Verifiable and Scalable Mission-Plan Synthesis for Autonomous Agents

Rong Gu, Eduard Enoiu, Cristina Seceleanu, and Kristina Lundqvist

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

Abstract. The problem of synthesizing mission plans for multiple autonomous agents, including path planning and task scheduling, is often complex. Employing model checking alone to solve the problem might not be feasible, 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 limitation. 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 means to synthesize mission plans for multi-agent 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 two similar approaches, TAMAA and UPPAAL STRATEGO. The evaluation shows that MCRL performs better for a number of agents greater than three.

1 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 [14]. As this kind of systems bear the promise of increasing people's safety, as well as industrial productivity, by automating repetitive tasks, autonomous technologies are drawing growing attention from both researchers and practitioners. In order to realize their functions, autonomous agents need *mission planning*, including path planning and task scheduling, which is most critical to solve [8]. As path-planning algorithms focus just on calculating collision-free paths towards the destination, they do not cover requirements concerning logical 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 [15], 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 does not scale for a large number of agents, 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 [9] for verification, therefore many studies have explored ways of fighting it [21]. In this paper, we propose a novel method called **MCRL** that combines model checking with reinforcement learning [22] to restrict the state space, in order to synthesize mission plans for large numbers of agents. Our method is based on UPPAAL [5] and leverages the model of autonomous agents generated by TAMAA. Instead of exhaustively exploring and storing the states of the model, MCRL utilizes random 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 used just to sample execution traces randomly. Next, a reinforcement learning algorithm, namely Q-learning [24], 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 aim to synthesize, so we inject it 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 safety requirements, for instance.

We select relevant scenarios involving autonomous agents in a construction site, and conduct experiments with MCRL, as well as TAMAA, and a similar tool called UPPAAL STRATEGO [12]. The experimental results show that MCRL performs much better than the other two, when the number of agents is greater than three. The time of synthesizing mission plans using MCRL increases linearly with the number of agents, whereas for the other two methods it increases exponentially.

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 verifying the synthesis results by model checking.
- An evaluation of the scalability of MCRL, via experiments conducted with the latter, as well as the TAMAA, and UPPAAL STRATEGO tools, on relevant scenarios involving autonomous agents.

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 approach

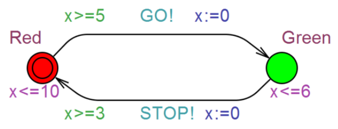


Fig. 1. An example of a timed automaton of a traffic light

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 the paper in Section 7.

2 Preliminaries

In this section, we introduce the background knowledge of this paper.

2.1 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 [2]. UPPAAL [5] 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 UPPAAL TA model of a traffic lights example. Two circles labelled `Red` and `Green`, called *locations*, model the two colors of the traffic light. A clock variable `x` that measures the elapse of time is used in the *invariants* (Boolean expressions over clocks) on locations (e.g., `x<=10`) to enforce an upper bound of delaying in each location, respectively. The directed lines used to connect locations are called *edges*, and can be decorated by *guards*. Guards are Boolean conditions over clocks or discrete variables, and enable traversing the respective edge once they evaluate to true. In UPPAAL, locations denoted by encircled c are called *committed* locations, and require that time does not elapse in those locations and the next edge to be traversed must start from one of them. Clocks can be reset over edges, e.g., `x:=0` in Figure 1, whereas discrete variables can be assigned values, accordingly, via updates on the edges, or via functions that are implemented in the declaration of the TA, by a subset of the C language. A *network* of TA, $B_0 \parallel ... \parallel B_{n-1}$, models a parallel composition of n TA that can synchronize 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 the model.

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

2.2 UPPAAL STRATEGO

UPPAAL STRATEGO [12] is a tool that integrates UPPAAL with two of its branches, i.e., UPPAAL SMC [11] (statistical model checking) and UPPAAL TIGA

[4] (policy synthesis for timed games). 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 is controlled by the players, whereas the latter is controlled by the environment. We refer readers to the literature [12] for details of this tool.

2.3 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 [22]. In this paper, we use one of the model-free reinforcement learning algorithms called *Q-learning* [24], which is usually adopted to learn policies that indicate agents what actions to take at different states. A policy is associated with a state-action value function called *Q function.* The optimal Q function satisfies the Bellman optimality equation:

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

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, and $\underset{a'}{max}\ q^*(s',a')$ 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^*. Although Q-learning is a model-free algorithm, the learning process often relies on a simulation that depends on the form of the model. In this paper, we use the simulation query in UPPAAL to gather the information of state-action pairs, and invoke the Q-learning algorithm to populate a Q-table.

3 Problem Description: An Autonomous Quarry

In this section, we introduce an industrial use case of an autonomous quarry provided by VOLVO CE. The quarry contains various autonomous vehicles, e.g., trucks, wheel loaders, etc. For instance, 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 first carry the stones from the stone piles to the so-called primary crushers, where stones are crushed into fractions, and then proceed to carry the crushed stones to the secondary crushers, where the destination is. 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,

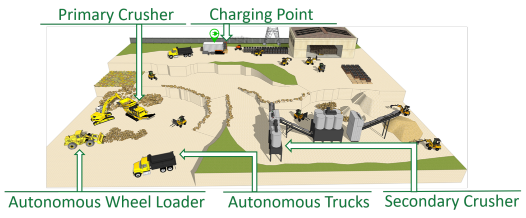


Fig. 2. An example of an autonomous quarry

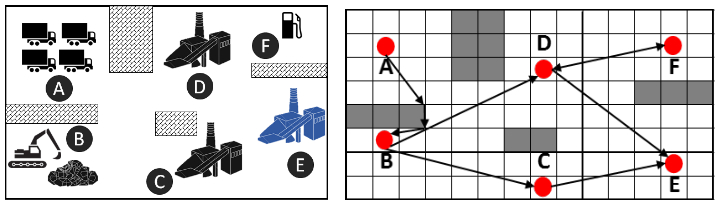
in this paper. To achieve their goals, respectively, the agents need to be able to calculate collision-free paths and schedule their tasks efficiently. Hence, our research problem involves task scheduling, path planning and following, and collision avoidance for multiple agents. In our previous work [17], 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 focus on the static layer, to synthesize verifiable mission plans, while assuming that the collision avoidance of dynamic obstacles functions correctly.

3.1 Problem Analysis

For simplicity, henceforth, we call the problem of path planning and task scheduling for autonomous 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* [10] and RRT [18]. 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 autonomous agents need to follow 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 places and order must be correct. Additionally, as the agents 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 a prescribed time limit.
- *Event Reaction.* Some tasks are triggered by events, 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 the *job-shop* problem [13]. The problem is NP-hard, so even a simple instance with very restrictive constraints remains difficult to solve [1]. Although the task scheduling in this paper shares many similarities with the job-shop problem, e.g., tasks are non preemptive, our problem poses some unique challenges, as described below.



(a) An example of a quarry (b) Map decomposition and the calculated paths

Fig. 3. An example decomposing a map of a quarry and calculating paths on it

3.2 Uncertainties and Scalability of Mission Planning

The classic job-shop problem is deterministic as the information is known and fixed a priori. However, the task-scheduling problem in this paper contains two types of uncertainties, i.e., the uncertain execution time of tasks and uncertain duration of agent movement.

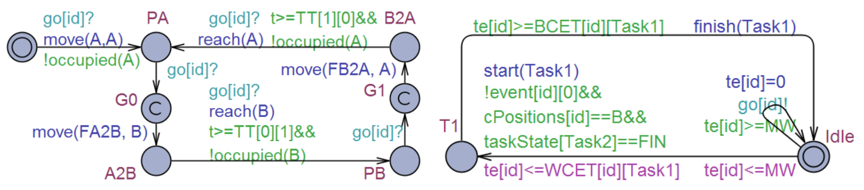
– *Uncertain execution time of tasks.* The execution time of tasks is modeled by time intervals between the BCET (best-case execution time) and WCET (worst-case execution time), which are usually different.
– *Uncertain movement time.* The traveling time between milestones is not fixed for any agent, 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. Moreover, when the number of agents increases, the complexity of the problem grows exponentially.

In our previous work [15], we propose a timed-automata-based approach called TAMAA to solve the mission planning problem. One the one hand, although the approach manages to generate mission plans that satisfy complex requirements, and is able to handle up to 100 milestones and tasks, the synthesized mission plans are restricted to the fastest, shortest, or random strategies, as they are the diagnostic traces resulting from the UPPAAL-based verification. In addition, when the number of agents increases to 5, model checking the TAMAA model exhausts the physical memory due to the notorious state-space explosion problem of model checking [9]. On the other hand, a similar existing approach supported by UPPAAL STRATEGO [12] is able to synthesize strategies considering all possible task execution and traveling times. However, UPPAAL STRATEGO is only able to generate results when the number of agents is less than 3 (see Section 5), as it depends on UPPAAL TIGA that uses exhaustive model checking. In a nutshell, task scheduling for multiple autonomous agents, as an NP-hard problem, remains unsolved when the number of agents is large.

4 MCRL: Combining Model Checking and Reinforcement Learning in UPPAAL

In this section, we introduce our proposed approach, namely MCRL, for mission planning of multiple autonomous agents, which combines model checking with



(a) Part of a movement TA (b) Part of a task execution TA

Fig. 4. The TA models of agent movement and task execution 3

reinforcement learning to alleviate the state-space-explosion problem. The TA model in MCRL originates from TAMAA, therefore, we first briefly introduce TAMAA to lay the foundation of this method. The formal definitions of the movement and task execution, as well as the model generation algorithms are described in our previous work [15], which readers are referred to for details.

4.1 Timed-Automata-Based Model for Mission-Plan Synthesis

We elaborate the TA model in TAMAA by an example illustrated in Figure 3(a). In this quarry, there are four autonomous trucks starting from milestone A, which aim to transport stones from 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 [10] 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, in Figure 4(a), we show a part of the TA model describing the movement of the autonomous trucks between milestones A and B. The outgoing edge from the initial location to location `A` indicates that the trucks start from milestone A. Locations `A2B` and `B2A` are created to measure the duration of traveling between `A` and `B`. Variable $\texttt{TT}[\texttt{m}_1][\texttt{m}_2]$ denotes the travelling time between milestones m_1 and m_2. Locations `G0` and `G1` are used to diverge the movement to multiple targets. Since some of the milestones are exclusive, the guard function `occupied` is utilized to judge if the milestones are occupied or not. When the function returns *false*, the edge is enabled but not triggered, which means that the agent can stay at this location rather than go to the target. Therefore, the channel `go[id]` is used to synchronize the transitions in this TA with the task execution TA (Figure 4(b)) so that the moving actions are triggered periodically. The updating functions `move` and `reach` simply change the values of integers representing the agents' positions and the status of the milestones, respectively. The movement to other milestones is modeled in a similar way.

When an agent is at a milestone, it has three options for the next motion: staying, moving, or executing tasks. TAMAA generates tasks execution TA that model these motions. One such TA is partly depicted in Figure 4(b), where location `Idle` represents the status of no operation, when the agent is allowed

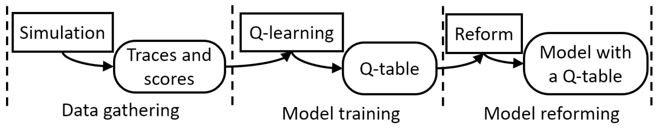


Fig. 5. The process of creating a model using a Q-table

to move. The invariant and self loop of location `Idle` represent the scheduling of the moving action. Every `MW` time units, the task execution TA informs the movement TA, via the synchronization channel `go[id]`, that the agent is ready to move. Location `T1` represents the task "loading", and the guard on its incoming edge regulates that the task must be carried out at milestone `B`, when `Task 2` finishes, provided that the charging event does not occur. Location `T1` has an invariant indicating that the actual execution time of "loading" must not exceed its WCET. Similarly, the guard on the outgoing edge of `T1` ensures that the TA leaves the location when the execution time is greater 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, that is, the shortest, the fastest, and random ones, respectively, we can generate mission plans that take the least number of steps (shortest), or the shortest time (fastest), or are random. However, the verification is based on exhaustive model checking, which means that the entire state space is built and stored during the process. The number of states of the model grows exponentially with the number of agents, hence the computation time and memory consumption increase dramatically. In the following, we show how we alleviate this shortcoming, by applying Q-learning and simulation to explore the state space of the TA model.

4.2 MCRL Method Description

Figure 5 depicts the process of MCRL. First, in the *data-gathering* phase, we obtain the execution traces of the model by the simulation function 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 neither hold the properties nor contain deadlocks are ignored and not used in the next phase. In the *model-training* phase, we adopt Q-learning to process the traces and populate a Q-table, which is then used to form a new model whose state space is restricted.

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

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

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

where TP is a real number denoting the time point of leaving the state, and $MATCH$ 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, i.e., occurred or not,*
– *ST is a set of integers of EST (execution statuses of tasks) of all the agents.* □

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

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

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 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 but complete no tasks. The Q-states that have the same values of other attributes but own a much larger value of "*TP*" can be omitted. "*ST*" in Definition 1 represents the execution status of tasks (*EST*), for all agents in the environment. It has three possible 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 are located. As each agent owns a Q-table, respectively, when it needs to make a decision of where to go, it must be aware of the *EST* of other agents to avoid unnecessary waiting. "*MT*" in Definition 2 has two possible values, that is, 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, hence, 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 models. The state-action pairs and their values are calculated and stored in the array during the random simulation. UPPAAL 4.1.22[1] provides a new function of simulation that prints information only when certain predicates are true. For instance, in the query below, the model is simulated for `R` rounds and `T` time units in 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.

$$\texttt{simulate[<=T; R]}\{...\} : \texttt{taskAllFinish} == \texttt{true} \tag{2}$$

By using this function, we can control the simulation to print the array of the Q-tables when all tasks are finished (good traces), or any of the agents is stuck in a deadlock (bad traces). At the end of each simulation round, 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 $(T - C)^2$, where T is the entire simulation time as used in Query (2), C is the time point of finishing all tasks, whereas penalties have the same fixed

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

value. In this way, the traces that reach the states that satisfy the predicates faster get higher rewards and are thus 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 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 distribution, 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 the entire mission, otherwise the predicate remains false, hence no good trace can be obtained 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, the state-action pairs and their values are printed into files, which are parsed and used in the next phase.

Model Training and Reforming After the state-action pairs are gathered in the simulation, we input these data into the Q-learning algorithm, which is implemented as a Java program, to populate a Q-table. The Q-table is a two-dimensional array of a data structure in the following form:

$$\{Agent\ ID\} \mid \{Q\text{-}state,\ Q\text{-}action,\ Q\text{-}value\},$$

where *Agent ID* is the first dimension of the array, whereas *Q-state*, *Q-action*, and *Q-value* are the elements of the data structure. After running the Q-learning algorithm, the Java program produces C-language code of this array, which is then injected back to the TA model. 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; it 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, respectively. 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 action that owns the highest value among those that match the current Q-state of the agent. Note that we only need to compare the attributes in "*MATCH*" but not "*TP*", as the latter has infinite values and 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

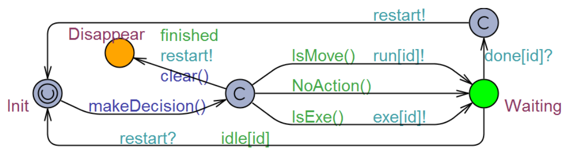


Fig. 6. The conductor TA in the new model equipped with a Q-table

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 when it responds to its conductor via the synchronization channel `done`[`id`].

The fact that all other locations, except 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 alone 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. In the movement TA, channel `go`[`id`] is replaced by a new channel, namely `run`[`id`], as the movement TA is now scheduled by the conductor TA instead of being synchronized with the task execution TA. Similarly, the task execution TA is synchronized with the conductor TA via channel `exe`[`id`] when it starts tasks, and the self-loop of location `Idle` is omitted, since it does not need to trigger the moving transitions any more. In the functions `move()` and `start()`, a Boolean variable `idle`[`id`], which is also used in the conductor TA, is turned to *false*, indicating that the agent is scheduled to start working. However, if the target position is exclusive and occupied at the moment, 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* since 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. However, as the time of carrying out tasks is not determined, the conductor does not know when to restart, hence the non-deterministic delay at location `Waiting` in Figure 6. Therefore, the channel `done`[`id`] is used to synchronize the task execution TA and movement TA with the conductor TA, 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 broadcast channel `restart?` and guard `idle`[`id`], indicating that another agent has finished something and changed its state, so if the current agent 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 to 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.

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:

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

where T is the simulation time of each round, R is the number of simulation rounds, N is the number of agents, and 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 shown in Figure 2. The predicate specifies that if the N agents accomplish X rounds of work, then the information between parentheses ({...}), i.e., the state-action pairs and their rewards/penalties, is printed into text files. Next, the text files are parsed by our Java program, which also implements the Q-learning algorithm and populates Q-tables. Finally, the Q-tables are used to construct the conductor TA that govern the movement and task execution TA.

By verifying queries of the forms below, we can prove that the synthesized mission plans satisfy our requirements described in Section 3. In these queries, $\texttt{te}_n$ and $\texttt{move}_n$ are the task execution TA and movement TA of agent n, respectively. `tasks` is a two-dimensional integer array of agents' task execution status, e.g., finished, or unfinished, `event` is a two-dimensional Boolean array of events' status, and `x` is a clock variable.

– *Milestone Matching.* Query (3) checks that agent's n position is always at milestone P_i, when it is executing task T_i.

$$\texttt{A}\Box\ (\texttt{te}_n\texttt{.T}_i\ \texttt{imply move}_n\texttt{.P}_i) \tag{3}$$

– *Task Sequencing.* Query (4) checks if agent's n precedent task T_{i-1} is always finished (`FIN` means finish), when agent n is executing task T_i.

$$\texttt{A}\Box\ (\texttt{te}_n\texttt{.T}_i\ \texttt{imply tasks[n][i-1]==FIN}) \tag{4}$$

– *Timing.* Query (5) checks if agent n can always finish all its tasks within TL time units, where `M` indicates the last task and `TL` is an integer of time limit.

$$\texttt{A}\Box\ \texttt{((forall(i:int[0,M-1]) tasks[n][i]==FIN) imply x}\leq\texttt{TL)} \tag{5}$$

– *Event Reaction.* Query (6) checks if agent i can always reach milestone P_k within EL time units if event j occurs, where `EL` is an integer of time limit.

$$\texttt{event[i][j] -->}\ (\texttt{move}_n\texttt{.P}_k\ \&\&\ \texttt{x} \leq \texttt{EL}) \tag{6}$$

These queries are impossible to be verified by traditional model checking alone in cases containing large numbers of agents, due to the exponentially grown state space.

5 Experimental Evaluation

In this section, we evaluate MCRL by comparing it with TAMAA and UPPAAL STRATEGO in several experiments. This experimental evaluation is 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 used in this experiment is depicted in Figure 3(a), and contains 4 static obstacles, 6 milestones, several autonomous trucks and 1 autonomous wheel loader. For comparing TAMAA and UPPAAL STRATEGO, we vary the number of agents from 2 to 6.

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 carries all the stones 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:

$$\texttt{E}\Diamond\ \texttt{(stone==0}\ \&\&\ \texttt{x}\leq\texttt{LIMIT),} \tag{7}$$

where variable "stone" represents the volume of the stone pile, whose value is updated in the function "finish()" in the task execution TA, and "x ≤ LIMIT" regulates the time limit of finishing the job. The verification results[2] show that TAMAA can generate mission plans that 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. As an example, in Figure 4(a), 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:

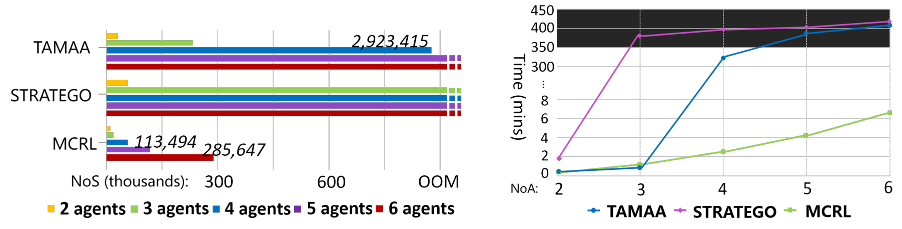
$$\texttt{strategy MP = control: A}\Diamond\ \texttt{stone==0} \tag{8}$$

$$\texttt{E}\Diamond\ \texttt{(stone==0}\ \&\&\ \texttt{x}\leq\texttt{LIMIT) under MP} \tag{9}$$

Query (8) utilizes a special syntactical keyword of UPPAAL STRATEGO, namely "control", to synthesize strategies that enable the model to transfer all the stones to the destination under any circumstances (i.e., "A◊"). Query (9) verifies the model to see whether the agents are able to transfer stones within a time limit, 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 (8), 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 4.2. Then, we synthesize mission plans for 2 to 6 autonomous agents. Figure 7(a) shows the comparison of

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



(a) The number of explored states for checking Query (7) or (8)

(b) The computation time of checking Query (7) or (8)

Fig. 7. Experimental result of the algorithm performance of synthesizing mission plans for different numbers of agents using three methods

the number of explored states in the verification, using different methods, where "NoS" means the number of explored states, "OOM" means that the verification runs out of memory and fails to generate a result. MCRL is able to generate a result for all the cases and explores much less states than the other two methods. Figure 7(b) shows the computation times of different methods, where "NoA" means the number of agents. Since the difference between times is 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. However, as the number of agents grows, the increase of computation time for MCRL is nearly linear. In case of 3 agents, TAMAA costs the least time, as UPPAAL STRATEGO and MCRL consider all the situations of uncertain task execution and movement time, which are not dealt with by TAMAA. In case of 4 agents, TAMAA can still generate results but costs more than 5 hours, whereas MCRL only needs about 3 minutes. This demonstrates that the new approach is applicable and scalable to solve the mission-planning problem for larger numbers of agents. We experiment up to 6 agents, however we believe that MCRL is able to handle even larger numbers of agents.

After the mission plans are synthesized, one can verify them by using various queries supported by UPPAAL, e.g., queries (3) to (6). Additionally, by simulating the reformed model, one can further analyze the mission plans in order to optimize the resources in the environment. We refer readers to previous work [16] for detailed introduction of the further analysis of the mission plans.

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 or optimal 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.

6 Related Work

Recently, there has been a rising interest in policy synthesis for autonomous systems. Wang et al. [23] propose a novel Partially Observable Markov Decision Processes (POMDP) formulation to synthesis policies over a vast space of probability distributions. Bouton et al. [6] also employ POMDP for modeling, and their solution enables the autonomous vehicles to adapt to the behavior of other agents. Nikou et al. [20] 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. Compared with these studies, our approach combines model checking and reinforcement learning so that both methods' strengths 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. [19] utilize the expressiveness of formal specification languages to capture complex requirements of robotic systems and construct reward functions of reinforcement learning so that they are interpretable. Bouton et al. [7] propose a generic approach to enforce probabilistic guarantees on agents trained by reinforcement learning. As aforementioned, UPPAAL STRATEGO is a new branch of UPPAAL designed by David et al. [12], 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-agent systems, 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. [3]. These authors propose a bounded rational verification approach for on-the-fly model checking. However, this method is limited to non-timing LTL properties.

7 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 computation time of MCRL increases linearly with the number of agents, where 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 time, which is not supported by TAMAA. We present means for verifying and analyzing the synthesized mission plans using model checking to ensure safety-critical requirements. Future work 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. Synthesizing mission plans of agents working for much longer time will be another direction, which would complicate the problem even more.

Acknowledgement. The research leading to the presented results has been under-taken within the research profile DPAC - Dependable Platform for Autonomous Sys-tems and Control project, funded by the Swedish Knowledge Foundation, grant num-ber: 20150022.

References

1. Abdeddaı, Y., Asarin, E., Maler, O., et al.: Scheduling with timed automata. vol. 354. Elsevier (2006)
2. Alur, R., Dill, D.: Automata for Modeling Real-time Systems. In: Automata, languages and programming, pp. 322–335. Springer (1990)
3. Behjati, R., Sirjani, M., Ahmadabadi, M.N.: Bounded rational search for on-the-fly model checking of ltl properties. In: FSE. pp. 292–307. Springer (2009)
4. Behrmann, G., David, A., Fleury, E., Larsen, K., Lime, D., Nantes, E.: Uppaal-tiga: Time for playing games! (tool paper). In: Proceedings of the 2007 Computer Aided Verification. Springer Berlin Heidelberg (2007)
5. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. vol. 3098, pp. 87–124. Springer (2004)
6. Bouton, M., Cosgun, A., Kochenderfer, M.J.: Belief state planning for autonomously navigating urban intersections. In: Intelligent Vehicles Symposium. pp. 825–830. IEEE (2017)
7. Bouton, M., Karlsson, J., Nakhaei, A., Fujimura, K., Kochenderfer, M.J., Tumova, J.: Reinforcement learning with probabilistic guarantees for autonomous driving. arXiv preprint arXiv:1904.07189 (2019)
8. Chandler, P., Pachter, M.: Research issues in autonomous control of tactical uavs. In: Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207). IEEE (1998)
9. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: LASER Summer School. pp. 1–30. Springer (2011)
10. Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. vol. 39. Journal of Artificial Intelligence Research (2010)
11. David, A., Du, D., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Statistical model checking for stochastic hybrid systems (2012)
12. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal stratego. In: TACAS. Springer (2015)
13. Fisher, H.: Probabilistic learning combinations of local job-shop scheduling rules. In: Industrial scheduling. pp. 225–251. Englewood Cliffs, NJ: Prentice Hall. (1963)
14. Franklin, S., Graesser, A.: Is it an agent, or just a program?: A taxonomy for autonomous agents. In: International Workshop on Agent Theories, Architectures, and Languages. pp. 21–35. Springer (1996)
15. Gu, R., Enoiu, E.P., Seceleanu, C.: 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)
16. Gu, R., Enoiu, E.P., Seceleanu, C., Lundqvist, K.: Combining model checking and reinforcement learning for scalable mission planning of autonomous agents. Tech. rep. (May 2020)
17. Gu, R., Marinescu, R., Seceleanu, C., Lundqvist, K.: Towards a two-layer framework for verifying autonomous vehicles. In: NASA Formal Methods Symposium. pp. 186–203. Springer (2019)
18. LaValle, S.M.: Rapidly-exploring random trees: A new tool for path planning. In: Technical Report (1998)
19. Li, X., Serlin, Z., Yang, G., Belta, C.: A formal methods approach to interpretable reinforcement learning for robotic planning. vol. 4. Science Robotics (2019)
20. Nikou, A., Boskos, D., Tumova, J., Dimarogonas, D.V.: On the timed temporal logic planning of coupled multi-agent systems. vol. 97, pp. 339–345. Elsevier (2018)

21. Pelánek, R.: Fighting state space explosion: Review and evaluation. In: FMICS Workshop. Springer (2008)
22. Sutton, R.S., Barto, A.G., et al.: Introduction to reinforcement learning, vol. 2. MIT press Cambridge (1998)
23. Wang, Y., Chaudhuri, S., Kavraki, L.E.: Bounded policy synthesis for pomdps with safe-reachability objectives. In: International Conference on Autonomous Agents and Multi Agent Systems. IFAAMS (2018)
24. Watkins, C.J.H.: Learning from delayed rewards. King's College, Cambridge (1989)