}{2} \leqslant z_x \leqslant x + \frac{\epsilon}{2}, \text{ and } y - \frac{\epsilon}{2} \leqslant z_y \leqslant y + \frac{\epsilon}{2} \tag{4}$$

If an object (static or dynamic) is located at the intersection of x and y axes, the object's position is marked by × as shown in Figure 6. If the intersection is occupied, no other object can move to that point anymore. In our model, each intersection point is assigned 0 or 1, denoting that the point is empty or occupied, respectively. Furthermore, we assume that a dynamic obstacle occupies one point only, whereas a static obstacle can occupy more than one point as one can see in Figure 6. Based on this abstraction, the map is defined as a 2-dimensional array in UPPAAL, where each element represents a point on the map, and is assigned 0 or 1. A vertex is defined as a structure *Vertex* with two elements, integers x and y, representing the coordinates on x and y axes, respectively; the static obstacle is defined as a constant array of *Vertex*, representing the coordinates of each of its vertices (see Code 1).

Code 1: Vertex and static obstacle definitions

```
const int N = 15;
typedef struct
{
   int[0, N] x;
   int[0, N] y;
}Vertex;
const Vertex staticObstacle[10] =
     {{3,0},{3,1},{3,2},{3,3},{3,4},{4,4},{4,3},{4,2},{4,1},{4,0}};
```

4.2 Movements Abstraction

As the objects' locations are mapped onto the line intersections in the map, their movements are then restricted to the edges or the diagonals of the cells, as depicted in Figure 6. In our model, we separate the path into several path segments that are defined as pairs of vertices. A vertex is denoted by (z_x, z_y) as in formula (4), whereas v denotes the velocity of the AWL. Consequently, the path is defined as a sequence of path segments:

$$p = (z_{x_0}, z_{y_0})(z_{x_1}, z_{y_1}) \cdots (z_{x_{n-1}}, z_{y_{n-1}})(z_{x_n}, z_{y_n}) \tag{5}$$

$$\begin{cases} z_{x_i} = z_{x_{i-1}} \pm v, \text{ where } x_i \geq 1 \\ z_{y_i} = z_{y_{i-1}} \pm v, \text{ where } y_i \geq 1 \end{cases} \tag{6}$$

As mentioned previously, the AWL cannot occupy the vertices of a static obstacle, as shown in Figure 7.

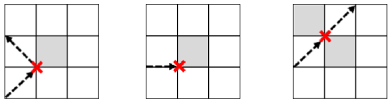


Figure 7: Three types of forbidden movements

When the loader starts to move, it accelerates from the minimum velocity (modeled as 0) to the maximum velocity (modeled as 2). The AWL stays at the current position for 2 time units at speed 0, then the duration decreases by 1 as the speed increases by 1, until it reaches the maximum velocity. The time unit is the execution period of *Main Task* in the control unit of the AWL.

We model the dynamic obstacle as a TA in UPPAAL, with a self-looping location that encodes the movements of the obstacle. The changing position of the obstacle is implemented by a function executed when the self-loop edge is traversed.

4.3 Formal Model of AWL's Control System

As shown in Figure 2, the control system consists of three units: vision unit, execution unit, and control unit. The vision unit acquires data from LIDAR and executes the recognition algorithms to identify the shapes, types, moving directions, etc., of the obstacles. However, in our model, we do not include this algorithm per se. Instead, our model checks the values of the map's points (0 or 1) to detect the obstacles present in the vicinity of the AWL, assumed as an area of maximum 3-cells distance from the AWL. In Figure 8, we see that object A cannot be detected as it is out of range, but object B is reported as an obstacle.

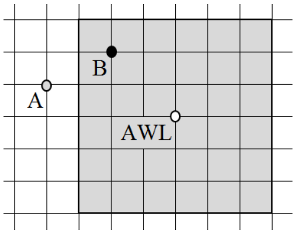
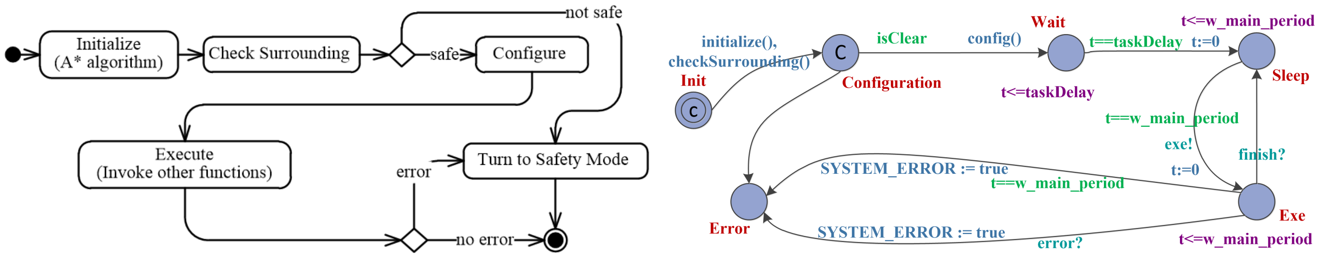


Figure 8: Detection range of the AWL

To model the AWL's control system, we create a TA for each task and function presented in Figure 3, whose procedures are captured by activity diagrams (e.g., Figure 9(a)) and sequence diagrams (e.g., Figure 4). We map the elements of such diagrams (e.g., decision nodes and action nodes in the activity diagrams) to our model, so that each TA's structure is respectively constructed. Even if the TA are manually created, we have used a 1-to-1 mapping in this process, described as follows:



(a) Activity diagram of Main Task

(b) Main Task TA

Figure 9: Modeling Main Task from the activity diagram

(1) For each action node of the activity diagram (except for those that call other functions), we create functions and corresponding locations and edges to ensure the same order of execution of the respective task, as in the original diagram.
(2) Decision nodes of the activity diagram are represented as locations with multiple outgoing edges, with edges enabled based on the associated guards.
(3) For action nodes that call other functions, we use synchronization among TA to model the invoking relation. This step is elaborated in the following examples.
(4) After the structure of the TA is constructed, we implement the A* and dipole field algorithms as C-code functions in the TA, respectively.

The final model contains 12 TA (i.e., 11 TA for the tasks and functions, 1 for the dynamic obstacle), and 61 C-code functions. Due to space limitation, we select to describe the respective behaviors and communication among Main Task, Calculate Path Task, and Get Path Function TA.

According to Main Task's activity diagram of Figure 9(a), the first two action nodes aim to initialize the system, check its surroundings, and run the A* algorithm. Hence, two locations, namely Init and Configuration, and the edge connecting them are created in the Main Task automaton shown in Figure 9(b). Along the edge, two functions, namely initialize() and checkSurrounding(), initialize the system's variables and check for obstacles around AWL, when executed. The A* algorithm is also executed in initialize() to generate the initial path. Next, two mutually exclusive outgoing edges from location Configuration are added, corresponding to the decision node in Figure 9(a) that follows the action node Check Surrounding, and indicating that if some error occurs during the initialization, the system moves to location Error to freeze the AWL. In case of no error, it moves to location Wait, where the task waits to be invoked. The other TA (e.g., Figure 10 and 11) are constructed by following similar steps.

After representing each individual task by a corresponding TA, the communication and scheduling of tasks need to be modeled. Every task has an execution period and is scheduled in a certain order. To achieve this, we add extra locations and invariants in the model, which are not corresponding to the action nodes and decision nodes of the activity diagram, such that some TA are executed periodically. The tasks for detecting obstacles and acquiring positions must be started earlier and executed more frequently than other tasks, such that the control system always makes decisions on the latest information. Hence, as it is shown in Figure 9(b), the automaton of Main Task, which awaits position information from other tasks, stays at location Wait until clock t reaches the value of taskDelay, which is 7 in this case. This delay enables Main Task to start later than the tasks using the system clock. To model the period of the task, after it moves to location Sleep, the automaton waits again until clock t reaches its task period, i.e., constant integer w_main_period, when it can be executed.

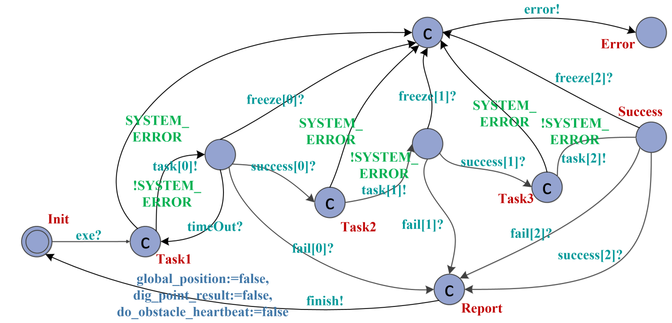


Figure 10: Timed automaton of Execution function

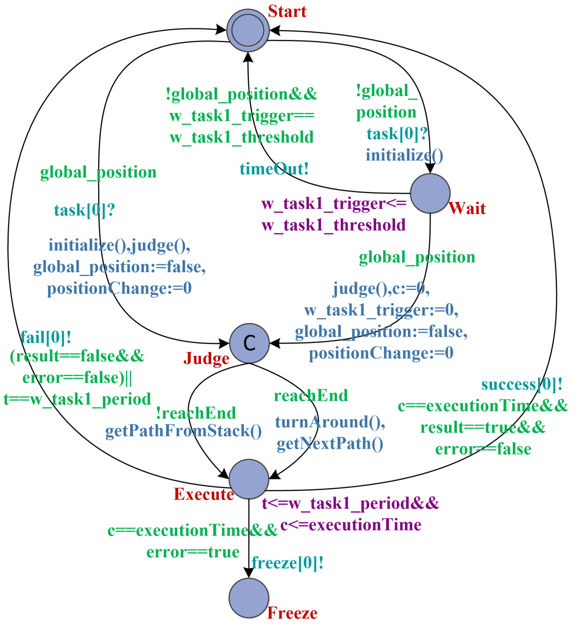


Figure 11: Timed automaton of Get-Path function

According to the sequence diagram of Figure 4, Main Task calls sub-functions. Hence, the automaton of Main Task is synchronized with Execution Function, via channel exe that decorates the edge connecting locations Wait and Exe. Then Main Task delays in location Exe until it synchronizes again with Execution Function via channel finish or error, indicating that the work is done or some error occurs.

The automaton Execution Function is also synchronized with other automata for the same reason. For instance, as shown in Figures 10 and 11, on channel task[0], the automaton Get Path Function is synchronized with Execution Function, indicating that Get Path Function is called by Execution Function. In addition, waiting for data from another task is modeled by locations and invariants added to Get Path Function automaton. For example, Figure 11 depicts that Get Path Function automaton waits for position data (global_position), in location Wait until clock w_task1_trigger reaches its limit w_task1_ threshold, when both w_task1_trigger is set to w_task1_threshold and variable global_position is set to true by other TA, indicating that the AWL's position has been acquired, or the variable global_position remains false until the invariant is violated; if the latter, the automaton moves back to the initial location Start, meaning that a time-out event occurs in Get Path Function.

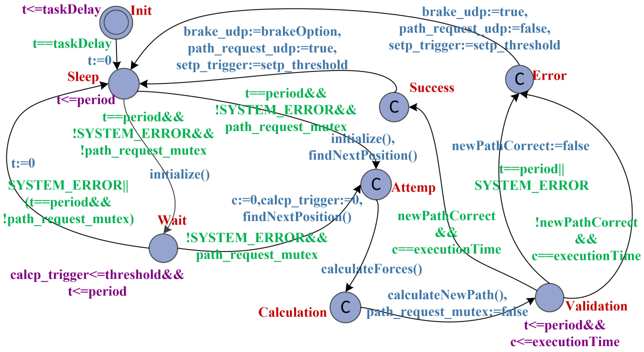


Figure 12: Timed automaton of Calculate Path task

The dipole field algorithm is implemented as C functions in the automaton Calculate Path Task, as shown in Figure 12. The task is executed in case an obstacle is detected, or AWL deviates from the initial path, so in the task's automaton, the first function being executed after *initialize()* is *findNextPosition()*, where the grid point in the initial path closest to the current position is returned as the new next position, which cannot be ensured to be safe. Hence, the forces applied on the AWL are computed in function *calculateForces()* based on the new position and equations described in Section 3.3. After that, a new path segment, if it exists, is calculated in function *calculateNewPath()* according to the field forces, which guarantees the safety of the AWL. The implementation is explained in detail in Section 5. If the new path segment does not exist (newPathCorrect==false), the automaton moves to location Error and sends out the brake command (brake_request_udp := true).

4.4 AWL's Model Verification

By applying the modeling process described in Sections 4.1, 4.2, and 4.3, we create the formal model of the AWL and its environment as a network of TA. As mentioned, the formal model consists of 12 TA (11 TA for the tasks and functions presented in Figure 3, plus one TA for the dynamic obstacle), four data structures (one for the map, two for the A* algorithm, and one for the path stack, which is used to store the path segments), 23 clocks, 49 global variables, 61 C-code functions, etc. To verify whether this model satisfies the informal requirements given in Section 2, we formalize the latter as TCTL queries that we check with UPPAAL. Two versions of the map are used in the verification. As depicted in Figure 13, we use a map with a static obstacle that occupies 10 grid points, and where the stone pile and crusher are located at (1,1) and (14,6), respectively. Next, as shown in Figure 14, we add one dynamic obstacle to this map, which starts at point (9,8) and moves along a predefined path.

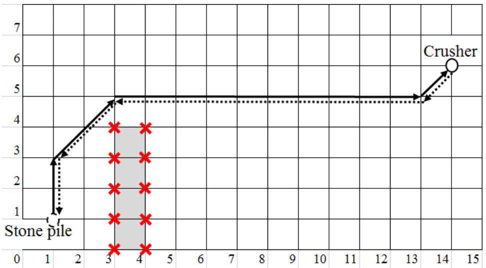


Figure 13: The AWL's trajectory on the map with a static obstacle

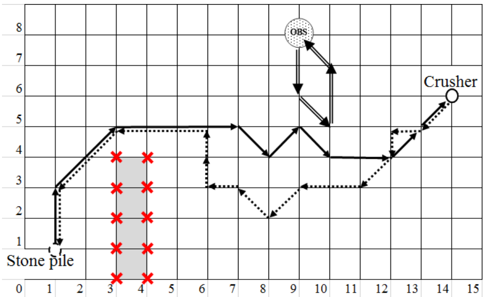


Figure 14: The AWL's trajectory on the map with a static and a dynamic obstacle

Table 1 presents the TCTL queries and the verification results. In the rest of this section, we describe these results.

Initial path computation. To verify that the AWL cruises between the stone pile and the crusher, and avoids the static obstacle, we use the map of Figure 13. Seven queries are specified to verify this requirement.

Queries Q1.0 and Q1.1 require that the initial path is calculated after the automaton *mainTask* moves to location *Wait* (mainTask.Wait). The integer variable *lenOfPathStack*, whose initial value is 0, is assigned by the length of the path stack, where the initial path is stored. Once the variable becomes greater than 0, the initial path is generated. Query Q1.3 states that, if the AWL is at the stone pile (currentPosition == pile) and its destination is the crusher (destination == crusher), the AWL will indeed eventually reach the crusher (currentPosition == crusher). Since UPPAAL's "leads to" operator ($p \rightsquigarrow q$) is equivalent to $A\square(p \Rightarrow A\lozenge q)$, we first check in query Q1.2 if the antecedent of Q1.3, that is, (currentPosition == pile and destination == crusher) is reachable. In our scenario, the AWL's initial location is the stone pile and the target is the crusher, thus query Q1.2 is satisfied by the initial state of the model and the verification explores only one state, the initial state. Similarly, in queries Q1.4 and Q1.5 we verify whether the AWL moves back from the crusher to the stone pile, whereas in query Q1.6 we check that

Table 1: Verification queries and results

Requirement	Query	Result	States explored	Time
Initial path computation	Q1.0: E<> mainTask.Wait	Pass	2	110 ms
	Q1.1: A<> mainTask.Wait imply lenOfPathStack > 0	Pass	8780	484 ms
	Q1.2: E<> currentPosition == pile and destination == crusher	Pass	1	0 ms
	Q1.3: (currentPosition == pile and destination == crusher) −> currentPosition == crusher	Pass	14191	1125 ms
	Q1.4: E<> currentPosition == crusher and destination == pile	Pass	2339	297 ms
	Q1.5: (currentPosition == crusher and destination == pile) −> currentPosition == pile	Pass	14204	782 ms
	Q1.6: A[] forall(i:int[0,9]) currentPosition != staticObstacle[i]	Pass	8780	485 ms
Obstacle avoidance	Q2.0: A[] currentPosition != currentObstacle	Pass	125941	6297 ms
	Q1.3: (currentPosition == pile and destination == crusher) −> currentPosition == crusher	Pass	227646	13969 ms
	Q1.4: E<> currentPosition == crusher and destination == pile	Pass	2678	375 ms
	Q1.5: (currentPosition == crusher and destination == pile) −> currentPosition == pile	Pass	192406	10656 ms
Mode switch: error A	Q3.1: E<> errorStart == true	Pass	30	234 ms
	Q3.2: error_start==true −> (SYSTEM_ERROR==true and reaction_time<=20)	Pass	91	250 ms
Mode switch: error B	Q3.1: E<> errorStart == true	Pass	29	234 ms
	Q3.2: error_start==true −> (SYSTEM_ERROR==true and reaction_time<=15)	Pass	320	266 ms
End-to-end deadline	Q4.0: (currentPosition==pile and destination==crusher) −> (currentPosition==pile and destination==pile and gClock <= 2200)	Pass	590326	36641 ms

the autonomous loader avoids the static obstacle. Concretely, query Q1.6 requires that the AWL must never occupy one of the grid points of the static obstacle (A □ forall(i:int[0,9]) currentPosition != staticObstacle[i]).

The combination of the first seven queries of Table 1 verifies that the autonomous loader cruises safely between the stone pile and crusher and back, without colliding with the static obstacle, thus showing that the AWL has the ability to compute a safe initial path. Furthermore, in order to visualize this path, we use the following queries:

E ◊ currentPosition == pile and destination == pile
E ◊ currentPosition == crusher and destination == crusher

These queries require that there exists at least one execution path in which the AWL eventually reaches the stone pile and the crusher, respectively. They are weaker than queries Q1.3 and Q1.5, but for these queries, the model checker generates a witness trace, which represents the initial path. The path presented in Figure 13 is generated in this way.

Obstacle avoidance and path recalculation. To verify this requirement, one dynamic obstacle (e.g., another vehicle) is added to the map, as shown in Figure 14. We assume that this object is not equipped with an obstacle avoidance feature, thus it does not change its path when it approaches the AWL. To verify this requirement, we need to check again that the AWL can reach the crusher and the stone pile, respectively, that is, queries Q1.2 to Q1.4, assuming the updated map. In query Q2.0, we check that AWL does not collide with the dynamic obstacle (A □ currentPosition != currentObstacle) for all possible execution paths of the model. As presented in Table 1, the number of states explored and the verification time for queries Q1.3, Q1.4 and Q1.5, in this case, are drastically increased as compared to the initial path computation case, since the dynamic obstacle increases the complexity of the model. As previously, we generate the path followed by AWL, which is depicted in Figure 14. The solid arrows represent the path to the crusher, the dashed arrows represent the way back to the stone pile, and the double-line arrows represent the preset path of the dynamic obstacle.

Mode switch. This requirement is verified on the map that contains the dynamic obstacle. To verify that AWL switches to safety mode and freezes its motion whenever it malfunctions, we introduce a global boolean variable SYSTEM_ERROR that "injects" errors into the model. For instance, in Figure 10, the TA moves to location Error whenever SYSTEM_ERROR becomes true. To check that this variable never evaluates to true, we use the following query:

A □ !SYSTEM_ERROR

As expected, this query is satisfied unless we inject faults into the model. In this paper, we model two faults that mimic real malfunctions:

- Error A: we set the boolean variable *do_obstacle_heartbeat* in the *Do Obstacle Task* to false, when it is sending a message and is resetting the clock *reaction_time* to zero at the same time, indicating that the information on obstacles cannot be reported to the control unit, which can be very dangerous.
- Error B: we set the boolean variable *position_udp* in the *Position Task* to false when it is sending the position information through the Ethernet, implying that the information is lost during the transmission.

For both these errors, two queries (Q3.1 and Q3.2) are formulated for verification. Query Q3.1 checks that there exists at least one execution path in which the error eventually happens (E ◊ errorStart == true). Formula Q3.2 requires that once an error occurs (SYSTEM_ERROR == true), the system must detect and react to the error within a certain time bound. This time bound is 20 time units for error A (reaction_time <= 20 in Q3.2 A), and 15 times units for error B (reaction_time <= 15 in Q3.2 B). The verification results show that, when error A or error B occur, the system moves to the SYSTEM_ERROR mode within the required time bound.

End-to-end deadline. The autonomous wheel loader must not only be able to travel to the crusher and then return to the stone pile position, but it also needs to accomplish this task within a certain time bound, which is its end-to-end deadline of 2200 time units. This requirement is verified by query Q4.0, which is a time-bounded leads to property, whose antecedent (that is, currentPosition ==

pile and destination == crusher) is the initial state of the model, and the consequent requires AWL to return to the stone pile before the deadline (currentPosition == pile and destination == pile and gClock <= 2200). Since Q4.0 is a leads-to property, we also need to verify that its antecedent is reachable, by proving Q1.2. Furthermore, by checking the query:

$$\text{E}\Diamond\ \text{currentPosition==pile and destination==pile},$$

we can request the model checker to generate the fastest diagnostic trace, which gives us the fastest time (1620 ms) to complete one cruise.

5 DISCUSSION

The issue with our abstraction of movements is that the shortest path that A* algorithm generates in the discrete area is not equivalent to the shortest path in the continuous area, because it constrains paths to be formed by the edges or diagonals of the cells. Some other path-planning algorithms, e.g., Theta* algorithm, overcome this drawback by changing the path to an any-angle path that does not necessarily follow the edges of the cells[16]. However, as traditional UPPAAL only supports integers, it is very difficult to implement algorithms like Theta* or dipole field as such, therefore they must be simplified. Hence, in our model, the forces in the dipole field algorithm that is employed by AWL for collision avoidance, are calculated using integers, based on the formulas in Section 3.3. Moreover, instead of using Newton's law of motion, which involves real numbers to calculate the loader's position, we use the sign of the combination of forces to decide the next position. Formula 7 shows the relation between the signs of forces and the AWL position, where (x',y') models the next position of AWL, (x,y) represents the current coordinates of AWL, F_x, F_y model the combination of attractive and repulsive forces on x axis and y axis, respectively, and T is the threshold for movements. This formula restricts the AWL to move only along the edges or diagonals of the cells, which is exactly our abstraction for movements.

$$(x', y') = \begin{cases} (x+1, y+1), \text{ if } F_x \geq T, F_y \geq T \\ (x+1, y), \text{ if } F_x \geq T, F_y < T \\ (x, y+1), \text{ if } F_x < T, F_y \geq T \\ (x, y), \text{ if } F_x < T, F_y < T \end{cases} \qquad T \in \mathbb{Z}^+ \qquad (7)$$

To fully implement the dipole field algorithm, a tool that fully supports floating point numbers is desirable. UPPAAL SMC (Statistical Model Checker) satisfies this requirement while still enjoying most of the useful features of UPPAAL [8]. With UPPAAL SMC, we can also model stochastic behaviors, e.g., the occurrence of dynamic obstacles, the reliability problem of Ethernet, etc. However, UPPAAL SMC does not provide exhaustive model checking, for it provides the probability of satisfying the queries.

We have also verified the AWL model in different scenarios, e.g., by letting the dynamic obstacle move arbitrarily within the map rather than along a preset path. It turns out to be very difficult to satisfy the requirements of reaching the destination while avoiding the obstacle under such circumstances. Two scenarios are generated by UPPAAL, where the AWL either collides with the dynamic obstacle or is stuck into a "livelock". As depicted in Figure 15, the AWL and the obstacle move back and forth on the same axis because there is no force on the other axis that turns the AWL at an angle to the obstacle. Therefore, the AWL consistently moves back and forth on this axis but never gets to the destination.

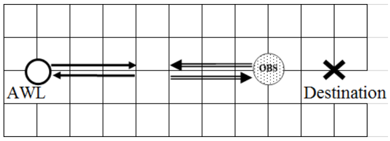


Figure 15: A livelock scenario

In another scenario the obstacle keeps "pushing" the AWL until both of them reach the edge of the map and stop, on grounds of our assumption that the dynamic obstacle does not avoid the AWL even though they come close to each other. One possible solution is to optimize the implementation of the dipole field algorithm so that the AWL can actively and angularly move towards the obstacle's moving direction, such that the AWL will bypass the obstacle from behind instead of being pushed away by the obstacle.

6 RELATED WORK

A number of formal methods have been applied to the verification of autonomous vehicles. Saberi et al. [20] propose using high-level languages, namely mCRL2 and Modal μ-calculus, for specifying and verifying multi-robot systems. Smith et al. [22] propose a method, based on weighted transition systems and Buchi automata, to find the optimal trajectory for the robot, which satisfies the requirements described in LTL. Koo et al. [14] propose a framework for the coordination of a network of autonomous robots with respect to formal requirements specifications in temporal logics, in which hybrid automata and Cadence's SMV model checker are used. Quottrup et al. [17] [2] design a high level abstraction of a multi-robot system using timed automata. Their model consider 4-directions movements of autonomous robots. Moreover, they also generate the shortest path with UPPAAL. Our research is inspired by such studies and provides modeling and verification of a complex control system of an autonomous wheel loader. However, instead of using a model checker to generate paths, we employ intelligent algorithms for path planning and collision avoidance (via dipole field) that we formally verify together with the control system.

There are also different approaches for modeling and analyzing the path-planning algorithms of autonomous vehicles. Fainekos et al. [9] apply temporal logic and model checking tools to generate discrete path plans that are later translated to continuous trajectories using hybrid control. The approach that they propose is built upon an existing framework [5] and proves that discrete plans and continuous trajectories are bisimilar, so that the satisfaction of LTL properties on the former is preserved by the latter. Kripke models and model checking techniques have been employed by Jeyaraman et al. in their study of modeling and verification of cooperative unmanned aerial vehicle (UAV) teams [12]. Rabiah et al. [18] use the Z specification language to formally specify the A* path planning algorithm, and verify the correctness of the algorithm by theorem proving. Saddem et al. [21] also use UPPAAL and CTL to verify reachability properties of autonomous behavior, including path finding. They propose a decomposition methodology to reduce the memory requirement and execution time of model checking. What makes our work different from the above studies is that we carry out a more extensive verification of the autonomous vehicle against a rich set of complex and realistic safety properties expressed in TCTL, e.g., whether the system can react to an error within a certain time limit. In addition, our formal model is more detailed, including the tasks in the control system and their communication, the algorithms, acceleration and deceleration of the vehicle. The conjunction of all these elements increases the size of model's state space dramatically, and hence the complexity of verification.

Some studies focus on the formal modeling and verification of the control logic or internal architecture of the automation system. Chouali et al. [7] propose an approach to model and ensure formally the reliability of automotive applications. They use SYSML to model the system before verifying the model described using interface automata. Hanisch et al. [24] [10] adopt the Net Condition/Event Systems (a modular extension of Petri nets) in their modeling and verification of several automated systems in intelligent manufacturing area. In comparison to these studies, our model includes not only the components in the control system but also the behaviors of the AWL and its environment, which allows us to simulate the model in a reactive mode, and our verification includes properties that are crucial for real-time automotive systems (e.g., end-to-end deadlines).

7 CONCLUSIONS

In this paper, we have presented the formal modeling and verification of an industrial prototype of an Autonomous Wheel Loader equipped with path planning and intelligent obstacle avoidance. Our modeling process maps the elements in activity diagrams to timed automata and implements the algorithms as C-code functions of the model, such that the model represents the entire control system of AWL. The (T)CTL queries used for verification completely express the informal requirements written in natural language, and provided by industry. The counter-examples that we have found during verification are helpful for the future optimization of the control system and the design of algorithms. Our model is the abstraction of the actual system, which serves to check correctness of the system at design level. Future work includes proving the correctness of the transformation from activity diagrams to UPPAAL TA and automating this process, modeling the dynamics of the AWL, injecting probabilistic events in the model in order to construct and verify a model that is closer to reality, etc.

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

REFERENCES

[1] Rajeev Alur and David Dill. 1991. The theory of timed automata. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*. Springer, 45–73.

[2] Michael S Andersen, Rune S Jensen, Thomas Bak, and Michael M Quottrup. 2004. Motion planning in multi-robot systems using timed automata. *IFAC Proceedings Volumes* 37, 8 (2004), 597–602.

[3] Adina Aniculaesei, Daniel Arnsberger, Falk Howar, and Andreas Rausch. 2016. Towards the Verification of Safety-critical Autonomous Systems in Dynamic Environments. *arXiv preprint arXiv:1612.04977* (2016).

[4] Gerd Behrmann, Alexandre David, and Kim G. Larsen. 2006. A Tutorial on Uppaal 4.0. (2006).

[5] Calin Belta and LCGJM Habets. 2004. Constructing decidable hybrid systems with velocity bounds. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, Vol. 1. IEEE, 467–472.

[6] Johan Bengtsson and Wang Yi. 2004. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science* 3098 (2004), 87–124.

[7] Samir Chouali, Azzedine Boukerche, and Ahmed Mostefaoui. 2017. Ensuring the Reliability of an Autonomous Vehicle: A Formal Approach based on Component Interaction Protocols. In *Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems*. ACM, 317–321.

[8] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. 2015. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer* 17, 4 (2015), 397–415.

[9] Georgios E Fainekos, Hadas Kress-Gazit, and George J Pappas. 2005. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2020–2025.

[10] Hans-Michael Hanisch, Andrei Lobov, Jose L Martinez Lastra, Reijo Tuokko, and Valeriy Vyatkin. 2006. Formal validation of intelligent-automated production systems: towards industrial applications. *International Journal of Manufacturing Technology and Management* 8, 1-3 (2006), 75–106.

[11] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[12] Suresh Jeyaraman, Antonios Tsourdos, Ratal Zbikowski, and Brian White. 2005. Formal techniques for the modelling and validation of a co-operating UAV team that uses Dubins set for path planning. In *American Control Conference, 2005. Proceedings of the 2005*. IEEE, 4690–4695.

[13] T.J. Koo, R.Q. Li, M.M. Quottrup, C.A. Clifton, R. Izadi-Zamanabadi, and T. Bak. 2012. A framework for multi-robot motion planning from temporal logic specifications. *Science China Information Sciences* 55, 7 (2012), 1675–1692. https://doi.org/10.1007/s11432-012-4605-8 cited By 4.

[14] T John Koo, Rongqing Li, Michael M Quottrup, Charles A Clifton, Roozbeh Izadi-Zamanabadi, and Thomas Bak. 2012. A framework for multi-robot motion planning from temporal logic specifications. *Science China Information Sciences* (2012), 1–18.

[15] Kim G Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a nutshell. *International journal on software tools for technology transfer* 1, 1-2 (1997), 134–152.

[16] Alex Nash and Sven Koenig. 2013. Any-angle path planning. *AI Magazine* 34, 4 (2013), 85–107.

[17] Michael Melholt Quottrup, Thomas Bak, and RI Zamanabadi. 2004. Multi-robot planning: A timed automata approach. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, Vol. 5. IEEE, 4417–4422.

[18] Eman Rabiah and Boumediene Belkhouche. 2016. Formal specification, refinement, and implementation of path planning. In *Innovations in Information Technology (IIT), 2016 12th International Conference on*. IEEE, 1–6.

[19] Steve Rabin. 2000. Game Programming Gems, chapter A* Aesthetic Optimizations. *Charles River Media* (2000).

[20] Arash Khabbaz Saberi, Jan Friso Groote, and Sarmen Keshishzadeh. 2013. Analysis of Path Planning Algorithms: a Formal Verification-based Approach.. In *ECAL*. 232–239.

[21] Rim Saddem, Olivier Naud, Karen Godary Dejean, and Didier Crestani. 2017. Decomposing the model-checking of mobile robotics actions on a grid. *IFAC-PapersOnLine* 50, 1 (2017), 11156–11162.

[22] S. L. Smith, J. TÅŕmovÃą, C. Belta, and D. Rus. 2010. Optimal path planning under temporal logic constraints. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 3288–3293. https://doi.org/10.1109/IROS.2010.5650896

[23] LanAnh Trinh, Mikael Ekström, and Baran Çürüklü. 2017. Dipole Flow Field for Dependable Path Planning of Multiple Agents. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. http://www.es.mdh.se/publications/4883-

[24] Valeriy Vyatkin and Hans-Michael Hanisch. 2003. Verification of distributed control systems in intelligent manufacturing. *Journal of Intelligent Manufacturing* 14, 1 (2003), 123–136.