Combining Model Checking and Reinforcement Learning for Scalable Mission Planning of Autonomous Agents

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

***Abstract*—The problem of mission planning for multiple autonomous agents, including path planning and task scheduling, is often complex, especially when the number of agents grows or requirements include real-time constraints. In this paper, we propose a novel approach called MCRL that integrates model checking and reinforcement learning to overcome this difficulty. Our approach employs timed automata and timed computation tree logic to describe the autonomous agents' behavior and requirements, and trains the model by a reinforcement learning algorithm, namely Q-learning, to populate a table used to restrict the state space of the model. Our method provides a means to synthesize mission plans for autonomous systems whose complexity exceeds the scalability boundaries of exhaustive model checking, but also to analyze and verify synthesized mission plans to ensure given requirements. We evaluate the proposed method on various scenarios involving autonomous agents, as well as present comparisons with other methods and tools.**

***Index Terms*—autonomous agents, mission planning, model checking, reinforcement learning**

I. Introduction

Autonomous agents are systems that usually move and operate in a possibly unpredictable environment, can sense and act on it, over time, while pursuing their goals [1]. As this kind of systems bear the promise of facilitating people's daily lives and increasing safety and industrial productivity by automating repetitive tasks, autonomous technologies are drawing an increased attention from both researchers and practitioners. In an attempt to realize their functions, mission planning of autonomous agents, including path planning and task scheduling, is one of the most critical problems to solve [2]. As path-planning algorithms focus on calculating collision-free paths towards the destination, they do not handle requirements concerning logic and temporal constraints, e.g., delivering goods in a prioritized order, and within a certain time limit. In addition, when considering a group of agents that need to collaborate with each other and usually work alongside humans, the job of synthesizing correctness-guaranteed mission plans becomes crucial and more difficult.

In our previous work [3], we have proposed an approach based on Timed Automata (TA) and Timed Computation Tree Logic (TCTL) to formally capture the agents' behavior and requirements, respectively, and synthesize mission plans for autonomous agents by model checking. Our approach is successfully implemented in a tool called TAMAA, and shown to be applicable to solving the mission-planning problem of industrial autonomous agents. However, TAMAA alone is not scalable when the number of agents is large, as the state space of the model explodes when the number of agents grows.

The state-space-explosion problem is one of the most stringent issues when employing model checking [4] for verification, therefore many studies have explored ways of fighting it [5]. In this paper, we propose a novel method called **MCRL** that combines model checking with reinforcement learning [6] to restrict the state space in order to synthesize mission plans for large numbers of agents. Our method is based on UPPAAL [7] and leverages the model of autonomous agents generated by TAMAA. Instead of exhaustively exploring and storing states of the model, MCRL utilizes Monte Carlo simulations to obtain the execution traces leading to the desired states or deadlocks. Note that in TAMAA timing uncertainties are modeled by non-deterministic delays bounded from below as well as above, rather than by probability distributions. Therefore, the simulation is simply for randomly sampling execution traces. Then, a reinforcement learning algorithm, namely Q-learning [8], is employed to process the execution traces, and populate a Q-table containing the state-action pairs and their values. The Q-table is recognized as the mission plan that we have aimed to synthesize, which is injected back into the old TAMAA model, forming a new model. As the Q-table regulates the behavior of the agent model, the state space is greatly reduced, which makes it possible to verify mission plans for large numbers of agents. Moreover, MCRL enables the model equipped with Q-tables to make best decisions when the task execution time and duration of movement are uncertain, which is not supported by TAMAA. As MCRL is based on formal modeling, it complements classic reinforcement learning algorithms with means to verify the synthesized mission plans against, for instance, safety requirements.

We select relevant scenarios involving autonomous agents in a construction site, and conduct experiments with MCRL, TAMAA, and UPPAAL STRATEGO [9] that is a tool often used for generating winning strategies for stochastic priced timed games. The experimental results show that MCRL performs better than TAMAA and UPPAAL STRATEGO, when the number of agents is greater than five. The time of synthesizing mission plans using MCRL increases linearly with the number of agents, whereas for the other two methods it increases exponentially.

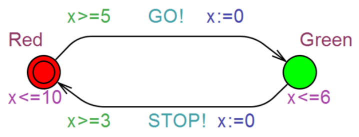


Fig. 1. An example of a time automaton of a traffic light

To summarize, the contributions of this paper are:

- A novel approach called MCRL for synthesizing mission plans of large numbers of autonomous agents by reinforcement learning, combined with model checking the synthesis results.
- An evaluation of the scalability of MCRL via experiments conducted with tools such as TAMAA, UPPAAL STRATEGO, and MCRL, on relevant scenarios involving autonomous agents. The experimental results show that MCRL can scale to large numbers of agents that cannot be handled by other methods.

The remainder of the paper is organized as follows. In Section 2, we introduce the preliminaries of this paper. Section 3 describes the problem that we attempt to solve and its challenges, whereas in Section 4 we introduce our novel approach for taming the scalability of model checking, which combines the latter with reinforcement learning. In Section 5, we explain the experiments and their results on TAMAA, UPPAAL STRATEGO, and MCRL. In Section 6 we compare to related work, before concluding and outlining possible future work in Section 7.

II. Preliminaries

In this section, we introduce timed automata, UPPAAL, UPPAAL STRATEGO, and reinforcement learning.

A. Timed Automata and UPPAAL

A *timed automaton* (TA) is a finite-state automaton extended with real-valued variables, called *clocks*, suitable for modeling real-time systems [10]. UPPAAL [7] is a tool for modeling, simulation, and model checking of real-time systems, which uses an extension of TA as the modeling formalism. Figure 1 depicts a simple example of a UPPAAL TA modeling traffic lights. Two locations `Red` and `Green` model the two colors of a traffic light. A clock variable `x` is used in the *invariants* (boolean expressions over clocks) on locations (e.g., x<=10) to enforce an upper bound of delaying in that location (in our case, after 10 time units, the automaton must leave location Red). Edges are directed lines used to connect locations, and they are decorated by guards, which are boolean conditions over clocks or discrete variables, which enable the automaton to traverse the respective edge once they evaluate to true. In our case, when x>= 5, the TA may move from the Red to the Green location. In UPPAAL, there is a special type of location, namely *committed* (denoted by encircled c). It requires that time does not elapse in these types of locations and the next edge to be traversed needs to start from a committed location. Clocks can be reset over edges, e.g., x:= 0 in Figure 1, whereas discrete typed variables can be assigned values, accordingly, via updates on the edges, or via functions that are implemented by a subset of the C language in the declaration of the TA. A network of TA, $B_0 \parallel ... \parallel B_{n-1}$, is a parallel composition of n TA via *synchronization channels* (i.e., $a!$ is synchronized with $a?$ by handshake). In Figure 1, the edges are labeled with channels named `STOP` and `GO`, which synchronize this TA with other TA of vehicles.

The UPPAAL queries that we verify in this paper are properties of the form: (i) **Invariance**: $A \square p$ means that for all paths, for all states in each path, p is satisfied, (ii) **Liveness**: $A \Diamond p$ means that for all paths, p is satisfied by at least one state in each path, (iii) **Reachability**: $E \Diamond p$ means that there exists a path where p is satisfied by at least one state of the path, and (iv) **Time-bounded Leads to**: $p \rightsquigarrow_{\leq t} q$, which means that whenever p holds, q must hold within at most t time units thereafter; it is equivalent to the property: $A \square (p \Rightarrow A \Diamond_{\leq t} q)$.

B. UPPAAL STRATEGO

UPPAAL has several branches that extends it to deal with various specific problems. UPPAAL STRATEGO [9] is a tool that integrates UPPAAL with two of its branches, that is, UPPAAL SMC [11] (statistical model checking) and UPPAAL TIGA [12] (policy synthesis for timed games). In this paper, we employ UPPAAL STRATEGO to solve the same mission-planning problem for autonomous agents, in order to compare the result with our MCRL approach. UPPAAL STRATEGO is designed to synthesize strategies for stochastic priced timed games. A game is a mathematical model of a system consisting of several players that compete in a common environment and aim to achieve their independent goals. Since it is based on UPPAAL, its modeling language is an extension of timed automata, which differentiates actions into two types: controllable and uncontrollable. The former ones are actions controlled by the players, whereas the latter ones are controlled by the environment. We refer readers to the literature [9] for details of this tool. A *strategy* is a policy of a player's actions for any possible situation that guides the player to reach its final goal. A *winning strategy* contains sequences of controllable actions that lead players to the states satisfying desired properties, regardless of the executed uncontrollable actions. In UPPAAL STRATEGO, one can synthesize winning strategies in form of: strategy S = control: P, where "=" is an assignment sign, P is the TCTL property to be met, and verify the synthesized strategies in the form of: P' under S, where P' is a stronger property that the model is verified against, with its behavior regulated by strategy S.

C. Reinforcement Learning

Reinforcement learning is a branch of machine learning aiming to calculate how agents should take actions in an environment, in order to maximize the accumulated reward obtained from the environment [6]. In this paper, we use one of the model-free reinforcement learning algorithms called *Q-learning* [8], which is usually adopted to learn policies that indicate agents the actions to take at different states. A policy is associated with a state action value function called

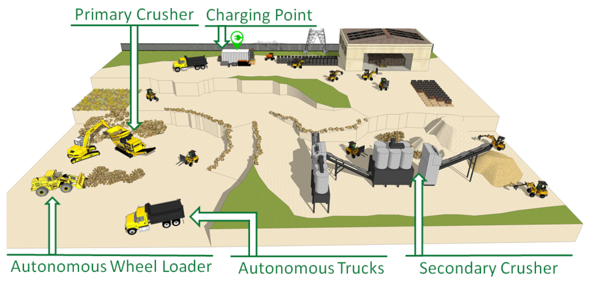


Fig. 2. An example of an autonomous quarry

Q function, where "Q" stands for "quality". The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \max_{a'} q^*(s',a')], \tag{1}$$

where $q^*(s,a)$ represents the expected reward of executing action a at state s, $\mathbb{E}$ denotes the expected value function, $R(s,a)$ is the reward obtained by taking the action a at state s, γ is a discounting value, s' is the new state coming from state s by taking action a, $\max_{a'} q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair (s',a'). The equation means that the expected reward of the state-action pair (s,a) is the sum of the current reward and the discounted maximum future reward. As the learning process iterates, the Q-value of each state-action pair converges to the maximum Q-value, i.e., q^*, and the parameters are updated using gradient descent [13]. Although Q-learning is a model-free algorithm, the learning process often relies on a simulation environment that depends on the form of the model. In this paper, we use the simulation function in UPPAAL to gather the information of state-action pairs, and invoke the Q-learning algorithm to populate a Q-table that stores state-action pairs and their Q-values.

III. Problem Description

In this section, we introduce an industrial use case of an autonomous quarry, containing various autonomous vehicles, e.g., trucks, wheel loaders, etc. For example, as shown in Figure 2, we consider the mission of transporting stones in a quarry site, where a wheel loader digs and loads stones, and trucks transport stones. They need to carry the stones from stone piles to the primary crushers, where stones are crushed into fractions, and proceed to carry the crushed stones to the secondary crushers, which is the destination. During this process, the vehicles must avoid static obstacles (e.g, holes and rocks on the ground, larger than given sizes) and go to the charging point when their battery level is low. In an autonomous quarry, all the operations are performed automatically without human intervention, and the vehicles are autonomous agents that we call *agents* for short, in this paper. To achieve their goal, respectively, the agents need to be able to calculate collision-free paths and schedule their tasks efficiently. Hence, our research problem involves task scheduling, path planning and following, and collision avoidance for multiple autonomous agents. In our previous work [14], we have proposed a two-layer framework for the design of formal models of agents, where task scheduling and path planning belong to the so-called static layer, whereas the path following and avoidance of dynamic obstacles, including the case of overlapping paths of multiple agents, is being dealt with in the dynamic layer. In this paper, we assume that the collision avoidance of dynamic obstacles functions correctly, and focus on the static layer for synthesizing verifiable mission plans.

A. Problem Analysis

For simplicity, henceforth, we call the problem of path planning and task scheduling for agents as **mission planning**. Path planning deals with computing collision-free paths that visit all required target positions (a.k.a. milestones), via efficient algorithms such as Theta* [15] and RRT [16]. We adopt the Theta* algorithm in this paper, since the environment in the problem is a 2-D map, and the algorithm is especially good at generating smooth paths with any-angle turning points in 2-D maps. After the paths are calculated, the agents need to know the assignment and execution order of tasks. For instance, digging stones must be carried out at stone piles before the stones are unloaded into the primary crushers. In this case, digging stones and unloading stones are two tasks, and their execution positions and order must be correct. Additionally, as the machines must guarantee a certain level of productivity, the work has to be completed within some given time. As our solution aims to be general, regardless of the exact type of agents, we formulate the requirements generically, and categorize them as follows:

- *Milestone Matching*. Tasks must be performed at the right milestones.
- *Task Sequencing*. The task execution order must be correct.
- *Timing*. Tasks must be done within prescribed times.
- *Event Reaction*. Some special tasks are only triggered by events under certain circumstances, e.g., when the battery level is low, the agents must go to charge themselves.

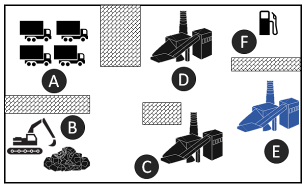
The task-scheduling problem in this paper is similar to a classic scheduling problem called *job-shop* problem [17], which consists of a finite set of jobs to be processed on a finite set of machines. Each job is a sequence of tasks to be executed in a certain order and no tasks can be preempted once started. Each machine can process at most one task at a time and the execution time varies for tasks, but it is fixed. The objective is to assign jobs to machines and decide their starting times in order to minimize the total execution time of all jobs. The problem is NP-hard, so even a simple instance with very restrictive constraints remains difficult to solve [18]. Although the task scheduling in this paper shares many similarities with the job-shop problem, e.g., tasks are non preemptive, our problem has some unique challenges that we introduce in the following section.

B. Uncertainties and Scalability of Mission Planning

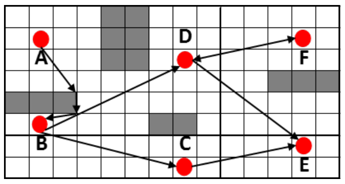
The classic job-shop problem is deterministic as the information is known and fixed. However, the task-scheduling

TABLE I
EVALUATION OF TAMAA AND UPPAAL STRATEGO

	Number of Agents	Number of Explored States	Time
TAMAA	4	2,058,132	20160 s
	5	Out of Memory	Out of Memory
UPPAAL STRATEGO	2	12,031	2670 s
	3	Out of Memory	Out of Memory



(a) A simple example of a quarry



(b) Map decomposition and paths calculation

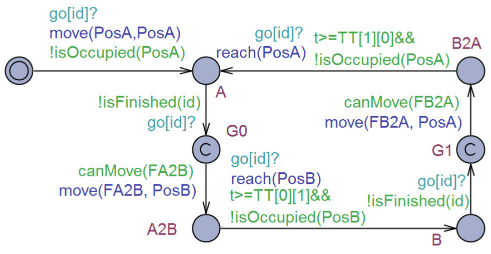
Fig. 3. An example of an autonomous quarry

problem in this paper contains two types of uncertainties, i.e., the uncertain execution time of tasks and uncertain duration of agent movement.

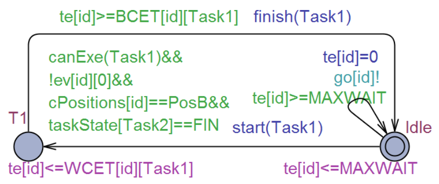
- *Uncertain execution time of tasks*. The execution time of tasks is modeled by time intervals between the BCET (best-case execution time) and WCET (worst-case execution time), which are usually different.
- *Uncertain movement time*. The traveling time between milestones of any agent is not fixed, due to the fact that the destination milestone can be occupied at some time, and thus the agent that is approaching it has then to wait until the destination is available, and the waiting time is uncertain.

These features make our problem more difficult than the classic job-shop problem. When the number of agents increases, the complexity of the problem grows exponentially.

In our previous work [3], we propose a timed-automata-based approach called TAMAA to solve this problem. Although the approach manages to generate mission plans satisfying complex requirements, when the number of agents increases to 5, model checking the TAMAA model exhausts the existing memory due to the state-space explosion problem of model checking [3], [4]. To compare with a similar existing approach, we also employ UPPAAL STRATEGO [9] to synthesize mission plans by verifying the model of TAMAA in this tool (Section 5). Researchers have utilized UPPAAL STRATEGO to solve similar scheduling problems like ours, for e.g., cruise control [19], and floor heating [20], which involve assigning motions to "players" in the environment, to obtain winning strategies. Thus, UPPAAL STRATEGO is considered to be suitable to solve such task-scheduling problems. However, UPPAAL STRATEGO is only able to generate results when the number of agents is less than 3, as it is shown in Table I. In a nutshell, task scheduling for multiple autonomous agents, as an NP-hard problem, remains unsolved when the number of agents is large.



(a) Part of an agent's movement TA



(b) Part of an agent's task execution TA

Fig. 4. The TA model of the example in Figure **??**

IV. MCRL: COMBINING MODEL CHECKING AND REINFORCEMENT LEARNING IN UPPAAL

In this section, we introduce our novel approach called MCRL for mission planning of multiple autonomous agents, which combines model checking with reinforcement learning to alleviate the state-space-explosion problem. The TA model in MCRL originates from TAMAA, therefore, we first briefly introduce TAMAA in the following section to lay the foundation of this method. The formal definitions of the movement and task execution in TAMAA, as well as the model generation algorithms are described in our previous work [3], which the interested reader is referred to for details[1].

A. Timed-Automata-Based Model for Mission Plan Synthesis

We elaborate the TA model in TAMAA by an example illustrated in Figure 3(a). In an autonomous quarry, there are four autonomous trucks starting from milestone `A`, aiming to transport stones at milestone `B`, to the primary crusher at milestone `C` or `D`, and eventually go to the secondary crusher at milestone `E`. There are also autonomous wheel loaders working at milestone `B`, digging stones and loading them into the trucks. A charging point is located at milestone `F`, where all the vehicles go for charging when their battery-level is low.

Initially, the environment is decomposed into a Cartesian grid and the Theta* algorithm [15] is executed to calculate the shortest paths among milestones `A` - `F` (See Figure 3(b)). Note that the trucks only need to choose one primary crusher at position `C` or `D`, to unload stones. Next, a TA-model is automatically generated by TAMAA, based on the decomposed environment. For brevity, an example of the TA model is is only partly shown in Figure 4(a). It models the movement of autonomous trucks between milestones `A` and `B`. The initial location of the automaton has only one outgoing edge to location `A`, indicating that milestone `A` is where the truck starts. Locations `A2B` and `B2A` are created to count the duration of traveling between `A` and `B`. Variable $\mathtt{TT}[m_1][m_2]$ is the

[1]A demo of TAMAA is in https://doi.org/10.5281/zenodo.3614128

travelling time between milestones m_1 and m_2. Locations `G0` and `G1` are committed locations that do not cost any time and are used to diverge the movement to multiple targets. Since some of the milestones are not accessible when they are occupied, the guard function "isOccupied" is utilized (see Figure 4(a)) to judge if the milestones are occupied or not. When the function returns `false`, the edge is enabled but does not trigger the transition, which means that the agent can stay at this location rather than go to the target. Therefore, the incoming edges of locations `A` and `B` are labelled with channels "go[id]?", where "id" represents the index of the agent, and it synchronizes the movement TA with the tasks execution TA.

When an agent is at a milestone, it has three options for the next motion: staying, moving, or executing tasks. TAMAA generates a TA for tasks execution that models these behaviors. This TA is partly depicted in Figure 4(b), with location `Idle` representing the no-operation task, where the agent is allowed to move. The invariant and self loop of location `Idle` represent the time unit of scheduling a moving action. Every "MAXWAIT" time unit, the tasks execution TA informs the movement TA that the agent is ready to move. Location `T1` represents the task "loading", and the guard on its incoming edge regulates that it must be carried out at milestone `B` and after task 2 finished, provided that the charging event does not occur. Location `T1` has an invariant that indicates that the actual execution time of task "loading" must not exceed its WCET. Similarly, the guard on the outgoing edge of `T1` ensures that the agent leaves the location when the execution time is no longer less than BCET.

After the resulting TA model is verified in UPPAAL, execution traces indicating the order of visiting milestones and operating tasks are generated. Since UPPAAL provides three types of execution traces, i.e., the shortest, the fastest, and random ones, we can generate mission plans that take the least number of steps (shortest), or the shortest time (fastest), or random. However, the verification is based on exhaustive model checking, which means that the entire state space is built and stored during the process. Therefore, the number of states of the model grows exponentially as the number of agents increases, and thus the computation time and memory consumption increase dramatically, as it is shown in Table I. In the following, we show how we alleviate this shortcoming, by applying a reinforcement learning algorithm to reduce the state space of model checking the TA model.

B. MCRL Method Description

In order to alleviate the state-space explosion problem, MCRL adopts random simulation instead of exhaustive model checking, and trains the model by the Q-learning algorithm. Figure 5 depicts the process of the method. First, in the data-gathering phase, we obtain the execution traces of the model by Monte Carlo simulation in UPPAAL. We assign rewards to the state-action pairs of the execution traces that satisfy the desired properties, and penalties to the ones containing deadlocks. The traces that either hold the properties or contain deadlocks are ignored and not used in the next phases. There-

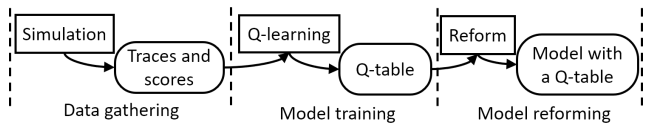


Fig. 5. The process of creating a model using a Q-table

after, in the model-training phase, we adopt the Q-learning algorithm, which is implemented as Java program, to process the traces and populate a Q-table, which is then used to form a new model whose state space is restricted. Details of this approach are presented in the following sections.

1) Model Design and Data Gathering: To differentiate between the state of TA and the state of Q-tables, we define Q-state and Q-action as follows:

Definition 1 (Q-state). *A Q-state is defined as a tuple:*

$$\mathcal{QS} = < TP, MATCH >,$$

where TP is a real number denoting the time point of leaving this state, $MATCH$ is a tuple $< RT, CT, CP, EV, ST >$, where

- *RT is an integer denoting the number of rounds for finishing all tasks,*
- *CT is an integer denoting the index of the current task,*
- *CP is an integer denoting the index of the current milestone,*
- *EV is a set of Boolean values of events, occurred or not,*
- *ST is a set of integers of EST (execution status of tasks) of all the agents in the environment.* ☐

Definition 2 (Q-action). *A Q-action is defined as a tuple:*

$$\mathcal{QA} = < BT, WT, MT, TT >,$$

where,

- *BT is a real number denoting the BCET of the action,*
- *WT is a real number denoting the WCET of the action,*
- *MT is an integer denoting the type of the motion,*
- *TT is an integer denoting the target of the motion.* ☐

"TP" in Definition 1 is created to distinguish "meaningless" execution traces of agents that simply move around and consume plenty of time but do not complete tasks. The Q-states that have the same values of other attributes but own a much larger value of "TP" can be omitted. Note that "ST" in Definition 1 represents the execution status of tasks (EST) of all agents in the environment. It has three possible integer values, i.e., 0: unfinished, 1: finished, or 2: will be finished by the time the current agent arrives at the milestones where other agents locate. As each agent owns a Q-table, when they need to make a decision, i.e., which milestone to go, or which task to execute, they must be aware of the EST of other agents to avoid unnecessary waiting. "MT" in Definition 2 has two possible values, i.e., 0: movement, 1: execution. Correspondingly, "TT" can be the index of the target milestone, or the index of the next task.

All the attributes of a Q-state and a Q-action can be elicited from the TA model generated by TAMAA, and thus, we create a 2-dimensional array in the global declaration of the TA model in UPPAAL to represent the Q-table for each of the agent mode. The state-action pairs in the Q-tables are calculated and stored during the random simulation of the model. UPPAAL 4.1.22[2] provides a new function of simulation that prints information only when certain predicates are true. For example, in the following query, the model is simulated 1000 rounds and 100 time units for each round. Only when the predicate following the simulation query is true, i.e., the Boolean variable "taskAllFinish" turns true, the information within the curly parentheses $(\{\dots\})$ is printed.

$$\text{simulate}[<= 100; 1000]\{...\} : \text{taskAllFinish} == \text{true}$$

By using this function, we can control the simulation to print data when all tasks are finished (good traces), or any of the agents is stuck in a deadlock (bad traces). At the end of each round of the simulation, if the predicate is satisfied, rewards (positive values) are assigned to the state-action pairs in the trace by the functions in the TA model; if a deadlock occurs, penalties (negative values) are assigned to them in a similar way. More precisely, the reward has a value of $MAX - CTime$, where MAX is the maximum simulation time, $CTime$ is the time point of finishing all tasks, whereas penalties have the same fixed value. In this way, the traces that reach the states that satisfy the predicates faster get higher rewards and thus are enhanced by Q-learning.

There are several things about the simulation that deserve further explanation. In UPPAAL, the simulation query subsumes Monte Carlo simulation to simulate the model, which is originally designed for statistical model checking [11]. However, in this paper, we do not adopt this feature of UPPAAL but only utilize the Monte Carlo simulation to explore the state space of the model, and the only two uncertainties in the problem, e.g., uncertain task execution and movement times, are modeled as time-bounded delays that follow a uniform probability distribution. One can change it to an arbitrary choice of time-bounded delays or other probability distributions and still use MCRL to solve the problem. Additionally, the simulation time of each round should not be shorter than the shortest time needed to finish tasks, otherwise the predicate remains false and thus no good trace can be gathered in the simulation. The number of simulation rounds should be set properly so that the gathered data is not only enough for training the model, but also not too large, which would entail unnecessarily long time to process it. When the simulation finishes, UPPAAL prints the state-action pairs into a file, which is used in the model-training phase.

2) Model Training and Reforming: After the state-action pairs are formed in the simulation, we input those data into the Q-learning algorithm, which is implemented as Java program, to populate a Q-table. We illustrate the format of the Q-table as follows:

|Q-state|Q-action|Q-value|

[2]UPPAAL 4.1.22 was published in March 2019 on http://www.uppaal.org/

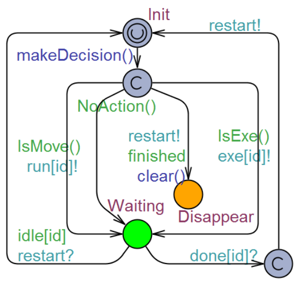


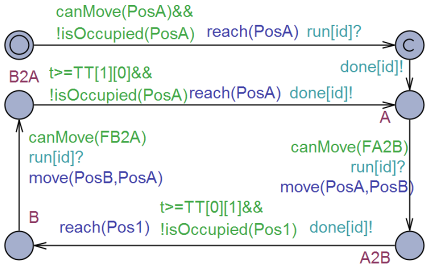
Fig. 6. The conductor TA in the new model with a Q-table

As aforementioned, the Q-tables for agents are stored in a two-dimensional array of the TA model. After running the Q-learning algorithms, Equation 1 guarantees that the Q-values of the state-action pairs are accumulated and converged.

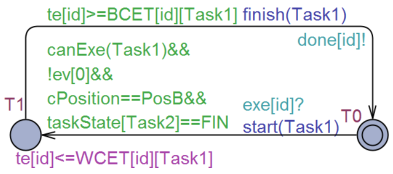
In the model-reforming phase, a new TA model, which we call conductor, is designed for each of the autonomous agents, which looks up the agent's Q-table and sends controlling commands. Since there is no centralized control in the environment, each agent model is equipped with one conductor. However, the conductor contains the Q-tables of all agents in order to decide which one has the priority to act, when multiple agents intend to perform some concurrent actions. Figure 6 depicts the TA model of conductors. The initial location `Init` is urgent to ensure that whenever the agent is ready, it is scheduled immediately. The function `makeDecision()` looks up the Q-table and chooses the state-action pair that owns the highest value among those that match the current state of the agent. Note that, here we only need to compare the attributes in "$MATCH$" but not "TP", because the former is enough to represent the states of the agent and environment. If the chosen action is "execution", the conductor sends an "executing" command to the task execution TA via channel "exe[id]". If the chosen action is "movement", the conductor looks up other agents' Q-tables to obtain their intentions of actions. If they also intend to go to the same milestone where agents are mutually exclusive, the one with the highest value of state-action pair is allowed to move, whereas others have to wait until the former finishes scheduling. Whatever the command is, the conductor TA transfers to the location `Waiting` to wait until the agent finishes its action and responds via the synchronization channel "done[id]".

The fact that locations, expect locations "`Disappear`" and "`Waiting`", are either urgent or committed guarantees that all agents are scheduled simultaneously. Meanwhile, UPPAAL sets the order of running the conductors to be arbitrary, which means agents could act in any order. However, the formal verification of the model equipped with Q-tables can prove that no matter what the acting order is, agents are guaranteed to satisfy the desired properties. This is what traditional RL algorithms cannot provide.

Consequently, the original TA of movement and task execution (see Figures 4(a) and 4(b) as examples) need to be slightly adjusted. As depicted in Figures 7(a) and 7(b), the edges with functions `move()` and `start()` are labelled with channels



(a) Reformed TA of an agent's movement



(b) Reformed TA of an agent's task execution

Fig. 7. Reformed TA model

"run[id]?" and "exe[id]?", respectively. In those two functions, a Boolean variable "idle[id]" is turned to *false*, indicating that the agent is scheduled to start working. However, if the target position is occupied at the moment and multiple agents are not allowed at this milestone, the movement TA should not transfer. Hence, the channel "run[id]" is broadcast so that it does not block the transition in the conductor TA, and the variable "idle[id]" remains *true* because the function `move()` is not invoked.

In this case, the conductor TA needs to be informed when the position is released in order to re-schedule the agent. When the action finishes, the conductor leaves location `Waiting` and moves back to the initial location to start another round of scheduling. As the times of such actions are not determined, the conductor does not know when to restart. Hence, the edges with functions `finish()` and `reach()` in the task execution TA and movement TA are synchronized with the conductor TA via channel "done[id]", so that whenever an agent completes an action, its conductor restarts. The conductor TA could also go back to its initial location via the edge labelled with a broadcast channel "restart?" and a guard "idle[id]", indicating that some other agent has changed its state, and if the current one is idle, it can be re-scheduled. A Boolean variable "finished" is used in the conductor TA. When the agent finishes the requested rounds of work, this variable turns to *true* on the edge going to the location `Disappear`, and the milestone occupied by this agent is released, indicating that it has left the site and stopped. This edge is also labelled with the channel "restart!" to inform other agents for re-scheduling.

3) Mission Plan Synthesis and Analysis: By introducing the conductor TA, the behavior of the autonomous agents is restricted by the Q-table. Hence, if the Q-table is formed by using the state-action pairs satisfying certain predicates, the reformed model is supposed to satisfy the predicates. For example, in the data-gathering phase, the simulation query is designed as follows:

$$\text{simulate}[<=T;\ R]\ \{...\}\text{: forall(i:int[0,N-1]) work[i]} \geq X,$$

TABLE II
TASKS FOR THE AUTONOMOUS AGENTS IN THE EXPERIMENT

	Task	BCET	WCET	precondition
Truck	Load	1	4	none
	Unload	4	4	Load
	Charge	15	15	none
Wheel loader	Dig	2	2	none
	Unload	1	4	Dig
	Charge	15	15	none

where T is the simulation time of each round, R is the number of simulation rounds, N is the number of agents, X is the requested rounds of work. In the case of autonomous trucks, one round of work means starting from the stone pile and eventually unloading stones at the secondary crusher as it is shown in Figure 2. The predicate regulates that if the N agents accomplish X rounds of work, the information in the parenthesis ({...}), i.e., the state-action pairs and their rewards/penalties, is printed. Hence, when the TA model is verified in UPPAAL, properties of the following form:

$$A\Diamond\ \text{forall(i:int[0,N-1]) work[i]} \geq X, \tag{2}$$

should be satisfied, which we demonstrate in Section V. Meeting this kind of properties proves that the Q-table serves as the mission plan that we intend to synthesize, and guides the agents to accomplish a requested amount of work. Additionally, one can also verify properties of the following form:

$$A\Box\ \text{forall(i:int[0, M-1]) positionOccupied[i]} \leq 1 \tag{3}$$

$$\text{batter==low}\ --> \text{movement.charging}\ \&\&\ \text{x} \leq \text{L} \tag{4}$$

Equation 3 requires that milestones are never occupied by multiple agents. Equation 4 requires that the agent goes to the charging point within L time units, when its battery level is low. One can design their own properties, or TA model, to express and verify specific requirements. These properties are impossible to be verified by traditional model checking alone in the cases containing large numbers of agents, due to the exponentially grown state space.

V. Experimental Evaluation

In this section, we evaluate our approach by conducting experiments on MCRL, TAMAA, and UPPAAL STRATEGO to make a comparison. The experiments are conducted in UPPAAL 4.1.22 and UPPAAL STRATEGO 4.1.20-7, on a laptop running an Intel Core i5 processor with 16 GB of RAM and a 64-bit Windows OS. The environment model in this experiment is the one depicted in Figure 3(a), containing 4 static obstacles, 6 milestones, and several autonomous trucks and 1 autonomous wheel loader. To make a comparison with TAMAA and UPPAAL STRATEGO, we vary the number of agents from 2 to 6. The tasks and their execution times for autonomous trucks and wheel loader are shown in Table II.

Experimentation using TAMAA. After configuring the environment, agents, and tasks in the TAMAA tool, we obtain the TA model of task execution, movement, and monitor for the battery-low event. To synthesize the mission plan that transfers all the stones to the secondary crusher with the minimum time consumption, we verify the model in UPPAAL and select the fastest diagnostic trace. The TCTL query designed for the verification is as follows:

$$\text{E}\Diamond\ (\text{stone==0}\ \&\&\ \text{time}\leq\text{LIMIT}), \tag{5}$$

where the variable "stone" represents the volume of the stone pile, whose value is updated in the function "finish()" in the task execution TA, and "time $\leq$ LIMIT" regulates the time limit of finishing the job. The verification results[3] show that TAMAA can generate mission plans that guide the agents to avoid static obstacles and carry all the stones to the destination. However, this approach can only synthesize a certain type of mission plans, e.g., fastest, shortest, or random, as UPPAAL provides these three types of diagnostic traces. When the execution times of tasks are uncertain, these types of mission plans are not sufficient to handle all situations.

Experimentation using UPPAAL STRATEGO. In order to synthesize mission plans in UPPAAL STRATEGO, the TA model in TAMAA needs to be adjusted slightly. See Figure 4(a) as an example, where edges from location `A2B` to location `B` and from location `B2A` to `A` in the movement TA are changed to "uncontrollable" ones, as they are controlled by the environment. Similarly, in the task execution TA, the incoming edges of location `Idle` are changed to "uncontrollable". Thereafter, we verify the model against queries as follows:

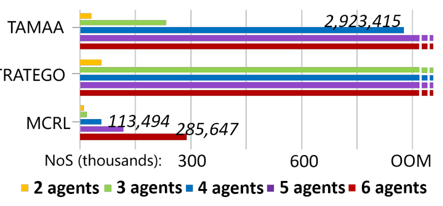
$$\text{strategy MP = control: A}\Diamond\ \text{stone==0} \tag{6}$$

$$\text{E}\Diamond\ (\text{stone==0}\ \&\&\ \text{x}\leq\text{MAXTIME})\ \text{under MP} \tag{7}$$

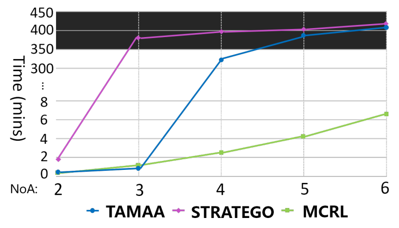
Query 6 utilizes a special syntactical keyword of UPPAAL STRATEGO "control" to synthesize strategies that enable the model to transfer all the stones to the secondary crusher under any circumstances (i.e., A$\Diamond$). Query 7 verifies the model to see whether the agents are able to transfer stones within a time limit (i.e., "time $\leq$ MAXTIME"), when their behaviors are restricted by the strategy (i.e., "under MP"). These queries provide a means of synthesizing and optimizing mission plans that handle the uncertain times of task execution and movement, which is better than TAMAA. However, as UPPAAL STRATEGO still adopts exhaustive model checking to generate mission plans (strategies) by queries like Query 6, the state-space explosion problem is inevitable when the system is large and complex.

Experimentation using MCRL. In this experiment, we train and reform the TA model of TAMAA in the way described in Section IV-B. Then, we synthesize mission plans for 2 to 6 autonomous agents. Figure 8(a) shows the comparison of the number of explored states in the verification using different methods, where "OOM" means the verification runs out of memory and fails to generate a result. As shown in Figure 8(a), MCRL is able to generate a result for all the cases and explores much less states than the other two methods. This demonstrates that the new approach is applicable and scalable to solve the mission-planning problem for larger numbers of

[3]Graphic mission plans in TAMAA: http://doi.org/10.5281/zenodo.3731960



(a) The number of states explored



(b) The time consumption

Fig. 8. Experimental result of the algorithm performance of synthesizing mission plans for different numbers of agents using three methods

agents. We experiment up to 6 agents, however we believe that MCRL is able to handle even larger numbers of agents.

A. Discussion

From the experimental results we can conclude that MCRL can generate results for up to 6 agents, TAMAA for maximum 4 agents, and UPPAAL STRATEGO for maximum 2 agents (see Figure 8(a)). Figure 8(b) shows the computation time of synthesizing mission plans using different methods. Since the difference between times are significantly large, in order to show the data in one graph, the Y-axis is not entirely equidistant, as from 8 we skip numbers. Since TAMAA and UPPAAL STRATEGO fail to generate results when agents are more than 4 and 2 respectively, the black portion of the graph indicates that the methods exhaust memory and return an "out of memory" error after large amounts of time, respectively. The computation time of MCRL is the sum of computing all phases, including data gathering, model training and reforming. As the number of agents grows, the time increase of computation is nearly linear. In the case of 3 agents, TAMAA costs the least time, as UPPAAL STRATEGO and MCRL consider all the situations of uncertain task execution and movement times, which are not dealt with by TAMAA. In the case of 4 agents, TAMAA can still generate results but costs more than 5 hours, whereas MCRL only needs nearly 3 minutes.

Beside the ability of handling larger number of agents, MCRL also provides a way to analyze the synthesized mission plans. Given the model with a Q-table, we can inspect sample mission plans via simulation query as follows:

$$\text{simulate}[<=45;\ 2]\{\ \text{position, task+6}\}, \tag{8}$$

where tasks and positions are encoded as different levels, and the simulation runs 2 rounds and 45 time units for each round. The result of the simulation query is depicted in Figure 9, which indicates that the agent probably goes to the primary crusher at milestone `C` or `D` (see Figure 3(a)), to carry out the unloading task. It is due to the fact that,

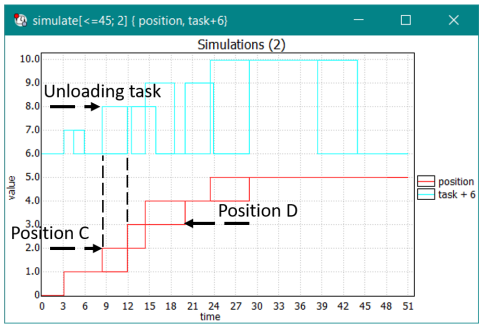


Fig. 9. Two samples of mission plans

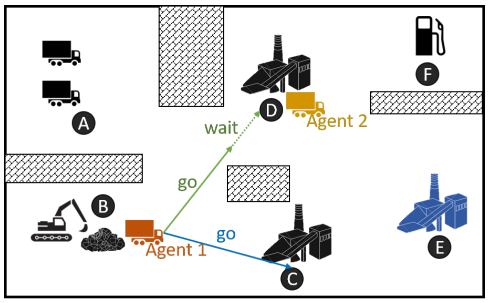


Fig. 10. A scenario where agent 1 learns in the training phase

in case either milestone is being occupied, the agent knows to go to the other one to avoid unnecessary waiting. This "intellegence" is obtained through the model-training phase, which is one of the benefits of adopting Q-learning. One can design various queries to analyze the synthesized mission plans in this integrated method, which is another contribution of MCRL.

By verifying Query 9, we can get the counter-example of the query that enables one to understand how the choice is made.

$$A\Box \text{ agent[1].unload==FIN } imply \text{ movement1.C} \tag{9}$$

As illustrated in Figure 10, when agent 1 finishes the loading task, agent 2 is occupying the primary crusher at position D and unloading stones. At this moment, if agent 1 goes to position D, it needs to predict whether agent 2 is still there, which entails that agent 1 has to wait. To achieve this, we employ the attribute "ST" (execution status of tasks) in Definition 1.

The moment agent 1 finishes its task at position B, it sends a request to obtain the execution status of the agent working at position D, which contains two elements: execution status (ES) and worst-case-execution-time of the current task ($WCET$). Based on the movement TA of agent 1, it is aware of its traveling time of reaching position D. Hence, ES of agent 2 can be easily predicted by the following formula:

$$ES_2(c+\mu_1) = \begin{cases} FIN, & ES_2(c) == FIN, \\ UFIN, & ES_2(c) \neq FIN \ \& \ c+\mu_1 < WCET_2, \\ WFIN, & ES_2(c) \neq FIN \ \& \ c+\mu_1 \geq WCET_2, \end{cases} \tag{10}$$

where c is the current time, and μ_1 is agent 1's traveling time to position D. Formula 10 means: (i) if agent 2's current task has finished at the moment, after the traveling time of agent 1, it is still "finished", or, (ii) if the future time point $(c+\mu_1)$ is less than the $WCET$ of agent 2's current task, it is "unfinished", otherwise (iii) it "will-be-finished". This formula provides a conservative prediction if the $WCET$ is different from the $BCET$ of the task. One can change $WCET$ in Formula 10 with $BCET$ to make aggressive predictions.

Once the states of the model are distinguished in this way, the learning algorithm is able to gradually acquire the optimal decisions for different situations, after multi-rounds of simulation. For example, in the data-gathering phase, we obtain the state-action pairs of agents going to positions C and D. The learning algorithm assigns higher values to the ones with less time consumption, therefore, like the situation in Figure 10, when the predicted execution status of agent 2 is unfinished, agent 1 going to position C is "reinforced" because it is faster. Moreover, Query 11, a modified version of Query 9, can be satisfied, which means the observation in the sample is generally held by the mission plan.

$$\begin{aligned}&(\text{agent[1].load} == FIN \ \&\& \ \text{agent[2].unload} == UFIN) \\ &-- >(\text{agent[1].unload} == FIN \ imply \ \text{movement1.C})\end{aligned} \tag{11}$$

Besides this example, one can specify various requirements by using CTL/TCL queries, and apply MCRL to synthesize mission plans and verify them by model checking. To the best of our knowledge, the ability of synthesizing verifiable mission plans for large numbers of agents is not provided by any existing solution in the literature.

Although promising, one observation of MCRL is that if the simulation rounds in the data gathering phase are not enough, and thus do not obtain enough data, the method is unable to synthesize valid mission plans, even when there exists one solution in the original model. Currently, the number of simulation rounds is decided based on the experience of designers, and a method to infer the number is needed in the future work. However, according to the experiments (see Figure 8(b)), we know that, even including all phases of MCRL, the total time consumption is much less than other two methods when the number of agents grows.

VI. RELATED WORK

Recently, there has been a rising interest in policy synthesis for autonomous systems. Wang et al. [21] propose a novel POMDP (Partially Observable Markov Decision Processes) formulation to synthesis policies over a vast space of probability distributions so that their approach is capable of handling uncertain obstacles. Bouton et al. [22] also employ POMDP for modeling, and their solution enables the autonomous vehicles to adapt to the behavior of other agents. Nikou et al. [23] propose an automata-based solution for controller synthesis of multi-agent path planning, where Metric Interval Temporal Logic (MITL) is used to describe each agent's individual high-level specification. In contrast to these studies, our approach combines model checking and reinforcement learning so that both merits benefit our solution that proves to be accurate and scalable.

The combination of formal methods and learning algorithms is a recent trend that attracts a large body of research work. Li et al. [24] utilize the expressiveness of formal specification languages to capture complex requirements of robotic systems to construct reward functions of reinforcement learning so that they are interpretable. Bouton et al. [25] propose a generic approach to enforce probabilistic guarantees on agents trained by reinforcement learning. Mason et al. [26] present an assured reinforcement learning algorithm using abstract Markov decision processes and probabilistic model checking to establish abstract policies for autonomous agents that are formally verified. As aforementioned, UPPAAL STRATEGO is a new branch of UPPAAL designed by David et al. [9], which adopts reinforcement learning algorithms to refine the synthesized strategies for winning priced timed games. However, as different from these studies, our approach focuses on using reinforcement learning to replace exhaustive model checking for mission-plan synthesis of multi-agents, so that the state-space explosion is alleviated.

To the best of our knowledge, the first attempt to solve the state-space-explosion problem of model checking using reinforcement learning is done by Behjati et al. [27]. These authors propose a bounded rational verification approach for on-the-fly model checking. However, this method is limited to non-timing LTL properties.

VII. Conclusion and Future Work

We present a novel mission-plan synthesis method called MCRL that can handle large numbers of autonomous agents. The method adopts formal modeling to capture the behavior of autonomous agents and Q-learning to train the model and synthesize mission plans in the form of Q-tables. We demonstrate MCRL's ability of handling multiple agents by an experiment, and compare the result with TAMAA and UPPAAL STRATEGO. The experimental results show that the computation time of MCRL increases linearly with the number of agents, whereas the other two approaches show an exponential increase of their computation time, respectively. MCRL is also able to cope with uncertain task execution and movement times, which is not supported by exhaustive model checking in TAMAA. We present means for verifying and analyzing the synthesized mission plans using model checking to ensure safety-critical requirements. As the current approach does not consider unforeseen situations such as undetected obstacles, one direction of the future work is to introduce statistical model checking into our method to cope with probabilistic situations. Another possible direction will focus on integrating Q-learning directly into the generation of the state space with UPPAAL, and possibly on applying other machine learning or AI algorithms to tame verification scalability or guide the model checking itself.

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

References

[1] S. Franklin and A. 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, pp. 21–35.
[2] P. Chandler and M. Pachter, "Research issues in autonomous control of tactical uavs," in *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*. IEEE, 1998.
[3] R. Gu, E. P. Enoiu, and C. 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.
[4] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School*. Springer, 2011, pp. 1–30.
[5] R. Pelánek, "Fighting state space explosion: Review and evaluation," in *FMICS Workshop*. Springer, 2008.
[6] R. S. Sutton, A. G. Barto *et al.*, *Introduction to reinforcement learning*. MIT press Cambridge, 1998, vol. 2, no. 4.
[7] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," *Lecture Notes in Computer Science*, vol. 3098, pp. 87–124, 2004.
[8] C. J. H. Watkins, "Learning from delayed rewards," 1989.
[9] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, and J. H. Taankvist, "Uppaal stratego," in *TACAS*. Springer, 2015.
[10] R. Alur and D. Dill, "Automata for Modeling Real-time Systems," in *Automata, languages and programming*. Springer, 1990, pp. 322–335.
[11] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards, "Statistical model checking for stochastic hybrid systems," *arXiv preprint arXiv:1208.3856*, 2012.
[12] G. Behrmann, A. David, E. Fleury, K. Larsen, D. Lime, and E. Nantes, "Uppaal-tiga: Time for playing games! (tool paper)," 2007.
[13] M. J. Kochenderfer, *Decision making under uncertainty: theory and application*. MIT press, 2015.
[14] R. Gu, R. Marinescu, C. Seceleanu, and K. Lundqvist, "Towards a two-layer framework for verifying autonomous vehicles," in *NASA Formal Methods Symposium*. Springer, 2019, pp. 186–203.
[15] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Artificial Intelligence Research*, vol. 39, 2010.
[16] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.
[17] H. Fisher, "Probabilistic learning combinations of local job-shop scheduling rules," *Industrial scheduling*, pp. 225–251, 1963.
[18] Y. Abdeddaı, E. Asarin, O. Maler *et al.*, "Scheduling with timed automata," *Theoretical Computer Science*, vol. 354, no. 2, 2006.
[19] K. G. Larsen, M. Mikučionis, and J. H. Taankvist, "Safe and optimal adaptive cruise control," in *Correct System Design*. Springer, 2015.
[20] K. G. Larsen, M. Mikučionis, M. Muniz, J. Srba, and J. H. Taankvist, "Online and compositional learning of controllers with application to floor heating," in *TACAS*. Springer, 2016.
[21] Y. Wang, S. Chaudhuri, and L. E. Kavraki, "Bounded policy synthesis for pomdps with safe-reachability objectives," in *International Conference on Autonomous Agents and Multi Agent Systems*. IFAAMS, 2018.
[22] M. Bouton, A. Cosgun, and M. J. Kochenderfer, "Belief state planning for autonomously navigating urban intersections," in *Intelligent Vehicles Symposium*. IEEE, 2017, pp. 825–830.
[23] A. Nikou, D. Boskos, J. Tumova, and D. V. Dimarogonas, "On the timed temporal logic planning of coupled multi-agent systems," *Automatica*, vol. 97, pp. 339–345, 2018.
[24] X. Li, Z. Serlin, G. Yang, and C. Belta, "A formal methods approach to interpretable reinforcement learning for robotic planning," *Science Robotics*, vol. 4, no. 37, 2019.
[25] M. Bouton, J. Karlsson, A. Nakhaei, K. Fujimura, M. J. Kochenderfer, and J. Tumova, "Reinforcement learning with probabilistic guarantees for autonomous driving," *arXiv preprint arXiv:1904.07189*, 2019.
[26] G. R. Mason, R. C. Calinescu, D. Kudenko, and A. Banks, "Assured reinforcement learning with formally verified abstract policies," in *ICAART*, 2017.
[27] R. Behjati, M. Sirjani, and M. N. Ahmadabadi, "Bounded rational search for on-the-fly model checking of ltl properties," in *FSE*. Springer, 2009, pp. 292–307.