International Journal on Software Tools for Technology Transfer (2022) 24:395–414
https://doi.org/10.1007/s10009-022-00657-z

GENERAL

Special Issue: FMICS 2019/2020

Verifiable strategy synthesis for multiple autonomous agents: a scalable approach

Rong Gu[1] · Peter G. Jensen[2] · Danny B. Poulsen[2] · Cristina Seceleanu[1] · Eduard Enoiu[1] · Kristina Lundqvist[1]

Accepted: 8 March 2022 / Published online: 30 March 2022

Abstract
Path planning and task scheduling are two challenging problems in the design of multiple autonomous agents. Both problems can be solved by the use of exhaustive search techniques such as model checking and algorithmic game theory. However, model checking suffers from the infamous state-space explosion problem that makes it inefficient at solving the problems when the number of agents is large, which is often the case in realistic scenarios. In this paper, we propose a new version of our novel approach called MCRL that integrates model checking and reinforcement learning to alleviate this scalability limitation. We apply this new technique to synthesize path planning and task scheduling strategies for multiple autonomous agents. Our method is capable of handling a larger number of agents if compared to what is feasibly handled by the model-checking technique alone. Additionally, MCRL also guarantees the correctness of the synthesis results via post-verification. The method is implemented in UPPAAL STRATEGO and leverages our tool *MALTA* for model generation, such that one can use the method with less effort of model construction and higher efficiency of learning than those of the original MCRL. We demonstrate the feasibility of our approach on an industrial case study: an autonomous quarry, and discuss the strengths and weaknesses of the methods.

Keywords Autonomous agents · Synthesis · Model checking · Reinforcement learning

1 Introduction

With the rise of artificial intelligence (AI), autonomous agents, such as driverless cars, drones, and autonomous construction equipment, are increasingly integrated in all aspects of society. Autonomy requires that the involved agents are able to sense the often unpredictable environment and act on changes over time in order to pursue their goals [20]. For instance, in a construction site, the autonomy of the agents (machines) bear the promise of increasing people's safety, while improving industrial productivity by automating repetitive tasks. Two major problems need to be solved to achieve the autonomy during operations: *path planning* and *task scheduling*. Computing both automatically is called *mission plan synthesis*. Path planning aims to calculate a path that visits all target positions (a.k.a. milestones) and avoids static obstacles. Algorithms like A* [35], Theta* [15], and Rapidly exploring Random Tree [29] are adopted widely for calculating the shortest path between two points in a 2-D map. While path plans specify the movement between every two milestones, the order in which tasks should be completed at milestones is often dealt with as a subsequent optimization problem. The optimization problem of task scheduling is often paired with additional constraints, such as finishing tasks in a certain order, and repeating some tasks until the agents are informed to execute other tasks. The requirements on tasks can involve temporal conditions, e.g., "always start

✉ Rong Gu
rong.gu@mdu.se

Peter G. Jensen
pgj@cs.aau.dk

Danny B. Poulsen
dannybpoulsen@cs.aau.dk

Cristina Seceleanu
cristina.seceleanu@mdu.se

Eduard Enoiu
eduard.paul.enoiu@mdu.se

Kristina Lundqvist
kristina.lundqvist@mdu.se

[1] Mälardalen University, Västerås, Sweden

[2] Aalborg University, Ålborg, Denmark

task *A* before task *B* is finished," and timing constraints, e.g., "always finish all tasks within 8 hours". All these constraints make the task scheduling difficult to complete in practice, in particular if constraints on computation time are given. In fact, a simplified version of task scheduling is the classic *job-shop* problem [19], which is NP hard [1].

In our previous work [22], we proposed an approach based on Timed Automata (TA) and Timed Computation Tree Logic (TCTL) to formally describe the agents' movement and task execution, as well as their requirements, respectively, to facilitate synthesis of plans by model checking. The approach has been implemented as a tool named TAMAA (Timed-Automata-based Mission planner for Autonomous Agents). The tool shows the feasibility of solving the mission-planning problem by using model checking when time of movement and task execution are fixed. However, TAMAA has two limitations: (i) if moving and executing tasks take unpredictable durations, TAMAA fails to generate complete mission plans that address all eventualities; (ii) TAMAA alone does not scale well with the number of agents growing, as the state space of the model explodes when the number of agents becomes large.

In this paper, we first solve problem (i) by synthesizing comprehensive strategies of timed games (TG), which use time intervals instead of fixed times as the moving and task execution times. TG are solvable by UPPAAL TIGA [5], which is for synthesizing strategies of TG. The TAMAA-generated TA can be re-used and easily converted into TG by labeling actions as controllable and uncontrollable ones in UPPAAL TIGA. As the TG models consider all possible times of task execution and moving within given intervals, and UPPAAL TIGA utilizes liveness properties to find the state-action pairs of the models that always eventually reach the goal states, the results represent the complete mission plans that address all eventualities.

However, as the synthesis in UPPAAL TIGA is still based on exhaustive symbolic exploration, this method inevitably suffers from the same state-space explosion problem as ordinary TAMAA. The state-space explosion problem is one of the most stringent issues when employing exhaustive search techniques such as model checking [14]; therefore, many studies have explored ways of fighting it [9,34]. To solve problem (ii), we proposed a novel method called *MCRL* [23] that combines model checking with reinforcement learning [36] to synthesize mission plans for large numbers of agents. Instead of exhaustively exploring the state space, MCRL samples the state space randomly within a time frame, and then uses these samples to train the agent models so that their behavior becomes increasingly efficient in reaching their goals such as finishing all tasks. Since the method does not need to traverse every state of the model, state-space explosion is avoided.

In this paper, we improve the original MCRL by integrating it with UPPAAL STRATEGO[1] [18], which is a tool that integrates the UPPAAL model checker, simulation, algorithmic synthesis (i.e., UPPAAL TIGA), and learning-based synthesis. Thanks to the integration, we can merge the sampling phase and the training phase of MCRL so that the temporary synthesis results can be used in the simulation and accelerate it to get to the goal state. Specifically, after each round of simulation, the sampled trace is provided to a learning module, which runs *Q-learning* [38] to populate a Q-table. Q-learning is a reinforcement learning algorithm that calculates a value for each state-action pair in the trace. The state-action pairs and their values are stored in the Q-table, which is then used as a strategy. *Strategies* are mission plans that constantly provide suggestions of actions to the agents, at each of their states. The suggested actions include moving to a certain milestone, or executing a certain task. After the learning algorithm is invoked, an intermediary strategy is generated, which does not necessarily cover all the eventualities in the unpredictable environment. However, it is still input into the next round of simulation so that the simulator explores the state space in a heuristic way, by increasing the probabilities of choosing the actions that have higher values than other actions of the same state in the Q-table. In this way, the simulation can get to the goal state increasingly likely and faster.

Although exhaustive model checking suffers from state-space explosion, it is beneficial at ensuring the correctness of the synthesized strategies, that is, the latter satisfy all requirements, and the completeness, meaning that the synthesized strategies cover all the eventualities in the unpredictable environment. Therefore, we leverage exhaustive model checking after a strategy is synthesized by the Q-learning algorithm, to verify if the agent models behave according to the requirements specification, under the control of the strategy. In this work, we further extend the model checker of UPPAAL STRATEGO to support the exhaustive verification of the learned strategies. In this way, model checking and reinforcement learning are combined effectively by our method (MCRL), in solving the mission-plan synthesis problem of multiple autonomous agents. Moreover, the new version of MCRL reuses the automation of model construction provided by our toolset named *MALTA*.

To summarize, this paper is an extension of our previous work [23] and the new contributions are:

- An improved version of TAMAA that employs UPPAAL TIGA to synthesize mission plans (i.e., strategies) that consider all the eventualities in the respective environments.
- A new version of MCRL integrated in UPPAAL STRATEGO, which provides advantages such as a merged phase of

[1] https://people.cs.aau.dk/~marius/stratego/

sampling and training that benefit each other.
The new version of MCRL is implemented in an extensible scheme with the help of UPPAAL STRATEGO, so that users can replace the learning algorithm with their own pre-compiled libraries.
- Experimental evaluation of the new methods by applying them on an industrial case study to demonstrate their merits and weaknesses.

The remainder of the paper is organized as follows. In Sect. 2, we introduce the preliminaries of this paper, that is, definitions of timed automata, (stochastic) timed games, (stochastic) strategies, and reinforcement learning. Section 3 describes the problem of strategy synthesis for multiple agents, as well as its challenges. Section 4 presents all the solutions and their application ranges. This section provides a general view of the methods and describes their differences. In Sect. 5, we overview our previous method TAMAA, which is the foundation of the new methods, and introduce the improved version of TAMAA in UPPAAL TIGA. Section 6 continues with the introduction of the learning-based method for strategy synthesis. It first analyzes the root of the scalability problem of TAMAA, after which it describes the new MCRL. Section 7 presents the implementation of MCRL and the integration with UPPAAL STRATEGO, as well as the automated model generation supported by our existing toolset. Next, we describe the evaluation experiments in Sect. 8, where we present the results of the experiments, as well as a discussion of the merits and weaknesses of the simulation-based methods and the improved version of TAMAA. In Sect. 9, we compare to related work, before concluding the paper in Sect. 10.

2 Preliminaries

2.1 Timed automata and timed games

Definition 1 A *Timed Automaton* TA [2] is a tuple:

$$\mathcal{A} =< L, l_0, X, \Sigma, E, I >, \tag{1}$$

where L is a finite set of locations, l_0 is the initial location, X is a finite set of nonnegative real-valued clocks, Σ is a finite set of actions, $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is a finite set of edges, where $\mathcal{B}(X)$ is the set of *guards* over X, that is, conjunctive formulas of clock constraints of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in X, n \in \mathbb{N}, \bowtie \in \{<, \leq, =, \geq, >\}$, and $I : L \to \mathcal{B}(X)$ assigns an invariant to each location.

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

(i) *delay transitions*: $(l, v) \xrightarrow{d} (l, v \oplus d)$, where $v \oplus d$ is the result obtained by incrementing all clocks of the automaton with the delay amount d such that $v \oplus d \models I(l)$, and

(ii) *discrete transitions*: $(l, v) \xrightarrow{a} (l', v')$, corresponding to traversing an edge $l \xrightarrow{g,a,r} l'$ for which the guard g evaluates to *true* in the source state (l, v), $a \in \Sigma$ is an action, r is the clock reset set, and clock valuation v' of the target state (l', v') is obtained from v by resetting all clocks in r such that $v' \models I(l')$.

We denote the timed transition system of a TA $\mathcal{A}$ by $S_{\mathcal{A}}$. A run π of a TA $\mathcal{A}$ is a sequence of alternating delay and discrete transitions of its $S_{\mathcal{A}}$: $\pi = (l_0, v_0) \xrightarrow{d_1} (l_0, v_1) \xrightarrow{a_1} (l_1, v_1') \xrightarrow{d_2} \ldots \xrightarrow{d_n} (l_{n-1}, v_n) \xrightarrow{a_n} (l_n, v_n')$, where d_i refers to a delay transition and a_i refers to a discrete transition. We denote the set of finite runs of $\mathcal{A}$ starting from (l_0, v_0) as $\Pi_f(\mathcal{A})$.

A *Timed Game* $\mathcal{G}$ (TG) [13] is a TA whose actions Σ are partitioned into controllable (Σ_c) and uncontrollable (Σ_u) actions. The timed transition system, runs, and a set of runs of a TG are denoted as $S_{\mathcal{G}}$, π, and $\Pi(\mathcal{G})$, respectively. TG is a useful mathematical model, suitable to describe a system consisting of several players that compete or collaborate to win the game, e.g., by finishing their tasks. Each player can take arbitrary numbers of actions before other players act. The numbers depend on the design of the TG. Informally, a strategy is a function that during the course of the TG constantly suggests the players what to do next in order to win the game. The suggestion is either a controllable action $a \in \Sigma_c$ or a delay. Delays in strategies are denoted as λ, which do not indicate the lengths of delays, whereas the symbol d_i used in the definition of runs refers to concrete delays with specific lengths. The formal definition of strategies is as follows, where $last(\pi_f)$ is used to denote the last state of a finite run π_f:

Definition 2 (*Strategy*) Let $\mathcal{G} =< L, l_0, X, \Sigma_c \cup \Sigma_u, I >$ be a TG. A strategy σ over $\mathcal{G}$ is a partial function: $\pi_f \to \Sigma_c \cup \{\lambda\}$ such that for any finite run π_f ending in state q (i.e., $q = last(\pi_f)$), if $a \in \sigma(\pi_f) \cap \Sigma_c$, then there must exist a transition $q \xrightarrow{a} q' \in S_{\mathcal{G}}$.

Definition 2 indicates that a strategy is a function that takes finite runs of the TG as input and output controllable actions or delays as suggestions of actions to the agents. If the strategy σ is *memoryless*, that is, the decisions on actions depend only on the current state, it can be represented as a function: $last(\pi_f) \to \Sigma_c \cup \{\lambda\}$. In this paper, we focus on *memoryless* and *non-lazy winning* strategies [17], which

either urgently decide on a controllable action or *wait* until the environment acts.[2]

2.2 Stochastic timed games and stochastic strategies

In principle, more information is often known of the environment, for instance, the likelihood of actions or the probability distribution of delays. In this section, we consider *Stochastic Timed Games*, where a stochastic environment is assumed. The environment makes choices of delay and uncontrollable actions stochastically, according to a density function for a given state.

We define the Stochastic Timed Game as a Timed Markov Decision Process (TMDP) [17]:

Definition 3 (*Stochastic Timed Games*) A Stochastic Timed Game (STG) is a TMDP $\mathcal{P} =< \mathcal{G}, \preceq^{\mathrm{u}} >$, where $\mathcal{G} = < L, l_0, X, \Sigma_c \cup \Sigma_u, E, I >$ is a TG, and μ^{u} is a family of density-functions. Let $\mu^{\mathrm{u}}_{\mathrm{q}}(d, u) \in \mathbb{R}_{\geq 0}$ be a member of μ^u, which assigns a probability density of the environment taking the uncontrollable action "u" after a delay of "d" at the state "q," where $\{\mu^{\mathrm{u}}_{\mathrm{q}} : \exists l \exists v.q = (l, v)\}$, $u \in \Sigma_u$ is an uncontrollable action, and q is a state (l, v).

Stochastic strategies [17] for STG are correspondingly defined as follows:

Definition 4 (*Stochastic Strategy*) A stochastic strategy μ^c for a STG is a family of density-functions. Let $\mu^{\mathrm{c}}_{\mathrm{q}}(d, c) \in \mathbb{R}_{\geq 0}$ be a member of μ^c, which assigns a probability density of the controller taking the controllable action "c" after a delay of "d" from state "q," where $\{\mu^{\mathrm{c}}_{\mathrm{q}} : \exists l \exists v.q = (l, v)\}$, $c \in \Sigma_c$ is a controllable action, and q is a state (l, v).

Remark 1 The STG models are for sampling the state-action pairs in the corresponding TG. They are used in the simulation and learning phases of MCRL. The TG models, which reflect the agents' behavior more realistically, are used in the verification phase of MCRL, and the algorithmic synthesis in UPPAAL TIGA.

2.3 UPPAAL, UPPAAL TIGA, and UPPAAL STRATEGO

2.3.1 UPPAAL

UPPAAL [6] is a state-of-the-art model checker for real-time systems. It supports modeling, simulation, and model checking, and uses an extension of TA as the modeling formalism.

We use an example depicted in Fig. 1 to illustrate a simple UPPAAL TA (UTA) modeling traffic lights. *Locations* are

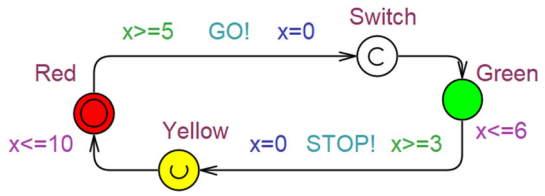


Fig. 1 An example of a UPPAAL timed automaton (UTA) of a traffic light

circles, such as the ones labeled `Red` and `Green`, which model the two colors of the traffic lights. The initial location is the double circle (i.e., `Red`). One UTA can have only one initial location. The UTA's *edges* are directed lines that connect locations, which can be decorated by *guards*. A clock variable `x` is defined to measure the elapse of time, and used in the *invariants* on locations (e.g., `x<=6`), which specify how long the UTA can delay on that location, and *guards* on edges (e.g., `x>=3`).

A *network* of UTA models a parallel composition of UTA that can synchronize via *channels* (i.e., $a!$ is synchronized with $a?$ by handshake). In Fig. 1, the edges are labeled with channels named `STOP` and `GO`, which synchronize this UTA with other UTA. In UPPAAL, there are two special kinds of locations, namely *urgent* and *committed* locations. *Urgent* locations are denoted by encircled u, and require that the time does not elapse on those locations (e.g., `Yellow`); *committed* locations are denoted by encircled c, and require that not only no time elapses there but also the next edge to be traversed must start from one of the committed locations in the network of UTA (e.g., `Switch`). UTA also extends TA by introducing discrete data variables that can be updated via functions on edges. Functions are written in a subset of the C language. Clocks can be reset over edges, e.g., `x=0` in Fig. 1.

The UPPAAL queries that we verify in this paper are properties of the following form, where p is an atomic proposition over the locations, clocks, and data variables of the UTA: (i) **Invariance**: A`[]` p meaning that for all runs, for all states in each run, p is satisfied, (ii) **Liveness**: A`<>`p meaning that for all runs, p is satisfied by at least one state in each run, and (iii) **Reachability**: E`<>`p meaning that there exists a run where p is satisfied by at least one state of the run.

2.3.2 UPPAAL TIGA and UPPAAL STRATEGO

UPPAAL TIGA [5] is an extension of UPPAAL, which supports solving games based on TG with respect to the temporal properties aforementioned. In this paper, we use UPPAAL TIGA to solve our task scheduling problem in the first solution based on game-theoretic synthesis. UPPAAL STRATEGO [18] is a tool that integrates UPPAAL with two of its branches, that is, UPPAAL SMC [16] (statistical model checking) and UPPAAL TIGA [5]. In addition, it also supports learning-based algo-

[2] This kind of strategies is shown to suffice for optimal scheduling of Duration Probabilistic Automata [27].

rithms for solving STG, and we use this tool to develop our second solution to strategy synthesis that is based on simulation and learning.

2.4 Reinforcement learning

MCRL employs *reinforcement learning* (*RL*) for strategy synthesis. *RL* is a kind of machine learning method for training reactive systems by rewarding desired behaviors and/or punishing undesired ones. Agents that constantly act in an environment and receive feedback (i.e., rewards/penalties) from the environment are reactive systems. *RL* aims to calculate how agents should take actions in an environment, in order to maximize the accumulated rewards of actions. Model-free *RL*, such as *Actor-Critic algorithms* [28], relies on samples from the environment, which can be a model or a real environment, to estimate the rewards of the next state-action pairs. Model-based *RL*, such as *Dynamic Programming* [36], uses the model's predictions or distributions of the next state-action pairs and their rewards to calculate optimal actions.

Q-learning is one of the model-free algorithms, which, at the limit, converges to *optimal policies* for reactive agents in a stochastic environment. Policies are associated with a state-action value function called *Q function*. The optimal Q function satisfies the Bellman optimality equation:

$$q^*(s,a) = \mathbb{E}[R(s,a) + \gamma \max_{a'} q^*(s',a')], \tag{2}$$

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 $\max_{a'} q^*(s',a')$ represents the maximum reward that can be achieved by any possible next state-action pair (s',a'). The Bellman equation calculates the rewards of state-action pairs by considering both the current reward and the discounted maximum future reward. Watkins [38] shows that under the assumption of sufficient repeated sampling, the Q-learning algorithm converges towards the optimal Q-values and thus the solution to the Bellman equations. These values are stored in Q-tables, which serve as the strategies that we aim to synthesize.

3 Problem description and analysis

In this section, we introduce the *autonomous quarry* that serves as the industrial case study provided by VOLVO Construction Equipment (CE) in Sweden. Based on this practical case study, we formulate our research problem and two associated challenges.

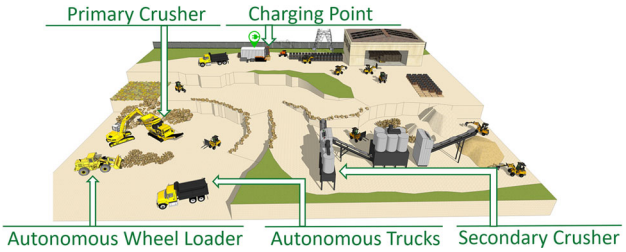


Fig. 2 An example of an autonomous quarry

3.1 An industrial case study: the autonomous quarry

As depicted in Fig. 2, the quarry contains various autonomous vehicles, e.g., trucks and wheel loaders, which are the agents in the environment.

A typical mission of the agents is to transport stones from stone piles to crushers. Specifically, wheel loaders first dig stones at the stones piles and load them into trucks that are responsible for transporting the stones to crushers. The primary crushers crush the stones into fractions, after which trucks load the crushed stones and transport the material to the secondary crushers, which is the final destination of the stones. During the transportation, the agents (that is, autonomous wheel loaders and trucks) must avoid static obstacles (e.g., holes and rocks on the ground, larger than given sizes) and dynamic obstacles (e.g., humans, other mobile machines). In brief, these agents must be able to plan their paths to the target positions (a.k.a. milestones) and schedule their tasks so that the entire mission could be accomplished respecting some requirements, e.g., quarrying 1500 m^3 of stones per day.

Generalizing from this case study, our research problem of mission planning involves task scheduling, path planning and following, and collision avoidance for multiple agents. In our previous work [24,25], we have introduced a solution for the collision-avoidance problem of dynamic obstacles, and proposed a method for verifying this function. In this paper, we focus on synthesizing static mission plans, while assuming that the dynamic avoidance among agents as well as other moving obstacles functions correctly.

3.2 Problem analysis

Algorithms such as Theta* [15] and RRT [29] are capable of computing collision-free paths between two milestones. We adopt the Theta* algorithm to solve path-planning in this study and the experiments, as the algorithm is especially good at generating smooth paths with any-angle turning points, in 2-D maps. Note that, our toolset (introduced in Sect. 7) supports more path-planning algorithms. After the paths are calculated, the execution order of tasks on milestones must be decided to achieve correct and efficient strategies. Based

on the requirements from VOLVO CE, we formulate and categorize the requirements of tasks as follows:

- *Milestone Matching* Tasks must be performed at the right milestones, e.g., digging stones must be carried out at stone piles.
- *Task Sequencing* The task execution order must be correct, e.g., unloading stones into the primary crusher must be executed after digging stones is finished, but before loading stones starts.
- *Timing* All tasks that contribute to the goal (e.g. transporting 10 tons of stones to the secondary crusher) must be finished within a prescribed time (e.g. within 1 hour).

Task scheduling now reduces to synthesizing a plan of task execution such that, by following the plan, agents can work independently or collectively to accomplish the entire mission according to the requirements. The classic scheduling problem called the *job-shop* problem [19] is a simplified version of the task scheduling. Being an NP-hard problem, even a simple instance of the job-shop problem with very restrictive constraints remains difficult to solve [1]. Additionally, our task-scheduling problem poses some unique extra challenges, as described in the next section. For simplicity, henceforth, we call the problem of path planning and task scheduling for autonomous agents as *mission planning*.

3.3 Non-determinism and scalability of mission planning

Different from the classic job-shop problem, there are two types of uncertainties existing in the environment that must be considered in the *mission-planning* phase, that is, the *non-deterministic execution time* of tasks and *non-deterministic duration* of agent movement.

- *Non-deterministic task execution time* The execution time of a task is usually a time interval between the BCET (best-case execution time) and WCET (worst-case execution time) of the respective task.
- *Non-deterministic movement time* The devices at some milestones could be exclusively occupied by agents. Therefore, other agents that are approaching these milestones must wait until those devices are released, respectively, and then start their tasks. This yields a non-deterministic movement time.

These features complicate our task scheduling even more than in the classic job-shop case. Our target is not only calculating mission plans, but also guaranteeing their correctness, that is, showing that the synthesized mission plan (a.k.a., strategy) satisfies all the requirements, and that it is complete, namely, covers all eventualities in the environment.

In our previous work [22], we have proposed an approach called TAMAA, based on the model-checking technique, to synthesize mission plans for agents. This approach can automatically generate mission plans, assuming feasible numbers of milestones and tasks up to 100. However, the approach cannot cover all eventualities when the environment is non-deterministic. Additionally, when the number of agents exceeds 5, TAMAA exhausts the physical memory due to the notorious state-space explosion problem of model checking [14].

4 Overall description of the solutions

Facing the limitation of TAMAA, we propose two solutions in this paper, that is, (1) a game-theoretic synthesis (i.e., TIGA), and (2) a simulation-based synthesis (i.e., MCRL).

Table 1 lists the solutions and their characteristics. TAMAA uses UTA as the modeling language and is suitable for 1-player games, in which agents have the full control over their environment. As depicted in Fig. 3a, the agent models in TAMAA have no uncontrollable actions, which means the agents can totally control their movement and task execution times. Therefore, the goal of TAMAA is to find the best mission plans that finish all tasks the fastest.

However, strategies of $1\frac{1}{2}$-player and 2-player games can only choose controllable actions, whereas the uncontrollable actions are taken by the environment either non-deterministically in 2-player games, or stochastically in $1\frac{1}{2}$-player games (Fig. 3b). Therefore, the goal of 2-player games, solvable by UPPAAL TIGA and MCRL, is to find the comprehensive strategies that enable the agents to finish their tasks no matter which and when uncontrollable actions are taken. Taking into account the probabilities of performing the uncontrollable actions, the goal of $1\frac{1}{2}$-player games, solvable by UPPAAL STRATEGO, is to find the strategies that have the highest probability of finishing all tasks.

In summary, different methods are suitable for different applications, and have their own advantages and disadvantages. When stochastic behaviors are observed in the system, $1\frac{1}{2}$-player games and UPPAAL STRATEGO can provide a suitable solution. When agents can fully control their task execution times, 1-player games and TAMAA can be the right choice (Sect. 5). When the task execution times are flexible rather than fixed and the uncontrollable actions are non-deterministic, UPPAAL TIGA (Sect. 5) and MCRL (Sect. 6) are capable of handling the problem.

UPPAAL TIGA is sound and complete in the sense that when a strategy is synthesized, it is guaranteed to be correct by construction, and conversely, when such a strategy exists in the state space of the model, UPPAAL TIGA is able to find it. However, UPPAAL TIGA suffers from the scalability problem as the method relies on the exhaustive graphic search. MCRL uses a

Table 1 Summary of all solutions

	TAMAA	TIGA	MCRL	STRATEGO
Model	UTA	TG	TG & STG	STG
Game	1 player	2 player	2 player	$1\frac{1}{2}$ player
Techniques	Model checking [22]	Symbolic on-the-fly algorithm [13]	Reinforcement learning & Model checking [23]	Reinforcement learning [18,26]

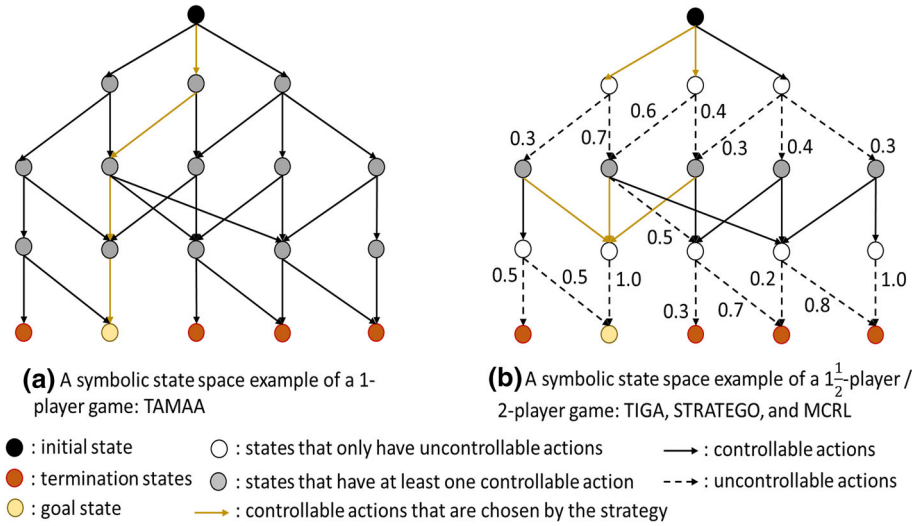


Fig. 3 Examples of symbolic state spaces of different models of games. Probabilities in **b** are only used in $1\frac{1}{2}$-player games

simulation-based method for synthesis and proposes a post-verification of the synthesized strategies, which alleviates the scalability problem while sacrificing the completeness of the method, that is, although MCRL has the ability to deal with more agents than UPPAAL TIGA, it does not guarantee to synthesize a strategy even if such strategy exists.

We will introduce these methods in detail in Sects. 5 and 6, and then compare their performance in different application scenarios in Sect. 8.

5 Solution 1: game-theoretic synthesis

In this section, we introduce the first solution, that is, our *game-theoretic synthesis*, which is based on an exhaustive search of the state spaces of agent models. We have two methods belonging to such kind of synthesis, namely the original TAMAA [22] and TAMAA in UPPAAL TIGA [5]. As aforementioned, the original TAMAA is designed to solve 1-player games, whereas TAMAA in UPPAAL TIGA leverage the models of TAMAA and the algorithmic method of synthesis of UPPAAL TIGA to synthesize complete plans that take into account any (possibly antagonistic) environmental action. First, we overview TAMAA, which provides an automatic model generation and synthesis of mission plans for 1-player games.

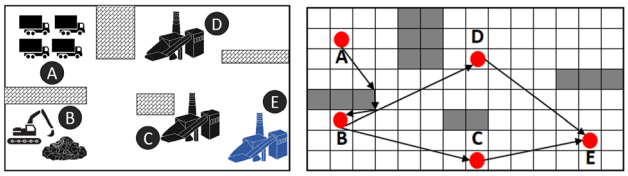
5.1 Overall description of TAMAA

TAMAA [22] enables users to configure their agents, tasks, and working environment in a graphical user interface (GUI), and automatically generate UTA networks that model the movement and task execution of agents. After users finish the configuration, UPPAAL is called to verify the UTA models in order to generate runs that satisfy various properties. The verification used in TAMAA is not for checking if the model is correct or not, but for generating runs of the model, which are then parsed to generate mission plans.

5.2 Mission-plan synthesis by TAMAA

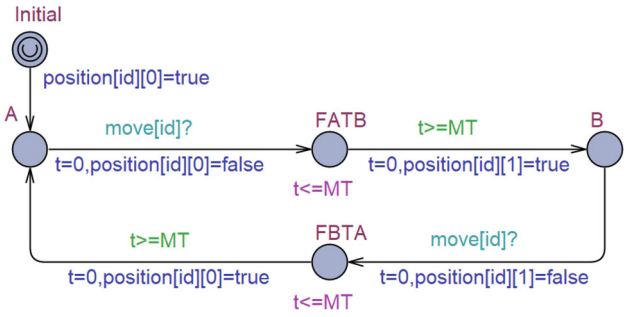
To pave the foundation of the synthesis method, we first elaborate the UTA models generated by TAMAA by an example illustrated in Fig. 4a. These models are also used in the improved solution of TAMAA with some slight adjustments (see Sect. 5.3).

In the quarry example, there are four autonomous trucks starting at 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*. A wheel loader is working at milestone *C* to dig stones and load them into the trucks. Only the autonomous trucks are the agents that we are interested in. First, the environment is

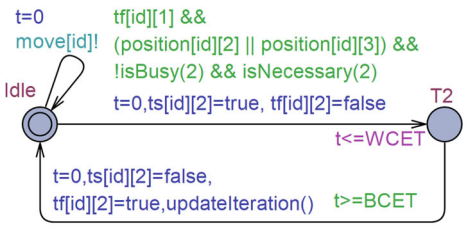


(a) An example of quarries **(b)** Map decomposition and the calculated paths

Fig. 4 An example of calculating paths by decomposing the map and running the Theta* algorithm



(a) Part of a movement UTA



(b) Part of a task execution UTA

Fig. 5 TA models of an agent's movement and task execution

decomposed into a Cartesian grid and the Theta* algorithm [15] is executed to calculate the shortest paths among milestones *A*–*E* (See Fig. 4b). Note that the trucks only need to choose one primary crusher at position *C* or *D*, to unload stones.

Next, UTA models are automatically generated by TAMAA, based on the shortest paths. For brevity, in Fig. 5a, we show a part of the UTA model in UPPAAL describing the movement of the autonomous trucks between milestones *A* and *B*. The movement to other milestones is modeled in a similar way. *Locations* `A` and `B` represent milestones *A* and *B*, respectively. The outgoing edge from the urgent initial location to location `A` indicates that the trucks start from milestone *A*. *Locations* `FATB` and `FBTA` are created to count the traveling time between `A` and `B`.

A constant variable `MT` stores the traveling time. The agent is only allowed to move when it is not executing any tasks. Therefore, *channel* `move[id]` is used to synchronize the transitions in the movement UTA with the task execution UTA (Fig. 5b) so that the moving actions are only enabled when the agent is idle, where the variable `id` refers to the current agent in both the movement UTA and the task execution UTA. A two-dimensional Boolean array named `position` is updated in this UTA, in which each element stores whether a certain milestone is being occupied by an agent. To model the agents' movement on the paths in Fig. 4b, TAMAA instantiates movement UTA similar to Fig. 5a.

The task execution UTA models the actions that an agent can choose to execute at a milestone. One such UTA is partly depicted in Fig. 5b, where location `Idle` represents the status of "no operation," when the agent is allowed to move, and location `T2` represents the task of unloading stones into a primary crusher. The self-loop of location `Idle` labeled by *channel* `move[id]` regulates the movement UTA to start to move only when the task execution UTA is at location `Idle`. Two Boolean arrays named `ts` and `tf` are updated in the UTA, representing whether a task has been started or finished, respectively. Assuming trucks need to iterate their tasks multiple times before transporting all the stones, the guard of the incoming edge of location `T2` enables this edge if the following conditions are *true*: (i) the task of loading stones from the wheel loader is done (i.e., `tf[id][1]` is *true*), (ii) the agent must be at milestone *C* or *D*, where the primary crushers are located (i.e., `position[id][2]` ‖ `position[id][3]` is *true*), (iii) no other agent is executing this task (i.e., `!isBusy(2)` is *true*), and (iv) the task has not been done in this round of transportation (i.e., `isNecessary(2)` is *true*). Location `T2` has an invariant indicating that the execution time of the task must not exceed its WCET. Similarly, the guard on the outgoing edge of location `T2` ensures that task is finished after the execution time is greater than or equal to BCET. The function `updateIteration()` updates the integer of task iteration.

After the resulting UTA model is automatically generated by TAMAA, properties that formalize the requirements mentioned in Sect. 3.2 are also generated by using the configuration information and well-designed TCTL templates. The *Timing* requirement is used for synthesizing mission plans that finish all tasks within a prescribed time limit. Others are for verifying if the models guarantee that the mission plans are functionally correct. For brevity, we only show the TCTL property of the *Timing* requirement used in UPPAAL. The rest of the properties are reported in our previous work [22]. The TCTL reachability Query (3) checks if agents can accomplish their missions within *TL* time units, where `ite` is an integer array storing the iteration of the tasks, that is, finishing all tasks once counts for one round, `x` is a clock variable that is never reset, `N` and `M` are two integers indicating the number of agents and the requested iterations of tasks, respectively:

```
E<> ((forall(i:int[0,N-1]) ite[i]>=M)&&
x ≤ TL)                                    (3)
```

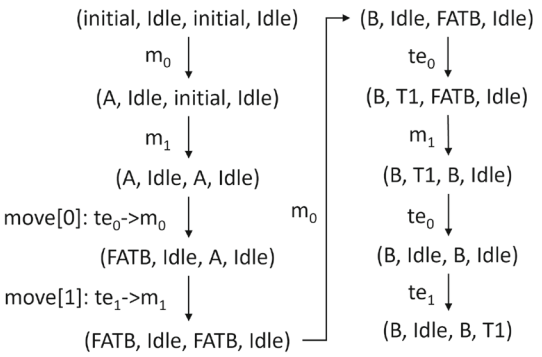


Fig. 6 A segment of a run generated by TAMAA

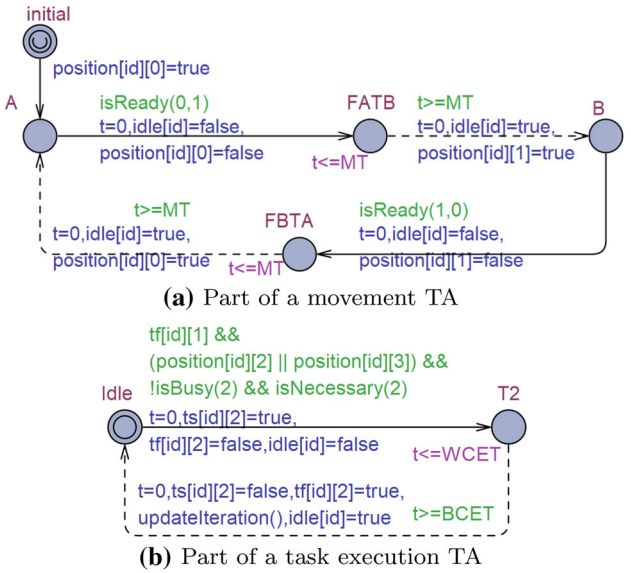


Fig. 7 TG models of an agent's movement and task execution in UPPAAL TIGA

The target of mission planning in a 1-player game is to find the run that reaches the goal state where, for example, agents finish all the tasks. Figure 6 depicts a segment of such run belonging to a model of 2 agents, where states of the models are symbolically represented by the locations of the UTA, $\mathtt{m}_i$ and $\mathtt{te}_i$ stand for actions in movement and task execution UTA of agent i, respectively, and $\mathtt{move[i]:te}_i\mathtt{->m}_i$ stands for the synchronized actions of starting to move. As depicted, all the actions are controllable by the agents (i.e., solid lines), which consecutively or alternately move the respective agent and execute tasks. They can stay at the same milestone (e.g., `B`) but cannot execute the same task (e.g., two `T1` cannot appear at the same state) unless the agents are not mutual exclusive of the task.

As explained in Sect. 4, runs like the one in Fig. 6 are mission plans of 1-player games. To obtain such runs, TAMAA uses the model checker of UPPAAL to verify the UTA models of agents against reachability properties in the form of Query (3). If the properties are satisfied, UPPAAL can generate runs that can be either the fastest, shortest, or random run, respectively. Hence, TAMAA can generate these three kinds of mission plans.

However, when the problem becomes a $1\frac{1}{2}$-player game or a 2-player game, TAMAA is not able to solve it, because the task execution times are decided by the uncontrollable actions taken by the environment. We need another method to deal with these problems such that the mission plans can cover all possible scenarios, even in the face of an antagonistic environment.

5.3 Synthesizing strategies in UPPAAL TIGA

In this subsection, we apply UPPAAL TIGA instead of UPPAAL to synthesize strategies defined by Definition 2, which serve as the complete mission plans that the original TAMAA is not able to synthesize.

We recast the models of TAMAA from the UTA formalism into the TG formalism of UPPAAL TIGA as follows. As depicted in Fig. 7a, the edge from location `FATB` (resp., `FBTA`) to location `B` (resp., `A`) is marked as uncontrollable. This change indicates that the decision of choosing a milestone to visit is made by the agent, whereas the duration of the movement to reach the milestone is determined by the environment. Note that the invariant on `FATB` (i.e., `t<=MT`) and the guard on the outgoing edge of `FATB` (i.e., `t>=MT`) force the duration to be `MT`. Similarly, in Fig. 7b, the edge from location `T2` to location `Idle` is marked as uncontrollable, indicating that the duration of the task is determined by the environment.

Besides the change of uncontrollable actions, the synchronization between the task execution UTA and movement UTA is removed to avoid the input non-determinism of random simulation in UPPAAL. Instead, a global Boolean array `idle` is introduced in the TG models to store whether the agents are idle or not. This array is used in the function named `isReady` in the movement TG, which returns *true* when the corresponding element in `idle` is *true* and the agent has not finished its requested iteration of tasks.

Query (3) is also adjusted to synthesize complete strategies that deal with the non-determinism of the environment, as follows:

```
strategy st = control: A<>
  ((forall(i:int[0,N-1])ite[i] ≥ M)&&
  x ≤ TL)                                    (4)
```

Query (4) applies the universal quantifier `A` on runs and the "eventually" temporal operator `<>` on states, which means that the synthesized strategy `st` must always guide the agents

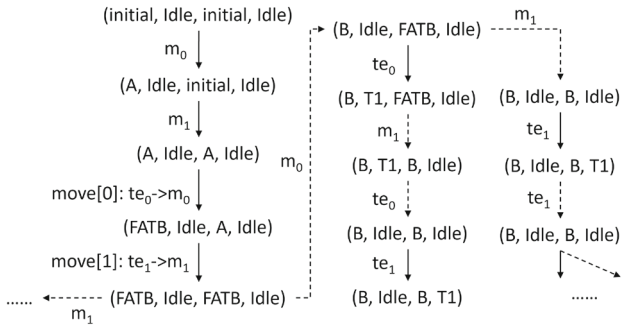


Fig. 8 A segment of a strategy generated by UPPAAL TIGA

to finish their tasks for M rounds within TL time limit, no matter how long the time of task execution is.

As depicted by Fig. 8, the runs generated by UPPAAL TIGA contain controllable (solid lines) and uncontrollable actions (dashed lines). The other notions of the figure are the same as in Fig. 6. The first four steps in Figs. 6 and 8 are the same, being all controllable actions. The fifth step in Fig. 8 starts to be different, because it is an uncontrollable action, which means that the environment decides which actions to perform, instead of the agents. Assuming that the agents travel at the same speed, at state `(FATB, Idle, FATB, Idle)`, the environment can choose agent 0 to arrive at milestone *B* first via the uncontrollable action in m_0; or choose agent 1 to arrive first via the uncontrollable action in m_1. The actions of finishing tasks are also uncontrollable, so the task execution times are uncertain from the agents' point of view. The strategies synthesized by UPPAAL TIGA are complete in the sense that no matter which and when uncontrollable actions are taken, the agents can always finish their tasks with respect to various requirements by following the strategies.

Although the strategies are now complete, since UPPAAL TIGA is also (in the worst case) exhaustively exploring the state space to synthesize strategies, the scalability problem of TAMAA still exists in UPPAAL TIGA. As depicted in Table 2, the number of explored states, and the computation time increase exponentially with the agent number growing linearly, which implies that UPPAAL TIGA encounters the state space explosion.

6 Solution 2: simulation-based synthesis

In this section, we introduce our second solution for the task-scheduling problem, which is based on simulation and learning. First, we describe the root of the state-space explosion problem that the both the original and improved versions of TAMAA have.

6.1 State-space exploration of TAMAA

The states of the agent models are the Cartesian product of states in each individual agent. Therefore, the state space of multiple agents grows exponentially with the number of agents growing linearly. The interleaving actions among the agents also increase the state space. Running TAMAA in UPPAAL TIGA requires searching the state space of the model. The essence of the method is about searching and storing the state space in order to find the runs that reach (respectively, avoid) certain states for reachability properties (respectively, safety properties). Since it relies on an implementation of an on-the-fly symbolic algorithm, UPPAAL TIGA may terminate before having explored the entire state space, which alleviates the state-space explosion problem. However, the searching algorithm is either breadth-first, depth-first, or random, which is not heuristic because it constantly follows the same order of searching without using the historical information of the searched state space. Therefore, the synthesis method in UPPAAL TIGA can take a long time to find the desired runs when the state space is large. The simulation-based synthesis, which is presented in the next subsection, improves the method in this aspect.

6.2 Learning strategies

Instead of using a symbolic and potentially exhaustive method, we study here the use of simulation-based synthesis algorithms such as *Q-learning* [38]. Rather than exploring the state space exhaustively, simulation-based methods sample the state space strategically, which happens often in a reactive manner, hence they can avoid state-space explosion. Nonetheless, simulation-based approaches sacrifice completeness over speed of synthesis, but gain the ability to accommodate a stochastic resolution of environment choices.

In this subsection, we go through the workflow of the new version of MCRL, which is integrated with UPPAAL STRATEGO. In the rest of Sect. 6, we introduce the new features of the new MCRL while briefly introducing the functions and parameters in UPPAAL STRATEGO. For technical details of UPPAAL STRATEGO, readers are referred to the literature [18,26].

As depicted in Fig. 9, MCRL explores the state space of the TG model via random simulation at the initial step, during which runs of the model are sampled. These runs serve as input to the learning algorithm to compute the rewards or penalties of the state-action pairs. As a result, the pairs belonging to the runs that reach the states where tasks are finished faster than those in other runs are assigned with higher rewards, whereas the pairs that end up into deadlocked states, or are wondering meaninglessly, are assigned with lower rewards or even penalties. The accumulated values (i.e., rewards or penalties) of the state-action pairs contribute to synthesize an intermediary strategy, which is then used

Table 2 Performance evaluation of synthesis in UPPAAL TIGA with different number of agents running 3 tasks among 3 milestones

Number of agents	Number of explored states	Computation time
2	775	5 ms
3	222,88	220 ms
4	764,001	18.1 s
5	33,312,229	53.8 mins
6	Out of memory	Unknown

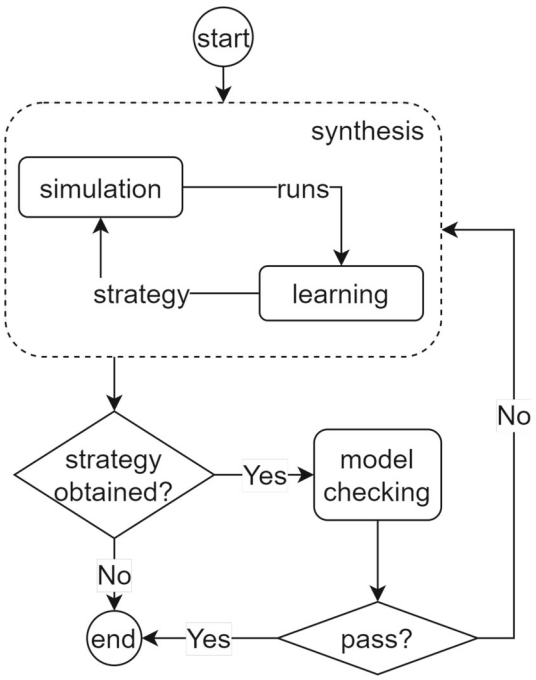


Fig. 9 Workflow of MCRL

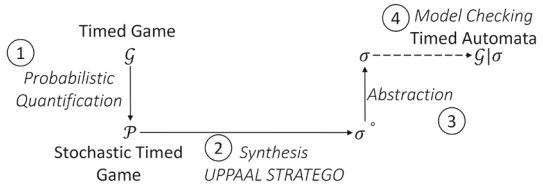


Fig. 10 Overview of various models and strategies and their relations in UPPAAL TIGA and UPPAAL STRATEGO (adapted from [18])

in the next round of simulation until a user-defined number of runs is sampled. Specifically, the simulator exploits the intermediary strategy in its following rounds of simulation by increasing (respectively, decreasing) the probabilities of choosing the actions with higher values (respectively, lower values). In this way, the simulator can reach the goal state faster and easier than the previous rounds of simulation do. This integration of simulation and learning is not provided by the initial version of MCRL [23].

When a user-defined number of runs is sampled, a strategy is considered to be produced. The simulation-based synthesis cannot guarantee the correctness of the strategies. Therefore, we propose a post-verification of the strategies by using model checking. Specifically, the TG models are verified together with the synthesized strategies. When the model checker encounters multiple controllable actions during verification, it enquires the strategy to choose the ones with the highest values. Details of the verification are presented in Sect. 6.2.3. Strategies that pass the verification are guaranteed to be correct in the sense that they satisfy the temporal constraints of requirements. If the verification fails, a new iteration of the synthesis and verification can be carried out, where the user-defined number of runs is increased for a more thorough learning. In addition, the state space of the model is restricted by the synthesized strategy, which enables MCRL to deal with more complicated problems than TAMAA and UPPAAL TIGA do.

In the following subsections, we introduce the key definitions, algorithms, and techniques that are used in the new version of MCRL. Since UPPAAL STRATEGO integrates UPPAAL, UPPAAL SMC, and UPPAAL TIGA, the following algorithms and techniques are designed and implemented collectively in UPPAAL STRATEGO.

6.2.1 Model conversion

The initial step of MCRL is random simulation (see Fig. 9). The non-deterministic choices of actions in the synthesis of UPPAAL TIGA are replaced by random sampling of actions in the simulation. Consequently, the TG must be converted into STG by assigning probabilities to actions.

Figure 10 shows the model conversion in the process of learning and verifying a strategy.

In step 1, the task execution TG is changed to an STG, where the probability of finishing a task between its BCET and WCET is uniformly distributed. Note that the uniform distribution can be changed to a user-defined distribution that can make the agents finish their tasks easier, in the simulation. However, it does not change the fact that the synthesized strategies lack a correctness guarantee, which means that the post-verification is needed anyway. The reason why we choose to use the uniform distribution is because it is the default

distribution on time-bounded delays in UPPAAL STRATEGO.[3] Therefore, the syntactic structure of the TG does not need to be changed. We name the first step *probabilistic quantification* because it assigns quantitative probabilities to the actions that are originally chosen non-deterministically in UPPAAL TIGA.

Step 2 is MCRL's synthesis (also seen in Fig. 9), which learns a stochastic strategy σ° based on the STG. σ° is then abstracted to a strategy that does not contain any probability, in step 3. The abstraction of stochastic strategies is introduced in Sect. 6.2.3. In the final step, the TG and the synthesized strategy σ are verified together by the model checker of UPPAAL STRATEGO.[4] This is supported by queries in the form of Query (7) that is introduced in Sect. 6.2.3.

The model conversion does not spoil our assumption of the environment, because the probabilities assigned to the uncontrollable actions in the TG are only used in the learning phase. The formal verification of the synthesized strategy is still by exhaustive model checking, which guarantees that the agents satisfy the requirements regardless of how the environment behaves.

6.2.2 Q-learning algorithm

Although we adopt Q-learning [38] in this work, our framework is open for extension with any other learning algorithms. In order to apply Q-learning on our STG models of the agents, we first define the states and actions of the Q-table generated by the learning algorithm. To differentiate the states of STG, we define Q-states and Q-actions as follows.

Definition 5 (*Q-State*) A Q-state is defined as the following tuple:

$$\mathcal{QS} =< RT, CT, CP, ST >,$$

where:

- $RT \in \mathbb{N}^d$ is a set of natural numbers denoting the iteration of executing all tasks for each agent, where d is the number of agents,
- $CT \in \mathbb{N}$ denotes the index of the current task,
- $CP \in \mathbb{N}$ denotes the index of the current milestone,
- ST is a set of Boolean variables encoding the respective execution statuses of tasks (EST) of all agents.

Definition 6 (*Q-Action*) A Q-action is defined as the following tuple:

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

where:

- $MT \in \{1, 2\}$ denotes the type of motion, i.e., 1 : moving, 2 : executing a task, and
- $TT \in \mathbb{N}$ denotes the target of the motion, which can be a milestone or a task.

In practice, "*RT*" is declared as an array of integers in our UTA models. "*CT*" and "*CP*" are represented by the current locations of the movement UTA and task execution UTA, respectively. "*ST*" is declared as a two-dimensional array of Boolean variables that stores the execution statuses of tasks (*EST*), that is, *finished* (*true*) or *unfinished* (*false*), for all agents in the environment. "*TT*" can be the index of the target milestone, or the index of the next task.

Note that a *Q-state* does not contain continuous variables such as clocks, because it is impossible to sample all the possible values of continuous variables in the simulation. Moreover, the mission-planning problem concerns the *EST* of agents, which are covered by *Q-states* already. Introducing other variables, e.g., a global clock variable that measures the entire time of mission execution, would be redundant. In addition, to symbolize Q-states with clock variables, we need to use *zones* [6] instead of their concrete values, which complicates the problem unnecessarily. The existence of "*ST*" in *Q-states* requires the agents to communicate with each other, which introduces overhead and unreliability in the implementation of these agents. However, to solve the mission-planning problem with uncertain task execution times, this cost is necessary.

To apply Q-learning, we need to define a formula to calculate the rewards for state-action pairs. The rewards should encourage the agents to accomplish their tasks as fast as possible. Hence, a global clock variable named `gt` that measures the total execution time of agents is defined in our UTA model, although it is not included in the *Q-states*. UPPAAL STRATEGO provides a special query [26] that allows us to simulate the model, sample the specific runs, and pass them to the learning algorithm (e.g., Q-learning):

```
strategy opt = minE(x)[<=T]{dv}->{cv}:<> P
```
(5)

In Query (5), `minE(x)` simulates the model while executing the learning algorithm to minimize "`x`," which can be a variable or an expression. Parameter `T` is the maximum simulation time, `dv` is a set of discrete variables, and `cv` is a set of continuous variables. The learning algorithm observes

[3] The uniform distribution is used in UPPAAL SMC by default. UPPAAL STRATEGO includes UPPAAL SMC.

[4] The model checker is UPPAAL [6], which is included in UPPAAL STRATEGO.

the state space of the model partially, by detecting the values of the variables in dv and cv. The formula "<>P" is a (T)CTL property satisfied by the runs sampled from the simulation. These runs are used as input to the learning algorithm to evaluate state-action pairs. In this mission-planning problem, the global clock variable gt is x, the attributes of *Q-state* constitute dv, cv is an empty set, and P is as follows, being also used in Query (4):

```
(forall(i:int[0,N-1]) ite[i] ≥ M)&&gt ≤ TL
```
(6)

Algorithm 1 presents the process of executing queries in the form of Query (5) in UPPAAL STRATEGO. Parameters *stg, iterationNum, totalNum*, and *goodNum, formula* represent the STG model, the user-defined number of iterations of learning, the user-defined maximum rounds of simulation, the maximum number of runs that satisfy the property, and the property (<> P in Query (5)), respectively.

Algorithm 1: Simplified algorithm behind the minE-query of UPPAAL STRATEGO

```
1  Main(stg, iterationNum, totalNum, goodNum, formula)
2  int iterations = 0
3  int bestFitness = ∞
4  Strategy best = empty
5  Strategy aStrategy = empty
6  for iterations < iterationNum do
7      int totalRuns = 0
8      int goodRuns = 0
9      for totalRuns < totalNum do
10         Run aRun = simulate(stg, aStrategy)
11         if aRun satisfies formula then
12             aStrategy = learn(aRun)
13             goodRuns++
14             if goodRuns ≥ goodNum then
15                 break
16         totalRuns++;
17     if goodRuns ≥ goodNum then
18         fitness = evaluate(aStrategy)
19         if fitness < bestFitness then
20             bestFitness = fitness
21             best = aStrategy
22     iterations++
23 return best;
```

At lines 4 and 5 of Algorithm 1, two empty strategies are defined, which are two arrays for storing Q-tables, in practice. In line 10 random simulation starts, from which the runs that satisfy "<> P" (a.k.a., good runs) are sent to the learning algorithm (line 12), which can be an internal function of UPPAAL STRATEGO or a pre-compiled library. The check of satisfaction of "<> P" is done by UPPAAL STRATEGO. For details, we refer the interested reader to the literature [18]. The learning algorithm calculates the rewards of the state-action pairs in these good runs based on the value of "x," and stores the rewards in the variable *aStrategy* (line 12).

The simulation and learning terminate under two conditions: (i) when the total rounds of simulation reach the limit (line 9), or (ii) when the number of good runs reaches the limit (line 15). When the simulation terminates in case (i), no strategy is generated as the good runs collected from the simulation do not support generating a complete strategy; if the simulation terminates in case (ii), a strategy is generated and stored as a *Q-table*. Lines 17 to 21 evaluate the learned strategy. The *fitness* of a strategy is the expectation of "x" when the model is under the control of the strategy up to the horizon provided in the query. If the query is minE (respectively, maxE), the evaluation observes the fitness of the current strategy and judges if its value is less (respectively, larger) than the value of the best strategy, and updates the best strategy accordingly.

6.2.3 Verification of the synthesized strategies

As depicted in Fig. 9, after a strategy is synthesized, the model checker of UPPAAL STRATEGO is employed to verify the TG of the system under the control of the strategy. UPPAAL STRATEGO provides a special query to realize this function, which is shown in Query (7), where P is a Boolean expression, e.g., Query (6), opt is the strategy that is synthesized by Query (5) and that controls the behavior of the model:

```
A<> P under opt
```
(7)

Specifically, when UPPAAL STRATEGO reaches a state where it faces multiple controllable actions, the model checker can filter out non-optimal choices (according to the strategy) from the exploration of the system.

We extend UPPAAL STRATEGO such that a subset of strategies generated by Query (5) can also be verified by Query (7). This is an extension of the original work on UPPAAL STRATEGO [18] where only strategies generated by the game-theoretic synthesis of UPPAAL TIGA [5] can be verified. The subset of strategies here refers to the ones that do not have clock variables. As defined in Definitions 5 and 6, the strategies (i.e., Q-tables) of MCRL do not contain clocks.

This verification is step 4 of the method (see Fig. 10). For the users of this method, synthesizing a strategy and verifying it are two consecutive operations of running Queries (5) and (7). However, there are two important steps of model conversion that are executed underneath, by the tool: *probabilistic quantification* and *abstraction* (Fig. 10). Probabilistic quantification is explained in Sect. 6.2.1. Now we introduce the abstraction from stochastic strategies σ° of STG to strategies σ of the corresponding TG. As stated in Definitions 4 and 2, σ° assigns probabilities to the controllable actions of

the agents, whereas σ explicitly informs the agents what is the next action to do at each state.

In practice, given a Q-table that contains the values of state-action pairs defined in Definitions 5 and 6, we construct strategies σ by using the rewards as the priorities of choosing actions at the corresponding states, that is, the actions with the highest values are always chosen by the model checker. When multiple actions have the same value at some states, the model checker will exhaustively select each one of them to execute and check, in a non-deterministic manner. In this way, we can verify if the strategies synthesized by Q-learning are guaranteed to be complete and correct in the sense that the new models of the agents, which are controlled by the strategies, satisfy the requirements considering all the possible task execution times. In addition, the state-space explosion of the original TAMAA is overcome, since the state space that is explored by the model checker for verifying the new models is much reduced by the strategies.

To guarantee that the synthesized strategies meet all the requirements mentioned in Sect. 3.2, we design queries as presented below. In these queries, $\texttt{te}_n$ and $\texttt{move}_n$ are the task execution TG and movement TG of agent n, respectively. The variable `tf` is a two-dimensional Boolean array of agents' task execution statuses, e.g., finished, or unfinished, `x` is a clock variable, and `opt` is the synthesized strategy.

- *Milestone Matching* Query (8) checks that agent's n position is always at milestone P_i, when it is executing task T_i:

$$\texttt{A[] (te}_n\texttt{.T}_i \texttt{ imply move}_n\texttt{.P}_i\texttt{) under opt} \quad (8)$$

- *Task Sequencing* Query (9) checks if the precedent task T_{i-1} is always finished, when agent n is executing task T_i:

$$\texttt{A[] (te}_n\texttt{.T}_i\texttt{imply tf[n][i-1]==true) under opt} \quad (9)$$

- *Timing* Query (10) checks if the agents can always finish all their tasks within *TL* time units, where `N` is the number of agents, `M` is the requested tasks iteration number, and `TL` is an integer of time limit:

$$\texttt{A<> ((forall(i:int[0,N-1]) fin[i} \geq \texttt{ M) imply x} \leq \texttt{TL) under opt} \quad (10)$$

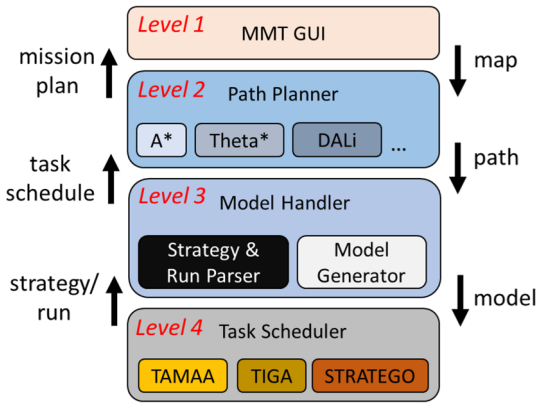


Fig. 11 The structure of the toolset

7 Tool support

In this section, we describe the automated support for our method, our toolset *MALTA*,[5] which is depicted in Fig. 11.

A GUI named *Mission Management Tool* (MMT) is designed at the top level to enable users to configure the map, agents, tasks, and milestones, etc., capturing the information of the environment. A module named *Path Planner* is designed at the second level, to support various path-planning algorithms, e.g., A* [35], Theta* [15], and DALi [33]. The *Path Planner* obtains the information of the map, including the navigation area, forbidden areas, milestones, etc., and calculates paths among the milestones and avoid all the forbidden areas. DALi can even select paths intelligently, when encountering temporary obstacles, crowed areas, etc. We refer the reader to literature [33] for details.

In the experiments of this paper, we use Theta* for path planning, due to its capability of calculating smooth paths that minimize the amount of sharp turns. The *Path Planner* sends the paths to the third level, where a module named *Model Generator* is designed to produce the TG/UTA models of agents, automatically. These models are used in the fourth level called *Task Scheduler*, which invokes TAMAA, UPPAAL TIGA, or UPPAAL STRATEGO, based on the requirement and scale of the problem of synthesizing strategies. Strategies (respectively, runs), synthesized by UPPAAL TIGA or UPPAAL STRATEGO (respectively, TAMAA), are then sent back to the third level, where a module named *Strategy & Run Parser* is designed to parse the strategy or runs into the format that is understandable for the second level. Last, the task schedule and the path plan are combined as a mission plan and shown in MMT GUI.

A detailed description of levels 1 to 3 of the toolset can be found in previous work [22]. We focus on level 4, *Task Scheduler*, in this section. In our previous work [23], we

[5] MALTA is published: https://github.com/rgu01/MALTA

have proposed an implementation of MCRL, which uses the *simulation* query in UPPAAL SMC to randomly simulate the models, gathers enough runs that satisfy a condition, and prints the rewards of the state-action pairs of the runs into text files. Next, the files are parsed and used as the source data for the Q-learning algorithm to populate Q-tables. The Q-tables are then injected back into the models. A new UTA named *conductor* is designed to read the Q-tables every time when the agent needs to make a decision. The *conductor* sends signals to the movement and task execution UTA, in order to control them to perform different actions according to the Q-table.

This implementation separates the data-gathering phase from the learning phase, so the rewards of state-action pairs accumulated in the data-gathering phase cannot easily be exploited for guiding the sampling in a strategic manner. In every round of simulation, the *simulator* explores the state space randomly, with unchanged probabilities of the actions. Moreover, the UTA models allow the most permissive behaviors of agents, such as wondering without executing any tasks. The separation of phases makes the simulation unlikely to reach the states where rare events happen, e.g., multiple iterations of tasks, or finishing a large number of tasks in a strict time frame. Therefore, the new version of the method embeds MCRL into UPPAAL STRATEGO to fix this inconvenience, which will be introduced in the next subsection.

7.1 Integration of task scheduler and UPPAAL STRATEGO

As shown in Algorithm 1, in the new version of MCRL, once a run that satisfies our requirement is obtained from the simulation (line 11), it is directly fed into the learning algorithm to synthesize a strategy (line 12). The strategy is not necessarily complete, but it is then input into the next round of simulation (line 10), where the *simulator* can exploit the existing strategy by using the rewards accumulated in the past rounds of simulation as the probabilities of actions (see Sect. 6.2.2). Therefore, the actions with higher rewards will be chosen more likely than the ones with lower rewards, and thus, the learning phase is accelerated.

After a certain rounds of simulation (the number is configurable), a candidate strategy is produced and sent to the model checker to verify if it guarantees to enable the agents to finish all tasks according to the requirement, regardless of how the environment reacts. This process iterates until the verification passes. In our previous implementation of MCRL [23], the movement and task execution UTA are modified, and a UTA named *conductor* is created to control the models according to the Q-table. In our current implementation, the original movement and task execution TG are directly verified in UPPAAL STRATEGO by running queries in the format of Query (7). When UPPAAL STRATEGO verifies the models against these queries, it calls back a function in the external library of the learning algorithm, where the Q-table is stored, whenever it faces multiple available controllable actions. This function searches the Q-table and returns the highest priority for the actions with the highest rewards to the model checker. UPPAAL STRATEGO then exhaustively explores the actions that have the same highest priority, but ignores the ones with lower priorities. In this way, the models' behavior is under the control of the strategy without introducing new models, such as a *conductor* UTA.

Additionally, the new version of MCRL is implemented in an extensible scheme. The learning algorithm is programmed in standard C or $C++$, and compiled into an external library, which UPPAAL STRATEGO calls back when learning is required. Hence, the users of the tool can replace the Q-learning with their own learning algorithms, and leverage the formal aspects of the method to guarantee the completeness and correctness of the synthesized strategies.

8 Experimental evaluation

In this section, we evaluate the improved version of TAMAA and the new version of MCRL in several experiments. The experiments are conducted on a laptop running an Intel Core i7 processor with 12 cores, 16 GB of RAM, and a 64-bit Linux OS.

8.1 Design of experiments

Figure 12 depicts a working environment of agents created in MMT. Our mission planner calculates paths that enable the agents to visit the milestones in a certain order so that they finish their tasks in a correct and efficient way.

According to previous investigation [22], the number of agents is the factor that impacts the computation time of the mission planners most profoundly. As shown in Table 2, UPPAAL TIGA could not handle more than 5 agents. Therefore, we vary the number of agents from 3 to 6 in the experiments in order to show that the new MCRL is capable of dealing with more agents than the improved version of TAMAA in UPPAAL TIGA. To demonstrate the extensibility of MCRL, the experiments are conducted on two versions of MCRL. One uses an external library of Q-learning and one uses the Q-learning function in UPPAAL STRATEGO [26], which are called external and internal Q-learning for brevity,[6] respectively. We experiment with both an internal and an external version of Q-learning to study the impact of (1) a fully extensible implementation of the learning algorithm,

[6] Although the internal Q-learning is a part of UPPAAL STRATEGO, MCRL provides a post-verification to it and thus it is called MCRL with internal Q-learning.

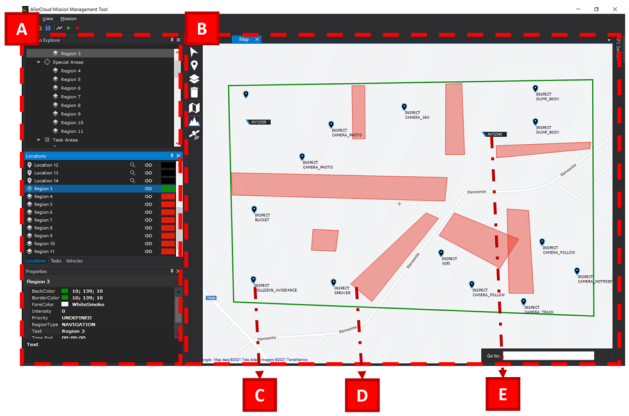


Fig. 12 A working environment of agents in MMT. Module *A* is the configuration panel, where users configure the parameters of the map, vehicles, tasks, etc. Module *B* is the map, where the environment is visualized. Synthesized mission plans will also be shown in this module. Pinpoints like *C* are the milestones, where tasks are assigned to. Red areas like *D* are the special areas, which can be fixed/temporary forbidden areas (a.k.a., static obstacles), crowed areas, etc. Tags like *E* are the initial positions of agents

and (2) the overhead of communication between UPPAAL STRATEGO and the external library. In addition, this construction allows us to define a custom strategy output format for an integration into our toolset *MALTA*. We also vary the number of milestones (correspondingly, tasks) to see how this factor influences the computation time.

8.2 Results of experiments

Table 3 shows the numbers of explored states and computation time for the three mission planners synthesizing mission plans for 3, 4, 5, and 6 agents, respectively.

The numbers of milestones and tasks are fixed, such that the difference of the results among the mission planners would only be caused by the increased number of agents. We run the experiments 5 times for each mission planner in each scenario containing different number of agents, and use the mean values as the results. Clearly, UPPAAL TIGA can only deal with situations with less than 6 agents, whereas MCRL can cope with 6 agents within reasonable computation times: 7.9 minutes or 14.8 minutes. The difference in computation times of the two versions of MCRL is due to the different implementations of Q-learning and the overhead of communication between UPPAAL STRATEGO and the external library. They collectively confirm the conclusion that MCRL outperforms TAMAA when the number of agents is large.

As depicted in Table 4, strategies synthesized by MCRL with the external and internal Q-learning are complete in the sense that they satisfy liveness queries in the form of Query (7). When the number of agents is greater than 4, internal Q-learning needs more simulation rounds to sample enough runs for learning than that of the external Q-learning. The reason for this is discussed in the next subsection.

Table 5 shows the number of explored states and computation time for the three mission planners synthesizing mission plans for 2 agents, but different numbers of milestones and tasks (tasks are assigned to milestones, thus N milestones imply N tasks). As presented in the table, the number of explored states and computation time of UPPAAL TIGA do not increase very fast with the increasing numbers of milestones and tasks, which is consistent with our previous investigation [22]. However, two versions of MCRL with the internal and external Q-learning perform much worse than UPPAAL TIGA when the numbers of milestones and tasks are greater than 5. Note that, when the numbers of milestones and tasks are 10, the rates of synthesizing complete strategies by using the external and internal Q-learning are lower than 10%. We discuss the reason for this result in the next subsection.

8.3 Discussion of the experimental results

As MCRL randomly searches the state space multiple times during the learning process, the numbers of explored states of these two planners do not reflect the size of the agent model. Hence, we compare the numbers of explored states obtained with UPPAAL TIGA in Tables 3 and 5, and conclude that the size of the state space of the agent model is mainly influenced by the number of agents. The numbers of milestones and tasks increase the state space much less significantly, but the trend of increase is still exponential.

The reason why MCRL with the external Q-learning needs less total rounds of simulation than that of the internal Q-learning is because the intermediary strategies are adopted in the external Q-learning during the course of simulation for a heuristic exploration of the state space. The difference of heuristic exploration in two versions of MCRL results in the different requested rounds of simulation, which also contributes to the worse performance of the simulation-based algorithms when the numbers of milestones and tasks are more than 8.

Currently, the learning algorithms can only leverage the "good" runs that satisfy our requested condition (e.g., Formula (6)). When the milestones and tasks are few, random simulation can easily get to the states where the property holds. As the numbers of milestones and tasks increase, it becomes increasingly unlikely to reach the terminal state by random simulations, which in turns implies that a guided search cannot occur in the simulation. Runs that do not satisfy the specified condition are not provided to the learning algorithm, and thus, do not contribute to the heuristic exploration of the state space. Therefore, in cases with large numbers of milestones and tasks, large numbers of simulation rounds are needed to generate enough "good" runs, which result in a high number of explored states and a long computation time.

Table 3 Explored states and computation time of synthesizing mission plans for different numbers of agents

	States	Time	Agents
TIGA	222,88	220 ms	3
MCRL with Internal Q-learning	143,044	572 ms	
MCRL with External Q-learning	428,550	2.0 s	
TIGA	764,001	18.1 s	4
MCRL with Internal Q-learning	772,619	2.2 s	
MCRL with External Q-learning	1,150,349	7.3 s	
TIGA	33,312,229	53.8 mins	5
MCRL with Internal Q-learning	9,822,914	38.2 s	
MCRL with External Q-learning	6,700,782	53.0 s	
TIGA	Out of memory	Unknown	6
MCRL with Internal Q-learning	10,322,666	7.9 mins	
MCRL with External Q-learning	100,901,760	14.8 mins	

The environment contains 3 milestones and 3 tasks

Table 4 The numbers of sampled traces and total simulation rounds that are needed for synthesizing strategies by using the MCRL with the internal and external Q-learning, as well as the completeness of the synthesized strategies

	Sampled traces	Total runs	Completeness	Agents
External Q-learning	100	2000	True	4
Internal Q-learning	100	2000	True	
External Q-learning	200	10,000	True	5
Internal Q-learning	200	20,000	True	
External Q-learning	200	100,000	True	6
Internal Q-learning	200	150,000	True	

Table 5 Explored states and computation time of three methods synthesizing mission plans for different numbers of milestones and tasks

	States	Time	Milestones & tasks
TIGA	11,746	61 ms	5
MCRL Internal Q-learning	136,113	347 ms	
MCRL External Q-learning	200,963	1.1 s	
TIGA	161,953	1 s	8
MCRL Internal Q-learning	49,489,463	3.4 mins	
MCRL External Q-learning	63,858,459	8.2 mins	
TIGA	586,124	3.9 s	10
MCRL Internal Q-learning	324,257,087	33.7 mins	
MCRL External Q-learning	324,283,558	46.4 mins	

The environment contains 2 agents

We leave the improvement of the method for future work, but hypothesize that the inclusion of negative reinforcement feedback (that is, runs that do not meet the goal will receive a penalty) will improve the performance of MCRL significantly.

In summary, when the number of agents is large, MCRL with the internal Q-learning is the first option because it scales better than UPPAAL TIGA and needs less computation time than MCRL with the external Q-learning does in this case. In the cases where the numbers of milestones and tasks are large, UPPAAL TIGA is the first option as it scales and provides strategies that are guaranteed to cover all possible scenarios. When the users need to embed their own learning algorithms in the method, MCRL with the external Q-learning is the first option because the learning module is an external library that can be replaced easily.

9 Related work

Synthesis of strategies for multiple autonomous agents has become an increasingly studied area. Wang et al. [37] attempt to address the scalability challenge of solving POMDP (Partially Observable Markov Decision Processes) with safe-reachability objectives. Similar to the bounded model-checking technique [8], their method constrains the state-space of the model by using a goal-constrained belief

space instead of the entire belief space. Bouton et al. [10] focus on a concrete scenario of autonomous cars: navigation in unsignalized intersections. Their method is based on POMDP and Monte Carlo sampling, thus avoiding the scalability problem. However, their method does not provide formal-verification-based guarantees of correctness. Nikou et al. [32] propose a solution to synthesize controllers of agents for path planning. Their synthesized controllers also satisfy complex high-level constraints of tasks. However, no proof of scalability with the number of agents is provided. Our approach is accompanied by a toolset that is capable of handling mission-plan synthesis for multiple agents, mitigating the associated lack of scalability caused by the numbers of agents, milestones, and tasks.

Similar to our work, some studies also combine formal verification with learning algorithms. The UPPAAL STRATEGO [18] tool facilitates both sample-based optimization and correct-by-construction controller synthesis. In addition, both these methods can be combined for safe and (near-)optimal synthesis. Basile et al. [3] use UPPAAL STRATEGO to solve the strategy synthesis problem for autonomous driving in a moving block railway system. They leverage the game-theoretic method to synthesize safe strategies and reinforcement learning to optimize the strategies. To achieve formal correctness of a learned controller, UPPAAL STRATEGO relies on learning under a prior construction of the safe controller, specifically guarding the learning against unsound actions. This is contrary to our simulation-based approach (namely, MCRL) where learning is conducted on the original models directly to synthesize strategies with no guarantee of correctness. The post-verification in MCRL adds correctness guarantees on the learned strategies, which eliminates the state-space explosion problem that exists in the original models.

Similar to TAMAA, the approach of Gleirscher et al. [21] is also based on graphic search. Their approach is able to synthesize and verify safety controllers for human-robot collaboration. Bersani et al. [7] present the PuRSUE (Planner for RobotS in Uncontrollable Environments) approach, which supports users to configure their robotic applications and automatically generate their controllers by using UPPAAL TIGA. The main difference between MCRL and theirs is that their synthesis is based on graphic search and thus limited on scalability.

Li et al. [31] focus on capturing complex and domain-specific requirements of robotic systems by using formal specification languages. Their method also makes the reward generation of the learning process interpretable and guarantees the satisfaction of specification, for critical components of the systems. The method proposed by Bouton et al. [11] enforces probabilistic guarantees on agents during the course of reinforcement learning. Brázdil et al. [12] provide algorithms for searching MDP (Markov Decision Processes) to verify various reachability properties. Legay et al. [30] present a scalable approach of verification for MDP. When comparing to these studies, we apply model checking on the learned strategies and facilitate the verification for complex models with large state spaces by using reinforcement learning, rather than constructing initially a safe restriction of the system. Our work is orthogonal to that of Brázdil, Legay and Bouton, that is, their methods could be utilized for the initial construction of strategies to demonstrate the nonexistence of rare events. Our method can be then used to verify their strategies. In addition, our method has the ability to handle timed systems and distributions over durations.

To the best of our knowledge, the earliest attempt to employ reinforcement learning for solving the state-space explosion problem of model checking is done by Behjati et al. [4]. The authors propose a bounded rational verification approach for on-the-fly model checking. However, this method is limited to LTL properties, and it has not been applied on autonomous agents.

10 Conclusion and future work

In this paper, we have presented our method of solving the mission-plan synthesis problem of multiple autonomous agents. The method is based on our tool named TAMAA and improves the original TAMAA with the ability of handling uncertain movement time and task execution time of agents. Additionally, our method, called MCRL, combines model checking with reinforcement learning, so that it is capable of dealing with more agents than the improved TAMAA, which applies model checking alone. MCRL provides a means for verifying and analyzing the synthesized mission plans by using model checking, to ensure that safety-critical requirements are met. The method is fully integrated with UPPAAL STRATEGO. We demonstrate MCRL's ability of handling multiple agents by experiments, and compare the result with the original and improved TAMAA. The number of explored states and computation time of MCRL increase much slower than the two versions of TAMAA when the number of agents increases. However, the improved version of TAMAA in UPPAAL TIGA performs better than MCRL when the number of agents is less than two but the numbers of milestones and tasks are more than five.

One of the future directions of work is to improve the learning algorithm of MCRL to perform better in environments with large numbers of milestones and tasks. Another future work direction focuses on estimating the existence of strategies of timed games before synthesis starts. Introducing variables that evolve continuously, e.g., time, energy consumption, in the models and strategies is another interesting direction of future research, which would dramatically complicate the strategy-synthesis problem.

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

Funding Open access funding provided by Mälardalen University.

References

1. Abdeddaı, Y., Asarin, E., Maler, O., et al.: Scheduling with timed automata. Theor. Comput. Sci. **354**(2), 272–300 (2006)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
3. Basile, D., ter Beek, M.H., Legay, A.: Strategy synthesis for autonomous driving in a moving block railway system with uppaal stratego. In: International Conference on Formal Techniques for Distributed Objects, Components, and Systems. Springer (2020)
4. Behjati, R., Sirjani, M., Ahmadabadi, M.N.: Bounded rational search for on-the-fly model checking of LTL properties. In: Symposium on the Foundations of Software Engineering. Springer (2009)
5. Behrmann, G., David, A., Fleury, E., Larsen, K., Lime, D., Nantes, E.: Uppaal-Tiga: Time for playing games! (tool paper). In: International Conference on Computer Aided Verification. Springer Berlin Heidelberg (2007)
6. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. Lecture Notes in Computer Science (2004)
7. Bersani, M.M., Soldo, M., Menghi, C., Pelliccione, P., Rossi, M.: Pursue-from specification of robotic environments to synthesis of controllers. Formal Aspects of Computing (2020)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Carnegie Mellon University (2003)
9. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Muñiz, M., Srba, J.: Stubborn set reduction for two-player reachability games. Preprint arXiv:1912.09875 (2019)
10. Bouton, M., Cosgun, A., Kochenderfer, M.J.: Belief state planning for autonomously navigating urban intersections. In: Intelligent Vehicles Symposium. IEEE (2017)
11. Bouton, M., Karlsson, J., Nakhaei, A., Fujimura, K., Kochenderfer, M.J., Tumova, J.: Reinforcement learning with probabilistic guarantees for autonomous driving. Preprint arXiv:1904.07189 (2019)
12. Brázdil, T., Chatterjee, K., Chmelik, M., Forejt, V., Křetínskỳ, J., Kwiatkowska, M., Parker, D., Ujma, M.: Verification of markov decision processes using learning algorithms. In: International Symposium on Automated Technology for Verification and Analysis. Springer (2014)
13. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: International Conference on Concurrency Theory. Springer (2005)
14. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: LASER Summer School. Springer (2011)
15. Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: any-angle path planning on grids. J. Artif. Intell. Res. **39**, 533–79 (2010)
16. David, A., Du, D., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., Sedwards, S.: Statistical model checking for stochastic hybrid systems. Preprint arXiv:1208.3856 (2012)
17. David, A., Jensen, P.G., Larsen, K.G., Legay, A., Lime, D., Sørensen, M.G., Taankvist, J.H.: On time with minimal expected cost! In: International Symposium on Automated Technology for Verification and Analysis. Springer (2014)
18. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal Stratego. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2015)
19. Fisher, H.: Probabilistic learning combinations of local job-shop scheduling rules. In: Industrial Scheduling. Prentice Hall, Englewood Cliffs (1963)
20. 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. Springer (1996)
21. Gleirscher, M., Calinescu, R., Douthwaite, J., Lesage, B., Paterson, C., Aitken, J., Alexander, R., Law, J.: Verified synthesis of optimal safety controllers for human-robot collaboration. Preprint arXiv:2106.06604 (2021)
22. Gu, R., Enoiu, E.P., Seceleanu, C.: TAMAA: UPPAAL-based mission planning for autonomous agents. In: ACM/SIGAPP Symposium On Applied Computing (2020)
23. Gu, R., Enoiu, E.P., Seceleanu, C., Lundqvist, K.: Verifiable and scalable mission-plan synthesis for multiple autonomous agents. In: International Conference on Formal Methods for Industrial Critical Systems. Springer (2020)
24. Gu, R., Marinescu, R., Seceleanu, C., Lundqvist, K.: Towards a two-layer framework for verifying autonomous vehicles. In: NASA Formal Methods Symposium. Springer (2019)
25. Gu, R., Seceleanu, C., Enoiu, E.P., Lundqvist, K.: Model checking collision avoidance of nonlinear autonomous vehicle models. In: Formal Methods 2021 (2021)
26. Jaeger, M., Jensen, P.G., Larsen, K.G., Legay, A., Sedwards, S., Taankvist, J.H.: Teaching stratego to play ball: Optimal synthesis for continuous space MDPs. In: International Symposium on Automated Technology for Verification and Analysis. Springer (2019)
27. Kempf, J.F., Bozga, M., Maler, O.: As soon as probable: Optimal scheduling under stochastic uncertainty. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer (2013)
28. Konda, V.R., Tsitsiklis, J.N.: Actor-critic algorithms. In: Advances in neural information processing systems (2000)
29. LaValle, S.M.: Rapidly-exploring random trees: a new tool for path planning. In: Technical Report (1998)
30. Legay, A., Sedwards, S., Traonouez, L.M.: Scalable verification of markov decision processes. In: International Conference on Software Engineering and Formal Methods. Springer (2014)
31. Li, X., Serlin, Z., Yang, G., Belta, C.: A formal methods approach to interpretable reinforcement learning for robotic planning. Science Robotics (2019)
32. Nikou, A., Boskos, D., Tumova, J., Dimarogonas, D.V.: On the timed temporal logic planning of coupled multi-agent systems. Automatica (2018)
33. Palopoli, L., Argyros, A., Birchbauer, J., Colombo, A., Fontanelli, D., Legay, A., Garulli, A., Giannitrapani, A., Macii, D., Moro, F., et al.: Navigation assistance and guidance of older adults across complex public spaces: the DALi approach. Intelligent Service Robotics (2015)

34. Pelánek, R.: Fighting state space explosion: Review and evaluation. In: International Conference on Formal Methods for Industrial Critical Systems. Springer (2008)
35. Rabin, S.: Game Programming Gems, Chapter a* Aesthetic Optimizations. Charles River Media (2000)
36. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press (2018)
37. 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. Springer (2018)
38. Watkins, C.J.C.H.: Learning from Delayed Rewards. King's College, Cambridge United Kingdom (1989)