TAMAA: UPPAAL-based Mission Planning for Autonomous Agents

Rong Gu, Eduard Enoiu, and Cristina Seceleanu
Mälardalen University, Västerås, Sweden
(first.last)@mdh.se

Abstract

Autonomous vehicles, such as construction machines, operate in hazardous environments, while being required to function at high productivity. To meet both safety and productivity, planning obstacle-avoiding routes in an efficient and effective manner is of primary importance, especially when relying on autonomous vehicles to safely perform their missions. This work explores the use of model checking for the automatic generation of mission plans for autonomous vehicles, which are guaranteed to meet certain functional and extra-functional requirements (e.g., timing). We propose modeling of autonomous vehicles as agents in timed automata together with monitors for supervising their behavior in time (e.g., battery level). We automate this approach by implementing it in a tool called TAMAA (Timed-Automata-based Planner for Multiple Autonomous Agents) and integrating it with a mission-management tool. We demonstrate the applicability of our approach on an industrial autonomous wheel loader use case.

Keywords

autonomous agents, mission planning, UPPAAL

1 Introduction

Autonomous vehicles [3] are complex systems that combine mechanical elements, electromechanical devices, digital circuits and software programs in embedded controllers. Their operation is subjected to many constraints, due to cluttered environments and objectives that can change over time. Mission planning is the process of determining what each autonomous vehicle should do to achieve the goals of the mission as described by the high-level system specifications. This includes autonomous path planning such that (static) obstacles are avoided, tasks assignment and scheduling, and re-planning in unforeseen circumstances. The challenge in this area is the development of modeling and verification frameworks [6, 16] able to accommodate the operating complexity of these systems, while allowing for the verification of their designs early in the development process. One way of ensuring the quality of mission design for autonomous vehicles is to employ model-checking for generating mission plans with guaranteed correctness. In this study, we propose such an approach to synthesize mission plans for Autonomous Wheel Loaders (AWL), which are part of an industrial use case provided by Volvo Construction Equipment (VCE). An AWL machine is designed to autonomously move and execute missions (like digging stones, loading and unloading) in quarries. An example of a complex mission plan for an AWL can be expressed as follows: *"Dig stones at the stone pile. Carry and unload them into a primary crusher 500 meters away. Avoid static obstacles and keep repeating these tasks until the stone pile is empty or the AWL needs to charge."* To model the AWL and specify this kind of requirement rigorously, synthesize mission plans and verify their execution formally, we adopt the two-layer framework approach for modeling and verifying autonomous agents' operations, proposed in our previous work. This framework consists of a static layer and a dynamic layer. The static layer focuses on path planning and task scheduling and the dynamic layer focuses on modeling the kinematics and dynamics of the agents to verify if they can accomplish the tasks and circumvent risks, such as moving or unforeseen obstacles. The contribution of this paper targets only the design of the static layer of such a framework, which provides rigorous algorithms for model generation and a user-friendly way for model configuration.

Specifically, we use Timed Automata [2] and Timed Computation Tree Logic (TCTL) [8] for capturing formally the AWL's behavior and requirements specification, respectively. Formal definitions of the concepts, e.g., tasks, are given for formal analysis and synthesizing solutions. Based on the definitions, we propose model-generation algorithms that we integrate with an advanced path-planning algorithm (Theta*) to generate formal models automatically. For simplicity, we use mission plans to denote path-and-task plans in this paper. The formal models are built in order to be able to synthesize mission plans satisfying requirements like the one we aforementioned. These requirements often concern three aspects: i) **safety**: all obstacles of the generated paths should be avoided, ii) **execution constraints**: tasks should be executed with respect to given logical and temporal constraints, iii) **timeliness**: the final goal should be achieved within a certain amount of time for productivity reasons. Given such a mixed palette of requirements, it is not trivial to generate automatically mission plans that will guarantee all of them. Moreover, it is desirable that such synthesis of plans is supported by an easy-to-use tool, in which the user can visualize

and modify the mission plans. Hence, in this work, we propose a method supported by a tool, called Timed-Automata-based Planner for Multiple Autonomous Agents (TAMAA). Our approach integrates the state-of-the-art model checker UPPAAL [5] with a toolkit for mission configuration called MMT (Mission Management Tool) [14]. TAMAA implements the model-generation algorithms and provides a graphic interface to configure the environment, agents, and tasks and organizes the information to build formal models, including the movement of agents, task execution, and monitors. Next, within TAMAA, one can verify the generated model with UPPAAL, against the TCTL queries that formalize the natural-language requirements and generate diagnostic traces. The traces are parsed by TAMAA to synthesize mission plans. Eventually, the synthesis result is shown in MMT. If there is a valid path, it is guaranteed to be correct and optimal in the sense that it is generated via exhaustive model checking. If no valid path exists, a counter-example is depicted to illustrate the contradictions in the model configuration. We demonstrate the applicability and scalability of TAMAA by applying it to scenarios of an industrial use case.

The nolvety of TAMAA is that it addresses not only path generation but also takes into account complex requirements (functional and timing ones). Moreover, our solution combines a rigorous, formal encoding of algorithms for computation with a user-friendly interface for visualizing model configurations. Model-generation algorithms provide a systematic and automatic way for obtaining formal models and properties from industrial requirements, which is less time-consuming and error-prone. The remainder of the paper is organized as follows. In Section 2 we introduce the preliminaries of this paper. Section 3 describes the actual contribution, that is, TAMAA, whereas in Section 4 we introduce the implementation and evaluation of TAMAA. In Section 5 we compare to relate work, before concluding and outlining possible future work in Section 6.

2 Preliminaries

In this section, we briefly overview the background information related to timed automata and UPPAAL model checker, which we employ in TAMAA.

2.1 UPPAAL Timed Automata

A *timed automaton* (TA) is an extended finite-state automaton suitable for modeling real-time systems [1]. UPPAAL [5] is a tool for modeling, simulation, and model checking of real-time systems, and uses an extension of TA as the modeling formalism [8]. A UPPAAL timed automaton is defined as a tuple: $< L, l_0, A, V, C, E, I >$, where L is a finite set of *locations*, $l_0 \in L$ is the *initial location*, $A = \Sigma \bigcup \tau$ is a set of *actions*, where Σ is a finite set of *synchronizing actions* and $\tau \notin \Sigma$ are internal actions, V is a set of *data variables*, C is a set of real-valued variables called *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 $B(V)$, and $I : L \mapsto B_{dc}(C)$ is a function assigning *invariants* to locations where $B_{dc}(C) \subseteq B(C)$ denotes a subset of clock constraints resulting from the restriction to upper bounds $\triangleleft \in \{<, \leq\}$.

The semantics of a TA is given by 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 $u \models g$ 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 $d \in \mathbb{R}_{\geq 0}$. The following transitions ($\rightarrow$) can happen in a timed automaton:

- Delay transitions: $< l, u > \xrightarrow{d} < l, u + d >$ if $u \models I(l)$ and $(u + d') \models I(l)$, for $0 \leq d' \leq d$, and
- Action transitions: $< l, u > \xrightarrow{a} < l', u' >$ if $l \xrightarrow{g,a,r} l', a \in \Sigma, u \models g$, 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 $u' \models I(l')$.

A network of TA, $B_0 \parallel \ldots \parallel B_{n-1}$, is a parallel composition of n TA over C, A and synchronization channels (i.e., $a!$ is synchronized with $a?$ by handshake). We refer the reader to literature [1] for more information on the theory of TA.

UPPAAL uses a decidable subset of (Timed) Computation Tree Logic [8] as the query language. It consists of path formulae and state formulae. Specifically, we use the following path-specific temporal operators: "Always" ($\Box$) temporal operator for which a given formula is true in all states of a path, and the "Eventually" ($\Diamond$) operator used to show that a formula becomes true in finite time, in some state along a path. In this paper, we use queries of the following categories: (i) *Invariance* (i.e., $A \Box p$), stating that p should be true in all reachable states for all paths, and (ii) *Reachability* (i.e, $E \Diamond p$), stating that there exists a path starting at the initial state, such that p is eventually satisfied along that path.

3 TAMAA Approach

In this section, we describe an approach to automatically synthesize mission plans for autonomous agents. We first describe the function and architecture of an industrial use case, the Autonomous Wheel Loader (AWL), in Section 3.1, which motivates the design of TAMAA. Next, in Section 3.2, we introduce the components and workflow of TAMAA, followed by formal definitions of autonomous agents, their movement, tasks and their execution, which are all needed for the automatic model generation. Last but not least, we describe the model-generation algorithms in Section 3.4.

3.1 Use Case: Autonomous Wheel Loader

In this section, we introduce our use case, which is based on an industrial system provided by Volvo Construction Equipment (Sweden). The use case contains Autonomous Wheel Loaders (AWL) that are used in construction sites to perform operations without human intervention. For example, as shown in Figure 1, we consider the case of AWL that are utilized to transport materials in a quarry site. According to the provided requirements, an AWL digs a given stone pile and carries an amount of stones in its bucket before it moves to the primary crusher and unloads the stones onto the conveyor belt. After this first step, the AWL moves to

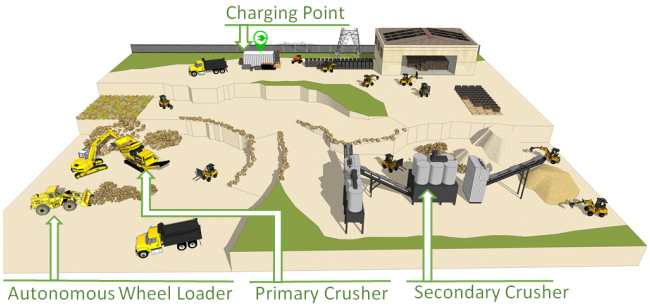


Figure 1: An example of a working environment for an autonomous wheel loader

the other end of the primary crusher and loads the crushed stones. It then continues moving to the secondary crusher to unload the stones and finishes its one-round job. During this process, the AWL carries out its tasks autonomously and moves to the charging point when its battery level is low. The AWL has to also avoid static obstacles (e.g, holes and rocks on the ground). The problem involves mission planning, path following, and collision avoidance.

In this paper, we focus on generating valid paths for autonomous vehicles, guaranteed to avoid static obstacles, as well as correct schedules for the operational tasks of the machines. We assume a two-layer approach of the design, as proposed in our previous work, with mission planning belonging to the static layer, while the avoidance of dynamic obstacles, including the case of overlapping paths of multiple vehicles working on the site, is being dealt with in the dynamic layer. We assume that the latter functions correctly, and we focus only on synthesizing mission plans for our autonomous machines.

Intuitively, the mission-planning problem requires the AWL to: (i) generate a path plan that includes visiting (in the right order) the milestones where the loader needs to stop to carry out a given operation, (ii) avoid all the static obstacles on the way, and (iii) guarantee to execute certain operations at particular milestones. Specifically, the requirements provided by our industrial partner can be divided into the following categories:

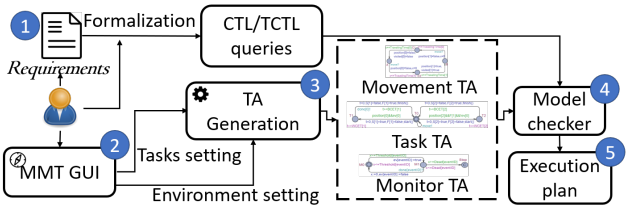


Figure 2: Overview of the process of model generation and mission plan synthesis in TAMAA

- *Task Coverage.* The AWL must execute all tasks and repeat them until the ultimate goal is achieved (e.g., all stones are transferred to the secondary crusher).
- *Task Matching.* The AWL must accomplish a certain task at each particular milestone (e.g., digging is only allowed at stone piles).
- *Task Sequencing.* The order of tasks execution must be correct.
- *Timing.* The AWL must finish the tasks within a prescribed time, to keep the desired level of productivity (e.g., the AWL must complete digging and carrying a ton of stones in 0.5 hours).
- *Event Reaction.* Some special tasks are only triggered by events under certain circumstances. For instance, when the battery level is below a certain level, the AWL must move to the charging point to charge itself.

Overall Challenge. *Given an environment containing one or several AWL with accurate speed control and a deterministic speed range, predefined milestones and static obstacles, and a set of requirements (e.g., task coverage, task matching and sequencing, timing, and reacting to events), we need to synthesize mission plans for these AWL in this environment, such that the requirements are satisfied.*

3.2 Workflow of TAMAA

Given this challenge, we propose a method called TAMAA (Timed-Automata-based Planner for Multiple Autonomous Agents) for making the optimal plan for the AWL to accomplish a sequence of tasks based on a set of given requirements. Overall, the approach is composed of the steps shown in Figure 2. i) Step 1 - formalizing the requirement into CTL/TCTL queries, ii) Step 2 - configuring the information of the environment and tasks in MMT, iii) Step 3 - automatically generating environment and tasks models as UPPAAL TA, iv) Step 4 - verifying models of Step 3 in UPPAAL against the requirements in Step 1, and generating execution traces, and v) Step 5 - using the traces to obtain the mission plans. Since this is an automatic approach, users are only involved in the first two steps in the configuration phase. All steps are described in detail in the following sections.

3.3 Model Formalization and Definitions of Concepts

In this section we define formally the elements of TAMAA, that is: an autonomous agent, its movement, and the notion of an autonomous task. To illustrate the formal definitions and algorithms, we use a running example extracted from our use case through out this section.

Running example. *As depicted in Figure 3(a), the AWL starts from A, goes to the stone pile at B and digs stones and moves to the crusher at C to unload stones and comes back eventually.*

The AWL can be considered as an autonomous agent that is situated within an environment, it can sense the environment and act on it, over time, in pursuit of its own goals [12]. In this paper we focus on mission planning of autonomous agents, whose movement and tasks are simply abstracted as time duration without considering any real-time feedback from the environment. Therefore, we assume that autonomous agents can be considered automated agents at this level of abstraction and defined as follows: *An automated agent is a system that receives instructions from its mission plan and executes its instructions with no human control and no interaction with its environment.* There are many definitions of what automated agents are according to their use in different fields of research [12]. In this paper we assume the definition above, and we formalize an automated agent (that actually fits our AWL use case) as follows.

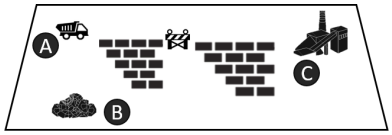
Definition 1 (Automated Agent). *An automated agent (AA) is defined as a tuple:*

$$AA \triangleq < \mathcal{S}, \mathcal{M}, \mathcal{T} > \quad (1)$$

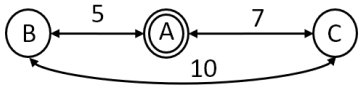
where,

- $\mathcal{S}$ *is the speed of the moving vehicle,*
- $\mathcal{M}$ *is a set of motion primitives that make the agent move and execute tasks,*
- $\mathcal{T}$ *is a set of tasks that the agent has to accomplish.* □

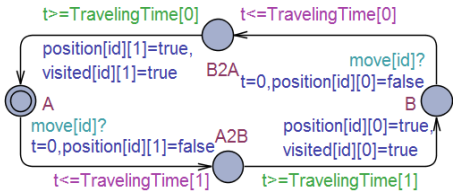
The working environment of an agent is a closed space including some static obstacles that the agent should avoid, and some milestones where the tasks should be carried out. However, when an agent is reaching a milestone, it does not necessarily stop. According to the mission plan, the agent can stop and execute the corresponding task or simply pass. Static obstacles and milestones are represented as a set of X-Y coordinates in the environment. The working environment of an agent is defined as a weighted graph $G = (V_g, E_g)$, where V_g is a set of vertices denoting the milestones, $E_g \subseteq V_g \times N_g \times V_g$ is a set of edges, where $N_g \subseteq \mathbb{R}_{\geq 0}$ denotes a set of traveling times between vertices. Edges only connect the vertices that are directly reachable from each other, which means the shortest path between two connected vertices does not pass any other vertices.



(a) An example extracted from the use case



(b) The weighted graph modeling the example



(c) TA encoding of an Agent's movement in the example

Figure 3: A running example and its corresponding weighted graph and TA

We assume that automated agents are equipped with a set of motion primitives that allow them to deterministically move from v to v' for each $(v, t, v') \in E_g$, with $v, v' \in V_g$. Hence, the traveling time t between two vertices is constant, and it is calculated by graph-search algorithms such as Theta* algorithm [11]. The weighted graph extracted from the example of Figure 3(a) is depicted in Figure 3(b).

When the agent starts to move, changing the position depends on the connectivity of vertices and the traveling time between them. Hence, the movement of an agent involves discrete changes of position and the continuous evolution of time, which makes TA a suitable formalism for modeling the movement of agents.

Definition 2 (Movement of AA). *The movement of an automated agent is defined as a timed automaton in a restricted form:*

$$Mm \triangleq < P, p_0, x_m, A_m, E_m, I_m > \quad (2)$$

where,

- $P = P_v \cup P_e$ *is a finite set of locations, where* P_v *denotes the vertices of the weighted graph of the environment, and* P_e *denotes the locations of components that represent the transitions between vertices. The component consists of an incoming edge, a location in* P_e *and an outgoing edge;*
- $p_0 \in P_v$ *is the initial location denoting the initial position;*
- x_m *is a clock variable defined to measure the traveling time;*
- $A_m = \{move\} \cup \tau$ *is a set of actions, where "move" is for synchronizing with the automaton encoding the agent's tasks, and* $\tau \notin \Sigma$ *denotes internal or empty actions without synchronization;*
- $E_m \subseteq P_v \times A_m \times B_m(x_m) \times 2^C \times P_e$ *is a set of edges, where* $B_m(x_m)$ *is a set of guards contains clock constraints of the*

form $x_m \geq \Upsilon$, where $\Upsilon \in \mathbb{R}_{\geq 0}$ is a constant value of the traveling time between two locations, and $C = \{x_m\}$;
- *$I_m : P_e \mapsto B_e(x_m)$ is a function that assigns invariants to locations in P_e, where $B_e(x_m)$ contains clock constraints of the form $x_m \leq \Upsilon$.* □

Based on Definition 2, a part of the TA modeling the movement of agents is depicted in Figure 3(c), where the agent moves from A to B and and vice versa. Locations A2B and B2A belonging to P_e and the their associated invariants, respectively, are created to model the duration of traveling.

Any automated agent should carry out tasks that can be an operation (e.g., loading, digging) or simply a state of stop and wait. A task is allowed to be carried out only at certain predefined positions, with an execution time given as an interval. For example, an AWL only unloads rocks at a primary crusher or a secondary crusher. Some tasks, like charging, are triggered in special circumstances, but once they are triggered they must be prioritized. Given an agent $(\mathcal{S}, \mathcal{M}, \mathcal{T})$ and a set of events Ev triggering $T_i \in \mathcal{T}$, one needs to formally capture the agent's tasks and their execution, which we introduce by the following Definitions 3 and 4, respectively.

Definition 3 (AA Task). *A task is defined as a tuple:*

$$Task \triangleq (B, W, \Delta, S, F, R, O, M, V, G) \tag{3}$$

where,
- *B is the best case execution time,*
- *W is the worst case execution time,*
- *Δ is the time that has elapsed during the execution of a task,*
- *S is a Boolean variable denoting if the task has started,*
- *F is a Boolean variable denoting if the task has finished,*
- *R is a precondition that must be satisfied before the task starts,*
- *O is a postcondition that must be satisfied after the task finishes,*
- *M is a set of indices of milestones where the task is allowed to be executed,*
- *V is a set of variables that are changed after the task finishes,*
- *G is a set of Boolean variables (events) that trigger the task.* □

To simplify the notation, T_i is used to denote a task for any i and we use "." to access an element in a tuple. $T_i.B$ and $T_i.W$ are different for different tasks and agents, and $T_i.\Delta$ is designed to measure the total execution time of a task, so $B \leq \Delta \leq W$. The precondition is $T_i.R = \theta_t(T_0.F, ..., T_k.F) \wedge \theta_e(ev_0, ..., ev_m)$, where θ_t and θ_e are predicates reflecting the execution order of tasks $\{T_0, ..., T_k\} \subseteq \mathcal{T} \setminus T_i$, and the status of events $\{ev_0, ..., ev_m\} = Ev$. The postcondition is $T_i.O = \bigwedge_{i=1}^{n} \neg ev_i \wedge T_i.F$, where $ev_i \in T_i.G$ and $n = |T_i.G|$.

There are three tasks $\{T_1, T_2, T_3\}$ in the example of Figure 3(a), namely digging the stone pile, loading, and unloading, respectively. The rules of execution are: T_3 can start after T_1 and T_2 finish, T_2 can start after T_1 finishes, ev_0 triggers T_1, then the preconditions and postconditions of these tasks are:

$$T_1.R = ev_0,\ T_1.O = \neg ev_0 \wedge T_1.f$$
$$T_2.R = T_1.f \wedge \neg ev_0,\ T_2.O = T_2.f$$
$$T_3.R = T_1.f \wedge T_2.f \wedge \neg ev_0,\ T_3.O = T_3.f$$

For some tasks, e.g., digging stones, finishing an execution means a decrease of the volume of the stone pile. This feature is reflected in the value change of the variables in $T_i.V$. When an agent is executing a regular task, it must not move. After finishing tasks, the agent must switch to a special task called no-op task before it starts to move. The no-op task indicates no task is being executed, and it is denoted as $T_0(0, \infty_+, \Delta, S, F, \emptyset, \emptyset, M, \emptyset, \emptyset)$. In T_0, B is 0 and W is ∞_+ implying the execution time can be any length, R and O are "$\emptyset$" implying the agent can get to or out of this task without restrictions, M is the complete set of all the milestones in the *environment* implying that this task is allowed at any position (except obstacles), and V and G are $\emptyset$ implying that this task does not change any data variable and is not triggered by any event. Based on Definition 3, we define the execution of tasks of an automated agent AA, as follows.

Definition 4 (Task Execution of AA). *For an automated agent $(\mathcal{S}, \mathcal{M}, \mathcal{T})$, the execution of tasks in $\mathcal{T}$ is defined as a timed automaton in a restricted form:*

$$Taa \triangleq (N, l_0, x_e, A_e, V_e, E_e, I_e, M_e) \tag{4}$$

where,
- *N is a set of locations representing the tasks in $\mathcal{T}$,*
- *$l_0 \in N$ is the initial location representing the no-op task T_0,*
- *x_e is a clock that is reset whenever a task finishes,*
- *$A_e = \{move,\ done_0, ..., done_n\} \cup \tau$ is a set of actions,*
- *V_e is a set of variables containing variables of all the tasks in $\mathcal{T}$, i.e., $V_e = \bigcup_{i=1}^{S} T_i.V$, $S = |\mathcal{T}|$,*
- *$E_e \subseteq l_0 \times A_e \times B_e(x_e, \mathcal{T}) \times 2^C \times 2^{\mathcal{T}} \times N$ is a set of edges connecting l_0 and $l \in N$ with a set of actions and guards, where $C = \{x_e\}$,*
- *$I_e : N \setminus l_0 \mapsto B_i(x_e)$ is a function assigning invariants to locations except l_0,*
- *$M_e : N \mapsto \mathcal{T}$ is a function assigning tasks to locations.* □

In A_e, "*move*" and "$done_0, ..., done_n$" are used for the synchronization between the *task TA* and the *movement TA* and the *monitor TA* respectively. The *monitor TA* are for supervising some indices of the agents that we will introduce later. For $\forall e_i \in E_e$, they are always between l_0 and $l \in N$, because the agents have to switch to the no-op task before they move or execute the next task. For $\forall\ T_i \in \mathcal{T} \setminus T_0$, the invariant $B_i(x_e)$ is of the form $x_e \leq T_i.W$. The guard on the incoming edge of T_i is of the form $P_j \wedge T_i.R$, where the Boolean variable P_j denotes if the current position of the agent is a

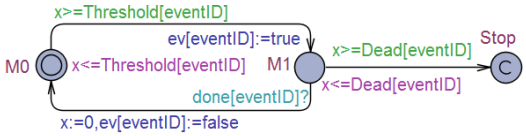


Figure 4: A monitor as an UPPAAL TA

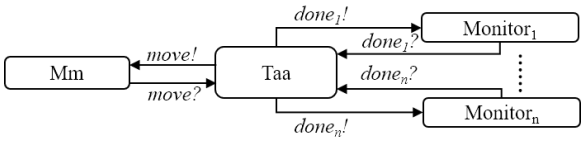


Figure 5: A network of TA obtained by TAMAA

milestone $m_j \in T_i.M$. The guard on the outgoing edge of T_i is $x_e \geq T_i.B$. Clock x_e, variables $v_i \in T_i.V$, task flags $T_i.S$ and $T_i.F$ are updated on the edge.

As some tasks are triggered by events (e.g., when the battery level is below a certain level, a battery-low event occurs and it triggers the agent to charge), we need monitors to supervise the status of the agents and inform in a real-time manner when the values of the indices are below or above thresholds. In fact, these indices are rates of consumption that could be represented by real-number values of time, e.g., the fuel/electricity consumes 80% when the agent travels a certain period of time. In this paper, we assume that all events only concern the indices changing monotonically and continuously over time. An example of monitor TA is depicted in Figure 4. The invariant of $M0$ and the guard of its outgoing edge are used to guarantee that the monitor is progressing to $M1$ when the clock's value reaches the threshold. Invariant of M1, and guard of its outgoing edge are used when switching to *Stop* when the clock's value reaches a certain threshold, meaning that the agent has no resources to move anymore. Hence, the monitor gives the agent a time horizon between the threshold and deadline to react to the event. However, if the agent ignores it for too long time, energy (fuel, battery, etc.) is consumed so it cannot move, which is represented as a deadlock in the TA.

A network of TA $Mm \,||\, Taa \,||\, Monitor_1 \,||\, \ldots \,||\, Monitor_n$ over (A, X) is a composition of TA for the movement, tasks, and monitors (Figure 5), where $A = \{move, done_1, \ldots, done_n\}$, $X = \{Mm.x_m, Taa.x_e, Monitor_1.x_r, \ldots, Monitor_n.x_r\}$, $n = |Ev|$. Taa sends out synchronization signals *move* to inform the movement TA that it is allowed to move, and $done_i$ to $Monitor_i$ informing the monitor TA that the task reacting on event ev_i has finished.

3.4 Automatic Generation of Autonomous Mission Models via TAMAA

In this section, we describe the steps and the algorithms used for building the resulting TA and we also show how we formalize the requirements as UPPAAL CTL/TCTL queries.

Algorithm 1: TA Generation of the Movement of an AA

```
1  Function CreateTA(Environment env)
2      grid :=new CartesianGrid(env)
3      ms :=new Milestones(env)
4      ta :=new TimedAutomata()
5      int tt[][] :=new int[grid.size][grid.size]
6      for m_i ∈ ms do
7          for m_j ∈ ms ∧ m_i ≠ m_j do
8              tt[m_i][m_j] :=ThetaStar(grid,m_i, m_j)
9      while ms ≠ ∅ do
10         Select a m_i ∈ ms, create a location A in ta representing it
11         for B ∈ ms ∧ B ≠ A ∧ tt[A][B] < MAX do
12             Create a location C in ta
13             Label C with a guard: ta.c ≤ tt[A][B]
14             CreateConnection(A, C, B, ta)
15             CreateConnection(B, C, A, ta)
16         Remove m_i from ms
17     return ta
18 Function CreateConnection(L1, T, L2, ta)
19     Create an edge e in ta from L1 to T
20     Label e with a channel move?
21     Label e with assignments: ta.c := 0, position[L1] := false
22     Create an edge e′ in ta from T to L2
23     Label e′ with a guard: ta.c ≥ tt[L1][L2]
24     Label e′ with assignments: position[L2] := true,
         visited[L2] := true
```

3.4.1 Generation of TA modeling the Movement of AA. To abstract the continuous-space environment as discrete models (as shown in Definition 2 in Section 3.3), we decompose the environment into a set of regions. There are two types of decompositions that have been investigated previously in the literature [10, 13]. The *geometry-ignoring* decomposition [10] concerns only a set of regions of interest and ignores the actual geometry of these regions. In contrast, the *geometry-using* decomposition [13] divides the environment by using different types of geometries, like rectangles, triangles, or convex polygons. Both approaches to environment decomposition ensure that propositions are well preserved by the discrete model of the environment and are therefore called *proposition-preserving decompositions* [10].

Our approach combines these two previous approaches by dividing the environment into square cells for path calculation between milestones and abstracting the environment as a TA where milestones and transitions among them are represented. The concrete description is shown in Algorithm 1. We first decompose the environment as a Cartesian grid and abstract the set of milestones as a two-dimensional array (i.e., *ms* in Algorithm 1) for storing the coordinates. An array *tt* of integers is used for storing the traveling time between milestones (lines 2 - 5). The Theta* algorithm is used to generate paths and traveling time (lines 6 to 8). In addition, we traverse the elements in *ms* and create a location (A) in ta

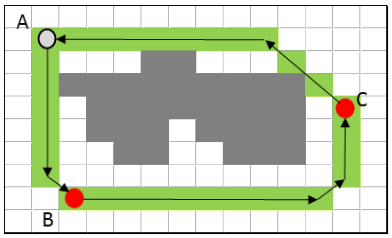


Figure 6: Environment decomposition and paths calculation applying for the environment of Figure 3(a)

for each of these elements (lines 9 and 10). "*MAX*" is the maximum value of integers. After selecting another location (B) in ta other than A, we connect them via a new location C (lines 11 - 15). We use a function "*CreateConnection*" for assigning guards and channels to edges as shown in Definition 4 of Section 3.3 (lines 19 - 23). Once the agent moves to a milestone, the corresponding element in the array *position* flips to true, and it is turned to false when the agent leaves the milestone. Similarly, the array *visit* is used for storing the visited milestones (lines 21 and 24).

Figure 6 illustrates the decomposition of the environment of Figure 3(a). The cells in the Cartesian grid are the decomposition unit. The ones that are completely or partially occupied by obstacles are marked as forbidden cells (colored in grey in Figure 6). Consequently, this is a conservative approach for obstacle detection that leads to unnecessary avoidance. This can be solved by increasing the grid resolution, which might however increase the computation time.

3.4.2 Generation of the Task TA for Automated Agents. Based on the concepts shown in Definition 4 of Section 3.3, we describe the process of building task TA (i.e., Algorithm 2).

Note that, the line numbers mentioned in this paragraph refer to Algorithm 2. We first create a TA and an initial location l_0 to represent the no-op task and label the self-loop edge of l_0 with a channel "*move*" for synchronization with the movement TA. In addition, we traverse every task $T_i \in AA.\mathcal{T}$ and create a location l_i in ta to represent it (lines 6 and 7). The l_i edge is labeled with an invariant $ta.c \leq T_i.W$ which ensures that the execution of the task must not be longer than its worst-case execution time. We create a new edge connecting l_0 to l_i and label it with a guard and assignments (lines 9, 10, and 11). The edge denotes the start of T_i and the guard is used to model that the agent must be at one of the locations whose index belong to $T_i.M$ and that the task's precondition $T_i.R$ must be true (line 10). The assignment on the edge resets the clock and flips the *starting* flag to true and the *finishing* flag to false (line 11). We create an edge connecting l_i to l_0 (lines 12 to 17) and label it with a guard, a channel, and assignments. The tasks triggered by events are labeled with "$done[i]$" to inform the monitor TA that the events are responded (lines 13 - 15). For all tasks, the assignments reset the clock and update the starting and finishing flags (line 16). For exemplification, we show in Figure 7 a task TA, which models the execution of a subset

Algorithm 2: Task Automaton Generation

1 **Function** *CreateTaskAutomaton(Agent aa, Bool position[], EventSet Ev)*
2 ta :=new TimedAutomata()
3 Create an initial location l_0 in ta representing the no-op task
4 Create a self-loop edge of l_0 and label it with $move$!
5 **while** $aa.\mathcal{T} \neq \emptyset$ **do**
6 Select a task $T_i \in aa.\mathcal{T}$
7 Create a location l_i in ta representing T_i
8 Label l_i with an invariant: $ta.c \leq T_i.W$
9 Create an edge e connecting l_0 to l_i
10 Label e with a guard: $\bigvee_{j=k}^{m} position[j] \wedge T_i.R$, where $\{k, \ldots, m\} = T_i.M$
11 Label e with assignments: $ta.c := 0$, $T_i.S := true$, $T_i.F := false$
12 Create an edge e' connecting l_i to l_0
13 **for** $ev_i \in Ev$ **do**
14 **if** ev_i *triggers* T_i **then**
15 Label e' with a channel: $done[i]$!
16 Label e' with assignments: $ta.c := 0$, $T_i.S := false$, $T_i.F := true$
17 Label e' with a guard: $ta.c \geq T_i.B$
18 Delete T_i from aa.$\mathcal{T}$
19 **return** ta

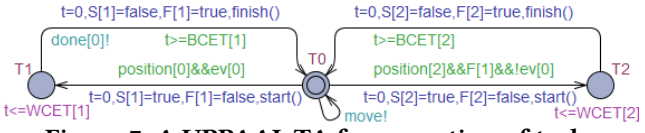


Figure 7: A UPPAAL TA for execution of tasks

of tasks in our running example, namely digging holes (i.e., T_1) and loading stones (i.e., T_2). T_0 is the no-op task.

3.4.3 Composition of TA. A network of *movement* TA and *task* TA and *monitor* TA is constructed to synthesize mission plans satisfying various properties. As shown in Figures 3(c) and 7, the *task* TA and the *movement* TA are synchronized using the "*move*" channel. Specifically, in the task TA in Figure 7, this channel is only labeled in the self-loop of T_0, because the agent is only allowed to move when it has no operation to perform. We mention here that the agent does not necessarily move to milestone C (i.e., $position[2]$) for executing the corresponding task T_2. It probably simply passes it to go to another milestone. Therefore the transition from T_0 to T_2 is not synchonized with the movement TA. The synchronizations between the *task* TA and *monitor* TA are modeled in a similar way. The network of TA is then used for model checking against certain CTL/TCTL queries in UPPAAL. The resulting execution traces from model checking representing transitions between milestones and tasks will be used to synthesize mission plans.

3.4.4 UPPAAL Queries Design. We use the requirements in our use case provided by VCE (as described in Section 3.1) to show the design of UPPAAL queries in the following way:

- *Task Coverage.* Given that the agent must finish all tasks, the corresponding CTL query can be written as:
$$E\Diamond\ (F[1] \wedge F[2] \wedge \ldots \wedge F[j] \wedge stonePileVol == 0) \quad (5)$$
As shown in Definition 3 in Section 3.3, "$F[i]$" represents the finish of task T_i and $stonePileVol$ is a variable indicating the volume of the stone pile. Hence, this query requires the agents to finish all the tasks and repeat them. When the query is verified in UPPAAL, a diagnostic trace is generated for the synthesis of mission plans. We make use of UPPAAL's ability to generate traces witnessing a submitted reachability property. Currently, UPPAAL supports three options for trace generation: some trace leading to a goal state, the shortest trace with the minimum number of transitions, and fastest trace with the shortest time delay.
- *Task Matching.* This requirement requires that an agent executes particular tasks at particular milestones, which is guaranteed by the guards in task TA defined in Definition 4. Hence, it does not need to be checked against a query.
- *Task Sequencing.* This requirement specifies that the order of task execution must be correct. To prove the correctness of some possible complex requirements, one can design the query in the following form:
$$E\Diamond\ S[i+1] \quad (6)$$
$$A\Box\ S[i+1]\ imply\ F[i] \quad (7)$$
In this case, task T_i must be finished before T_{i+1} starts. The invariance is used to guarantee that the order of execution holds on the model for all execution paths.
- *Timing Requirement.* For this requirement, the agent must guarantee to finish its tasks within a time limit. Assume the agent must carry all the stones within N time units, and c is a clock variable, the TCTL query can be as follows:
$$E\Diamond\ (F[1] \wedge F[2] \wedge \ldots \wedge F[j] \wedge stonePileVol == 0 \wedge c \leq N) \quad (8)$$
- *Event Reaction.* For this requirement, special tasks that are triggered by events under some circumstances need to be executed and prioritized. For instance, for battery level checking, a monitor would activate an event when the battery level is lower than a threshold. As the agent model describes all possible combinations of behavior, there is a possibility that the agent keeps staying at one location or moves meaninglessly without executing any task until its battery is consumed. Nevertheless, the satisfaction of query (5) or (8) guarantees that the synthesized mission plan subsumes that the agent charges itself whenever the low-battery event occurs, because if a deadlock happens in the monitor TA, there is no way to finish all the tasks (i.e., queries (5) and (8) cannot be satisfied).

4 TAMAA Implementation and Evaluation

In this section we outline some of the main aspects of TAMAA, including a high-level implementation description and an evaluation of its applicability and scalability in different realistic scenarios.

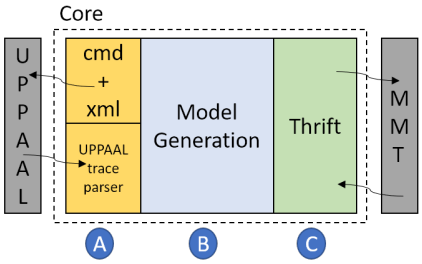
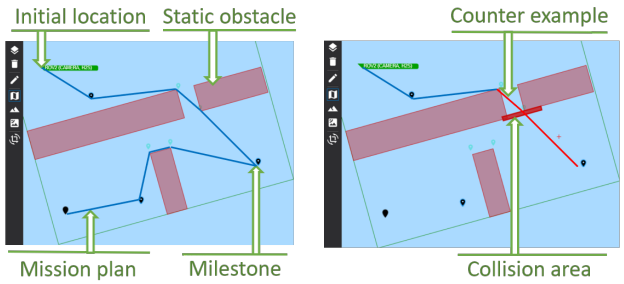


Figure 8: The architecture of TAMAA



(a) A mission plan generated in a reasonable environment

(b) A counter example generated in an unreasonable environment

Figure 9: Two screenshots of the MMT user interface

4.1 Implementation and User Interface

We present several technical solutions used in the implementation of TAMAA to fully support the complexity required for model-checking the generated models. We have implemented the algorithms described in Section 3.4 in Java and have integrated the TAMAA tool with a Mission Management Tool (MMT). MMT is a tool allowing users to graphically create complex missions for agents [14]. One can drag and drop markers in the environment as milestones and assign specific tasks to them (See Figure 9). When the environment and tasks are configured in MMT, one can choose a planner from the interface to calculate a mission plan. Our TAMAA tool is linked to MMT as an explicit planner option, which runs the Theta* algorithm, generates the UPPAAL TA and calculates trajectories satisfying the given queries. As illustrated in Figure 8, TAMAA has two communication modules and one processing module which are implemented in Java. The communication module connecting to UPPAAL is shown as module A in Figure 8 and consists of two submodules: one for connecting to UPPAAL for model checking and the other one for analyzing the obtained trace. The module for automatic TA generation is module B. The communication module connecting to MMT is shown as module C in Figure 8. Module B first reads data from MMT via module C, which is implemented in Apache Thrift[1], to obtain information about the environment, agents and tasks. Next, TAMAA

[1]Apache Thrift is a software framework for scalable cross-language services development. https://thrift.apache.org

executes its model-generation engine for automatically creating the UPPAAL TA, which is represented as an xml file. Module B invokes UPPAAL and sends the generated model and the necessary commands as command-line arguments to module A. After UPPAAL executes its verification of the model, an execution trace is produced and parsed by module A so that module B can interpret it as a mission plan and transfer it to MMT by executing module C. Finally, if a satisfactory execution trace exists, the corresponding mission plan is depicted in MMT as it is shown in Figure 9(a). Otherwise, a counter example representing an invalid mission plan is also produced and shown in MMT GUI (See Figure 9(b)) for further debugging.

4.2 Applicability Evaluation

In this section, we consider various scenarios of AWL to show the applicability of this method and tool implementation in a realistic setting. The following environment is used in all scenarios: a 50×50 2D space containing 3 static obstacles and 4 milestones. The evaluation is conducted on a machine running an Intel Core i5 processor with 16 GB of RAM and a 64-bit Windows OS. We present here the scenarios and the evaluation results.

Scenario 1. An AWL needs to perform 3 tasks in the right order for one round. We design queries in the form of queries (5)–(8) to obtain execution traces and check if the model satisfies the requirements. As all queries are satisfied, a mission plan is synthesized and the computation time is only a few milliseconds.

Scenario 2. An AWL needs to repetitively execute four tasks until the stone pile is empty and travel to a certain location to charge itself when the battery is low. In this case, one more task (i.e., charging) is added that is being triggered by "low-battery" event. A monitor containing an auxiliary data variable (variable ev_0) is designed to inform the task TA when the clock value exceeds a certain threshold. A query in the form of query (8) is generated and the computation takes 0.5 seconds while exploring 113,719 states. The generated trace for this query shows that the AWL as specified in the model reacts to the event "low-battery" in time.

Scenario 3. In this case, three AWL cooperate to accomplish one complex task. They have to all gather at one milestone and start the task simultaneously. After that, they continue to finish their own tasks. In this situation, the synthesis of mission plans for three agents has to be conducted in one entire model. Similarly, queries in the form of queries (5)–(8) are checked and satisfied. Verifying invariance queries in the form of query (7) takes less than 9 s to explore more than 770,000 states. Overall, our results show that mission plans are successfully synthesized for all scenarios within a few seconds. This is an indication that the TAMAA approach is applicable to the industrial scenarios of AWL.

Table 1: Scalability evaluation results with different number of milestones and tasks and 1 agent.

Query	Numer of Milstones	Numer of Tasks	Numer of Explored States	Time
Reachability	30	30	20,363	0.2 s
	60	60	157,033	2.2 s
	100	100	712,721	14 s
Invariance	30	30	41,193	0.3 s
	60	60	317,703	4.5 s
	100	100	1,429,903	29 s

Table 2: Scalability evaluation results with different number of agents running 3 tasks among 3 milestones.

Query	Numer of Agents	Numer of Explored States	Time
Reachability	2	1,661	0.01 s
	3	159,632	2.0 s
	4	2,058,132	20160 s
	5	Out of Memory	Out of Memory
Invariance	2	3,533	0.03 s
	3	344,701	4.0 s
	4	Out of Memory	Out of Memory

4.3 Scalability Evaluation

In this section, we consider the scalability of TAMAA with regard to the number of milestones, tasks and agents considered. In all scenarios we are interested in two types of queries: reachability and invariance (queries (5) and (7) are used as examples). In this evaluation, we first vary the number of milestones and tasks between 30 and 100 for both. Meanwhile, we use one AWL for all variations as this is a realistic scenario for the use case. The result is presented in Table 1, and it shows that the computation time ranges between 0.2 s and 29 s and the number of states explored is increasing exponentially with the number of milestones and tasks for all queries. We mention here that even for a model containing 100 milestones and tasks the results are encouraging in term of model checking efficiency.

In addition, we evaluate the scalability of TAMAA by varying the number of AWL between 2 to 5. The results are shown in Table 2. The environment is kept the same for all variations and contains three milestones and three tasks. We conduct this evaluation using Scenario 3 described in Section 4.2. The number of explored states and computation time increases exponentially with the number of AWL. We observe that the results for the case with three AWL running in a 3-milestone environment executing 3 tasks are similar with the results for one agent executing in an environment with 60 milestones and 60 tasks shown in Table 1. This can be explained by the increase in the number of TA and clocks for

the models containing more agents, which results in more time zones and non-deterministic interleaving transitions. Thus, searching through models with more agents takes significantly longer. We note that the use of more than three agents is problematic and therefore restricts the handling of larger systems, due to the increased cost of computation time and states explored. Because of the use of clocks at locations and edges, partial order reduction [7] of the model is not suitable in this model. One of our ongoing work is to integrate reinforcement learning [18] in the model to leverage the historical exploration of the state space of the model to alleviate the scalability problem. We leave this to our future work to report.

5 Related Work

In recent decades, there has been a growing interest in formal modeling and verification of autonomous systems given mission planning problems with complex goals. Belta et al. [6] present a hierarchical structure and based on a three-level process they propose a method using Linear Temporal Logic (LTL). This is evaluated in several case studies [17, 19]. Bhatia et al. [9, 10] propose synthesis methods by constructing a multi-layered synergistic framework. This work uses temporal logic to specify goals and shows the use of geometry in the abstraction of environment results with significant computational speedups compared to previous studies. Dimarogonas et al. [4, 15] propose their method for motion planning of multiple-agent systems using various temporal logic. In contrast to these studies, our approach is focusing on integrating a state-of-the-art path-planning algorithm with temporal logic to leverage the heuristics and efficiency of the former and the rigorousness and expressiveness of the latter. In addition, our approach combines a model-checker with a mission-management tool to tackle this problem on an industrial case, which demonstrates the applicability and scalability of this approach in realistic scenarios. Instead of using LTL (e.g., [6]) for requirement specification, we explore the use of TCTL for expressing more complex requirements (i.e., both functional and timing requirements).

6 Conclusions and Future Work

In this paper we have presented an integrated approach (named TAMAA) for automatically generating mission plans for autonomous agents satisfying various requirements (functional and timing). As part of TAMAA, we provide formal definitions of the movement of autonomous agents and tasks. These definitions enable a rigorous way of formalizing a practical problem. We also provide algorithms for the automatic model generation before verifying the models in UPPAAL against CTL/TCTL queries expressing autonomous vehicle requirements important to their respective missions. For increasing the appeal of our method, we have implemented these algorithms in a tool written in Java and have integrated it with a mission-management tool to provide an easy-to-use automated support. Our approach has been evaluated in three scenarios proposed by industry demonstrating its applicability in realistic scenarios. The scalability evaluation shows that while the number of tasks and the scale of the environment do not significantly influence the cost of model checking in terms of computation time and the number of states explored, the synthesis efficiency dramatically decreases with the number of agents.

The future work has at least two potential directions. One is to combine model checking techniques with machine learning (e.g. reinforcement learning) to improve the efficiency of searching through the state space. Another direction is related to the integration of TAMAA with our two-layer framework with the goal of proposing and evaluating an entire solution for performing static planning and dynamic simulation and verification by taking into account the dynamics and kinematics of different types of agents.

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] R. Alur and D. Dill. 1990. Automata for Modeling Real-time Systems. In *Automata, languages and programming*. Springer, 322–335.
[2] Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235.
[3] Saeed Asadi Bagloee, Madjid Tavana, Mohsen Asadi, and Tracey Oliver. 2016. Autonomous vehicles: challenges, opportunities, and future implications for transportation policies. *Journal of modern transportation* 24, 4 (2016), 284–303.
[4] Fernando S Barbosa, Lars Lindemann, Dimos V Dimarogonas, and Jana Tumova. 2019. Integrated Motion Planning and Control Under Metric Interval Temporal Logic Specifications. In *2019 18th European Control Conference (ECC)*. IEEE.
[5] Gerd Behrmann, Alexandre David, and Kim G Larsen. 2006. A tutorial on Uppaal 4.0. *Department of computer science, Aalborg university* (2006).
[6] Calin Belta, Antonio Bicchi, Magnus Egerstedt, Emilio Frazzoli, Eric Klavins, and George J Pappas. 2007. Symbolic planning and control of robot motion [grand challenges of robotics]. *IEEE Robotics & Automation Magazine* 14, 1 (2007), 61–70.
[7] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. 1998. Partial order reductions for timed systems. In *International Conference on Concurrency Theory*. Springer, 485–500.
[8] Johan Bengtsson and Wang Yi. 2004. Timed automata: Semantics, algorithms and tools. *Lecture Notes in Computer Science* 3098 (2004), 87–124.
[9] Amit Bhatia, Lydia E Kavraki, and Moshe Y Vardi. 2010. Sampling-based motion planning with temporal goals. In *International Conference on Robotics and Automation*. IEEE, 2689–2696.
[10] Amit Bhatia, Matthew R Maly, Lydia E Kavraki, and Moshe Y Vardi. 2011. Motion planning with complex goals. *IEEE Robotics & Automation Magazine* 18, 3 (2011).
[11] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. 2010. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research* 39 (2010).
[12] Stan Franklin and Art Graesser. 1996. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 21–35.
[13] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. 2009. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics* 25, 6 (2009), 1370–1381.
[14] Branko Miloradović, Baran Cürüklü, Mikael Ekström, and Alessandro Papadopoulos. 2019. Extended Colored Traveling Salesperson for Modeling Multi-Agent Mission Planning Problems. In *International Conference on Operations Research and Enterprise Systems*. INSTICC, SciTePress, 237–244.

[15] Alexandros Nikou, Dimitris Boskos, Jana Tumova, and Dimos V Dimarogonas. 2018. On the timed temporal logic planning of coupled multi-agent systems. *Automatica* 97 (2018), 339–345.
[16] Scott Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Eng, Daniela Rus, and Marcelo Ang. 2017. Perception, planning, control, and coordination for autonomous vehicles. *Machines* 5, 1 (2017), 6.
[17] Stephen L Smith, Jana Tumova, Calin Belta, and Daniela Rus. 2011. Optimal path planning for surveillance with temporal-logic constraints. *International Journal of Robotics Research* 30, 14 (2011), 1695–1708.
[18] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[19] Alphan Ulusoy, Stephen L Smith, Xu Chu Ding, Calin Belta, and Daniela Rus. 2013. Optimality and robustness in multi-robot path planning with temporal logic constraints. *International Journal of Robotics Research* 32, 8 (2013), 889–911.