Mission Planning for Multiple Autonomous Agents under Complex Road Conditions: Model-Checking-Based Synthesis and Verification

RONG GU, Mälardalen University, Sweden
EDUARD BARANOV, UCLouvain, Belgium
AFSHIN AMERI, Mälardalen University, Sweden
EDUARD PAUL ENOIU, Mälardalen University, Sweden
BARAN CÜRÜKLÜ, Mälardalen University, Sweden
CRISTINA SECELEANU, Mälardalen University, Sweden
AXEL LEGAY, UCLouvain, Belgium
KRISTINA LUNDQVIST, Mälardalen University, Sweden

Mission planning for multi-agent autonomous systems aims to generate feasible and optimal mission plans that satisfy the given requirements. In this article, we propose a mission-planning methodology that combines (i) a path-planning algorithm for synthesizing path plans that are safe in environments with complex road conditions, and (ii) a task-scheduling method for synthesizing task plans that schedule the tasks in the right and fastest order, taking into account the planned paths. The task-scheduling method is based on model checking, which provides means of automatically generating task execution orders that satisfy the requirements and ensure the correctness and efficiency of the plans by construction. We implement our approach in a tool named MALTA, which offers a user-friendly GUI for configuring mission requirements, a module for path planning, an integration with the model checker UPPAAL, and functions for automatic generation of formal models, and parsing of the execution traces of models. Experiments with the tool demonstrate its applicability and performance in various configurations of an industrial case study of an autonomous quarry. We also show the adaptability of our tool by employing it on a special case of the industrial case study.

1 INTRODUCTION

Autonomous robotic systems are becoming common in our society. These systems can take different forms, e.g. a vehicle that is used for transportation in a factory or in a construction site, a mobile robot used for entertainment in our homes, or a mobile communication platform used in the homes of the elderly. Obviously, these robotic systems are associated with different requirements, thus having different shapes and overall design. Despite the differences, these systems share a common feature: the ability to function in an environment without or with minimum human intervention. This feature can be referred to as *autonomous operation*. However, the environment where these systems operate in could include humans and other obstacles, hence it is unpredictable and only partially known to the system. To realize the autonomy feature, the autonomous robotic systems must be able to perceive the environment, reason based on known facts, and act to meet the requirements associated to their goals (for the sake of brevity autonomous robotic agents are called "autonomous agents", or simply "agents" in the rest of the paper [FG96]).

One key challenge of designing agents that move and operate in a confined environment is *mission planning* (a.k.a., mission plan synthesis), which includes *path planning* and *task scheduling*. When obstacles are present in the environment, the ability to reach a destination without colliding with them is a problem that has been solved by existing path-planning algorithms, such as A* [Rab00]

Authors' addresses: Rong Gu, rong.gu@mdu.se, Mälardalen University, Sweden; Eduard Baranov, eduard.baranov@uclouvain.be, UCLouvain, Belgium; Afshin Ameri, afshin.ameri@mdu.se, Mälardalen University, Sweden; Eduard Paul Enoiu, eduard.paul.enoiu@mdu.se, Mälardalen University, Sweden; Baran Cürüklü, baran.curuklu@mdu.se, Mälardalen University, Sweden; Cristina Seceleanu, cristina.seceleanu@mdu.se, Mälardalen University, Sweden; Axel Legay, axel.legay@uclouvain.be, UCLouvain, Belgium; Kristina Lundqvist, kristina.lundqvist@mdu.se, Mälardalen University, Sweden.

and rapidly-exploring random tree (RRT) [LaV98] algorithms. However, the real environment might impose complex restrictions to path planning, stemming from obstacles that are temporary, or from existing special areas, like crowded or desired areas. Consequently, the *path-planning* algorithm should provide means to calculate a path plan for an agent that "wisely" chooses to wait, circumvent, or cross the respective areas. Furthermore, agents visit different positions (a.k.a. *milestones*) in the environment as part of their tasks, e.g., within a quarry, autonomous wheel loaders visit stones piles to load stones. Being able to guarantee that agents carry out the right tasks at the right milestones is important for the overall success of a mission. Additionally, tasks can have complex and temporal requirements, for instance, autonomous wheel-loaders must keep digging stones until trucks arrive, and then load the stones into the trucks before the latter transport the stones to crushers. Some applications require the agents to keep a certain level of productivity, e.g., autonomous trucks must transport all the stones to a crusher within a maximum time window of a few hours. Path-planning algorithms alone are not able to calculate *mission plans* that accomplish the tasks respecting such requirements. They must be combined with *task-scheduling* algorithms for synthesizing mission plans that ensure that the agents travel safely, that is, without any collision with static obstacles, and satisfy the requirements of tasks.

Task-scheduling algorithms aim to calculate an order of task execution to achieve the global goal, e.g., a pile of stones is dug and loaded, then transported to crushers, and then crushed into fractions. Based on the position of each agent, task scheduling assigns milestones and the visiting orders to the agents, respectively, so that the agents can finish their mission within a given time window. A classic presentation of this problem is called the *job-shop* problem, which is described as follows:

"*Given are a set of jobs and a set of machines, assume that: (i) Each machine can handle at most one job at a time, and (ii) Each job consists of a chain of operations, each of which needs to be processed during an uninterrupted time period of a given length on a given machine. The purpose is to find a schedule, i.e., an allocation of the operations to time intervals on the machines, that has minimum length.*" [Len92]

As an NP-hard problem, even a simple instance of the *job-shop* problem with very restrictive constraints remains difficult to solve [AAM+06]. Furthermore, task schedules that we aim to calculate must not only "have the minimum length", i.e., accomplish the tasks in the quickest way, but also satisfy the complex temporal requirements aforementioned. In a nutshell, we provide answers to the following research questions in this paper:

(1) *Path planning*: considering an environment with various road conditions, such as static obstacles (e.g., forbidden areas and temporary obstacles), crowded areas, and desired/undesired areas, how to calculate a path that reaches the desired milestone without a collision?
(2) *Task scheduling*: given a set of tasks and milestones where the tasks are supposed to be carried out, as well as a set of task execution constraints, how to synthesize a task-execution schedule that finishes tasks at their respective milestones and satisfies the task execution constraints?
(3) *Mission planning*: how to combine path planning with task scheduling and produce an optimal mission plan?
(4) *Automation*: given a mission-planning problem that matches the problem definition (Definition 1), how to easily configure the scenario of the problem and automatically calculate mission plans that combine the results of path planning and task scheduling?
(5) *Adaptability*: when the mission-planning problem does not match the problem definition, how to leverage the automation and easily adapt the models to solve the problem?
(6) *Visualization*: how to visualize mission plans for demonstration purpose?

In this article, our main contribution is a methodology and a tool support for answering the research questions (1) - (6) collectively, with an optimal result, that is, not only a correct mission plan that meets various requirements but also reaches the goal in the fastest manner.

In particular, we adapt and tune up the DALı algorithm [CFL+15] for path planning, which takes into account both environmental constraints and user preferences (research question (1)). We adapt a method called TAMAA (Timed-Automata based Mission planner for Autonomous Agents) [GES19] for task scheduling, which uses the well-known timed automata (TA) [AD94] formalism for modeling, and model checking in UPPAAL [HYP+06] for generating results with a correctness guarantee (research question (2)). Specifically, TAMAA automatically generates a TA model of agents, including the movement TA and task execution TA, and checks the model to find the execution traces that reaches the goal state the fastest and satisfy other requirements. The contribution of the new version of TAMAA is the combination with DALı. Previously, TAMAA uses the result of path planning once and generates a mission plan that only considers the permanently existing obstacles. In this article, complex road conditions such as temporary obstacles are considered. Hence, the initially correct mission plans may become invalid when temporary obstacles are activated. Hence, we design a validator in MALTA to check if the temporary mission plan meets the complex road conditions, and an iteration of execution between DALı and TAMAA until a correct mission plan is produced (research question (3)).

The design and implementation of our methodology, i.e., a toolset named *MALTA*, answers to research questions (4) - (6). MALTA has a client-server architecture, which integrates a graphical user interface (GUI) called *MMT* [ACME20] for mission management in the client, data exchange modules, and path-planning and task-scheduling algorithms in the server. The client side of MALTA is MMT where users can configure the environment, missions for the agents, and parameters of agents, like their speeds, respectively. The server side has two modules: middleware and a back end. The middleware is responsible for obtaining the mission information from the GUI, running a path-planning algorithm, and generating agent models by using the path-planning results and mission information. Next, the middleware sends the agent models to the back end, where TAMAA runs for task scheduling. As our methodology is designed to cope with complex road conditions, such as temporary obstacles, MALTA possibly runs more than one iteration rounds between the middleware and the back end until an optimal mission plan is produced.

If a mission plan exists, it is visualized by MALTA so that users can check the details of the mission plan, such as when and which agent starts to execute a certain task, and how temporary obstacles can affect the plan (research question (6)). MALTA generates models and parses traces automatically, which eases the work of model and mission plan construction, especially for the cases where the amount of agents or the size of the environment is large (research question (4)). One can also modify the models according to one's own applications and still enjoy the facility of other automation provided by MALTA, such as trace parsing and information extraction from the map (research question (5)). MALTA has been applied on an industrial use case of an autonomous quarry that exposes the challenges of synthesizing time-optimal mission plans of several autonomous vehicles with various requirements.

The rest of the paper is organized as follows. In Section 2, we introduce the industrial case study. Section 3 describes the preliminaries including timed automata and UPPAAL, and the DALı algorithm for path planning. Our methodology for solving the mission-planning problem for multiple agents is introduced in Section 4, followed by the description of the toolset in Section 5. In Section 6, we conduct experiments on a normal use case taken from the industrial case study, which matches our problem definition, for the evaluation of our methodology. Section 7 demonstrates how the method can be adapted to a special use case of our industrial case study. In Section 8, we

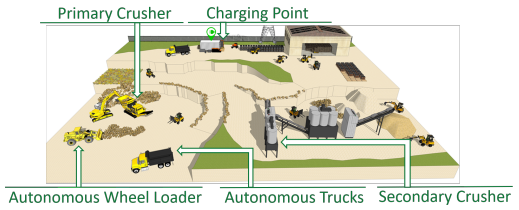


Fig. 1. An example of an autonomous quarry

present and compare to related work, whereas in Section 9, we conclude the paper and outline some directions of future work.

2 AN INDUSTRIAL CASE STUDY: THE AUTONOMOUS QUARRY

In this section, we introduce an industrial use case of an autonomous quarry provided by VOLVO Construction Equipment (CE) in Sweden. This use case serves as a running example through Sections 2 to 6, and as concrete motivation for our research. In the quarry, there are several stationary machines and vehicles. Examples of stationary machines are crushers that crush stones into certain sizes and are controlled manually. Typical examples of vehicles would be wheel loaders that dig stones, and autonomous trucks that transport stones into crushers. We assume that the wheel loaders are controlled manually and focus on the synthesizing mission plans for the autonomous trucks that are not necessarily identical, which means they can have different speed limits (e.g., 50 – 80 km/hour) and capability of transportation (e.g., 10 – 50 m^3). The vision is to deploy the autonomous trucks that performs certain tasks in order to fulfill the objectives defined by the operators of the quarry. The collection of such tasks together define a mission.

A simple quarry is illustrated in Figure 1. In this example, the stones should be dug up and loaded into the autonomous trucks by wheel loaders. Then trucks transport the stones and unload them into a primary crusher first, then later to a secondary crusher, the destination of the stones. Trucks must go to a charging pole when the battery level is low. The milestones are the positions where the tasks are carried out, e.g., at stone piles, crushers, and charging poles. A mission in this use case can be to dig, crush, and transport $1000m^3$ of stones in 24 hours.

Vehicle transportation should consider the environment. There could be static obstacles, that is, areas that are impassable by the vehicles (buildings, sizable holes in the road, etc.), and different road conditions that may cause the vehicles to slow down, e.g. muddy or bumpy roads, or form a temporarily inaccessible area, such as, during human intervention for maintenance, the corresponding area should not be crossed by any vehicle. Navigation must ensure a collision-free transportation in the presence of all the environmental constraints. The execution of tasks, e.g., loading, unloading, and charging, must be scheduled correctly and efficiently such that the trucks are guaranteed to accomplish the mission. In the autonomous quarry, all the planning work must be done automatically. Therefore, we need a planning methodology and a tool of planning that ensures completion of the mission and meeting additional requirements.

Based on this use case we formulate several requirements. We group the requirements for the *task scheduling* into the following categories:

- **Requirement Category I** (*milestone matching*): Tasks must be executed at the correct milestones, e.g., loading stones into primary crushers must be executed at a primary crusher.
- **Requirement Category II** (*task sequence*): Tasks must be executed in the right order, e.g., loading stones from stone piles must precede unloading stones into crushers.

- **Requirement Category III** (*timing*): Tasks can be executed multiple times and must be finished within a time frame in order to maintain a certain level of productivity, e.g., quarrying $1500m^3$ stones per day.

Path planning takes care of finding safe and fast paths for agents between milestones in complex environments. We consider the following types of environmental abnormalities that should be taken into account by the path planner:

- **Environmental Abnormality I** (*obstacles*): permanent and temporary obstacles must be avoided by agents. Permanent obstacles are always present, while temporary ones appear at known time points and disappear later on. Within this work we assume that all obstacles are static.
- **Environmental Abnormality II** (*road conditions*): muddy or bumpy roads cannot be passed at the full speed. The planner must decide whether it is faster to take a detour than to travel on these roads slowly.
- **Environmental Abnormality III** (*soft constraints*): for some reasons, e.g., safety, some areas might be considered undesirable for driving, and the planner must attempt to avoid such areas if there exists an alternative path. Such constraints should be satisfied unless they prevent the satisfaction of other requirements with higher priority. In the latter case, soft constraints can be ignored.

Note that path planning and task scheduling are not independent. Traveling time among milestones are needed by a task scheduler, while the presence of temporary obstacles can influence the starting time of travels. Therefore, path planning and task scheduling must work together to produce efficient mission plans.

In this paper, we work only with static scheduling: generated mission plans cannot change without full recomputation, therefore any additional tasks or unpredictable events are not considered, including unpredictable moving obstacles. Path following and the collision avoidance between agents are also outside the scope of this paper. We refer the interested reader to the literature [GMSL19] for these topics. In addition, the order of visiting milestones only depends on the constraints of task sequence. For example, for a mission containing two tasks A and B that must be carried out at milestones a and b, respectively, the order of visiting the milestones only depends on the task sequence of A and B. In another word, if tasks A and B can be executed in any order, then visiting milestone a and b can be any order too. To define the scope of automation of our methodology, we informally define the mission-planning problem and its solution as follows. The set of (non-negative) real numbers is denoted as $\mathbb{R}$ ($\mathbb{R}_{\geq 0}$).

Definition 1 (Mission Planning). *A mission-planning problem is a tuple:*

$$\mathcal{P} =< \mathcal{E},\ Ab,\ \mathcal{M},\ \mathcal{T},\ f,\ Req >, \tag{1}$$

where $\mathcal{E} \subset \mathbb{R}^n$ is a confined environment, $n \in \{2, 3\}$, $Ab \subseteq \mathcal{E}$ is a set of areas that belong to one of the Environmental Abnormalities I, II, and III, $\mathcal{M} \subset \mathcal{E}$ is a set of milestones, $\mathcal{T}$ is a set of tasks, $f : \mathcal{M} \to \mathcal{T}$ assigns each of the tasks to one or multiple milestones, and Req is a set of task requirements that matches the Requirement Categories I, II, and III.

Assuming an agent is equipped with the ability of doing two types of actions: moving to a milestone and executing a task, a solution of the mission-planning problem is informally defined as follows:

Definition 2 (Solution of Mission Planning). *Given a set of autonomous agents $\mathcal{V}$, a solution that enables the agents in $\mathcal{V}$ to solve a mission-planning problem $\mathcal{P} =< \mathcal{E},\ Ab,\ \mathcal{M},\ \mathcal{T},\ f,\ Req >$ is a*

tuple:

$$plan =< schedule,\ path >, \tag{2}$$

where $schedule$ *is a set of pairs* $(st,\ ft)$, $st,\ ft \in \mathbb{R}_{\geq 0}$ *are the time points of starting and finishing the agents' actions, respectively,* $path$ *is a set of sequences of points* $p \in \mathcal{E}$ *that the agents must follow in order to reach the milestones* $m \in \mathcal{M}$. *Let* $E1$, $E2$, *and* $E3$ *be the sets of environment constraints belonging to Environmental Abnormalities I, II, and III, respectively, s.t.,* $Ab = E1 \cup E2 \cup E3$, *and* $E3' \subseteq E3$ *be the subset of* $E3$, *which does not contradict* $E1$ *and* $E2$, *that is, the paths that meet* $E3'$ *do not violate* $E1$ *and* $E2$, *plan must hold the following two conditions:*

- $\forall e \in E1 \cup E2 \cup E3',\ path \vDash a$, *that is, the paths must meet environmental constraints in* $E1$, $E2$, *and* $E3'$,
- $\forall r \in Req,\ schedule \vDash r$, *that is, the task schedule must satisfy all the requirements in Req.*

In the rest of this paper, we introduce our methodology for automatically generating solutions of mission-planning problems, defined in Definitions 2 and 1, respectively. We also demonstrate the adaptability of the methodology to solve problems that does not match Definition 1 with a slight change of the models.

3 PRELIMINARIES

In this section, we introduce UPPAAL - the modeling, simulation, and verification tool that uses *Timed Automata* as the modeling formalism, which we apply for modeling agents movement and task execution. We also briefly describe the path-planning algorithm DALI that we employ in MALTA. We denote the set of natural numbers as $\mathbb{N}$.

3.1 Timed Automata and UPPAAL

Definition 3. *A* Timed Automaton *(TA) [AD94] is a tuple:*

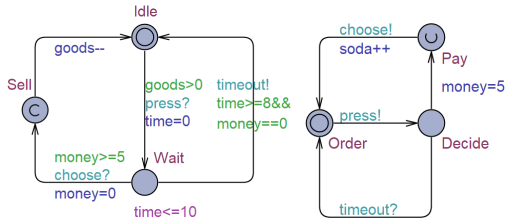
$$\mathcal{A} =< L, l_0, X, \Sigma, E, Inv >, \tag{3}$$

where L *is a finite set of locations,* l_0 *is the initial location,* X *is a finite set of non-negative real-valued clocks,* Σ *is a finite set of actions,* $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ *is a finite set of edges, where* $\mathcal{B}(X)$ *is the set of guards over* X, *that is, conjunctive formulas of clock constraints of the form* $x \bowtie n$ *or* $x - y \bowtie n$, *where* $x, y \in X$, $n \in \mathbb{N}$, $\bowtie\ \in \{<, \leq, =, \geq, >\}$, 2^X *is a set of clocks in* X *that are reset, and* $Inv : L \to \mathcal{B}(X)$ *is a partial function assigning invariants to locations.* □

The semantics of a TA $\mathcal{A}$ is defined as a *timed transition system* over states (l, v), where l is a location and $v \in \mathbb{R}^X$ represents the valuation of the clocks in that state, with the initial state $s_0 = (l_0, v_0)$, where v_0 assigns all clocks in X to zero. There are two kinds of transitions:

(1) *delay transitions*: $(l, v) \xrightarrow{d} (l, v \oplus d)$, where $v \oplus d$ is the result obtained by incrementing all clocks of the automaton with the delay amount $d \in \mathbb{R}^+$ such that $v \oplus d \vDash I(l)$, and
(2) *discrete transitions*: $(l, v) \xrightarrow{a} (l', v')$, corresponding to traversing an edge $l \xrightarrow{g,a,r} l'$ for which the guard g evaluates to *true* in the source state (l, v), $a \in \Sigma$ is an action, r is a set of clocks that are reset over the edge, and clock valuation v' of the target state (l', v') is obtained from v by resetting all clocks in r such that $v' \vDash Inv(l')$.

UPPAAL [HYP+06] is a state-of-the-art model checker for real-time systems. It supports modeling, simulation, and model checking, and uses an extension of TA with data variables, synchronization channels, urgent and committed locations etc., as the modeling formalism, which we call UPPAAL TA (UTA). In UPPAAL, UTA can be composed in parallel as a *network* of UTA synchronized via *channels* (an edge decorated with channel a! is synchronized with one decorated with a? by handshake). An example of UPPAAL model is depicted in Figure 2, consisting of two automata: a



(a) A UTA of a vending machine (b) A UTA of a customer

Fig. 2. An example of UTA modeling a customer ordering food from an impatient vending machine.

customer ordering sodas and an impatient vending machine kicking its customers out of the system when they do not order quickly enough (i.e., 10 time units is the maximum waiting time). A UTA of the vending machine shown in Figure 2(a) has 3 locations named `Idle`, `Sell`, and `Wait`. *Edges* are the direct lines that connect *locations*, which can be decorated by *guards*, *channels*, and *updates*. A clock variable `time` measures the elapse of time, and is used in the *invariants* on locations (e.g., `time<=10`) and in the *guards* on edges (e.g., `time>=8`). In UTA, locations can be labeled as *urgent*, denoted by encircled ∪, which forbids delaying in the locations (e.g., `Pay` of the customer TA in Figure 2(b)), or *committed*, denoted by encircled *C*, which not only forbid time from elapsing but also require the network of UTA to transit the next edge from one of the committed locations. Such an example of committed locations is *location* `Sell` of the vendor UTA in Figure 2(b). UTA also extends TA with data variables (integer and Boolean variables), which can be updated via C-code functions or assignments on edges. For example, on the *edge* from *locations* `Decide` to `Pay` of the customer TA, an integer variable `money` is updated to 5, meaning that the customer has paid 5 Swedish crowns to the vending machine.

UPPAAL can verify properties formalized as queries in a subset of *Timed Computation Tree Logic* (TCTL) [BY03]. Given an atomic proposition p over the locations, clocks, and data variables of the UTA, the UPPAAL queries that are used in this paper are: (i) **Invariance**: `A[]` 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) **Timed-bounded Reachability**: $E\diamond_{\leq T}$ p means that there exists a path where p is satisfied by at least one state of the path within T time units.

3.2 Devices for Assisted Living (DALı)

The DALı algorithm has been proposed previously [CFL+15], and is designed to provide motion planning in complex environments. The planner takes into account environmental constraints and user's preferences.

DALı consists of two parts: long-term and short-term planners. The former one is performed prior to actual navigation and searches for paths accounting for area topology, user goals, and foreseen obstacles or problems along the way. During navigation unforeseen obstacles can appear, e.g. a group of people obstructs the path, its detection starts a short term planner attempting to find a minimal deviation from the path that preserves all the constraints. Short term planning is often required to operate on low resources and to provide a quick response, therefore long term planner cannot be used for quick search for path deviations.

In this paper, we are interested in a long term planner of DALı. At the first step the algorithm transforms the area into a graph with a sufficient granularity to represent paths between points.

Permanent obstacles are excluded from the graph. Other environmental constraints and user preferences are stored within nodes and edges of the graph with one of the following options:

- Soft constraints taken from users' preferences and indicating zones that the planner would try to visit or to avoid. Each soft constraint is characterized by a triple (*center, radius, intensity*): it affects paths within the *radius* from its *center* and its intensity is the highest at the *center* decreasing with the distance from the *center*. A proposed path is deviated towards or outwards of the center within the radius based on the intensity level.
- A heat map of the environment indicating areas with high occupancy; navigation through such areas is slower by a specified factor.
- Anomalies or temporary obstacles that are areas inaccessible during certain periods of time. The long-term planner considers foreknown anomalies, assuming that their appearance and disappearance time is provided.

The long-term planner of DALI is based on Dijkstra's shortest path algorithm [D+59]. The algorithm maintains a set of nodes to which the shortest distances from a source node are computed. Starting from the source node, at each step the algorithm selects a new node that has the shortest distance from the source node and adds it to the set. DALI modifies Dijkstra's algorithm by taking into account environmental constraints and users' preferences during the choices of the nodes to be added into the set. Temporary obstacles prevent selection of the nodes inside these obstacles during the inaccessibility time. Heat maps affect the actual lengths of the edges. Soft constraints add virtual coefficients to the distances inside the zones covered by these constraints. Note that soft constraints do not affect the real travel time (the coefficient is virtual), thus DALI might return a path that is not the shortest. The coefficients of the soft constraints bound the extra length of the returned path, that is, if a detour for satisfying a soft constrain is too long to satisfy the time limit of reaching the destination, the constraint would be ignored.

4 MISSION PLANNING METHODOLOGY

In this section, we introduce our methodology for automated mission planning for multiple agents. We describe the overall procedure followed by the detailed description of each step.

Mission planning is composed of two main aspects: task scheduling and path planning. The former defines which tasks, in which order, at what time, and by which agent should be executed. The latter indicates the traveling path between milestones. Note the dependency of task scheduling on path planning: the knowledge of the traveling time is needed for scheduling.

We propose a *UTA-based mission planner* for agents. The path-planning aspect is based on an adapted version of the DALI algorithm. The DALI-based path planner takes the information of the map, including the navigation area, special areas (e.g., forbidden areas), milestones, tasks, and agents, and computes paths connecting milestones to each other regardless of the visiting order. The task-scheduling aspect is an adapted version of TAMAA (Timed-Automata based Mission Planner for Autonomous Agents) [GES19]. The TAMAA-based task scheduler employs UPPAAL to synthesize an optimal schedule satisfying all the task constraints such as the correct order of task execution. In general, task scheduling sets up the skeleton of the mission plan, which orders the actions of movement and task execution, and path planning fills in the generated concrete routes between every pair of milestones. While theoretically it is possible to employ UPPAAL for both path planning and task scheduling, in practice full mission planning in UPPAAL is infeasible due to the scalability problem, and the separation into two aspects is designed to simplify the computations [GES19].

To ensure the correct mission planning, path planning and task scheduling have to interact with each other: task scheduling requires the traveling time between milestones while path planning

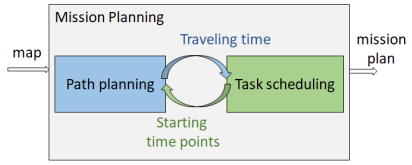
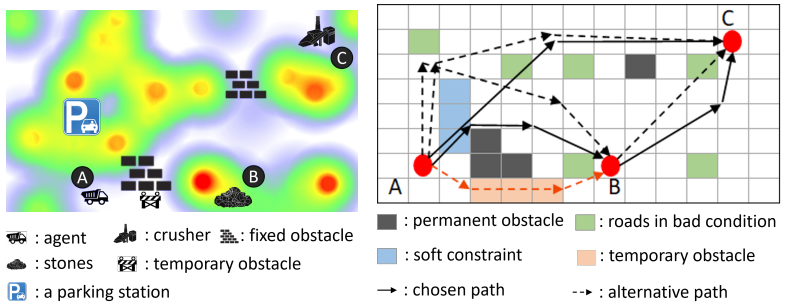


Fig. 3. Iterative process of mission planning

needs to know when the trip would take place in order to generate paths avoiding temporary obstacles. Therefore, both parts are executed in a loop until all constraints are satisfied by the resulting mission plan. The overall workflow consists of the following steps:

(1) Mission planning receives input information about the navigation area and its abnormalities, as well as required tasks and their constraints.
(2) Path planning calculates potential paths and traveling time for agents between every pair of milestones.
(3) TAMAA builds UTA models that are verified in UPPAAL. In case of successful verification, UPPAAL provides execution traces of the models that satisfy all task constraints. Subsequently, a task schedule is generated from the traces.
(4) Path planning checks whether the scheduled travels cross temporary obstacles when they are active. If a conflict is found, the planning returns to step 2 where the affected paths are updated.
(5) If both task and path constraints are satisfied, the iteration ends and a resulting mission plan is returned.

Figure 3 illustrates our solution. In the following subsections, we provide detailed descriptions



(a) An example of the autonomous quarry (b) A grid of the example

Fig. 4. An example of the autonomous quarry and a grid that discretizes the environment.

of the path planner (DALI) and of the task scheduler (TAMAA), as well as their integration. To illustrate the following models and algorithms, we use a running example based on the case study described in Section 2. Figure 4(a) depicts a small quarry, where an agent (i.e., an autonomous truck) originally located at milestone A should load stones at milestone B and deliver them to a crusher at milestone C. Two stationary obstacles are located in the quarry (brick walls in the figure). The area next to the stationary obstacle at the bottom is temporarily blocked. A parking station is an area that is recommended to be avoided. Moreover, road conditions are poor and the color scheme

indicates the speed reduction (white areas can be passed at full speed while red areas slow down the agents the most).

4.1 Improved DALı for Path Planning

The path planning algorithm is utilized to compute paths between all milestones. The traveling time of the paths is used by TAMAA for task scheduling. Note that not all paths are included in the final mission plan: only travels scheduled by TAMAA would be used. In the running example in Figure 4(a), the path planning algorithm computes paths between every pair of milestones, three in total (i.e., A to B, B to C, and A to C), yet the final mission plan would use only two of them (i.e., A to B and B to C).

Path planning has to be capable of providing navigation in complex environments, considering obstacles, road conditions, and users' additional preferences. The DALı algorithm supports different types of environmental constraints, thus we select it for our methodology. To adapt DALı to our use case, we transform DALı environmental constraints to be applicable in a quarry. In addition, we tune the algorithm with several optimizations as shown below.

The preliminary step of the DALı algorithm is a transformation of the navigation area into a graph and annotation of graph elements with environmental conditions. Figure 4(b) illustrates a discretization of the area with a Cartesian grid where each cell represents a node in the graph and neighbour cells are connected by edges. Each edge has a length equal to the distance between centers of cells connected by the edge. The initial positions of agents as well as milestones are assigned to the nodes corresponding to the cells that they are located at, respectively. Environmental constraints are encoded into the graph as follows.

- Permanent obstacles or areas that are always impassable (dark grey in Figure 4(b)) are excluded from the graph. There are no nodes corresponding to such areas and no edges connecting them.
- Temporary obstacles or areas that are impassable for a specified time interval (light orange in Figure 4(b)) are kept in the graph unlike permanent obstacles. Nodes in such areas are annotated with periods of inaccessibility that would be used by the path planning.
- Areas with bad road conditions are specified with a heat map used in DALı (green in Figure 4(b)). In the rest of the paper we refer to such areas as *heat areas*. Agents in a heat area have to reduce their respective speed by a given factor that is stored within the edges connecting nodes inside the area.
- Soft constraints marking some areas as "undesirable" (blue in Figure 4(b)) are defined differently from the soft constraints in DALı. Agents are allowed to pass through such areas however a virtual coefficient is added to lengths of edges making them less preferential to the path planner. Different from the original DALı where the coefficient depends on the distance to the area center, we consider the uniform coefficient in the whole undesirable area. In this way, we ensure that traversing the undesirable area even close to its boundary is discouraged. Note that, contrary to the heat areas that also affect the edges' lengths, this coefficient is virtual and only applied during the path finding but does not influence the actual traveling time on the paths.

Algorithm 1: Path Planning Algorithm

```
1  class Node
2      Edge edges[]                                                    // Outgoing edges
3      Double distance, vDistance          // Distance to source (virtual - with soft constraints)
4      Node previous                                                   // Previous node on the path
5      Double softConstraint                  // Soft constraint coefficient; 1 if no constraint
6      TimeInterval temporaryObstacle     // Inaccessibility time if inside a temporary obstacle
7
8  class Edge
9      Node start, end
10     Double length
11     Double heat                                  // heat map speed reduction factor; value ∈ [0, 1)
12
13 Function FindPath(Node[] area, Node source, Node target, Agent agent, Double startTime, Bool noTemp, Bool softExist)
14     PriorityQueue queue
15     foreach Node node : area do
16         node.distance := inf
17         queue.add(node)
18     source.distance := 0
19     Node processed[]
20     while queue ! = ∅ && target ∉ processed do
21         Node currentNode := queue.remove()
22         foreach Edge edge : currentNode.edges do
23             if noTemp || (edge.end ∈ processed || IsInsideTemporaryObstacle(edge, agent, startTime))
                 then
24                 skip
25             Double newDistance :=
                 currentNode.distance + softExist × edge.length × edge.end.softConstraint/(1 − edge.heat)
26             if edge.end.vDistance > newDistance then
27                 edge.end.vDistance := newDistance
28                 edge.end.previous := currentNode
29                 edge.end.distance := currentNode.distance + edge.length/(1 − edge.heat)
30             queue.add(edge.end)
31         processed.add(currentNode)
32     return ExtractPath(target)                  // traversal from target to source via node.previous
33
34 Function IsInsideTemporaryObstacle(Edge edge, Agent agent, Double startTime)
35     if edge.end.temporaryObstacle ≠ null then
36         Double time := startTime + (edge.length + edge.start.distance)/agent.speed
37         return time ∈ edge.end.temporaryObstacle
38     return False
```

The core step of the algorithm is a computation of the path between each pair of milestones on the created graph. The original DALI is a single-source single-target algorithm, i.e., it computes a path between a *source* and a *target*, and is based on Dijkstra's shortest path algorithm [D+59]. Algorithm 1 is called for each pair of milestones computing the shortest path between them while considering the environmental constraints. Each node stores a distance to the *source*, initially infinite. A priority queue orders nodes by their current distances to the *source* (line 14). The main loop (lines 20-31) takes a node with the smallest distance and updates the distances from all its

neighbours to the *source*. The update ensures the avoidance of temporary obstacles (lines 23 - 24) and calculates the distances taking in account soft constraints and heat areas (line 25). The algorithm terminates when the distance to the *target* node is computed and the path is obtained by moving in the graph via references to the previous nodes (line 32).

Within this work we propose two optimizations of the original DALI algorithm. The first optimization takes into account that our method requires to compute paths between every pair of milestones and that the Dijkstra's algorithm on which DALI is based can be modified into single-source multiple-target path planner. The extension of DALI is straightforward: the algorithm continues the main loop until the distances between every pair of milestones are computed. This optimization allows to reduce the number of calls to the path planner.

The second extension is inspired by the heuristics from the A* algorithm [HNR68]. Whenever the algorithm updates the distance of a node (line 25), it considers not only the distance to the *source* but also a distance estimation to the *target*. In particular, each node stores an estimation of the entire path length from the *source* to the *target* passing through the node and the priority queue sorts the nodes based on the estimation. For a node n the estimation is computed as $n.distance + dist(n,\ target)$, where $dist$ returns the Euclidean distance. Such heuristic guides the exploration of the graph towards the *target* and, in general, finds the shortest paths faster, especially in areas with few obstacles. In the remainer of the paper we call the DALI extension with the A* heuristic as DALI*. Note that the two extensions above are incompatible: an A* heuristic assumes that the algorithm is single-source single-target. The selection between the two optimizations must take into account the number of milestones, which affects the number of paths to compute, and the size of the Cartesian grid that discretizes the environment (Fig. 4(b)).

At the second step of the overall workflow (see the beginning of Section 4), a path planning algorithm computes the path between every pair of milestones. Since the path planner does not know at which time a path would be used, temporary obstacles are not considered during this step, i.e., the Boolean variable *noTemp* on line 23 is true. In our running example in Figure 4(b), the red path between A and B through the temporary obstacle would be selected. A path between B and C would not be direct: a heat area on the way would significantly slow down the agent and make a deviation faster. The path between A and C is also deviated due to the soft constraint, yet such path is far from optimal and may affect the timing constraints of the mission plan.

Temporary obstacles are considered during step 4 of the overall workflow (see the beginning of Section 4). At this point the selected paths and their starting time points are known, thus making it possible to check whether such obstacles have been passed at the wrong time when the temporary obstacles exist. If that is the case, such paths are recomputed with a consideration of the temporary obstacles (i.e., *noTemp* on line 23 is now false). In the running example (see Fig. 4(b)), the path between A and B is recomputed due to the temporary obstacle and a new and longer path avoiding both the temporary obstacle and the area with a soft constraint is computed. The new path and its length is then given to TAMAA for model generation.

It might be the case that paths avoiding areas with soft constraints are too long, which causes the timing constraint of a global mission to be violated. In this case, paths are recomputed by the path planner with soft constraints ineffective (i.e., `softExist` is false, which is converted to 0 at line 25). This recomputation may improve some paths and the updated traveling time is passed to TAMAA for task rescheduling. In the running example, such recomputation would output yet another path between A and B.

4.2 TAMAA for Task Scheduling

Now, we recall the mission-related requirements of agents, mentioned in Section 2, which agents need to fulfill. To ensure the correct work of the agents, we need a method that guarantees to meet

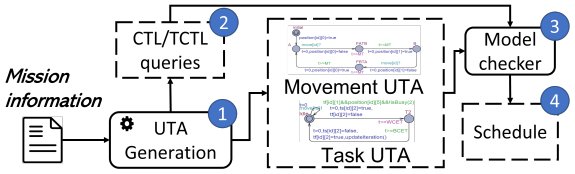


Fig. 5. Overview of the workflow of model generation and mission plan synthesis in TAMAA

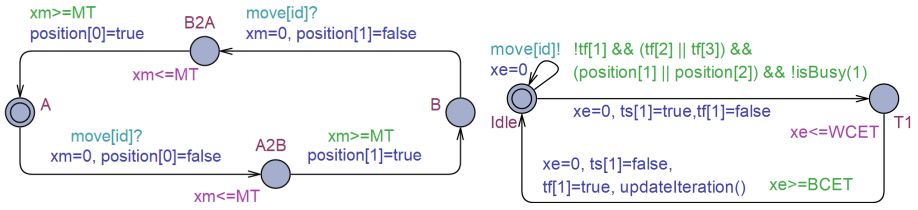
all the desired requirements, including correct task scheduling. Model checking can provide such guarantees given a modeling formalism for agents and constraints. We propose a method named *TAMAA (Timed-Automata-based Mission planner for Autonomous Agants)* that is able to automatically generate models and synthesize task schedules that satisfy the formalized requirements by using model checking. TAMAA employs UPPAAL as the model checker that uses the UPPAAL timed automata (UTA) formalism for modeling timed behavior of agents and (Timed) Computation Tree Language ((T)CTL) for requirement specification.

Figure 5 depicts the workflow of our approach:

(1) *UTA generation*: mission information, e.g., topology of the map, and information about the agents and their tasks, are input into the model-generation module, where a set of UTA that models the agents' movement and task execution are generated automatically.
(2) *Query generation*: (T)CTL queries for synthesizing schedules are generated automatically. Based on our templates of queries, users can manually modify the queries according to their own requirements.
(3) *Trace generation*: The UTA models are verified with UPPAAL against the (T)CTL queries. If the model checker finds an execution trace of the model that meets all the requirements, the trace is returned; otherwise, a verdict that the query is not satisfied is returned.
(4) *Schedule generation*: The returned trace is parsed and a schedule of actions (i.e., movement and task execution) is generated based on the trace.

The method automatically generates models, formalizes requirements, and synthesizes traces. The TAMAA method's automation of the process simplifies its application: no user action is required after setting up the tasks, milestones, and navigation area. Nevertheless, we leave the possibility to modify models and to add requirements so that our method can be used in applications with individual needs that are not expressible with the existing models. We introduce this in detail in Section 7. In addition, as the traces are generated by exhaustive traversal of the model state space, we can select the fastest ones that finish the tasks while holding the other requirements, e.g., *milestone matching*. In the following, we introduce the theoretical and technical details of TAMAA including formal definitions of the concepts, two algorithms for model generation, and the templates of queries that formalize the requirements presented in Section 2.

4.2.1 Definitions of Concepts. In our approach, autonomous agents are characterized by their speeds and a set of tasks that they are supposed to execute for accomplishing the entire mission. The environment where agents work in contains a number of milestones where the tasks are supposed to be carried out. Therefore, agents with a certain subset of tasks should visit the right milestones. To accomplish the mission, there are two types of actions that agents can perform, namely *moving* and *executing* tasks. Therefore, we split the agent model into two UTA, one taking care of the movement and one of the task execution.



(a) Agent movement as an UTA (b) Agent execution of tasks as an UTA

Fig. 6. Examples of agent movement UTA and task-execution UTA.

Definition 4 (Agent's Movement UTA). *Given an autonomous agent AA, the movement of AA is defined as a UTA in the following form:*

$$MV = < P_m, p_0, U_m, \Sigma_m, E_m, I_m >, \tag{4}$$

where:

- $P_m = P_m^s \cup P_m^t$ *is a finite set of locations, where* P_m^s *represents the set of milestones in the environment, and* P_m^t *is designed for measuring the traveling time between milestones;*
- $p_0 \in P_m^s$ *is an initial location representing the milestone where the agent is initially positioned;*
- $U_m = \{x_m, position\}$*, where* x_m *is a clock variable for measuring the traveling time, and "position", which is shared with other UTA, is an array of Boolean variables that stores whether the agent is at a milestone or not;*
- $\Sigma_m = \{move, \tau\}$ *is a set of channels, where "move" models the synchronization with task execution automaton described below in Definition 6, and* τ *denotes internal or empty actions without synchronization;*
- $E_m = E_m^c \cup E_m^u$ *is a finite set of edges connecting locations, where* $E_m^c \subseteq P_m^s \times \{true\} \times \{move\} \times F_m \times P_m^t$*, where* F_m *is a set of functions that update the value of the Boolean array "position" and reset the clock* x_m*, and* $E_m^u \subseteq P_m^t \times B_m(x_m) \times F_m \times P_m^s$*, where* $B_m(x_m)$ *is a set of guards containing clock constraints of the form* $x_m \geq \delta$*, where* $\delta \in \mathbb{R}_{\geq 0}$ *is the traveling time between two milestones;*
- $I_m : P_m^t \rightarrow B_t(x_m)$ *is a function that assigns invariants to locations in* P_m^t*, where* $B_t(x_m)$ *contains clock constraints of the form* $x_m \leq \delta$*, where* $\delta \in \mathbb{R}_{\geq 0}$ *is the traveling time between two milestones.*

Figure 6(a) illustrates the movement UTA for two milestones A and B. The UTA has four *locations*: two *milestone* locations representing the milestones, i.e., $P_m^s = \{A, B\}$, and two *traveling* locations representing intermediate positions necessary to capture traveling between milestones, i.e., $P_m^t = \{A2B, B2A\}$. Edges in E_m^c that connect the milestone locations with the traveling locations are labeled with a *channel* `move`, in our example, the edges from *locations* `A` to `A2B` and from *locations* `B` to `B2A`, respectively. The *channel* `move` is for synchronizing the movement UTA with the task execution UTA when the latter informs that the movement has started. A second group of edges (i.e., E_m^u) models the arrival to the destination. To model the traveling time we use invariants of the form $x_m \leq$ `MT` and guards in the form of $x_m \geq$ `MT`. The clock variable x_m is for measuring the traveling time between two milestones. The set F_m in this UTA contains the assignments that reset the clock x_m and modify a specific element in the Boolean array `position`. We use the array `position` to keep track of current position of an agent, which can be accessed by other UTA.

The second part of an agent model formalizes task execution. First, we define tasks as follows:

Definition 5 (Task). *A task is defined as a tuple, as follows:*

$$TS = (\texttt{BCET}, \texttt{WCET}, \texttt{isStarted}, \texttt{isFinished}, \texttt{Var}, \texttt{pre}, \texttt{Mil}), \tag{5}$$

where:

- BCET *is the best case execution time;*
- WCET *is the worst case execution time;*
- isStarted *is a Boolean variable denoting if the task has started;*
- isFinished *is a Boolean variable denoting if the task has finished;*
- Var *is a set of functions that update the variables in the task-execution UTA (Definition 6) after the task finishes;*
- pre *is a precondition that must be met to start the task, which can take into account execution status of other tasks and global variables. Formally,* $\texttt{pre} = p \mid \neg\texttt{pre} \mid \texttt{pre} \vee \texttt{pre} \mid \texttt{pre} \wedge \texttt{pre}$*, where* p *is an atomic proposition over* $\mathcal{S} \cup \mathcal{F} \cup \mathcal{V}$*, where* $\mathcal{S}$*,* $\mathcal{F}$*, and* $\mathcal{V}$ *are three sets consisting of Boolean variables* isStarted*,* isFinished*, and the variables updated in* Var *of all tasks, respectively;*
- Mil *is a set of milestones where the task is allowed to be executed. A task can be executed at multiple milestones.*

Table 1 shows two examples of tasks, which we use to illustrate Definition 5.

Table 1. An example of tasks for the autonomous trucks in Figure 4.

	BCET (mins)	WCET (mins)	isStarted	isFinished	Var	pre	Mil
Loading	5	10	false	false	full	/	B
Unloading	8	14	false	false	unload(stone)	Loading.isFinished	C

Initially, an agent starts to load stones near the stone piles at milestone B (*loading.isStarted* turns *true*). When the loading task is finished (*Loading.isFinished* turns *true*, while *Loading.Started* turns *false*), a Boolean variable *full* indicating whether the agent is fully loaded is set to *true*. At the next step, the agents are supposed to unload the stones into the primary crusher at milestone C. Therefore, the unloading task has a precondition *loading.isFinished*. Note that preconditions can be much more complex Boolean expressions, e.g., $A.pre = (B.isFinished \;||\; \neg C.isStarted)\;\&\; D.isStarted$, which means task A can start only when task B is done or task C has not started, and after task D has started. When the unloading task is done, an integer named *stone* is increased, which indicates the amount of crushed stones.

The execution time of a task is usually specified with a time interval (i.e., [BCET, WCET]). In a 1-player game [Jen18], the environment is under the total control of agents, which means agents can choose any time point within the time interval to finish the task. The goal of task scheduling in this paper is to find the schedules that finish tasks in the quickest way. Notably, the quickest schedules do not necessarily mean always using BCET for all tasks. In some cases, prolonging the execution time of some tasks can be more efficient for the entire mission, especially when multiple agents are working in the same environment. Additionally, some applications require the tasks to be executed multiple times before the final goal is reached. Like the example in Table 1, autonomous trucks are asked to repeat the trip of loading and unloading until all the stones are transferred to the primary crusher (i.e., variable *stone* is zero), and the tasks should be executed once and only once during one trip, i.e., before all tasks are completed. With all the requirements, how to assign starting time and ending time of tasks to each of the agents is the question that is answered by task scheduling. Now, we define the task execution UTA based on the definition of tasks.

Definition 6 (Task Execution UTA). *Given an agent AA and a set of tasks $\mathcal{T}$, task execution of AA is defined as a UTA of the form:*

$$TE = (N_e, n_0, U_e, \Sigma_e, E_e, I_e), \tag{6}$$

where:

- $N_e = \{n_0\} \cup N_e^t$ *is a set of locations, where* $N_e^t = \{n_t \mid t \in \mathcal{T}\}$;
- $n_0 \in N_e$ *is the initial location, which stands for the idle status of AA;*
- $U_e = \{x_e, t_s, t_f, ite\}$, *where* x_e *is a clock that is reset whenever a task finishes,* t_s *and* t_f *are Boolean arrays that store the statuses of tasks (isStarted and isFinished in Definition 5, respectively), and* $ite \in \mathbb{N}$ *stores the current iterations of all tasks. At the end of each iteration,* ite *is incremented by 1, while* t_s *and* t_f *are reset to false for a new round of tasks execution;*
- $\Sigma_e = \{move, \tau\}$ *is a set of channels;*
- $E_e = E_i^d \cup E_e^d \cup \{e_s\}$, *where* $E_i^d \subseteq \{n_0\} \times \{\tau\} \times B_e^i(U_e) \times F_e \times N_e^t$ *is a set of edges from* n_0 *to* $n_t \in N_e^t$, F_e *is a set of functions updating the variables in* U_e, $B_e^i(U_e)$ *is a set of guards consisting of the preconditions and milestone requirements of tasks,* $E_e^d \subseteq N_e^t \times \{\tau\} \times B_e^d(x_e) \times F_e \times \{n_0\}$ *is a set of edges from* $n_t \in N_e^t$ *to* n_0, $B_e^d(x_e)$ *is a set of guards containing clock constraints of the form* $x_e \geq BCET$ *of a task,* e_s *is a self-loop edge on* n_0 *that is labeled with channel move and an assignment* $x_e = 0$;
- $I_e : N_e^t \to B_i(x_e)$ *is a function assigning invariants to locations in* N_e^t. *The invariants are of form* $x_e \leq WCET$ *of a task.*

An example of the task execution UTA is depicted in Figure 6(b), where *location* `Idle` is the initial *location* n_0, and *location* `T1` represents a task. The *channel* `move` labels the self-loop *edge* on *location* `Idle`, which synchronizes the movement UTA and the task execution UTA. Due to the synchronization, movement cannot start during task execution but only when the agent is idle. *Location* `T1` and its outgoing *edge* are labeled with an invariant (`t<=WCET`) and a *guard* (`t>=BCET`), respectively, to ensure the execution time within *BCET* and *WCET*. Function `updateIteration` on the edge from `T1` to `Idle` is for incrementing the variable *ite* when an agent finishes all its tasks. For a specific case, additional functions can be added to this edge too, for example, the function `unload(stone)` in Table 1. Moreover, the guard `!tf[1]` forbids the multiple execution of the task during a single round of tasks iteration; the next execution of this task can be done only in the next iteration after the reset of the arrays `ts` and `tf`. The guard on the edge from *Idle* to `T1` presents the precondition (i.e., `!tf[1] && (tf[2] || tf[3])`), the milestone requirement (i.e., `position[1] || position[2]`), and the mutual-exclusive requirement of task *T1* (i.e., `!isBusy(1)`). The function `isBusy(1)` checks if `T1` is being executed by other agents or not. As preconditions defined in Definition 5, the function `isBusy(1)` uses the "isStarted" and "isFinished" variables of the same task executed by other agents to see if task *T1* is being executed.

Definitions 4 and 6 define the models of agent movement and task execution, respectively. A network of these UTA models the behavior of multiple agents working collectively to reach a common goal respecting a given set of constraints, such as mutual-exclusiveness of tasks. The definitions present the foundation for describing multi-agent systems with a formal modeling language. Next, we introduce how the course of modeling is automated by two algorithms.

4.2.2 Generation of UTA. In this section, we introduce the algorithms generating the movement and task execution UTA.

Figure 6 illustrates the results of algorithms application. Movement UTA (Definition 4) of an agent is generated by Algorithm 2. Its inputs are a two-dimensional array called *map*, which stores the distance between every pair of milestones, and a variable called *speed*, which is the agent's speed. The algorithm starts by creating a location for each milestone (lines 4 - 7). Next, traveling

between each pair of milestones is represented by a *traveling location* (lines 13 - 14) and two edges connecting the milestones via the traveling *location* (lines 16 - 23). *Guards*, *channels*, and *invariants* are assigned to location and edges according to Definition 4.

Algorithm 2: Movement UTA Generation

```
1  Function CreateMovementUTA(int[][] map, int speed)
2      UTA movementUTA
3      int i := 0, j := 0
4      for i < map.size do
5          Location l_i := createLocation("L_i")                    // Create a location with the name L_i
6          movementUTA.addLocation(l_i)
7          i++
8      i := 0
9      for i < map.size do
10         Location l_i := movementUTA.getLocation("L_i")           // Get a location with the name L_i
11         for j < map[i].size && i ≠ j do
12             Location l_j := movementUTA.getLocation("L_j")       // Get a location with the name L_j
13             Location l_FiTj := createLocation("F_iT_j")          // Create a location with the name F_iT_j
14             l_FiTj.invariant = "x_m ≤ map[i][j]/speed"
15             movementUTA.addLocation(l_FiTj)
16             Edge e_1 := createEdge(l_i, l_FiTj)                  // Create an edge from l_i to l_FiTj
17             e_1.channel := "move?"
18             e_1.assignments := "x_m = 0, position[i] = false"
19             movementUTA.addEdge(e_1)
20             Edge e_2 := createEdge(l_FiTj, l_j)                  // Create an edge from l_FiTj to l_j
21             e_2.guard := "x_m ≥ map[i][j]/speed"
22             e_2.assignments := "x_m = 0, position[j] = true"
23             movementUTA.addEdge(e_2)
24             j++
25         i++
26     return movementUTA
```

Algorithm 3 describes the generation of an agent's task execution UTA. Lines 3 - 8 create a *location* to represent the idle status of the agent and a self loop at this *location* to synchronize with the agent's movement UTA. This is a single point of synchronization between the two automata since task execution and movement are mutually exclusive. Lines 10 - 20 create *locations* representing the execution of each task and edges that connect these *locations* with the `idle` *location*. Line 14 assigns a *guard* to the *edges* coming from the *location* `Idle`. The *guard* regulates the UTA to start the task only when the task's precondition holds ($tasks[i].Pre$) and the agent is positioned at one of the right milestones ($(\bigvee_{l \in tasks[i].Mil} position[l])$), where $position$ is an array whose values are changed by the movement UTA of the agent. The guard $\neg tf[i]$ means that the task is not finished yet in this round of task iteration. The function $updateIteration()$ in line 19 increments the variable ite belonging to this UTA if all tasks of the corresponding agent are finished, and turns the variables in tf and ts to $false$, indicating a new round of iteration is about to start.

Algorithm 3: Task-execution UTA Generation

```
1  Function CreateTaskUTA(int agentID, TS tasks[])
2      UTA taskexeUTA
3      Location Idle := createLocation("Idle")                    // Create a location with the name Idle
4      taskexeUTA.addLocation(Idle)
5      e0 := createEdge(Idle, Idle)                               // Create a self-loop edge of location Idle
6      e0.channel := "move[agentID]!"
7      e0.assignment := "xe = 0"
8      taskexeUTA.addEdge(e0)
9      for i < tasks.size do
10         Location Ti := createLocation(Ti)                      // Create a location with the name Ti
11         Ti.invariant := xe ≤ tasks[i].WCET
12         taskexeUTA.addLocation(Ti)
13         Edge e1 := createEdge(Idle, Ti)                        // Create an edge from Idle to Ti
14         e1.guard := "¬tf[i] ∧ tasks[i].Pre ∧ (⋁l∈tasks[i].Mil position[l])∧!isBusy(i)"
15         e1.assignments := "x: = 0, ts[i] = true, tf[i] = false"
16         taskexeUTA.addEdge(e1)
17         Edge e2 := createEdge(Ti, Idle)                        // Create an edge from Ti to Idle
18         e2.guard := "xe ≥ tasks[i].BCET"
19         e2.assignment := "xe = 0, ts[i] = false, tf[i] := true, updateIteration()"
20         taskexeUTA.addEdge(e2)
21         i++
22     return taskexeUTA
```

4.2.3 Formalizing Requirements as UPPAAL Queries. In Section 2, we provide a list of typical requirements of our use case. In this section we describe how they are formalized for the model checking. Queries for verification of each agent are created based on the templates presented below (formulas (7) to (10)). We use index 'a' to denote an agent in queries. We use $\texttt{task}_a$ (respectively, $\texttt{move}_a$) to denote the task execution UTA (respectively, movement UTA) of an agent a, and Ti (respectively, Pi) to denote any *location* in the task execution UTA (respectively, movement UTA). A clock variable x is used to measure the global time. Two Boolean arrays named ts and tf indicate whether tasks have been started and finished, respectively. Two constant integers ALL and LIMIT denote the requested number of iterations of tasks and the time constraint to accomplish the entire mission, respectively.

- *Milestone matching*: Agents must be located at the right milestone while executing a task. Assuming task T_i must be carried out at one of the milestones: P_i, P_{i+1}, ..., P_k, the following queries are checked for each agent a:

$$\texttt{E<> ts}_a\texttt{[i]} \tag{7}$$

$$\texttt{A[] task}_a\texttt{.Ti imply (move}_a\texttt{.Pi || ... || move}_a\texttt{.Pk)} \tag{8}$$

  Query (7) is for verifying whether task T_i ever starts, after which Query (8) checks whether task T_i is carried out at the right milestone.
- *Task sequence*: Tasks must eventually be executed, and to start the execution, their pre-conditions must be satisfied. Assuming task T_i can start only after task T_j finishes, and at the beginning of each task iteration, all elements in both ts and tf are set to *false*, the corresponding queries are designed:

$$\texttt{A[] ts}_a\texttt{[i] imply tf}_a\texttt{[j]} \tag{9}$$

The first part of the requirement, i.e., whether task T_i ever starts, is verified in the requirement of *milestone matching* by checking Query (7). Query (9) checks the second part of the requirement: task T_i never starts before the required preceding task T_j has finished.

- *Timing*: Tasks must be finished within a time frame in order to maintain a certain level of productivity. The following query is used to capture this requirement:

$$\texttt{E<> iteration[a]>=ALL and gClock <= LIMIT}, \tag{10}$$

where *gClock* is a global clock that is not reset by any transition. Query (10) checks the reachability of a state where all tasks are executed for `ALL` rounds within `LIMIT` time units, and UPPAAL returns the trace reaching that state, in case the query is satisfied.

Note that satisfaction of Queries (7-9) is guaranteed by the construction of movement and task execution UTA. The queries can still be verified for a given mission, but they are not used in the mission planning. If Query (10) is satisfied, UPPAAL outputs the trace reaching the goal state. The trace is processed by TAMAA and a schedule is generated following the steps of the trace. We always look for the fastest trace. When the task execution time is a time interval rather than a fixed value, we assume the environment to be under the control of the agents, which means that the agents can choose any task execution time during the time intervals, respectively. If the environment reacts competitively or possibly antagonistically, we need a comprehensive schedule that considers all possible scenarios. We refer the interested readers to our previous work [GJP+22], in which we propose a method combining model checking and reinforcement learning to deal with uncooperative environments.

4.3 Mission Planning with DALı and TAMAA

Path planning and task scheduling aspects of the mission planning depend on each other. One of the task scheduling parameters is traveling time between pairs of milestones. Temporary obstacles affect shortest paths between milestones, therefore path planning requires to know the starting time points of travels in order to ensure avoidance of temporary obstacles. Therefore, mission planning might re-run the path-planning and task-scheduling modules multiple times until all requirements for the mission are satisfied: changing paths affect the traveling time and, consequently, might affect the task schedule; changes in the task schedule modify the starting time points of travels and, consequently, might affect the paths. In addition, task scheduler might find impossible to satisfy timing constraints with the given traveling time. In this case, DALı can attempt to improve the traveling time by ignoring soft constraints and calling TAMAA to reschedule tasks with the new traveling time as the input.

We illustrate the steps of the Algorithm 4 with the running example depicted in Figure 4(a). During the first step (line 5), DALı discretizes the entire environment into a Cartesian grid shown in Figure 4(b) and builds a graph of the environment which is used in all consequent path-planning calls. DALı computes the path between every pair of milestones while ignoring temporary obstacles during the first computation (lines 7-9). Indeed, at this point neither the order nor the starting time points of travels are known and the shortest paths between every two milestones are returned. In Figure 4(b), the red path between milestones A and B would be selected as the shortest even if it passes through a temporary obstacle (orange cells).

At the next step, the model generation module of TAMAA automatically generates the network of UTA that models the agents' movement and task execution (lines 10-13), after which a function named `Scheduling` is invoked to schedule the tasks taking into account the traveling time of the paths (line 14). In the `Scheduling` function, UPPAAL checks the existence of a model execution trace satisfying the Query (10) (line 32). Note that queries that formalize other requirements, e.g., Queries (7) - (9), can also be checked, which ensure the satisfaction of other requirements but do

not contribute to the mission plan synthesis. Next, if the query is satisfied and a trace is obtained, we convert the trace into a schedule ordering the movement and task execution actions (line 34). In the `traceParser` function, a schedule that orders the actions of movement and task execution for all the agents are generated and returned to the main function of mission planning (line 14). Note that the returned result of schedule can be empty (i.e., `null`), which indicates the non-existence of mission plans. Since the model checker exhaustively explores the entire state space of the model, the non-existence of mission plans is guaranteed. In such case, the mission planning tries to recompute the paths and the schedule ignoring soft constraints (line 19). If the Query (10) cannot be satisfied even without soft constraints (line 21), the algorithm terminates and suggests the users to modify their environment configuration or requirements.

Algorithm 4: Mission Planning Algorithm

```
1  Function MissionPlanning(Environment env, TS tasks[], Agent agents[], Query query)
2      NUTA model                                                    // A network of UTA
3      Bool noTemp := true
4      Bool softExist = true
5      Node area[] := CreateGrid(env)      // Create a discrete grid of the continuous environment
6      double startTimes[][] := 0
7      foreach agent : agents do
8          foreach m1, m2 : milestones do
9              agent.paths[m1][m2] :=
                 FindPath(area, m1, m2, agent, startTimes[m1][m2], noTemp, softExist)
10     foreach agent : agents do
11         UTA movement := CreateMovementUTA(agent.paths, agent.speed)
12         UTA taskExe := CreateTaskUTA(agent.ID, tasks)
13         model := compose(movement, taskExe)        // Compose the models into a network of UTA
14     Schedule schedule := Scheduling(model, query)
15     if schedule == null then
16         if softExist then
17             softExist := false
18             double startTimes[][] := 0
19             goto line 7                // Recompute paths and schedule ignoring soft constraints
20         else
21             return false                                          // No mission plan found
22     if !CheckTemporaryObstacles(schedule) then
23         noTemp := false
24         updateTimes(startTimes, schedule)                 // Update the start time of each path
25         goto line 7                                      // Recompute affected paths with DALi
26     foreach agent : agents do
27         agent.schedule := distribute(schedule)  // Dissolve and distribute the plan to each agent
28     return true
29
30 Function Scheduling(NUTA model, Query query)
31     Schedule schedule := null                                  // Stores the order of actions
32     Trace trace = check(model, query)     // Calls UPPAAL to check the model against Query (10)
33     if trace ≠ null then
34         schedule = traceParser(trace)      // Parse the trace and generate a resulting schedule
35     return schedule
```

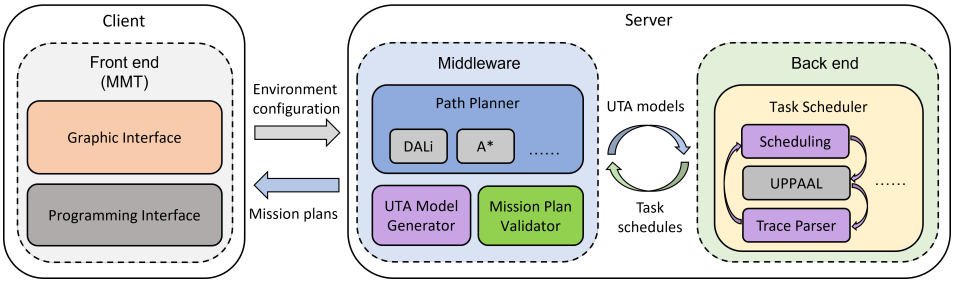


Fig. 7. Architecture and the information flow of the toolset

If the `Scheduling` function returns a non-empty result, DALi checks whether the plan happens to cross any temporary anomalies and at which time (line 22). The function *CheckTemporaryObstacles* looks at all the paths involved in the schedule, and, with the knowledge of the starting time of travels, checks that no temporary obstacle is entered during its inaccessibility period. If at least one path crosses a temporary obstacle during its existence, recomputation is run (line 25). Given scheduled starting time of actions, DALi updates paths that used to enter the temporary obstacles. New paths might become longer than the original paths, requiring TAMAA to reschedule tasks. For example, in Fig. 4(b), if a temporary obstacle is enabled then a different path between A and B is selected (dashed black lines). The new path is longer and might affect the satisfaction of timing constraints. In this case, another path is computed ignoring the soft constraint (solid line). TAMAA is called after each path modification and regenerates a new task schedule (line 14).

Next, the entire plan of actions are dissolved into several individual mission plans for the agents and distributed to each of them (line 27). The function `distribute` dissolves the entire schedule and puts the actions into the corresponding individual mission plan according to which agent they belong to. Finally, a mission plan that satisfies all constraints imposed on the tasks and paths is returned by the mission planner that integrates DALi and TAMAA. Again, since the satisfaction of constraints is guaranteed by the model checking technique used in the `Scheduling` function, the resulting mission plan is *correct-by-construction*.

5 DESCRIPTION OF THE TOOL

In this section, we present our toolset called *MALTA*[1] that consists of three main components: a GUI called Mission Management Tool (MMT), a path planner implementing DALi, and a task scheduler implementing TAMAA.

5.1 Overall Description

The toolset is built of 3 parts: a front end providing the graphic user interface (GUI), a middleware providing path planning and building mission plans from paths and schedules, and a back end dedicated to task scheduling. The toolset design adopts a Client/Server architecture. The reason is twofold: first, the front end of the toolset is a GUI that has been independently designed. Beside the GUI, the front end also provides a group of programming interfaces and data structures for extension and communication. Therefore, the front end is open for extension without touching its code. Second, the computation of mission plans can be quite expensive. As we show in Section 6, synthesizing mission plans for multiple agents can cost hours on a computationally powerful server.

[1] *MALTA* installation package and source code of path planner and task scheduler can be found at https://github.com/rgu01/MALTA

Therefore, the separation of front end GUI from the mission plan synthesis allows the users to move computations to a dedicated server, which is a user-friendly and efficient design pattern, also easy to maintain.

In the front end, users can configure their environment including the navigation areas, milestones, tasks, agents, etc., after which, the environmental configuration is transferred to the middleware. The *Mission Management Tool (MMT)* described in the following subsections provides the front end GUI.

The middleware receives the environmental configuration and passes it to a path planner. Any path planning algorithm that supports the desired environmental constraints can be used. In our implementation we offer a choice between A* and DALI algorithms described in Subsection 4.1. At the second step the middleware generates UTA models following the Algorithms 2 and 3. The *UTA model generator* is based on an open source library *j2uppaal*[2].

These UTA models are transferred to the back end, where we implement the TAMAA scheduler to synthesize schedules. The `Scheduling` module implementing the `Scheduling` function in Algorithm 4 invokes the model checker UPPAAL to check the UTA models against Query (10) and, in case of successful verification, UPPAAL generates an execution trace containing the sequence of actions that can be translated into a schedule by the *trace parser*, which uses the library for parsing traces provided by UPPAAL[3]. The back end can use other model-checking-based schedulers, for example missions in uncertain environment could use another scheduler combining model checking and reinforcement learning [GESL20, GJP+22].

The resulting schedule is stored as a standard format of Extensible Markup Language (XML) and sent to the middleware, where the schedule is combined with the paths to generate a mission plan. A module called *Mission Plan Validator* is designed to check if the mission plan happens to come across the temporary obstacles when they still exist. If the collision does happen, a new path plan that circumvents the collision is calculated by the *Path Planner* and the corresponding UTA models that reflect the new paths are generated and sent to the task scheduler again. The iteration of computation continues until a valid mission plan is generated or no valid path exists. The final mission plans are shown in the frond end when the *Mission Plan Validator* confirms that the results are correct, or a warning informs the users that no mission plan can be generated and suggests a configuration modification.

In the following subsections we introduce MALTA's GUI and demonstrate how can one configure the environment, and visualize the resulting mission plan.

5.2 Mission Management Tool

The Mission Management Tool (MMT) is a GUI that allows the operator to plan, execute and supervise missions involving multiple autonomous vehicles [ACME20]. In the context of this paper the focus is mission definition and plan visualization, hence we do not discuss the plan execution and supervision functionalities.

MMT's main window contains five main panels: (A) mission explorer, (B) assets, (C) properties, (D) map, and (E) plan outline (Fig. 8). The map shows an overall view of the mission area and the vehicles. It also provides tools to the operator for defining areas of interest for a mission. The mission explorer (Fig. 8.A) represents different assets involved in a mission and their relationship in a tree structure. The assets panel contains three sub-panels that contain vehicles, locations and tasks. These are either physical assets that the operator has access to (vehicles), or entities that the operator has defined himself/herself (tasks and locations). The properties panel shows different

[2]https://github.com/predragf/org.fmaes.j2uppaal
[3]https://github.com/UPPAALModelChecker/utap/wiki

properties of a selected asset and allows the operator to set their values if necessary. Finally the plan outline provides a Gantt chart representation of the whole plan for a mission (Fig. 9). MMT

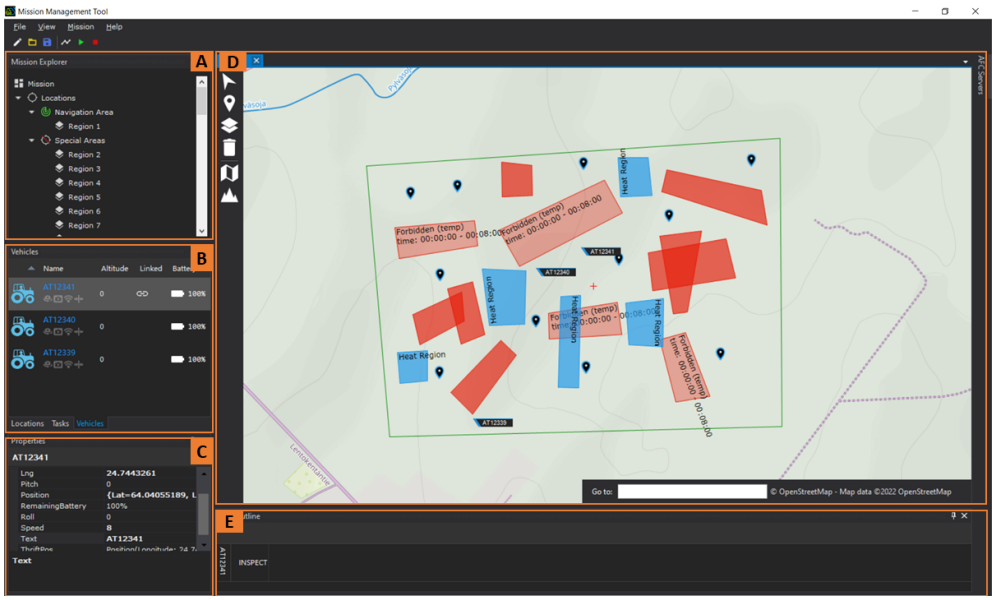


Fig. 8. Mission Management Tool. (A) Mission Explorer, (B) Assets, (C) Properties, (D) Map and (E) Plan Outline.

communicates with the mission planner through the Apache Thrift Framework[4]. This allows MMT and the planner to share definitions of different assets and communicate mission data as well as the final mission plan. Since the version of Apache Thrift that we use (0.9.3) does not fully support asynchronous communication for all the programming languages involved in this work, both MMT and the Planner provide one-way Thrift services to each other for communication. This means that upon a function call through Thrift framework, the client does not wait for a response and instead it provides it own Thrift server for receiving the results (the plan) whenever they are ready. Hence, the planners run a Thrift server that exposes functions allowing the MMT to send a mission definition to the planner and request a mission plan for it. MMT on the other hand, runs a Thrift server that includes a function, allowing the planner to send the final plan back to MMT.

5.3 Environmental Configuration with MMT

Mission definition through MMT is done in two steps: First the mission assets and their properties need to be defined, and next the relationship between these assets should be configured. Mission assets consist of vehicles, tasks and locations. Vehicles are not defined by the operator as they are real entities that are discovered by MMT. In case of vehicles, the operator can set some of their properties that might affect the visualization of vehicle on MMT (i.e. color) or a vehicle's performance in a mission (i.e. speed).

The two other types of assets need to be defined by the users. Defining locations/areas of interest for a mission takes place through the map. The operator can use map tools to set markers on the map or draw regions on it. The locations/areas are added to the locations sub-panel, where they

[4]https://thrift.apache.org/

can be accessed to set their visualization or mission-related properties. For an area, there are some important properties that can be set, which affect the planning phase:

- *Region Type*: This can be set to a forbidden area (vehicles should not go through that region), a preferred area (vehicles are preferred to go through that region), a less preferred area (vehicles can go through there but it is better to avoid it) and a heat area (hard to pass area).
- *Intensity*: This parameter indicates the intensity of heat areas and (less-)preferred areas. For heat areas it affects the speed drop in the area; for (less-)preferred areas it affects the decision for entering the areas.
- *Start and End Time*: These parameters define the time interval relative to the mission start when the area is active, e.g., temporary obstacles appear and last a while.

Defining the tasks of a mission is done through the tasks sub-panel. This panel allows the operator to define a new task or re-use a predefined one. Defining new tasks requires the operator to define the task type (either *inspect* or *survey*) and the equipment required for it. Inspect tasks are tasks that should be performed on a location (i.e. digging), whereas, survey tasks are performed on a whole area (i.e. spraying a field).

When all the assets are defined, the operator can define the mission by dragging and dropping the assets to the mission explorer. The mission explorer contains several entry points for mission assets as follows:

- *Navigation Area*: This is the main area of the mission. No vehicle will be allowed to move outside this area. Navigation area is visualized as a polygon with green borders in MMT (Fig. 8.D).
- *Special Areas*: These are the areas that have specific roles in the mission but are not part of a task. Examples include: forbidden areas, preferred areas, and heat areas. In MMT, forbidden areas are displayed with a red background and if they are temporarily forbidden a lighter shade of red is used. Temporarily forbidden areas are also annotated with a timespan during which they become inaccessible by the vehicles. Heat and preferred areas are displayed in light blue (Fig. 8.D).
- *Task Areas*: These are areas that are related to a task. For each task involved in the mission a new entry point is added, allowing the operator to add the locations/areas related to it. It is possible to add several locations/areas to a single task. If a task requires a specific sensor/actuator, this information is also written next to it on the map (Fig. 8.D).
- *Home Locations*: These are the locations where the vehicles should move to after finishing their mission.
- *Vehicles*: This part contains all the vehicles that are allowed to participate in the mission. This means that the operator is allowed to only drag some of the vehicles to this section which in turn means the planner can only use those for planning (Fig. 8.B).
- *Tasks*: This contains a list of all the tasks that should be performed in the mission. When the operator drags a task to this section, a new sub-section for this task is also added to the *Task Areas* section, allowing the operator to define the areas for this specific task (Fig. 8.B).

The final step in mission definition is defining location/area properties after they are added to a task. Please note that these properties are not directly bound to the location or the task itself, but are properties that represent that specific task while being done at that specific location. These properties are only accessible by clicking on the location/area under the task in task areas section of mission explorer (Fig. 8.C). These properties allow the operator to define task preconditions, BCET and WCET.

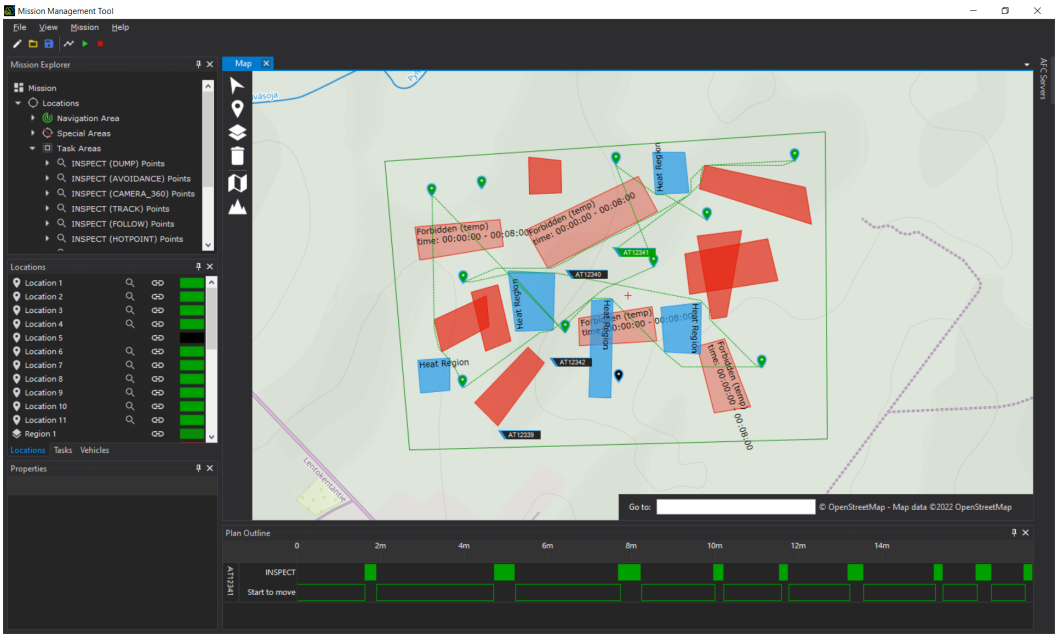


Fig. 9. Mission Management Tool.

After defining the mission, the operator can save it for later use or send it to the planner using the plan button on the toolbar. This sends the plan to the planner and awaits for a result. The final mission plan is then visualized on the map and plan outline.

5.4 Mission Plan Demonstration in MMT

Fig. 9 shows a plan visualized in MMT. The plan contains only one vehicle and nine tasks. The vehicle and its related path are color-coded (in this case they are all green). In cases with multiple vehicles, each vehicle, its related path and tasks will get a separate color. It can be observed that red areas are always avoided by the green lines (vehicle's path), while the light-red areas are sometimes avoided (during the related time span) and sometimes crossed (when the area is not forbidden anymore). The blue regions indicate areas with bad roads: vehicles can pass them but at a slower speed. The plan outline at the bottom also shows the order of actions to be taken by the vehicle and the amount of time each action consumes.

The mission starts from the initial location of the vehicle and the path shows the milestones that the vehicle will visit during its mission. Some milestones are visited several times as this is part of the task and mission definition. The milestone locations are marked with markers matching the color of the vehicle and a text shows the actual action which will be performed at that location.

6 EVALUATION

To test our prototype implementation we conducted a series of experiments directed to evaluate the performance and scalability of the proposed approach[5]. In this section, we present the design and results of the experiments.

[5]The mission configurations of the experiments are published so that one can replicate the experimental results: https://github.com/rgu01/MALTA.

6.1 Methodology

For the experiments we have created a mission shown in Figure 8 involving multiple milestones, permanent and temporary obstacles, and heat areas. The following parameters have been varied in the experiments.

- *Path-planning algorithm*: we compared A* algorithm, DALI without optimizations, and DALI* discussed in Section 4. We refer to the version without optimizations as DALI. All three algorithms have similar implementations being different only in the distance computations for the node selections, respectively, thus the comparison has no bias in implementation.
- *Granularity of a partition of the navigation area into nodes*: during the transformation of the navigation area into a graph, the parameter (namely, granularity henceforth) sets the sizes of Cartesian grid cells and the distance between neighbour nodes. A smaller distance results in finer granularity and a larger graph. We use the granularity in range $[2, 10]$ resulting in approximately 15000 nodes in the graph for the granularity 10 and 390000 for the granularity 2.
- *Presence of temporary obstacles*: in a part of the experiments where the performance of DALI or DALI* is compared with A*, we ignore all temporary obstacles.
- *Number of tasks/milestones* is in range $[1, 10]$.
- *Number of permanent obstacles* is in range $[1, 10]$.
- *Number of heat areas* is in range $[1, 5]$.
- *Number of vehicles* is in range $[1, 4]$.

In order to vary the numbers of milestones, permanent obstacles, and heat areas and to avoid the generation of a separate mission for each combination of the parameters, we use the following strategy. The mission contains a set of milestones (permanent obstacles, heat areas) and in each experiment we select a subset of all milestones (permanent obstacles, heat areas) of a desired size (ensuring that all task preconditions can be met) and perform mission planning with the selected milestones (permanent obstacles, heat areas).

In all experiments we compute the time needed to generate a graph, the total time used by the path-planning algorithms, and the time used by TAMAA (i.e., calls to UPPAAL). Each experiment involves multiple calls to the path-planning algorithm and to UPPAAL; we compute the total time for all calls. All experiments have been conducted 5 times and we consider the mean time as a result. The front end and the middleware are on a same PC with a 12-core i7 CPU, 16 GB RAM, and Windows 10 OS. The back end is on a server with a 48-core CPU (Intel Xeon E5-2678), 256 GB RAM, and Ubuntu 18.04 OS. The timeout of computation is set to be 1 hour in the experiments.

We perform a preliminary experiment comparing two optimizations of DALI. For the preliminary experiment, we use a single vehicle and remove temporary obstacles and heat areas from the mission to ensure that path planning is called only once. We vary the granularity and the number of milestones, and compare the time taken by the path-planning algorithms. The experimental results show that on our mission DALI* is faster than the single-source multiple-target optimization. Therefore, in the remaining experiments we do not use the single-source multiple-target optimization of DALI. The results of this preliminary experiment are discussed in Subsection 6.2.

The experiments have been divided into 4 groups shown as follows:

(1) *Group I*: agent amount: 1, presence of temporary obstacles and heat areas: false, path-planning algorithms: A*, DALI and DALI*.
(2) *Group II*: agent amount: 1, presence of temporary obstacles and heat areas: true, path-planning algorithms: DALI and DALI*.
(3) *Group III*: agent amount: 1 - 4, presence of temporary obstacles and heat areas: false, path-planning algorithms: A*, DALI and DALI*.

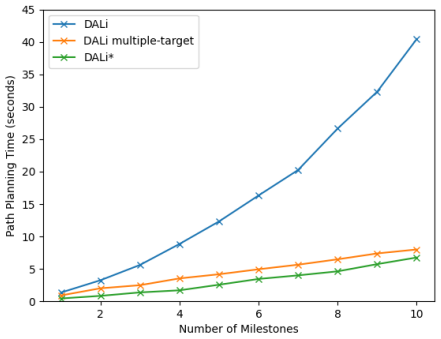


Fig. 10. Comparison of DALı and its optimizations

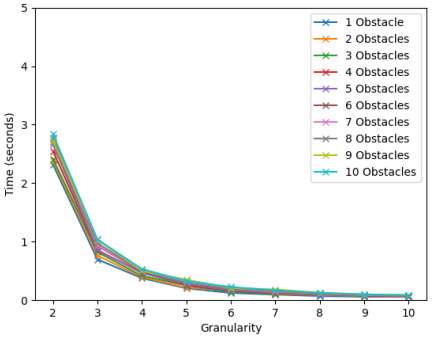


Fig. 11. The first group of experiments: the graph generation time w.r.t. the granularity and the number of obstacles

(4) *Group IV*: agent amount: 1 - 4, presence of temporary obstacles and heat areas: true, path-planning algorithm: DALı*.

Group I for evaluating the performance difference between A* and DALı algorithms (i.e., DALı and DALı*). Group II considers the effect of heat areas and temporary obstacles on the performance of DALı algorithms. The A* algorithm is not used in this group of experiments since it does not support navigation in the presence of heat areas and temporary obstacles. Groups III and IV consider multiple vehicles to see the influence of the vehicle numbers on the performance of the DALı algorithms.

In the Figures 10-21, we show the influence of 1 or 2 parameters on the execution time of our tool, while the remaining parameters are set to default values: 1 vehicle, 10 milestones, 10 obstacles, 0 heat areas, and granularity 4.

6.2 Comparison of DALı optimizations

In this preliminary experiment, we compare the path-planning time of the basic DALı algorithm and two optimizations proposed in Subsection 4.1. The first optimization converts DALı into the single-source multiple-target algorithm. Considering that our methodology requires to compute paths between every pair of milestones, such optimization drastically reduces the number of calls to the algorithm. The second optimization referenced as DALı* uses an heuristic from the A* algorithm.

The results show that both optimizations are significantly faster than the original DALı algorithm. Among the two optimizations, the DALı* is clearly the fastest. The results for the granularity set to 4 are shown in Figure 10. Considering the results of this experiment, we do not use the single-source multiple-target optimization in the following experiments.

6.3 Comparison of A* and DALı

In the first group of experiments we create a mission plan for a single vehicle with the A* algorithm and 2 versions of the DALı algorithm without a consideration of heat areas and temporary obstacles. We vary the number of milestones, the number of permanent obstacles, and the granularity. We report the mean value of the execution time of 5 runs. The time spent on the graph generation (Figure 11) and on the UPPAAL call (Figure 12) is independent from the choice of algorithms. The graph generation time depends on the granularity since the number of nodes in a graph is

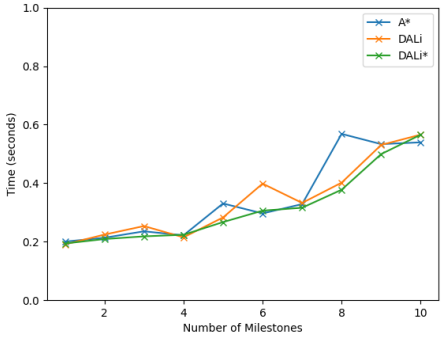


Fig. 12. The first group of experiments: the UPPAAL call time w.r.t. the number of milestones

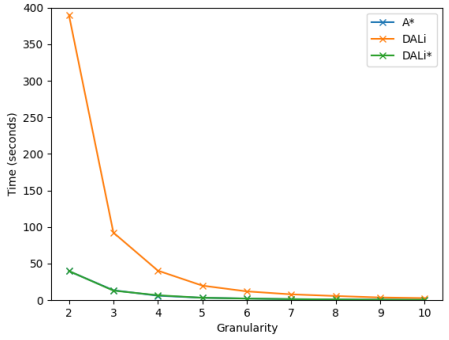


Fig. 13. The first group of experiments: the path-planning time w.r.t. the granularity

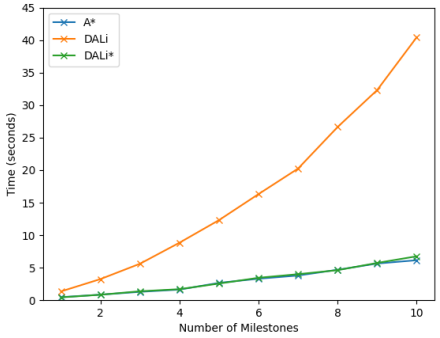


Fig. 14. The first group of experiments: the path-planning time w.r.t. the number of milestones

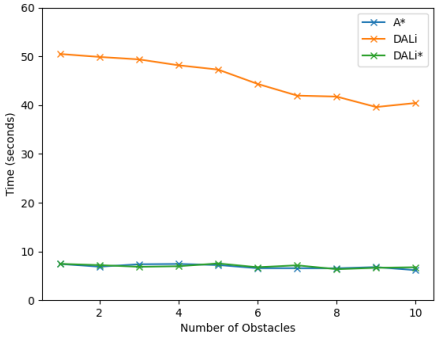


Fig. 15. The first group of experiments: the path-planning time w.r.t. the number of obstacles

inversely proportional to the square of the granularity. In this group of experiments, only a single call to UPPAAL is performed. The lack of temporary obstacles implies that the generated plan can satisfy all constraints after the single UPPAAL call. The number of milestones affects the size of the UPPAAL model and, consequently, the time needed to synthesize a plan. The computation time of all path-planning algorithms depends on the graph size, therefore the finer granularity the more time is used to compute paths (Figure 13). Since paths have to be computed between each pair of milestones, the total path-planning time grows with the number of milestones (Figure 14). On both figures we can notice that A* and DALi* have the same performance; whereas the original DALi is significantly slower and is more drastically affected by the granularity and the number of milestones. It is interesting to note that while A* and DALi* are barely affected by the number of obstacles, DALi performs 20% faster with ten obstacles than that with a single one (Figure 15). Indeed, obstacles reduce the number of nodes in the graph, which positively affects the path search time. Computation time of DALi* and A* are less affected by this parameter: first of all, they scale better with the number of nodes than DALi does and, therefore, a small reduction in the number of nodes has a minor effect on the algorithms' performance. Moreover, the heuristic used in A*

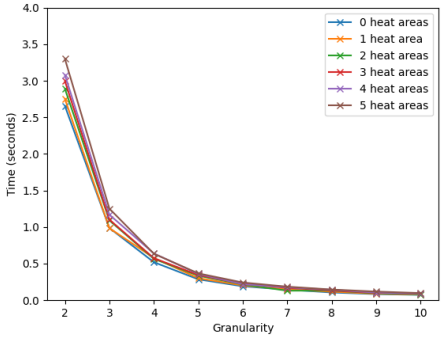


Fig. 16. The second group of experiments: the graph generation time w.r.t. the number of heat areas

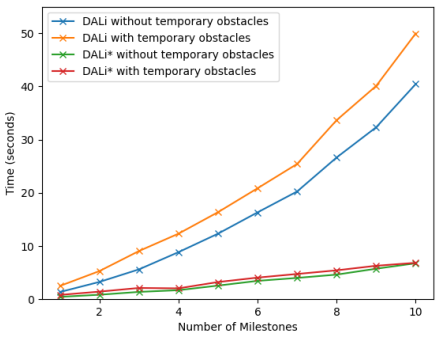


Fig. 17. The second group of experiments: the path-planning time w.r.t. the number of milestones

and DALi* uses the distance estimation to the target node and obstacles on the path make the estimation less reliable.

6.4 Evaluation of the Approach with Heat Areas and Temporary Obstacles

In the second group of experiments, only two versions of DALi are evaluated, because the A* algorithm cannot take heat areas and temporary obstacles into consideration. The mission has been specifically designed to ensure that paths computed before the first call to UPPAAL would cross the temporary obstacles. Therefore, the generation of a correct mission plan requires a recomputation of several paths and additional calls to UPPAAL.

The graph generation is unaffected by the presence of temporary obstacles as they are not considered during the graph construction, however nodes located inside the heat areas have to be marked. Thus, in Figure 16 we can notice that the graph generation time slightly rises with the number of heat areas. The time for a single UPPAAL call has not been affected by the presence of heat areas and temporary obstacles. Figure 17 shows the results for the two versions of DALi. For comparison, we add the corresponding results from the previous group of experiments without temporary obstacles. Due to the presence of temporary obstacles, TAMAA requires up to 4 additional calls to the path-planning algorithm and up to 2 additional calls to UPPAAL, though its effect on the computation time of path planning is minor: a few seconds for DALi and less than 1 second for DALi*. Indeed, for 1 vehicle all UPPAAL calls require less than 0.6 seconds and the path-planning calls (at granularity 4) take about 0.5 second for DALi and 0.07 seconds for DALi*, respectively (Figure 18). The presence of heat areas does not have a significant effect on the path-planning time (Figure 19). Nevertheless, heat areas can change the shortest path and, as a result, affect the number of recomputations of paths: a new path can lead towards a temporary obstacle or, conversely, can help to avoid it. In the experiment, we have the latter case: one of the heat areas changes a shortest path between the start point and the first milestone and avoids a temporary obstacle. As a result, the recomputation is not called, thus the path-planning and total time are smaller than those of the initial computation. In Figure 19, the path planning for 1 milestone and 5 heat areas takes only 0.4 seconds in comparison to 0.7 seconds for 1 milestone and $0-4$ heat areas. Note that the path-planning for 4 or more milestones and 5 heat areas is not significantly faster than for $0-4$ heat areas: paths between other milestones cross the temporary obstacles causing the recomputation.

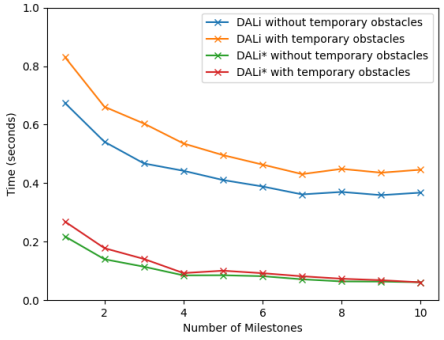


Fig. 18. The second group of experiments: the mean path-planning time

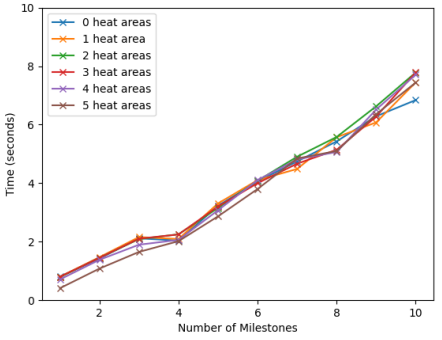


Fig. 19. The second group of experiments: the path-planning time w.r.t. the number of heat areas

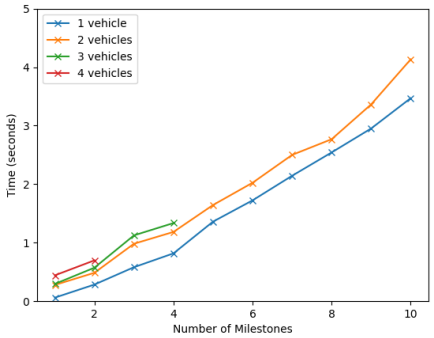


Fig. 20. The third group of experiments: the path-planning time w.r.t. the number of milestones

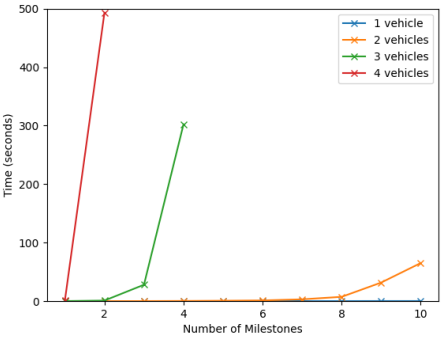


Fig. 21. The third group of experiments: the UPPAAL time w.r.t. the number of milestones

6.5 Results for Multiple Agents

In the third group of experiments, we evaluate how the tool performs with multiple agents. Within this group, the mission with temporary obstacles is used and the only considered path-planning algorithm is DALi*. Agents have different speeds and subsets of tasks to do. We assume that all agents can drive through any non-obstacle area and the heat maps have the same slowing factor for all agents. Therefore, without the consideration of temporary obstacles, the shortest path between a pair of milestones can be the same for all agents.

An involvement of multiple agents in the mission has low effect on the path-planning algorithm performance. The assumption above on the common shortest paths for agents allows us to reuse paths between milestones for all agents, thus an addition of an extra agent only requires to compute paths from its starting location to other milestones. The number of UTA used in TAMAA and, consequently, the complexity of the composed model of multiple agents depend on the number of agents.

Figures 20 and 21 show the time required by DALi* and by UPPAAL, respectively. Our tool times out during UPPAAL calls with no result in cases of 3 agents with 5 milestones, and 4 agents with 3 milestones. Therefore, figures 20 and 21 do not show the computation time of these cases.

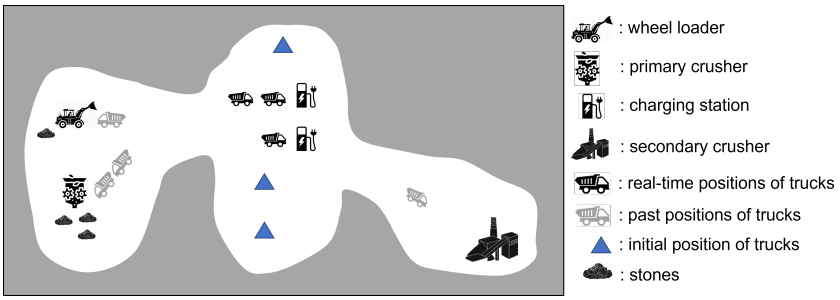


Fig. 22. An example of the autonomous quarry

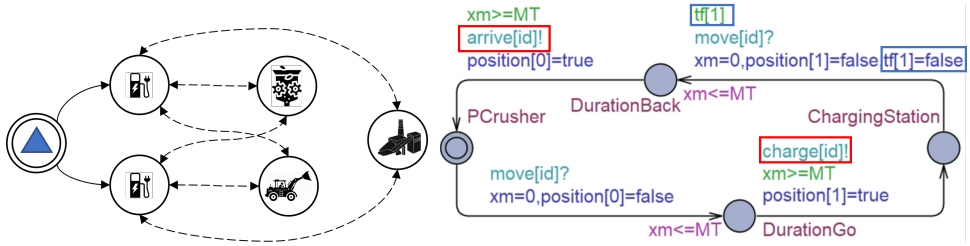
Introduction of temporary obstacles that requires additional calls to UPPAAL roughly multiplies the computation time by the number of calls to UPPAAL. Our previous work [GESL20] proposes a method named MCRL (model checking + reinforcement learning) to solve this scalability problem caused by large numbers of agents. We leave the integration of MCRL into MALTA as a future work.

7 ADAPTABILITY OF MALTA: A SPECIAL INDUSTRIAL USE CASE

Variability of mission planning problems is immense. Even within the autonomous quarry case study, there is a huge spectrum of requirements starting from safety properties to liveness properties [BK08], such as agents must never go across a certain area when humans are working there (safety property), and agents should repetitively enter the charging points until they accomplish the mission (liveness property). Due to high variability, it is infeasible to build a fully automated solution that fits all possible cases efficiently. Therefore, the adaptability of a solution plays a crucial role, which requires an easy way of adapting the agent models and queries to different applications and their requirements.

In this section, we show an example of a variant of our industrial case study: the autonomous quarry, which is sightly different from the problem definition (i.e., Definition 1). We explain how to adapt MALTA to the special use case depicted in Fig. 22. The quarry has 3 identical autonomous trucks transferring stones from a primary crusher to a secondary crusher. Stones are gathered at the left side of the quarry and can be loaded into the trucks either by the primary crusher or by a wheel loader. The primary crusher is required to minimize idle time, therefore the wheel loader can only be used if the primary crusher is already occupied by two trucks: one is being loaded and another is waiting in a queue. Stones should be unloaded from the trucks at the secondary crusher on the right part of the map. On the way between crushers there are two charging stations. The use case requires trucks to stop for charging every time they pass the charging stations, no matter how much battery they have left. Each charging stop takes 30 seconds.

The use case has 3 trucks, 1 wheel loader, 1 primary crusher, and 1 secondary crusher. Besides the special charging task, there are 2 regular tasks for trucks: loading stones that can either be performed at the primary crusher or at the wheel loader, and unloading stones at the secondary crusher.



(a) The topology of the autonomous quarry in Fig. 22

(b) A part of the adjusted movement TA

Fig. 23. Adjustments of the model for the industrial use case.

Table 2. Machine parameters in the autonomous quarry

	Machine	Speed	Rate	Capacity
Mobile	Autonomous truck	35 km/h	1.5 tons/s	15 tons
Stationary	Wheel loader	/	1.5 tons/s	/
	Primary crusher	/	0.25 tons/s	/
	Charging station	/	30 s/time	/

Table 2 shows the parameters of the machines in the quarry. Trucks can carry 15 tones of stones, and the primary crusher can load 0.25 tons of stones per second, therefore the primary crusher takes 60 seconds to fill in one truck, while the wheel loader can do that in 10 seconds. The trucks can unload 1.5 tons of stones per second, so trucks unloading a full bucket of stones into the secondary crusher takes 10 seconds. The mission goal is to transfer 90 tons of stones as fast as possible.

We use MALTA to create a mission plan for the trucks. The use case does not specify distances in the quarry, therefore we assign them in a manner that ensures that the trucks can finish the tasks without draining out. For path planning, the DALI* algorithm is applied. However, the use case imposes constraints on tasks that have not been covered in Section 4, in particular the special task of charging and priority between primary crusher and wheel loader. Therefore, we need to adjust the already introduced UTA models and queries for this use case.

7.1 Adjustments of the Models

The use case has two requirements that are not supported directly by the original model generated by MALTA. Besides, the topology of the map is also changed to match the special geographic arrangement of machines in the quarry. In this section we explain the modifications necessary to incorporate the new requirements.

The original movement UTA assumes a direct connection between every pair of milestones. However, the geographic arrangement of the use case is different. As the charging stations occupy the center of the quarry, trucks must pass them when traveling from the left to the right of the quarry, and vice versa. Hence, the topology of the quarry for this use case is adjusted. As depicted in Fig. 23(a), the primary crusher, wheel loader, and the secondary crusher are connected via the charging stations. The generated movement UTA enforces the topology, and thus only accepting the traveling between the charging stations and other milestones. From a truck's point of view, the wheel loader and primary crusher are both for loading stones, so there is no need to connect the wheel loader and primary crusher in the topology. The new requirement of the charging task also induces other adjustments of the movement UTA. Fig. 23(b) shows a part of the adjusted movement UTA, where changes are highlighted by the blue and red squares. When a truck, which is identified

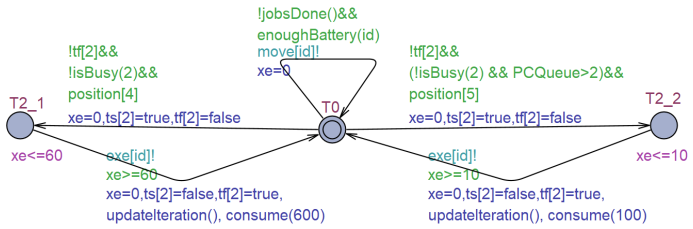


Fig. 24. A part of the adjusted task-execution UTA

by variable `id`, leaves a charging station, its charging task, which is represented by `tf[1]`, must be executed. After the truck leaves the charging station, the variable `tf[1]` flips to *false* so that the next time it reaches a charging station, the charging task can be carried out again.

The second additional requirement of this use case is the priority between the primary crusher and the wheel loader. To fulfill such requirement, an adaptation of the task execution UTA is necessary. As depicted in Fig. 24, we add an additional constraint $PCQueue > 2$ to the guard of the *edge* going to *location* `T2_2` (representing task "Loading at the wheel loader), where *PCQueue* is a global variable counting the number of vehicles at the primary crusher. Therefore, the adapted task execution UTA models that when the length of the waiting queue at the primary crusher is less than two, trucks must go to the crusher; otherwise, the trucks can choose to wait in the queue or go to the wheel loader for loading stones. The synthesized mission plan is supposed to make a wise choice in this case for the optimal productivity. In addition, in the movement UTA, *edges* going into and leaving from the location representing the primary crusher updates the *PCQueue* variable.

The mission of this use case is defined to carry all the stones to the secondary crusher rather than complete all the tasks a desired number of times. Therefore, we add the auxiliary variables *stone* and *load*, which represent the total volume of stones remained to be transferred and the vehicles capacity, respectively.

7.2 Additional Adaptation of Queries and Models

The requirements formalised by Queries (7) - (9) are not changed for the use case, however the Query (10) is replaced due to the different mission formulation by:

$$E <> stone == 0 \tag{11}$$

The use case has two additional requirements:

(1) Prioritizing the primary crusher before the wheel loader: a truck can choose the wheel loader only in case when there are two other vehicles at the primary crusher (one being served and one in a queue).
(2) Battery charging: whenever a truck passes by a charging station, it must stop there and charge for 30 seconds. The trucks' batteries must never be consumed before they finish the global mission.

The former requirement has a straightforward encoding into the following UPPAAL query:

$$\texttt{A[] PCQueue < 2 imply !(task}_0\texttt{.T2\_2 || ... || task}_n\texttt{.T2\_2),} \tag{12}$$

This query checks that at any moment in case of the queue at the primary crusher being not full, the task execution UTA of any truck is not at the *location* `T2_2` that represents being loaded at the wheel loader.

The latter requirement consists of two parts. First, it requires the trucks to always charge themselves right after they arrive at a charging station. Second, it requires the trucks to charge themselves timely so that their batteries are not consumed. A logic formalization of the first part is called "*always next*", i.e., the *next* action of moving to a charging station is *always* charging. Unfortunately, the "*always next*" property cannot be formalized in the query language supported by UPPAAL (a subset of TCTL [HYP+06]). To overcome this difficulty, we design an auxiliary UTA, to help us verify this requirement. Fig. 25 shows the auxiliary UTA, namely *monitor*, where *channels*

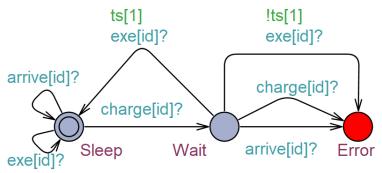


Fig. 25. The auxiliary UTA *monitor* for verifying the repetitive charging requirement.

`arrive` and `charge` synchronize the movement UTA and the monitor UTA (see Fig. 23(b) too). The *channel* `exe` is for synchronizing the task execution UTA and the monitor UTA when the truck is going to execute a task (see Fig. 24 too). Therefore, the monitor UTA initially stays at *location* `Sleep` when the truck moves and executes tasks (i.e., two self-loops at *location* `Sleep` in Fig. 25). If the truck is moving to a charging station, the monitor UTA transfers to *location* `Wait`, and synchronizes with the movement UTA via *channel* `charge`. Next, the monitor UTA has multiple choices of the next transition: charging or executing other tasks (i.e., synchronization via *channel* `exe`), or moving to other milestones (i.e., synchronization via *channel* `charge` or `arrive`). A Boolean variable `ts[1]` is used to indicate whether the current truck is charging or not. The *edge* from *location* `Wait` to *location* `Sleep` is guarded by this variable, meaning that only charging can make the monitor UTA go back to its initial *location*, whereas other transitions will end up to a *location* representing errors. In summary, the monitor UTA regulates the correct order of a truck's task execution, i.e., moving to a charging station must be always succeeded by charging. If the trucks violate this regulation, they will be stuck at the `Error` *location*. The contraposition of this statement forms the "*always next*" constraint in the battery-charging requirement: if the truck models never visit the `Error` *location*, the next action of moving to a charging station is always charging. Now, we encode an invariance query to verify this property:

$$\texttt{A[] !monitor}_0\texttt{.Error \&\& ... \&\& !monitor}_n\texttt{.Error} \tag{13}$$

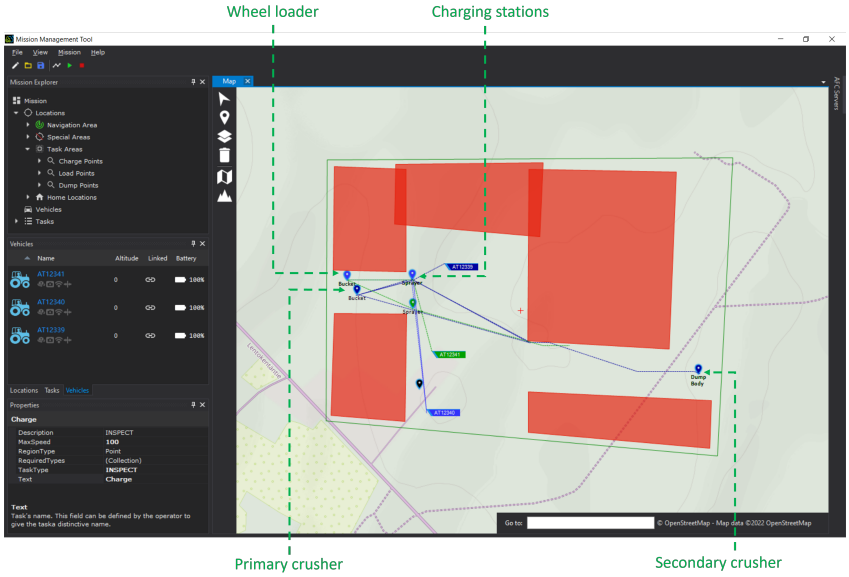
Query (13) requires that *monitors of trucks* never reach the `Error` *location*. Satisfying this query indicates that the trucks always charge themselves after arriving at a charging station.

To verify the second part of the battery-charging requirement, we need to add an array of integers to store the battery levels of trucks. When a truck moves, the corresponding integer of its battery level decreases. The consumption rate is assumed to be proportional to the truck's traveling time. If the truck charges, the integer increases. The CTL query that encodes this property is as follows, where `N` is the index of the last truck.

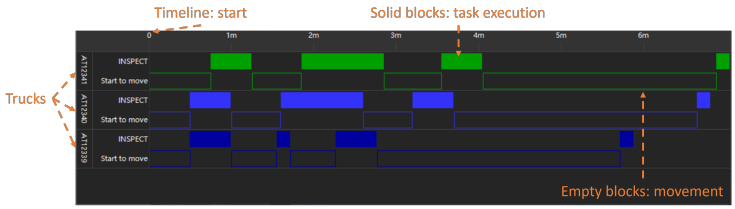
```
A[] forall(i:int[0,N]) battery[i]>0
```
(14)

7.3 Synthesis Results

Given the machine parameters shown in Table 2, we use *MALTA* to synthesize a mission plan that controls the three trucks to transport 90 tons of stones to the secondary crusher while satisfying all



(a) The path plans of trucks



(b) A part of the Gantt chat presenting the task schedules of trucks

Fig. 26. A mission plan of trucks working in the autonomous quarry of Fig. 22

the requirements of the use case, and find the fastest way to finish the global mission. After running *MALTA* to generate the fastest trace that satisfies Query (11), a synthesized result of mission plan is illustrated in *MMT* (Fig. 26). Fig. 26(a) shows the paths of avoiding static obstacles and visiting different milestones. The task schedule (Fig. 26(b)) shows the interleaving sequence of movement (empty rectangles) and task execution (solid rectangles). One can check the details of path plans by clicking on the empty rectangles, and then the corresponding path plans will be highlighted in the GUI of path plans. The mission plan shows that the fastest time of transporting all the stones is 6.9 minutes. The fact that a mission plan is depicted in *MMT* demonstrates that Queries (12) - (14) are satisfied.

8 RELATED WORK

Path planning, a.k.a., motion planning in the Artificial Intelligent (AI) community, has been an interest of research since the early days of robotics [N+84]. Sampling-based methods like Rapidly-exploring Random Tree (RRT) [LaV98] and a method based on probabilistic roadmaps for path planning [KSLO96], and graph-search-based methods like A* [Rab00] and Theta* [DNKF10] are

two typical branches of path-planning algorithms. The main contribution of these algorithms is to find collision-free paths in a static and continuous world, in which the topology of the moving space does not change. Moreover, when a robot starts to interact with the world, e.g., picking an object and carrying it to another position, the robot's and object's dynamics and kinematics are changed, which can cause the initial motion plan to be unsuitable. Alami *et al.* [ASL90] and Hauser *et al.* [HL10] propose a modal structure of the robots and their working environments. Since the switch of modes is discrete, the problem is about identifying the modes of the systems, defining the transitions among the modes, and traversing the state spaces in order to find a trace that satisfy some certain constraints. This is the so-called task planning in the AI community [GNT16]. These approaches do not guarantee correctness unless coupled with a formal verification technique.

Integrating task and motion planning (TAMP) provides us a good understanding of our problem: hybrid discrete-continuous search problem [GCH+21]. Research in this area often combines AI and robotics and seeks to provide a separation of concerns by designing hierarchical frameworks, in which high-level task planning and low-level motion planning are separated into different layers, and connected via an intermediate layer [GMSL19, STWZ10]. Downward refinement in the methods proposed by Bacchus *et al.* [BY94] and Nilsson *et al.* [N+84] first plans at the high level and then refines the high-level plans to low-level ones. The authors assume that their problems fulfil the *downward refinement property* [BY94], which is often not the case in reality. Our method, on the contrary, starts from the low level to calculate path plans that are collision-free, and then integrates the path-planning results into the model for task planning. Therefore, our task-planning results are naturally collision-free.

There is an important line of work of task planning that uses temporal logic to specify the high-level requirements of tasks [KGFP09, BYG17]. Linear Temporal Logic (LTL) is the most widely used logic for requirement specification [FJKG10, CPLK20, BKV10], because of its expressive power that is able to capture relatively complex requirements, such as repetitively filling the water tank if the water level is lower than a certain level. Different from these studies, we adopt Timed Computation Tree Logic (TCTL). (T)CTL and LTL are members of a temporal logic family named CTL* [BK08]. Each of them has its own expressive power and thus is used in different problems. TCTL enables one to express timed requirements such as digging 1000 m^3 of stones per 24 hours, which is of high industrial concern, in an attempt to ensure productivity when using autonomous vehicles. Most importantly, our task planning is fully integrated with path planning. Therefore, the results of our method comprehensively consider both the traveling time, as well as the task execution time and order.

In the formal methods community, task planning is being challenged by various formalisms and methods. When the formalisms only have stochastic models, the problems fall into a category called $\frac{1}{2}$-player games [Jen18]. In $\frac{1}{2}$-player games, neither agents nor environments get the control of their behaviors and the corresponding outcome, e.g., flipping a coin. By replacing the stochastic behaviors of agents with non-deterministic choice of actions, $\frac{1}{2}$-player games are changed to 1-player games, which is the problem that we are solving in this paper. Note that, 1-player games assume that the environment is fully controlled by the agents, so that the winning strategies are totally dependent on the behaviors of the agents [Jen18]. Besides UPPAAL, there are many other tools that aim to solve this kind of problems, e.g., Kronos [BDM+98], LTSim [BvdPW10], and SpaceEx [FLGD+11]. The major difference between our tool and these mentioned ones is that our MALTA tool integrates path-planning algorithms and task-scheduling algorithms, and has a dedicated GUI for mission planning, which provides interfaces for extension. Adding stochastic behaviors to environments makes the formalism represented as a $1\frac{1}{2}$-player game, and changing the stochastic behaviors of environments into non-deterministic ones that are independent from agents makes the formalism to be a 2-player game [Jen18], both of which are out of the scope of this paper. There are studies

that investigate synthesizing controllers from various temporal logic specifications. Alur *et al.* [AMT16, AMT18] propose a compositional method for synthesizing reactive controllers satisfying Linear Temporal Logic specifications for multi-agent systems. Tumova et al. [BLDT19, NBTD18] present their method for motion planning of multiple-agent systems using Metric Interval Temporal Logic (MITL). Inspired by these works, our study aims to bring up a methodology that is dedicated to collectively solve multi-agent mission planning that includes two components: path finding and task scheduling, and deal with complex environmental constraints and timing requirements of tasks.

In the field of robotics, design and development of GUI for planning, execution and supervision of missions involving several autonomous vehicles is getting increasingly much attention [ED06]. Such a GUI allows the operator of autonomous vehicles to plan and supervise several vehicles at the same time. Such user interfaces have been a research topic for different use cases including delivery services [SMCC18], military applications [KJK+15] and others [PMC+13, LKHM16, S+18]. Most of these GUI however, are designed for specific use cases and cannot be used as a generic graphical user interface for other domains. In this work we employ MMT which is a GUI that can be setup to communicate with different planners and autonomous vehicles. MMT has been used for planning and supervising underwater autonomous vehicles previously [ACME20].

9 CONCLUSIONS AND FUTURE WORK

In this article, we have presented a new methodology and a toolset to solve the mission planning problem of multiple autonomous agents. Our methodology includes an improved version of DALı for path planning, a timed-automata-based method for task scheduling, namely TAMAA, and an integration of these two methods to provide a complete solution of synthesis and verification of mission plans for multiple agents. As the improved version of DALı considers special road conditions, such as temporary obstacles, our method can deal with complex environments. TAMAA is based on formal modeling and model checking, so the synthesized task schedules are guaranteed to be the correct and the fastest to finish all tasks. The integration of DALı and TAMAA requires an iterative computation between path planning and task scheduling so that the result mission plans consider both the traveling time and task-execution time and are guaranteed to be the optimal solution. The methods have been implemented as a toolset named *MALTA*, which is made of three components. The front end of *MALTA* is a GUI for configuring the mission requirements and showing the results of mission planning. The back end of *MALTA*, which is responsible for running computational expensive functions, can be deployed locally or remotely. The middleware of *MALTA* bridges the front end and back end, so the users can focus on designing the map and tasks for the agents, and benefit from the algorithms of path planning and task scheduling without knowing the technical details. We have employed the toolset to solve a mission-planning problem of an industrial use case of an autonomous quarry. We have observed the computation time w.r.t. the numbers of obstacles, agents, heat areas, milestones, and the granularity of the map. The experimental results demonstrate the capability and limit of our method. An instantiated quarry is introduced and solved to show the flexibility of our method to fit various applications of mission planning.

There are two potential directions to extend our work in the future. One is to enrich the path-planning and task-scheduling algorithms supported by *MALTA*, so that the toolset can cope with more complex problems such as more agents or larger environments. To integrate the toolset with machine learning techniques is another direction. As the current task scheduling assumes the environment to be collaborative, it can be interesting to investigate how the method can be adapted when the environment contains some competitive agents.

ACKNOWLEDGMENTS

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

REFERENCES

[AAM+06] Yasmina Abdeddaı, Eugene Asarin, Oded Maler, et al. Scheduling with timed automata. *Theoretical Computer Science*, 2006.

[ACME20] E Afshin Ameri, Baran Cürüklü, Branko Miloradovic, and Mikael Ekström. Planning and supervising autonomous underwater vehicles through the mission management tool. In *Global Oceans 2020: Singapore–US Gulf Coast*, pages 1–7. IEEE, 2020.

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126, 1994.

[AMT16] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional synthesis of reactive controllers for multi-agent systems. In *International Conference on Computer Aided Verification*, pages 251–269. Springer, 2016.

[AMT18] Rajeev Alur, Salar Moarref, and Ufuk Topcu. Compositional and symbolic synthesis of reactive controllers for multi-agent systems. *Information and Computation*, 261:616–633, 2018.

[ASL90] Rachid Alami, Thierry Simeon, and Jean-Paul Laumond. A geometrical approach to planning manipulation tasks. the case of discrete placements and grasps. In *The fifth international symposium on Robotics research*. MIT Press, 1990.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A model-checking tool for real-time systems. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer, 1998.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[BKV10] Amit Bhatia, Lydia E Kavraki, and Moshe Y Vardi. Sampling-based motion planning with temporal goals. In *2010 IEEE International Conference on Robotics and Automation*. IEEE, 2010.

[BLDT19] Fernando S Barbosa, Lars Lindemann, Dimos V Dimarogonas, and Jana Tumova. Integrated motion planning and control under metric interval temporal logic specifications. In *2019 18th European Control Conference (ECC)*. IEEE, 2019.

[BvdPW10] Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In *International Conference on Computer Aided Verification*. Springer, 2010.

[BY94] Fahiem Bacchus and Qiang Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 1994.

[BY03] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Advanced Course on Petri Nets*. Springer, 2003.

[BYG17] Calin Belta, Boyan Yordanov, and Ebru Aydin Gol. *Formal methods for discrete-time dynamical systems*. Springer, 2017.

[CFL+15] Alessio Colombo, Daniele Fontanelli, Axel Legay, Luigi Palopoli, and Sean Sedwards. Efficient customisable dynamic motion planning for assistive robots in complex human environments. *Journal of ambient intelligence and smart environments*, 2015.

[CPLK20] Mingyu Cai, Hao Peng, Zhijun Li, and Zhen Kan. Learning-based probabilistic ltl motion planning with environment and motion uncertainties. *IEEE Transactions on Automatic Control*, 2020.

[D+59] Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959.

[DNKF10] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. *Journal of Artificial Intelligence Research*, 39:533–579, 2010.

[ED06] JW Eggers and Mark H Draper. Multi-uav control for tactical reconnaissance and close air support missions: operator perspectives and design challenges. In *Proc. NATO RTO Human Factors and Medicine Symp. HFM-135. NATO TRO, Neuilly-sur-Siene, CEDEX, Biarritz, France*, pages 2011–06, 2006.

[FG96] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 1996.

[FJKG10] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010.

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*. Springer, 2011.

[GCH+21] Caelan Reed Garrett, Rohan Chitnis, Rachel Holladay, Beomjoon Kim, Tom Silver, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Integrated task and motion planning. *Annual review of control, robotics, and autonomous*

systems, 2021.

[GES19] Rong Gu, Eduard Paul Enoiu, and Cristina Seceleanu. Tamaa: Uppaal-based mission planning for autonomous agents. In *The 35th ACM/SIGAPP Symposium On Applied Computing SAC2020, 30 Mar 2020, Brno, Czech Republic*, 2019.

[GESL20] Rong Gu, Eduard Paul Enoiu, Cristina Seceleanu, and Kristina Lundqvist. Verifiable and scalable mission-plan synthesis for multiple autonomous agents. In *25th International Conference on Formal Methods for Industrial Critical Systems*. Springer, 2020.

[GJP+22] Rong Gu, Peter Jensen, Danny Poulsen, Cristina Seceleanu, Eduard Paul Enoiu, and Kristina Lundqvist. Verifiable strategy synthesis for multiple autonomous agents: A scalable approach. *International Journal on Software Tools for Technology Transfer*, 2022.

[GMSL19] Rong Gu, Raluca Marinescu, Cristina Seceleanu, and Kristina Lundqvist. Towards a two-layer framework for verifying autonomous vehicles. In *NASA Formal Methods Symposium*. Springer, 2019.

[GNT16] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[HL10] Kris Hauser and Jean-Claude Latombe. Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, 2010.

[HNR68] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968.

[HYP+06] M. Hendriks, Wang Yi, P. Petterson, J. Hakansson, K.G. Larsen, A. David, and G. Behrmann. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*, 2006.

[Jen18] Peter Gjøl Jensen. *Efficient Analysis and Synthesis of Complex Quantitative Systems*. Aalborg Universitetsforlag, 2018.

[KGFP09] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE transactions on robotics*, 2009.

[KJK+15] Youngjoo Kim, Wooyoung Jung, Chanho Kim, Seongheon Lee, Kihyeon Tahk, and Hyochoong Bang. Development of multiple unmanned aircraft system and flight experiment. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2015.

[KSLO96] Lydia E Kavraki, Petr Svestka, J-C Latombe, and Mark H Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 1996.

[LaV98] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report*, 1998.

[Len92] J. K. Lenstra. Job shop scheduling. In Mustafa Akgül, Horst W. Hamacher, and Süleyman Tüfekçi, editors, *Combinatorial Optimization*. Springer, 1992.

[LKHM16] Bae Hyeon Lim, Jong Woo Kim, Seok Wun Ha, and Yong Ho Moon. Development of software platform for monitoring of multiple small uavs. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 2016.

[N+84] Nils J Nilsson et al. *Shakey the robot*. Articial Intelligence Center, SRI International Menlo Park, California, 1984.

[NBTD18] Alexandros Nikou, Dimitris Boskos, Jana Tumova, and Dimos V Dimarogonas. On the timed temporal logic planning of coupled multi-agent systems. *Automatica*, 97:339–345, 2018.

[PMC+13] Daniel Perez, Ivan Maza, Fernando Caballero, David Scarlatti, Enrique Casado, and Anibal Ollero. A ground control station for a multi-uav surveillance system. *Journal of Intelligent & Robotic Systems*, 2013.

[Rab00] Steve Rabin. Game programming gems, chapter a* aesthetic optimizations. *Charles River Media*, 2000.

[S+18] Espen Skjervold et al. Autonomous, cooperative uav operations using cots consumer drones and custom ground control station. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018.

[SMCC18] Khin Thida San, Sun Ju Mun, Yeong Hun Choe, and Yoon Seok Chang. Uav delivery monitoring system. In *MATEC Web of Conferences*. EDP Sciences, 2018.

[STWZ10] Gopinadh Sirigineedi, Antonios Tsourdos, Brian A White, and Rafal Zbikowski. Modelling and verification of multiple uav mission using smv. *arXiv preprint arXiv:1003.0381*, 2010.